sgap - Scala Genetic Algorithm Package

Marco Zanella

University of Padova

marco.zanella.9@studenti.unipd.it

May 8, 2015

Table of Contents

1. Introduction
   - Scala
   - Scenarios
2. Core concepts
   - Why Scala?
   - Test cases
3. Code implementation
   - Strategies
   - Result analysis
4. Similar works
   - JGAP
   - Scala for Machine Learning

Introduction

The goal of this work is to stress Scala's features on *real* scenarios:

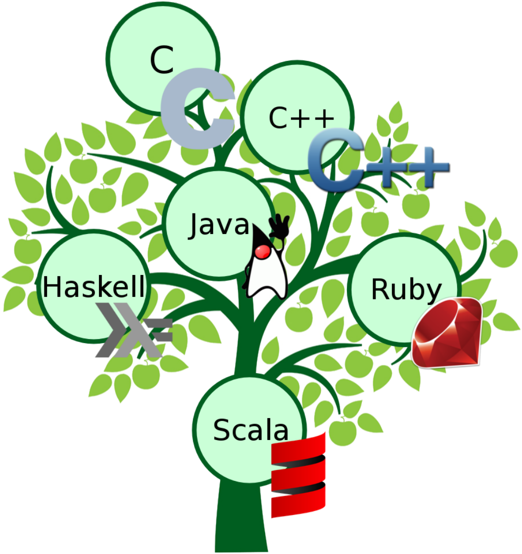
Genetic Algorithms to exploit Object-Oriented features

Greedy Algorithms to exploit functional features

And with *concrete* use cases:

- The Traveling Salesman Problem
- The Knapsack 1-0 problem

Introduction - Scala



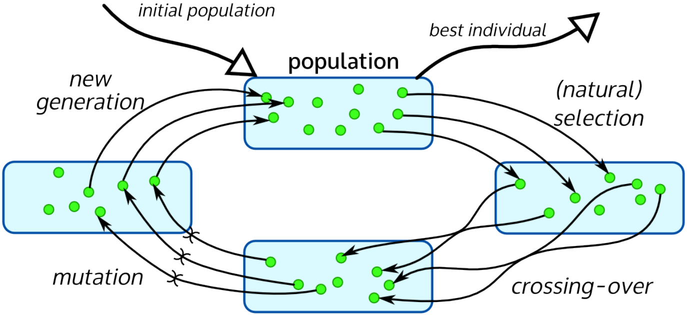
Modern, multi-paradigm programming language:

- Object-Oriented
- Functional
- Statically Typed
- Extensible

Famous for its actor-based concurrency model, it offers more interesting features.

Introduction - Genetic Algorithms

Search heuristic inspired by natural selection (Charles Darwin).



GAs use *no or little knowledge* about the properties of the problem.

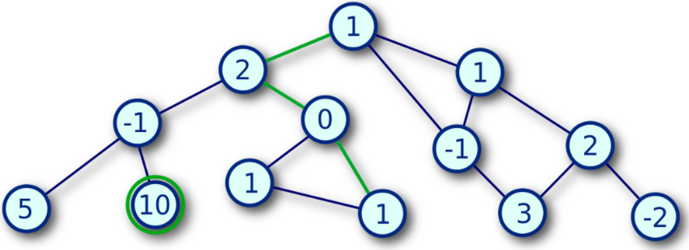
Marek Obitko

Introduction to Genetic Algorithms

Introduction - Greedy Algorithms

Simple heuristics which make a locally optimal choiche at each stage.

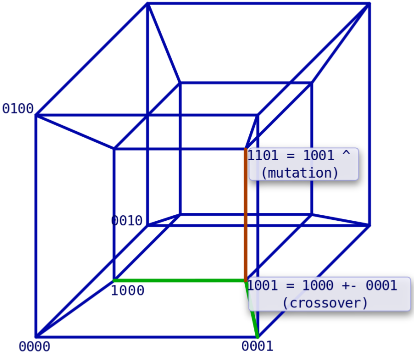
- optimal solution/termination is not guaranteed (except for matroids)
- very fast
- usually good solutions



Greedy algorithms require little knowledge about the problem.

Core concepts

A lot of (hard) mathematics behind Genetic Algorithms:



- search space as hypercubes
- axioms of closure under crossover and mutation
- convergence
- ...

Also negative results such as the *No Free Lunch Theorem* must be understood.

Thomas Weise
Global Optimization Algorithms

Core concepts - Why scala?

Many interesting benefits:

- Type safety
- Type inference (much less verbose than Java)
- Object Oriented... (unlike Haskell)
- ...and functional (unlike Java and C++)
- Traits (better than PHP's or Ruby's)
- (To try out something new...)

No real disadvantages so far, but:

- Slow compile phase (solved by incremenetal compilation)
- Some features not (yet) available

Core concepts - Test cases

Genetic Algorithms are commonly applied to a variety of fields. Programs based on combinatorial problems are common killer applications.

Traveling Salesman

-vs-

Knapsack

Both the *Traveling Salesman Problem (TSP)* and the *Knapsack 1-0 Problem* are well-known *NP-hard* problems.

Core concepts - Test cases

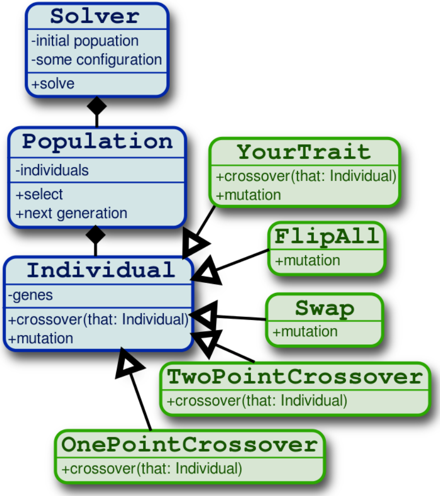
Knapsack 1-0 Which are the best items to put into a limited size knapsack? (no more than 1 of each item!)

TSP Given a set of cities, which is the shortest path which goes though them all?

Differences between Knapsack and TSP make them suitable test cases:

Knapsack 1-0	TSP
size of solution not know	size of solution know
maximize (Σ of values)	minimize (Σ of distances)
constraint (Σ of weights)	unconstrained
order of items does not matter	order of items matters

Code implementation



Functional and OO can live together:

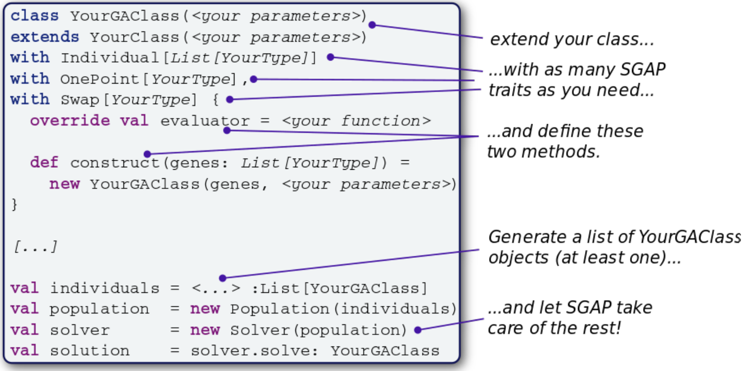
GA	Object Oriented
Greedy	functional

Greedy solutions as initial individuals for GAs.

📄 Patrick R. Nicolas
Scala for Machine Learning

Code implementation - Strategies

I have a Kapsack/TSP class, want to deploy GAs, and have
only one slide to show my teacher. How do I do?



```
class YourGAClass(<your parameters>)
extends YourClass(<your parameters>)
with Individual[List[YourType]]
with OnePoint[YourType],
with Swap[YourType] {
  override val evaluator = <your function>

  def construct(genes: List[YourType]) =
    new YourGAClass(genes, <your parameters>)
}

[...]

val individuals = <...> :List[YourGAClass]
val population  = new Population(individuals)
val solver      = new Solver(population)
val solution    = solver.solve: YourGAClass
```

extend your class...

...with as many SGAP traits as you need...

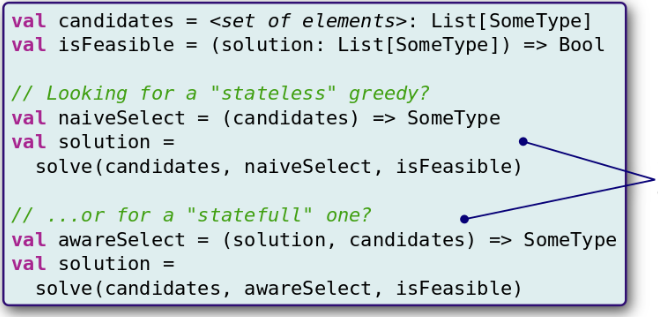
...and define these two methods.

Generate a list of YourGAClass objects (at least one)...

...and let SGAP take care of the rest!

Code implementation - Strategies

Same problem as before, but I
need a Greedy/GRASP this time...

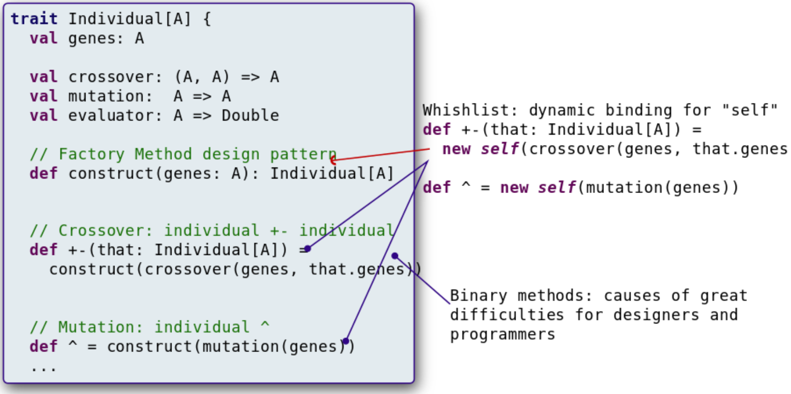


```
val candidates = <set of elements>: List[SomeType]
val isFeasible = (solution: List[SomeType]) => Bool

// Looking for a "stateless" greedy?
val naiveSelect = (candidates) => SomeType
val solution =
  solve(candidates, naiveSelect, isFeasible)

// ...or for a "statefull" one?
val awareSelect = (solution, candidates) => SomeType
val solution =
  solve(candidates, awareSelect, isFeasible)
```

will work with
either one...

Code implementation - Flaws

There are still *reasonable and desirable* features not available:



```
trait Individual[A] {
  val genes: A

  val crossover: (A, A) => A
  val mutation:  A => A
  val evaluator: A => Double

  // Factory Method design pattern
  def construct(genes: A): Individual[A]


  // Crossover: individual +- individual
  def +-(that: Individual[A]) =
    construct(crossover(genes, that.genes))


  // Mutation: individual ^
  def ^ = construct(mutation(genes))
  ...
```

```
Whishlist: dynamic binding for "self"
def +-(that: Individual[A]) =
  new self(crossover(genes, that.genes

def ^ = new self(mutation(genes))
```

```
Binary methods: causes of great
difficulties for designers and
programmers
```

Luca Cardelli et al.

On Binary Methods

Similar works

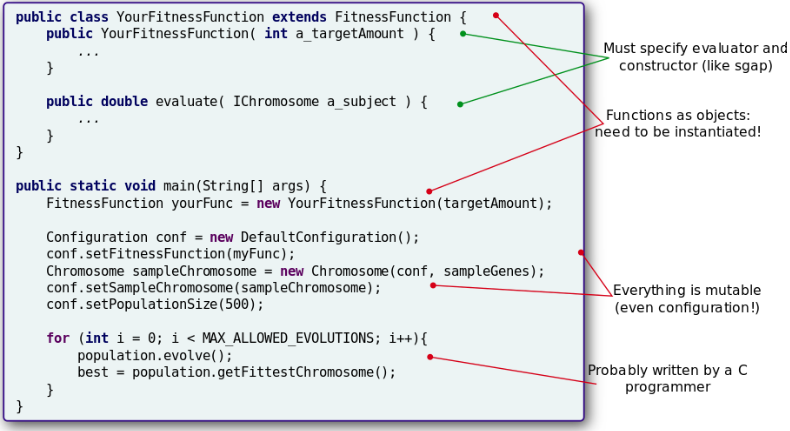
There are no notable Scala works with the same goal as SGAP. Similar works include:

Java Genetic Algorithms Package same goal, similar language

Scala for Machine Learning similar goal, same language

SGAP (tries to) take the best out of the two.

Similar works - JGAP



```
public class YourFitnessFunction extends FitnessFunction {
    public YourFitnessFunction( int a_targetAmount ) {
        ...
    }

    public double evaluate( IChromosome a_subject ) {
        ...
    }
}

public static void main(String[] args) {
    FitnessFunction yourFunc = new YourFitnessFunction(targetAmount);

    Configuration conf = new DefaultConfiguration();
    conf.setFitnessFunction(myFunc);
    Chromosome sampleChromosome = new Chromosome(conf, sampleGenes);
    conf.setSampleChromosome(sampleChromosome);
    conf.setPopulationSize(500);

    for (int i = 0; i < MAX_ALLOWED_EVOLUTIONS; i++){
        population.evolve();
        best = population.getFittestChromosome();
    }
}
```

Must specify evaluator and constructor (like sgap)

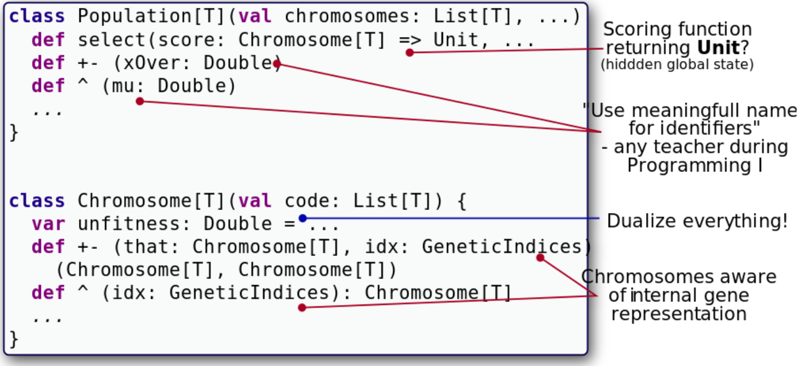
Functions as objects: need to be instantiated!

Everything is mutable (even configuration!)

Probably written by a C programmer

JGAP documentation
http://jgap.sourceforge.net/

Similar works - Scala for Machine Learning



```
class Population[T](val chromosomes: List[T], ...)
  def select(score: Chromosome[T] => Unit, ...
  def +- (xOver: Double)
  def ^ (mu: Double)
  ...
}


class Chromosome[T](val code: List[T]) {
  var unfitness: Double = ...
  def +- (that: Chromosome[T], idx: GeneticIndices)
    (Chromosome[T], Chromosome[T])
  def ^ (idx: GeneticIndices): Chromosome[T]
  ...
}
```

Scoring function
returning **Unit**?
(hiddden global state)

"Use meaningfull name
for identifiers"
- any teacher during
Programming I

Dualize everything!

Chromosomes aware
of internal gene
representation

Patrick R. Nicolas
Scala for Machine Learning

The End – Thank You

Conclusion

The goal of stressing Scala's features to develop a genetic algorithms package was achieved:

- ✓ both greedy and genetic algorithms
- ✓ flexible, stable code
- ✓ programming strategies supported by theoretical and practical considerations
- ✗ Scala is not as common as other alternatives (Java, C++)
- ✗ still open questions (Scala vs OCaml, binary methods...)