SPIS TREŚCI

SPIS RYSUNKÓW

SPIS LISTINGÓW

SPIS TABEL

1. INTRODUCTION

2. BACKGROUND

2.1. DOMAIN OVERVIEW

In artificial intelligence, an evolutionary algorithm (EA) is a subset of evolutionary computation, a generic population-based metaheuristic optimization algorithm. An EA uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the quality of the solutions. Evolution of the population then takes place after the repeated application of the above operators.

Evolutionary algorithms often perform well approximating solutions to all types of problems because they ideally do not make any assumption about the underlying fitness landscape. Techniques from evolutionary algorithms applied to the modeling of biological evolution are generally limited to explorations of microevolutionary processes and planning models based upon cellular processes. In most real applications of EAs, computational complexity is a prohibiting factor. In fact, this computational complexity is due to fitness function evaluation. Fitness approximation is one of the solutions to overcome this difficulty. However, seemingly simple EA can solve often complex problems[citation needed]; therefore, there may be no direct link between algorithm complexity and problem complexity.

Overall the optimization methods can be divided generally into two groups: the gradient methods, that operate on a single potential solution and look for some improvements in its neighborhood, and global optimization techniques – represented here by so called evolutionary methods – that maintain large sets (populations) of potential solutions and apply some recombination and selection operators on them. During the last decades, evolutionary methods have received a considerable attraction and have experienced a rapid development. Main paradigms are: genetic algorithm (binary or real coded), augmented simulated annealing (binary or real coded), evolution strategy and differential evolution. Still, each of these methods has many possible improvements. In this theses the author is concentrating on subfamily of evolution methods named genetic algorithms. It is one of the most popular representative of evolutionary methods and serves as a great example, because it allows to observe and study the properties of the whole family.

2.1.1. How evolutionary algorithms are used nowadays?

Typically, optimization methods arising in engineering design problems are computationally demanding because they require evaluation of a quite complicated objective function many times for different potential solutions. Moreover, the objective function is often multi-modal, non-smooth or even discontinuous, which means that traditional, gradient-based optimization algorithms fail and global optimization techniques, which generally need even a larger number of function calls, must be employed. Fortunately, the rapid development of computational technologies and hardware components allows us to treat these problems within a reasonable time.

Genetic algorithms have been applied in science, engineering, business and social sciences. Number of scientists has already solved many engineering problems using genetic algorithms. GA concepts can be applied to the engineering problem such as optimization of gas pipeline systems. Another important current area is structure optimization. The main objective in this problem is to minimize the weight of the structure subjected to maximum and minimum stress constrains on each member. GA is also used in medical imaging system. The GA is used to perform image registration as a part of larger digital subtraction angiographies. It can be found that GAs can be used over a wide range of applications [6]. Genetic algorithms can also be applied to production planning, air traffic problems, automobile, signal processing, communication networks, environmental engineering and so on. There is a lot of examples discussed in [1] from music, art in general, architecture and engineering design.

2.1.2. Disadvantages

As every other technique, genetic algorithm has its flaws. In this theses the author tries to attract attantion to the two most important disadvantage of genetic algorithms, due to subjective opinion. For each of them is proposed an alternative solution, which will later take place in functional requirements of the thesis work.

1. **Implementation complexity**
   There are many problems of different types, which may me optimized by genetic algorithms. The downside is that for each of those problems there is a need of specific implementation, which is at least time consuming. Even though the low level details, which depend on the problem GA is applied to, will change, the high level abstraction of genetic algorithm workflow remains the same: a population of individuals are processed in pipeline of genetic operators, resulting in a new fitter population.
   A solution for this problem would be a high level generic representation of genetic algorithm, which may be reusable among variety of application and independent from the low level details.

2. **Computation cost**
   Genetic algorithms tend to be very time consuming. This is strictly related to its computation model, as the function, used in GA, are usually very demanding.

This characteristic of genetic algorithms, however, may be negated by the usage of parallelization.

2.2. SOLUTION PROPOSAL

A solution for described flaws would be a tool, which allows to integrate a high-performance implementation of the genetic algorithm into user application and use it for the arbitrary optimization problem with minimum changes required to the existing codebase. Additionally, this should be a convenient toolbox for theoretical and practical problems which might be resolved using genetic algorithms, providing a ready-made implementation of the most popular algorithms with extra features, which save users from implementing one themselves. For a solution to be considered as fulfilled, it is neccessary to meet both functional and non-functional requirements predifined in advance.

2.2.1. Non-functional requirements

1. **Java Virtual Machine**
   This platform was chosen, because of its popularity and open nature. Java virtual machine is supported by many languages, what allows a user to pick a specific one, which is the best for a given circumstances. JVM is also vastly used and run on over 3 billion devices[1].

2. **Open implementation**
   The implementation of the tool has to be open to its users, allowing to view and expand source code to the specific needs.

3. **Modular architecture**
   The architecture solution used to build the tool has contain modularity among its fundamentals. This means, that allowing different features, its components have to be grouped according to their functions, so the users, who are interested in a small part of provided functionality wouldn't be forced to use all capabilities of the tool.

2.2.2. Functional requirements

1. **Limiting the evolution in terms of time and/or number of iterations** This is basic requirement for every genetic algorithm implementation. Evolution process takes time and in order to benefit from it, user must have access to the evolved version of population. Number of iterations corresponds to the number of evolution cycles performed on the population, when every genetic operator is applyed to it. On the other hand, the actual time of the evolution may be changed even in terms of the same number of iterations, as it strongly depends on performance of genetic operators and fitness function, so in

[1] According to official information provided by Oracle Corporation

some cases time may be the only valid criterion, which decides that evolution process should be stopped.

2. **Controll of the best individual through evolution**
   During the evolution process, due to the chosen selection, crossover or mutation strategies, even the fittest individual may not end in the next population. As for business needs it is important to get the best solution ever discovered by algorithm, it is important to store reference to that candidate separately.

3. **Fitness values caching** In genetic algorithms, the computation of the fitenss function provides the largest computational load for the algorithm. Each population genration is composed of individuals who are formed from previous gneeration via cloning, crossover, or mutation. It is also common for the individual to migrate to the next population without a change or, less common, a crossover of two different individuals may produce a genotype that already has been discovered earlier. If this occurres rather frequently, it is unprofitable to compute the fitness value all over again. As a solution one may consider caching the results of fitness computation for later reusage. Although it noticeably increases the speed of algorithm, the cache tends to grow dramatically, so the coise of data structure for this purposes is extrimely important.

4. **Parallel evolution processing**
   Genetic algorithms are generally known as a time-consuming technique, as it is often used with complex, high-dimensional problems. Evolution cycle may take up to hours and days of continuous computing, depending on the parameters of the algorithm and hardware in use, which comes with a great cost and very low flexibility, as with every mistake or change was done to algorithm implementation, evolution needs to be re-executed. On the other hand, even small upgrades in speed may result as a major cost cut.
   For this reason, it is highly important to use the maximum of the given resources, avoiding unreasonable workloads. It may be achieved through the computation parallelization between multiple virtual CPU cores within a single machine

5. **Asynchronous evaluation of fitness value**
   The solution must allow to compute fitness values on different machine or even cluster of machines, as well as using different low-level programming language or some specific optimization, with no need to switch the entire application to another platform. This type of environment also fits perfectly to cloud computing solutions, as fitness evaluation is entirely isolated from the rest of application.

6. **Generic representation of individual**
   There is a wide set of problems which might be resolved using genetic algorithms. Still, this optimization technique is rarely used in business applications and mainly

considered as a theoretical solution. One of the reasons for such phenomenon is the difficulty which comes with a try to use genetic algorithm optimization without a previous plan to do so. The process of reshaping the existing codebase in such way, so typical genetic algorithm could be applying often demands big costs and high level of understanding GA workflow and its possibilities.
Depending on the problem a user is trying to solve, a standard representation of each candidate solution would be an array of bits, integers or real numbers. The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size, which facilitates simple crossover operations. For the purposes of genetic programming and evolutionary programming tree-like and graph-form representation are used respectively, which causes the necessity in different implementations for each problem.
A solution to this problem is an implementation of the genetic algorithm, which is able to work on generic types, which represents a candidate solution for the applied problem. Due to this property, it is possible to reuse the implementation for different problems and create a generic tool, which is decoupled from the problem it is applied to.

7. **Evolution in distributed environment**
   As already mentioned, parallelization is a big part of speed optimization for genetic algorithms. While local parallelization is usefull and convinient for medium- and small-sized problems, working with complex use cases often demands the resources, which is may not fit into one machine. Distributed environment is a next step of parallelization, when evolution process is split between different machines, each computing its own part of the workload. This approach enables horizontal scaling, when each added machine increases performance of the algorithm.

8. **On-demand evolution**
   Genetic algorithm optimization is generally used for finding the best solution for a defined problem, which is the fittest candidate to the provided fitness function. It means, that the best solution is defined relative to all the solutions assigned earlier and cannot be proved as the optimal one, even when it is. But when applying to a nontrivial problem, finding the optimal solution is not always cost-effective, as it may take up to exponential time. For this reason, it is common to compute a 'good enough' solution in reasonable time. Depending on the business reasons, this time varies so sometimes it is possible to wait until the better solution is computed and sometimes it is not. For instance, if the application which uses genetic algorithm has a real-time interface and its user is interested in gaining a solution for the particular case in the range of seconds, it is justifiable to provide a solution which may not be optimal but is close to it.
   On the other hand, when a user is ready to allow the application to take its time in order to provide a better result, it has to still be possible to resume the evolution. Again the

application should not limit users in such important cases, as the end user needs may vary depending on different circumstances, which may not be predicted in advance. The evolution of a solution to a given problem should be available on-demand, without forcing unreasonable resource uses, and adjust to a specific case as much as possible.

3. EXISTING SOLUTIONS OVERVIEW

3.1. JAVA JENETICS

3.2. APACHE.MATH.GENETICS

4. WHAT IS MY SOLUTION TO A GIVEN PROBLEM?

4.1. IMPLEMENTATION OVERVIEW

The solution is presented as an open-source library, that provides the major part of genetic algorithm implementation, which can be shared between different application. Base algorithm consist of a pipeline separated by three stages: selection, crossover and mutation, which evolves the population of individuals, increasing average level of adjustment and finding the best individual according to provided fitness function; preimplemented most popular strategies of genetic selection, crossover and mutation, with possibility of extanding with the additional ones for specific reasons and a set of platforms which allow to run the pipeline in a different ways, best suited for existing hardware. The library is supplied with key features, basing on the concepts described earlier, wrapping all the components into sufficient framework.

4.1.1. Technology stack

Scala programming language

Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. Its name stands for "scalable language." The language is so named because it was designed to grow with the demands of its users. It can be applied to a wide range of programming tasks, from writing small scripts to building large systems. Scala is easy to get into. It runs on the standard Java platform and interoperates seamlessly with all Java libraries. It's quite a good language for writing scripts that pull together Java components. But it can apply its strengths even more when used for building large systems and frameworks of reusable components. [3]

Technically, Scala is a blend of object-oriented and functional programming concepts in a statically typed language. The fusion of object-oriented and functional programming shows up in many different aspects of Scala; it is probably more pervasive than in any other widely used language. The two programming styles have complementary strengths when it comes to scalability. Scala's functional programming constructs make it easy to build interesting things quickly from simple parts. Its object-oriented constructs make it easy to structure larger systems and adapt them to new demands. The combination of both styles in ScaalMatei Zaharia, “Spark: Cluster Computing with Working Sets makes it possible to express new kinds of programming patterns and component abstractions. It also leads to a legible and concise programming style [3]

Due to its programming style and variety of features Scala serves a great tool for developing this library. A combination of paradigms is extremely usefull in this case and was crutial during selection of programming language. Both of the programming styles were heavily used in order to gain the best possible result:

- Object-oriented style
  Scala is an object-oriented language in pure form: every value is an object and every operation is a method call. [3] It provides modularity and maintainability: all application is split into a self-sufficient containers, which store the relevant data and provide available operations, that may be performed on its data. These containers may be used as the assembling parts for another container, passed as function parameter and returned as function result. Such approach allows to keep track of the growing codebase, alligning its components into domain hierarchies and letting its parts to comunicate with one another.
- Functional style
  Functional programming provides expressiveness, composability and conciseness. Building the library with mathematical-like functions makes it more understandable, replacing the mutable state from the application. As the result, it becomes clear, unequivocal and may be safely used in multi-thread environment. One of the most important assets, which comes from functional programming, is scalability. A well constructed application may easily be enlarged for user needs to handle a growing amount of work; it is more adaptive to the changing needs and the performance of such system improves proportionally to the additional hardware.

Parallel collections

Encouraging usage of immutable objects, from version 2.8 Scala provides new collection API, which contains parallel collections package as part of it. Parallel collections were included in the Scala standard library in an effort to facilitate parallel programming by sparing users from low-level parallelization details, meanwhile providing them with a familiar and simple high-level abstraction. The idea is simple– collections are a well-understood and frequently-used programming abstraction. And given their regularity, they're able to be efficiently parallelized, transparently. By allowing a user to "swap out" sequential collections for ones that are operated on in parallel, Scala's parallel collections take a large step forward in enabling parallelism to be easily brought into more code. [5]

How parallel collections are built? What may be acquired with scala parallel collections?

Apache Spark

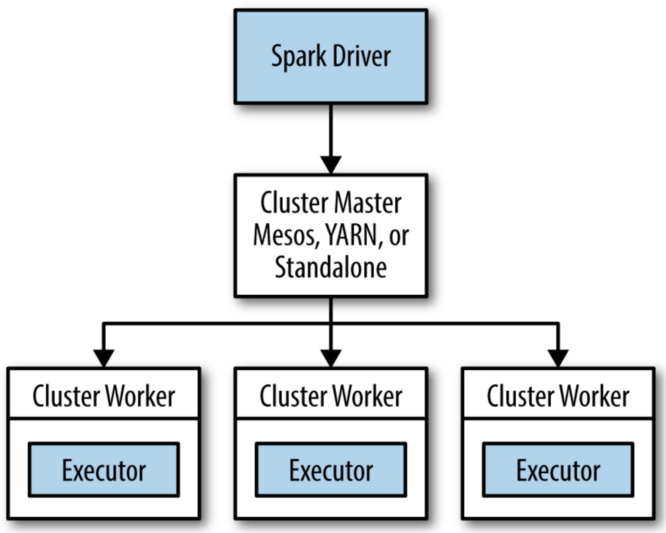
Apache Spark is a high-performance, general-purpose distributed computing system that has become the most active Apache open source project, with more than 1,000 active contributors. Spark enables users to process large quantities of data, beyond what can fit on a single machine, with a high-level, relatively easy-to-use API. Spark's design and interface are unique, and it is one of the fastest systems of its kind. Uniquely, Spark allows to write the logic of data transformations and machine learning algorithms in a way that is parallelizable, but relatively system agnostic. So it is often possible to write computations that are fast for distributed storage systems of varying kind and size.

On the generality side, Spark is designed to cover a wide range of workloads that previously required separate distributed systems, including batch applications, iterative algorithms, interactive queries, and streaming. By supporting these workloads in the same engine, Spark makes it easy and inexpensive to combine different processing types, which is often necessary in production data analysis pipelines.[2]

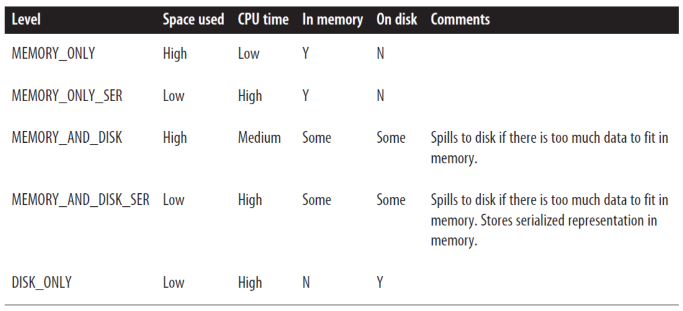
Spark can run over a variety of cluster managers to access the machines in a cluster. The easiest way is to run Spark by iteself on a set of machines. For this purpose Spark comes with built-in Standalone mode. Spark's Standalone manager offers a simple way to run applications on a cluster. It consists of a master and multiple workers, each with a configured amount of memory and CPU cores. When application is submited, one can choose how much memory its executors will use, as well as the total number of cores across all executors.

Spark Core contains the basic functionality of Spark, including components for task scheduling, memory management, fault recovery, interacting with storage systems, and more. Spark Core is also home to the API that defines resilient distributed data-sets (RDDs), which are Spark's main programming abstraction. RDDs represent a collection of items distributed across many compute nodes that can be manipulated in parallel. Spark Core provides many APIs for building and manipulating these collections. RDDs offer two types of operations: transformations and actions. Transformations construct a new RDD from a previous one. For example, one common transformation is filtering data that matches a predicate. Actions are operations, which produce different result than the transformation, like collecting the elements into a local collection on master node or counting the number of elements in RDD. Transformations and actions are different because of the way Spark computes RDDs. Although new RDDs can be defined in any time, Spark computes them only in a lazy fashion — that is, the first time they are used in an action. The transformations, on the other hand, are scheduled to a DAG (directed acyclic graph) of computations. This approach has a lot of advantages, among them: effective fault tollerance and ability to make a lot of optimization decisions before actually running operations. This would not be possible if it executed computations as soon as it got it.

In distributed mode, Spark uses a master/slave architecture with one central coordinator and many distributed workers. The central coordinator is called the driver. The driver communicates with a potentially large number of distributed workers called executors. The driver runs in its own Java process and each executor is a separate Java process. A



Pic. 4.1. The components of a distributed Spark application [2].



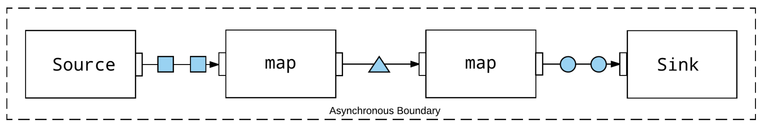
Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Pic. 4.2. Persistence levels from org.apache.spark.storage.StorageLevel and pyspark.StorageLevel [2].

driver and its executors are together termed a Spark application. Then a Spark application is launched on a set of machines using early described cluster manager.

This architectures allows to distribute the data between different executor nodes and perform operations on it locally until the shuffling is needed. Only then the data is shared between different nodes and the network communication is handled. In a distributed program, communication is very expensive, so laying out data to minimize network traffic can greatly improve performance. Much like how a single-node program needs to choose the right data structure for a collection of records, Spark programs can choose to control their RDDs' partitioning to reduce communication.

In order to be able to transforms large sets of data without redundant recomputation Spark provides data persistence. When the Spark is asked to persist an RDD, the nodes that compute the RDD store their partitions. Spark has many levels of persistence to choose from based on what the goals are. The list of storage levels with performance comparison may is shown on 4.2. Combining different persistence modes, an executor can effectively share the data between its disk and RAM.



Pic. 4.3. A typical stream flow

Akka Streams

Akka Streams is a library to process and transfer a sequence of elements using bounded buffer space. This latter property is what is ususally referred as boundedness and it is the defining feature of Akka Streams. Translated to everyday terms it is possible to express a chain of processing entities, each executing independently (and possibly concurrently) from the others while only buffering a limited number of elements at any given time.

ToDo: more info

4.1.2. Core components

ToDo: add introduction

- Individual
  An individual is a core component of genetic algorithm, often also reffered to as a *genotype, chromosome* or *a candidate*, which represents a candidate solution to a given problem. The number of individuals are evolved during the algorithm workflow, mixed and mutated in order to find the fittest one according to provided fitness function. This concept is purely abstract in the library, encoded as a generic type which may have an arbitrary representation defined by the users depending on their needs.
  In order to handle an individual through the process of evolution there should be defined a set of operators, where each encapsulates logic of some part of evolution procedure.

  · Fitness
    Fitness is a type class which represents a function of type `I => Double`, where `I` is type of individual. This function summarises how close a given design solution is to achieving the set aims. A ftness function must be devised for each problem to be solved, thus should be provided by the user and may not be preimplemented. Fitness function value is main criterion of future selection and is very important attribute of individual representation.
    ToDo - how much does it cost

· Join
  Join is a type class, which represents a cross function of type `(I, I) => (I, I)`, where `I` is type of individual. This function defines how two instances of type `I` may be combined together to produce a new pair of `I`. It describes a crossover operation, due to which all the individuals from a population combined into pairs will be mixed during evolution process. An instance of the type class may be created from a single function of type `(I, I) => (I, I)`, as well as a function of type `(I, I) => I`. The latter case is suitable when identical function is used to compute both of result elements or in case of commutative operation. These cases were separated in order to enable memory allocation optimization by reusing the same product object to construct result pair.

· Modification
  Modification is a type class, which represents a mutation function of type `I => I`, where `I` is type of individual. This operator is used to maintain generic diversity from one generation of a population to another. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state, during which the solution may change entirely from the former shape. Because of its loose representation (even identity function may be considered as modification) it is important to provide a set of laws, which will distinguish a modification, which actually brings a diversity into the population. A *lawfull* modification function is one, which holds following properties:

  1. A modified instance does not equal to original one (*e.i modification function does not equal to identity function*) 4.1

```
modify(i) != i
```

Listing 4.1: The first laws of Modification instance

  2. After a certain number of modification the same input produces different outputs (*e.i. modification function is randomized or depends on outer variables*) 4.2

```
def modify5(i: I) = modify(modify(modify(modify(modify(i)))))
modify5(i) != modify5(i)
```

Listing 4.2: The second laws of Modification instance

· Scheme
  Scheme is a type class, which is responsible for a creation of new individual. This is additional function, which may be replaced by predefined initial population

  during the start of evolution process, but still offers more functionality and may be used when individuals are created randomly, from limitless iteratator, smaller collection, which should be cycled or even single instance of individual. This concept is implemented as a covariant class, so the scheme object of parent class may be replaced by the scheme object of child class. It allows to easily use this class with hierarchies, avoiding redundant code.

  Together this satisfies predefined functional requirement 6

- Population

  Population is a collecion of individuals, which holds them during single evolution step. Individuals are processed together and are viewed only as a part of population from the perspective of algorithm. During the evolution step the representation of population or holding type may be changed (e.g. after selection stage further processed population contains pair of individuals, which are meant to be mixed later).

  From implementation perspective, population is a type alias to Scala vector collection. The choise of data structure was motivated by fast average performance and effectively constant random access time in particular.

- Evolution environment

  Evolution environment is a component, which taking the set of options constructs endless evolution flow of given population and makes it ready to be run. It is a parent type to all implementations of different approaches in processing computations, which may be sequential or parallel, local or distributed. This concept is very important from the usage perspective as it devides the implementation of genetic algorithm from its usage, so the end user may even not be aware of the way how the evolution is performed. It is also the end point of all the dependent components, as there is no direct dependency to any of concrete implementation of evolution environment. This abstraction may be considered as high level representation of genetic algorithm itself. The list of concrete derivatives of this interace contains:

  - Local environment

    This type of environment presumes local evoluation of evolution flow. It is the richest environments in terms of possible configurations. The logic, behind the strategy of applying genetic operators and evaluating fitness values is decoupled from one another into instances of `EvolutionCompanion` and `FitnessEvaluator` classes respectively. For each of the strategies the are sequential and parallel versions, with additional fitness caching option for repetitive individuals within population. Any fitness evaluation strategy may combined with any possible technique of applying genetic operators, creating a number of possibilites ready for use. *This satisfies predefined functional requirement 4*

  - Asynchronous environment

    Asynchronous environment delegates the function of fitness evaluation to another system, by sending non-blocking asynchronous calls for every element of the population. It may be extremely usefull for the problems with heavy fitness functions. This approach allows to compute fitness values on different machine or even cluster of machines, as well as using different low-level programming language or some specific optimization, with no need to switch the entire application to another platform. This type of environment also fits perfectly to cloud computing solutions, as fitness evaluation is entirely isolated from the rest of application. However, applying genetic operators in same way needs to send a lot more data over the network, which negates the benefits one can gain from this approach. For this reason, genetic operators are applied locally with any of the strategies described earlier.

    This satisfies predefined functional requirement 5

  - Distributed environment

    Distributed environment is very different from the previous options. Here the evolution is performed on the population in terms Spark RDD, distributed over the cluster. This technique enables processing of very large population, since it is split between many executor nodes. Once the evolution step is done executors shuffle the data between each other in order to perform selection of individuals for the next population and continue working on the local values. It minimizes the network communication, which may be a bottle-neck in such applications.

    This satisfies predefined functional requirement 7

- Evolution flow

  Evolution flow represents the endless evolution process of given population. It is implemented as the reactive stream with one output, which may be plugged in into further part of pipeline or executed standalone. During running, evolution flow emits the instance of population once it is computed and starts working on the next one. Since it is a reactive stream, the process of evolution is continued during there is a demand from the downstream. For example, if the sink of pipeline needs to take only 50 first populations, no more than 50 will be computed. This technique allows to leave the variety of possible stop conditions to the end user, including number of elements pushed from the flow, total time, idle time since the last push or any predicate based on the last produced population(s).

  This satisfies predefined functional requirements 1, 8, 2

- Fitness caching

  Fitness caching is implemented using decorator design pattern over the provided fitness function. It supports every fitness function provided by user, as it simply holds the record over the ranked individuals and returns stored result if it was encountered earlier. The data structure used for holding cached values is concurrent hash trie. It guarantees consistency in multithreaded environment, enabling safe usage with parallel workflows. From the performance side, concurrent hash tries allow to lookup element in logarithmic time, with costant snapshot time [4] (operation which is performed when two or more threads try to modify the same element).

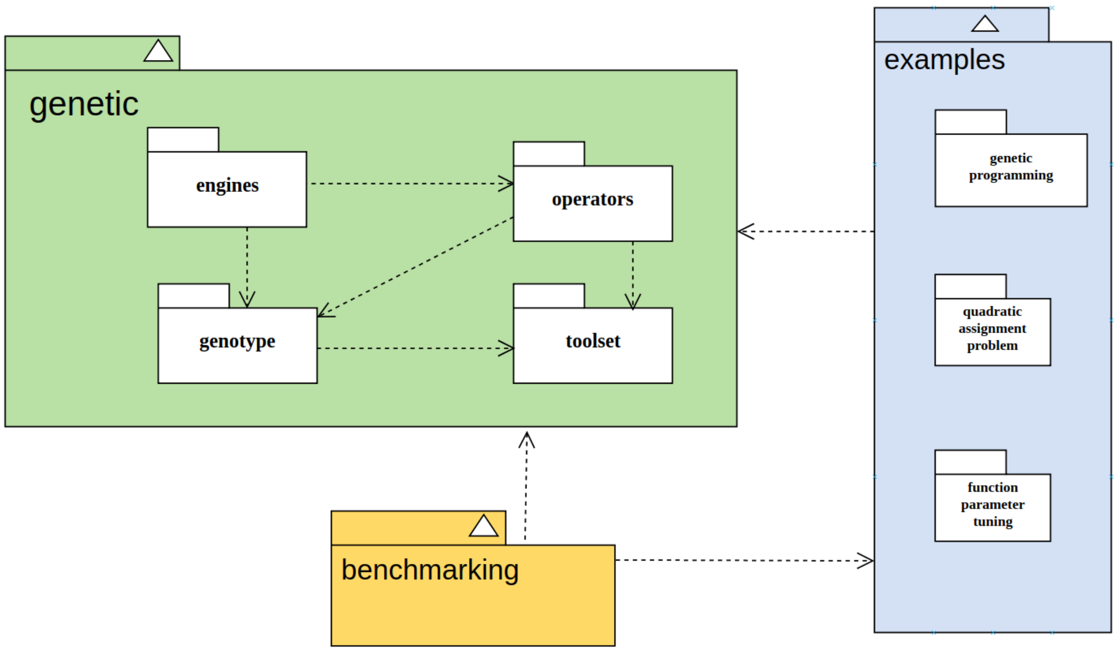
This satisfies predefined functional requirement 3

4.1.3. Project structure

The whole project is devided into three main packages:

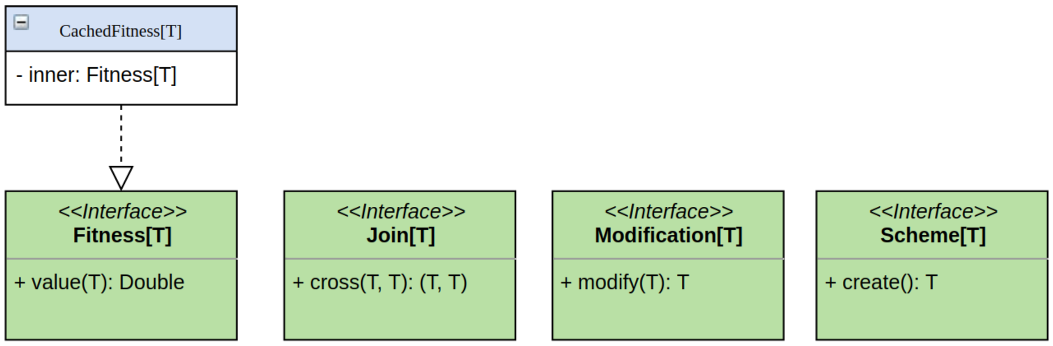
- *genetic*
- *examples*
- *benchmarking*

examples and *benchmarking* are additional packages, which demonstate how the library may be used and what results one may achieve by using it, while the main value have the classes from package *genetic*. The source code dependencies between the packages may be found on package diagram 4.4



Pic. 4.4. Packages and their source dependencies

genotype package contains classes, which correspond to the information used to describe behavior of an individual. This is achieved using 4 type classes: *Fitness, Join, Modification* and *Scheme*. This subpackage also contains *CachedFitness* class, which is a wrapper around provided fitness function, with additional caching behavior 4.5.



Pic. 4.5. Type classes used to describe an individual

.

engines package contains the heavy part of genetic algorithm implementation. Here one may find EvolutionEnvironment class and its concrete implementations. Every child class (parallel implementation) is placed in its specific subpage, so users have to import only the part, which they are interested in or everything, if this is desired. This approach may be seen in project structure 4.6, where the interfaces, which are communicating between each other are placed at the top of the package and different variants of their implementation are placed into specific subpackage, which later is important for user needs.

This satisfies predefined non-functional requirement 3

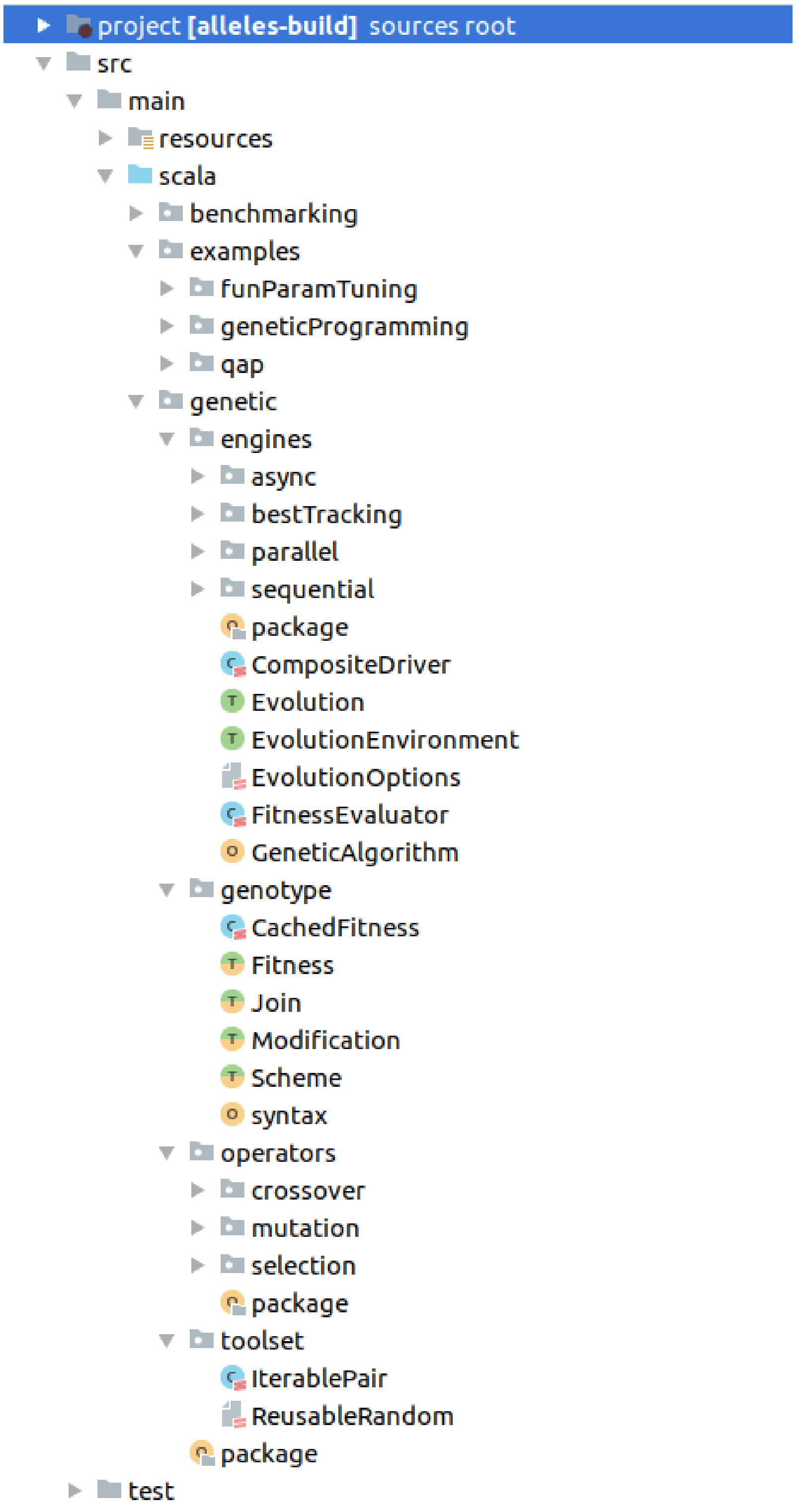
The way in which parts of *engines* package are related to each other may be observed in *genetic* class diagram4.7

4.2. USAGE

The library artifact may be added into the project as dependency using maven tool or sbt. After this, if library requirements are met, imported classes are ready to be used.

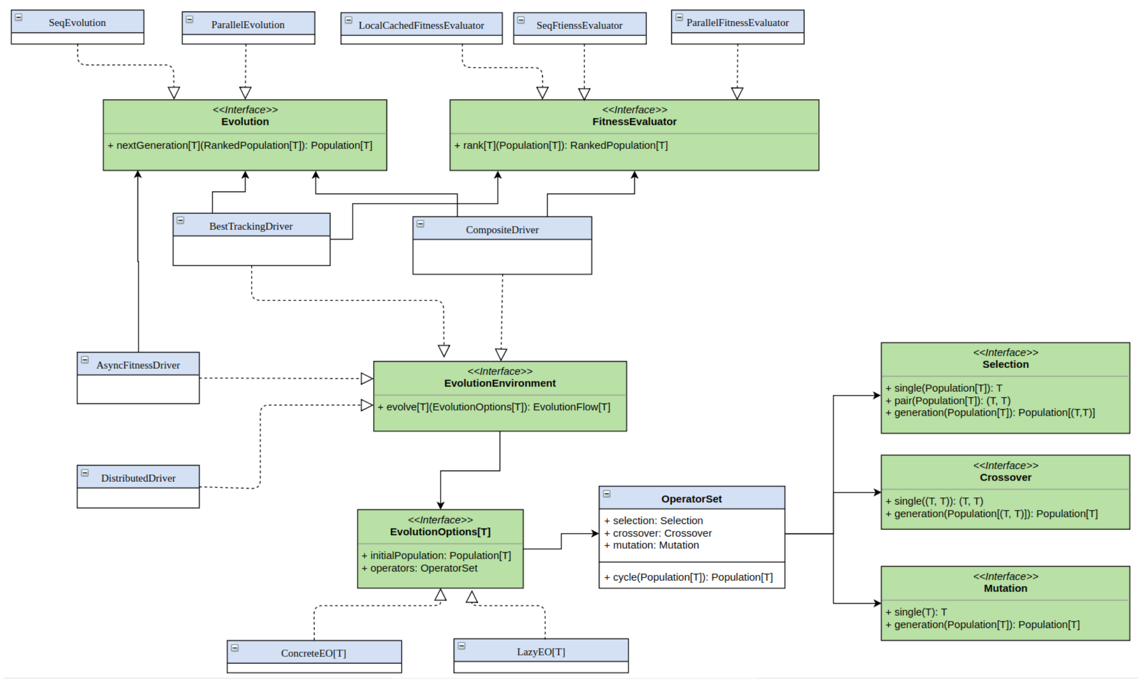
4.2.1. Requirements

- Scala 2.12.6
- sbt 1.2.1
- Apache Spark 2.3



```
▶ project [alleles-build] sources root
▼ src
    ▼ main
        ▶ resources
        ▼ scala
            ▶ benchmarking
            ▼ examples
                ▶ funParamTuning
                ▶ geneticProgramming
                ▶ qap
            ▼ genetic
                ▼ engines
                    ▶ async
                    ▶ bestTracking
                    ▶ parallel
                    ▶ sequential
                      package
                      CompositeDriver
                      Evolution
                      EvolutionEnvironment
                      EvolutionOptions
                      FitnessEvaluator
                      GeneticAlgorithm
                ▼ genotype
                      CachedFitness
                      Fitness
                      Join
                      Modification
                      Scheme
                      syntax
                ▼ operators
                    ▶ crossover
                    ▶ mutation
                    ▶ selection
                      package
                ▼ toolset
                      IterablePair
                      ReusableRandom
                  package
    ▶ test
```

Pic. 4.6. Expanded project structure



Pic. 4.7. Classes diagram from *genetic* package

4.2.2. API

Here is described the application programming interface of the library. Due to its specific type, the frameworks do not usually have use case diagram defined, but instead present its value through API. It is brief description of the classes and functions available to a user, in which they communicate and actually use a programming library.[1]

- `trait EvolutionEnvironment`
  - `def evolve[T: Fitness : Join : Modification](options: EvolutionOptions[T]): EvolutionFlow[Population[T]]`

    Creates an evolution flow out of evolotuion options of type T, with available implicit objects of class Fitness[T], Join[T], Modification[T]
- `type EvolutionFlow[T] = akka.util.Source[T, NotUsed]`

  A type alias to akka-streams Source. Contains all the methods of akka.util.Source
- `trait EvolutionOptions[T]`

  A trait which holds the parameter for constructing evolution flow
  - `def initialPopulation: Population[T]`
  - `def operators: OperatorSet[T]`
- `trait Evolution`

  Corresponds to the one step long evolution. Derives rated population to the new one using `class OperatorSet`

[1] Description of the rest of the classes was ommited due to its big size

- def nextGeneration[T: Join : Modification](ratedPop: Population[Rated[T]], operators: OperatorSet): Population[T]

- class FitnessEvaluator

  A class which basing on the instance of Functor[Population[T]] computes the fitness values for the population.

  - def rate(population: Population[T]): Population[Rated[T]]

- trait Fitness[T]
  - def value(t: T): Double

    Computes the fitness value for an individual of type T
  - def cached: CachedFitness[G]

    Returns a cached version of this fitness function

- trait Join[T]
  - def cross(a: T, b: T): IterablePair[T]

    Mixes a pair of individuals into new pair with shared genome

- trait Modification[T]
  - def modify(t: T): T

    Returns modified version of parameter t

- trait Scheme[+T]
  - def create: T

    Creates a new instance of type T

- trait Selection
  - def single[T](population: Population[Rated[T]]): (T, T)

    Selects a single pair from the Population of rated individuals
  - def generation[T](population: Population[Rated[T]]): Population[(T, T)]

    Selects pairs of individuals into new population of the same size

- trait Crossover
  - def single[T: Join](parents: (T, T)): IterablePair[T]

    Mixes one pair of individuals with provided Join[T] instance
  - def generation[G: Join](population: Population[(G, G)]): Population[G]

    Mixes every pair of given population into the new population with provided Join[T] instance

- trait Mutation
  - def single[T: Modification](individual: T): T

    Mutates a single individual with the Modification[T] instance
  - def generation[T: Modification](population: Population[T]): Population[T]

    Mutates all population with the Modification[T] instance

4.3. MEETING THE PREDEFINED REQUIREMENTS

With current implementation all predefined functional and non-functional requirements were successfully met.

5. SURVEY

BIBLIOGRAFIA

[1] Bentley, P., Corne, D., Engineering Information, I., Boden, M., *Creative Evolutionary Systems*, Evolutionary Computation Series (Elsevier Science, 2002).

[2] Karau, H., Konwinski, A., Wendell, P., Zaharia, M., *Learning Spark: Lightning-Fast Big Data Analytics*, 1st wyd. (O'Reilly Media, Inc., 2015).

[3] Odersky, M., Spoon, L., Venners, B., *Programming in Scala: Updated for Scala 2.12*, 3rd wyd. (Artima Incorporation, USA, 2016).

[4] Prokopec, A., Bronson, N.G., Bagwell, P., Odersky, M., *Concurrent tries with efficient non-blocking snapshots*, SIGPLAN Not. 2012, tom 47, 8, str. 151–160.

[5] Prokopec, A., Miller, H., *Parallel collections overview*, `https://docs.scala-lang.org/overviews/parallel-collections/overview.html`.

[6] Sivanandam, S.N., Deepa, S.N., *Introduction to Genetic Algorithms*, 1st wyd. (Springer Publishing Company, Incorporated, 2007).