The basics of genetic algorithms

Thomas Giraud Simon Chabot

November 25, 2013

Contents

1 Overview of the genetic algorithms **3**
1.1 Principle . . . 3
1.2 Illustration with a mathematical problem . . . 4

2 Examples **6**
2.1 The knapsack problem . . . 6
2.1.1 Definition . . . 6
2.1.2 Genetic approach . . . 6
2.1.3 A practical example . . . 7
2.2 Ant Colony Optimization . . . 7
2.2.1 How does it work . . . 9
2.2.2 A formal description . . . 10
2.2.3 Application to the traveling salesman problem . . . 10
2.2.4 Link with genetic algorithm . . . 11

A Source code **14**
A.1 Knapsack problem . . . 14

Introduction

This report aims to explain genetics algorithms which reproduce the principles of natural selection in different biological species. The genetic algorithms are used in scientific applications and in daily activities. In general, scientific problems have to comply with norms constraints that either performance or cost. However, goals could require the same parameter to be reached. Thus, optimization goals could be defined as a mathematical expression commonly known as the fitness function.

Chapter 1

Overview of the genetic algorithms

Genetic algorithms are quite simple and stochastic methods for optimization. Instead of improving a single solution, GA work with a set of individuals which represent possible solutions of the problem. It is only based on *probabilistic transition rules*. Each individual has an evaluation with respect to the desired solution. The next generation is created by the best suited individuals. The genetic algorithms evaluate each individual's value in the fitness function for each population. In fact, this is the optimization approach.

1.1 Principle

Selection, *reproduction* and *mutation* are the three main ingredients in a genetic algorithm. First of all, the selection process is to survive the next generation. In the nature, the more one individual is adapted to its environment, the more is its chance to transfer its genes to the next generation. In genetic algorithms, the next population is decided with an *evaluation of a fitness function*. Each candidate is as an array of bits and is assigned to a probability according to its aptitude in the fitness function. Thanks to this probability, the parents can be chosen during the reproduction process of the next generation.

The reproduction is a *recombination process* in which the new gene is only formed by the genes of the parents. Indeed, the parents exchange parts of their genes to form a child. For each child, new parents are selected and the process ends when a new population of appropriate size is created. The crossover can be singlepoint or multipoint. The following figure shows the crossover for one point:

$$\left.\begin{array}{ll} Gene1 & 01011\ 001 \\ Gene2 & 11100\ 010 \end{array}\right\} \rightarrow$$

Lastly, the mutation means that some changements occur due to mistakes

during the reproduction process. The mutation takes place when one individual has different characteristics that is not possibly exist at the beginning. From the point of view of genetic algorithms, it is equal to a random change of the value of one individual in the population.

1.2 Illustration with a mathematical problem

As a simple illustration of genetic algorithm, we would want to find an expression of which results 987 by mixing the digits 0 through 9 and the symbols +, -, / and

```
def gene_to_operand(gene):
    if gene == '0000':
        return '0'
    if gene == '0001':
        return '1'
    if gene == '0010':
        return '2'
    if gene == '0011':
        return '3'
    if gene == '0100':
        return '4'
    if gene == '0101':
        return '5'
    if gene == '0110':
        return '6'
    if gene == '0111':
        return '7'
    if gene == '1000':
        return '8'
    if gene == '1001':
        return '9'
    if gene == '1010':
        return '+'
    if gene == '1011':
        return '-'
    if gene == '1100':
        return '/'
    if gene == '1101':
        return '%'

    return ''
```

Firstly, we generate randomly 100 genes. Now that we have created this population of potential solutions, we need to evaluate each individual's probabilities assigned by its function fitness. For that purpose, we seperate the string of 48 bytes in 12 parts of 4 bytes. Hence, we could interpret the expression of each gene.

If the result of the expression means nothing (i.e. 2+/4), we give it as results a large number.

We choose the best 50 individuals for creating the next generation. In this list, we take 2 parents which will give birth to 4 children. The principle is to cut each parent's genes into two parts A and B, then assembling the part A of the first parent with the part B of the other and vice versa. The point of intersection is randomly determined. After this process, we obtain another population of 100 individuals which is supposed to be better than the previous one.

The successive generations are slowly converging to the solution. but sometimes the algorithm reachs a local extremum. It means that, at the beginning, the genes were not enough to find the solution. In order to avoid this issue, we use the mutation process. In this example, we randomly select one gene among the 100 and we reverse one byte of its 48. The rate of mutation is low, approximately $\frac{1}{1000}$. One individual in a thousand will mutate and thus the algorithm will avoid falling into a local extremum.

Finally, these are the kind of results we have:

```
Generation 0, best : [28, u'
    ↪ 100111100101100111011001000101100010100010000111']
Generation 1, best : [32, u'
    ↪ 100111100101100111011001000100110101001010110100']
...
Generation 10, best : [1, u'
    ↪ 010100110101101001011100011010001010010001010001']
Generation 11, best : [0, u'
    ↪ 001111100101101000100000001111101010011101001001']
Formula: 35+203+749
```

```
Generation 0, best : [11, u'
    ↪ 100110011000111111010111111101010100000101110101']
Generation 1, best : [11, u'
    ↪ 100110011000111111010111111101010100000101110101']
...
Generation 17, best : [1, u'
    ↪ 100101111000101010101001000011011010001011110000']
Generation 18, best : [0, u'
    ↪ 100110010010101000011011010100001101010011100100']
Formula: 992+1-50%44
```

```
Generation 0, best : [2, u'
    ↪ 100110000101110110010011010011101111010101111110']
...
Generation 6, best : [0, u'
    ↪ 101101000010110110010010111010101001111000110111']
Formula: -42%92+937
```

Chapter 2

Examples

This chapter is dedicated to two important examples. The examples we provide are examples where no polynomial solution is known currently, that is why there are, in our opinion, very interesting. Firstly we will introduce the *knapsack problem*, a problem with many industrial applications. Secondly, we will introduce a method to find a *good* solution the travel salesman problem using an ant colony.

2.1 The knapsack problem

2.1.1 Definition

The knapsack problem is a combinatorial optimization problem. Given different items with a value and a weight, try to maximize the value while the weight must not be higher than a given limit. This problem can be mathematically expressed as follow :

$$\arg\max_{\mathbf{x}} \sum_{i=1}^{n} v_i x_i \text{ and } \sum_{i=1}^{n} w_i x_i \leqslant L, \mathbf{x} \in \{0,1\}^n \tag{2.1}$$

where L is the maximum weight to not exceed.

2.1.2 Genetic approach

Now, let's see how quite good solutions to this problem can be given using a genetic approach.

As explained is the previous chapter, we need to define what an individual is in this case, and a function in charge of scoring individuals. In our case, we decided that an individual $\mathbf{x}$ is a vector of $\{0,1\}^n$. If $x_i = 1$, then a i-th item is taken in the bag otherwise it is not. Now, let's define the scoring function.

$$S(\mathbf{x}) = \begin{cases} \sum_{i=1}^{n} x_i v_i & \text{if } \sum_{i=1}^{n} x_i w_i \leqslant L \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

This function ensures that solutions exceeding L will be seen as *bad* (they will have the minimum score) and will have no probability to be selected to make the next generation. This function also ensures that the higher the value of a solution is, the higher its score is ; which is exactly what we want to maximize.

2.1.3 A practical example

As a practical example, we will use the one from `http://rosettacode.org/wiki/Knapsack_problem/0-1`. This is the story of a hiker. He wants to hike for the whole day, but he can carry 4 Kg only and he can not decide which items to take. So he wrote down the list of possible items and add two columns. One giving the weight of each items and an other giving the value of an item. The value of an item represents how much it is important. The table 2.1 gives the list of possible items and their weight and values.

We can help this guy thanks to the genetic algorithm we have made. We have run our algorithm over one hundred generation of one thousand individual each. We can give him the best solution among all the returned solutions. Having run our algorithm few times, and we can suggest those solutions :

— With a value of 1000 and a weight of 3.92 Kg : map, compass, water, sandwich, glucose, cheese, suntan cream, waterproof trousers, waterproof overclothes, note-case, sunglasses and socks ;
— With a value of 977 and a weight of 3.97 Kg : map, water, sandwich, glucose, cheese, suntan cream, waterproof trousers, waterproof overclothes, note-case, sunglasses, towel and sock ;
— With a value of 970 and a weight of 3.79 Kg : map, compass, water sandwich, glucose, banana, suntan cream, camera, waterproof overclothes, note-case and sock ;
— ...

Each run gives a solution, we cannot be sure that we have *the best* solution, but we do have a good solution, at least.

The python source code we wrote is given in appendices (cf A.1).

2.2 Ant Colony Optimization

In this section, we would like to talk about an method called *Ant Colony Optimization* used to find a short path in an unknown area. First of all, we will

Name	weight (Kg)	value
map	0.09	150
compass	0.13	35
water	1.53	200
sandwich	0.50	160
glucose	0.15	60
tin	0.68	45
banana	0.27	60
apple	0.39	40
cheese	0.23	30
beer	0.52	10
suntan cream	0.11	70
camera	0.32	30
T-shirt	0.24	15
trousers	0.48	10
umbrella	0.73	40
waterproof trousers	0.42	70
waterproof overclothes	0.43	75
note-case	0.22	80
sunglasses	0.07	20
towel	0.18	12
socks	0.04	50
book	0.30	10

Table 2.1: Items

explain how it works and finally we will explain why it can be seen as a genetic algorithm.

2.2.1 How does it work

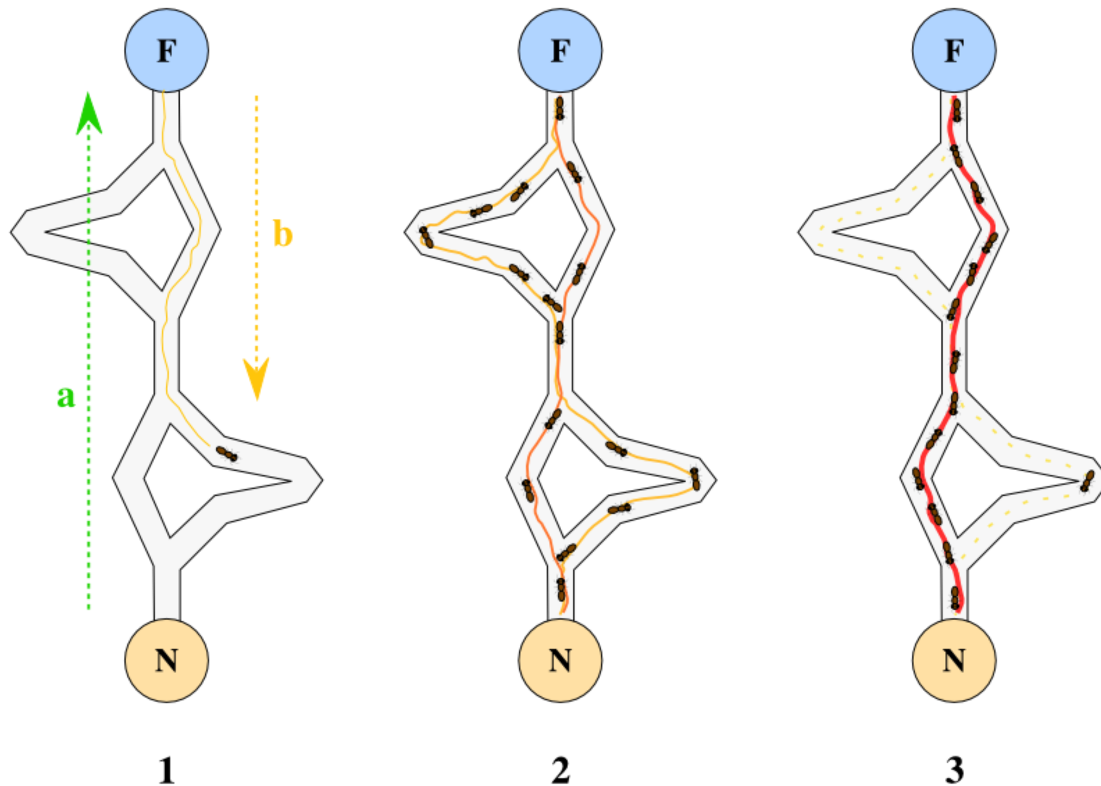


Figure 2.1: how does it work

There are two points, N (nest) and F (food). The goal is to find the shortest path from N to F. At the beginning an ant tries to find, randomly, a path from N to F. When a path is found, the ant goes back to N, using the same path leaving on it a pheromone. The quantity of pheromone is directly proportional to the length of the path. The following ants will also try to find a path to F, but a little bit less randomly because they will be attracted by the pheromones. Pheromones are not endless, they disappear with the time. So, in the end, the more ant will use a path, the more pheromone there will be. The shorter a path is, the more pheromones it has, so the long parts will not be used by the ants and they will take a short path – may be not the shortest, but a good one at least...

2.2.2 A formal description

Let's describe the method in a more formal way. We have m ants. Each ant moves according to a *moving rule*, giving the probability to move from i to j. The rule is :

$$p_{ij}^k(t) = \begin{cases} \frac{\tau_{ij}(t)^\alpha \eta_{ij}^\beta}{\sum_{l \in J_i^k} \tau_{il}(t)^\alpha \eta_{il}^\beta} & \text{if } j \in J_i^k \\ 0 & \text{otherwise} \end{cases} \tag{2.3}$$

where J_i^k is the set of all the possible movements of the k-th ant at the location i. η_{ij} is the visibility, usually taken as the inverse of the distance between i and j. τ_{ij} is the intensity of the pheromone on the edge from i to j at t. α and β are two control parameters.

Once each ant has visited each city, the k-th leaves a quantity $\Delta\tau_{ij}^k$ on each edge visited :

$$\Delta\tau_{ij}^k = \begin{cases} \frac{Q}{L^k(t)} & \text{if } (i,j) \in T^k(t) \\ 0 & \text{otherwise} \end{cases} \tag{2.4}$$

where $T^k(t)$ is the set of edges taken by the k-th ant at the t-th iteration, $L^k(t)$ is the length of the path and Q a control parameter. This function will favours short paths giving them more pheromones.

Then, the intensities have to be updated :

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^k(t) \tag{2.5}$$

where m is the total number of ants and ρ an evaporation parameters.

In the end, ants should all take the same path, and it may be a good solution to the problem. The ant colony algorithm can be long, but it always ends. It ends when they all take same path.

2.2.3 Application to the traveling salesman problem

The *traveling salesman problem* is a quite famous optimization problem. A traveling salesman has n cities to visit and he wants to minimize the distance to travel. Let's use an ant colony algorithm to find a good solution to this problem.

We assume that the graph, representing the cities, is complete. That is to say, there is a edge between each pair of vertices. The goal is to find the Hamiltonian path with the minimum value. Each ant starts from a random vertex and moves [1]

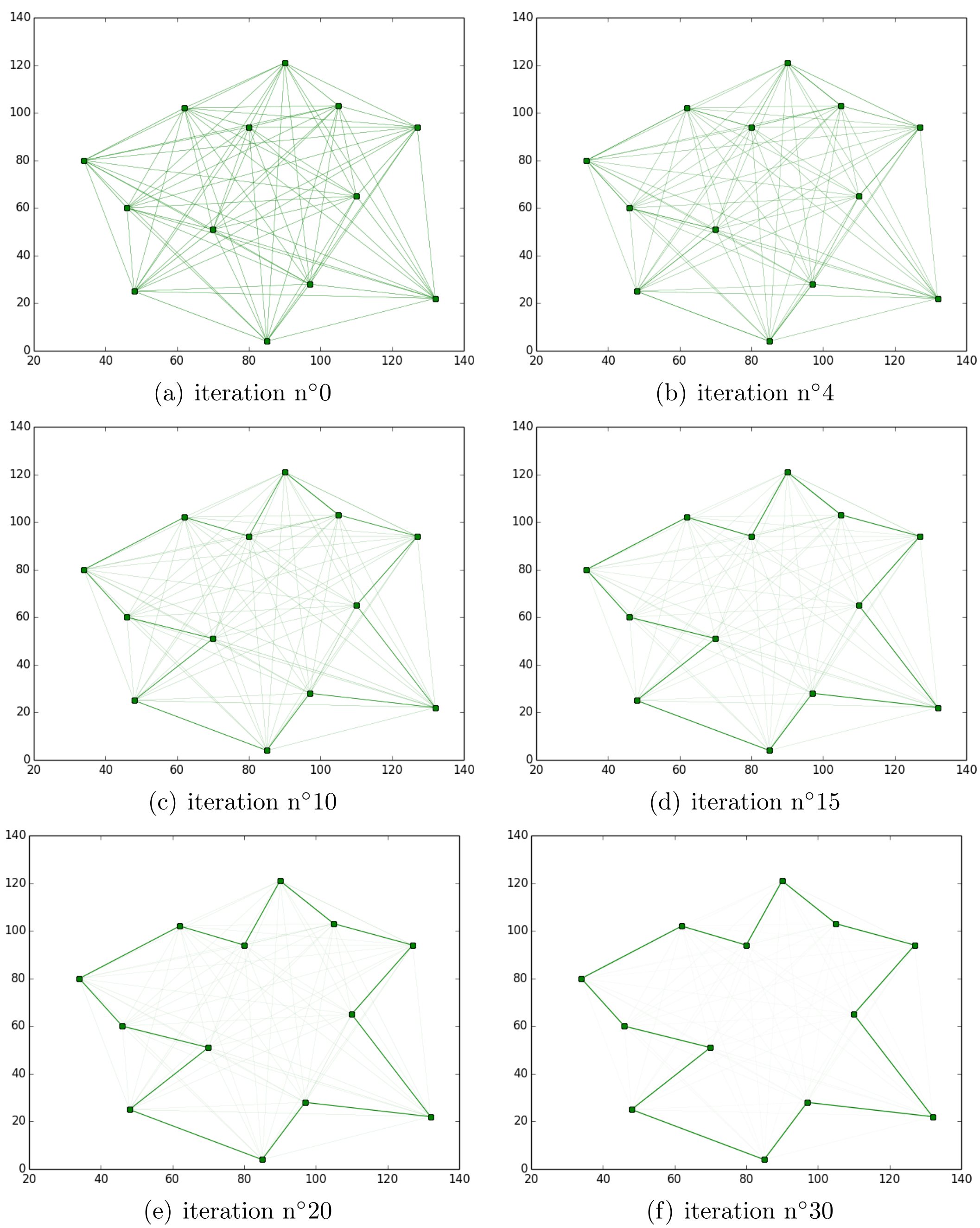
1. at $t = 0$, we have $\tau_{ij}(t) = \varepsilon, \forall(i,j)$ to avoid zero division errors...

using the rule (2.3). Each ant has a memory so that it cannot visit a vertex twice. When each ant has visited all the vertices − i.e. found an Hamiltonian path − the algorithm updates each edge using (2.5). So, for the next round, edges which has been providing a short path have a higher probability to be taken than the others. Quite quickly, the algorithm converges to a solution. It might not be the best one, but from a human point of view, it looks to be a good one. The figure 2.2 shows how the solution is found [2].

2.2.4 Link with genetic algorithm

This method might look, at first, very different than what we have been talking so far (with chromosomes, population, etc). But actually this method *can be seen as* a genetic algorithm. As a recall, a genetic algorithm is a method where there is a population, where each individual is a potential solution to a given problem. Each individual is scored and using the best fitted individuals, the next generation is built. In the case of the Ant Colony Optimization, a chromosome is an ant, an individual is a path. The *best* individuals are scored by the inverse of the path length (see (2.4)), so short paths might be taken again.

2. The width of an edge is proportional to the amount of pheromone on it.



(a) iteration n°0

(b) iteration n°4

(c) iteration n°10

(d) iteration n°15

(e) iteration n°20

(f) iteration n°30

Figure 2.2: Traveling salesman problem

Conclusion

In this report, the selected examples show great results for an optimization work. However, the genetic algorithm needs to be adapted to the problem, especially the selection method. The computation time required to find the solution of a problem is the only inconvenience of these algorithms. Overall, the genetic algorithms are really effective for finding an optimal value.

Appendix A

Source code

A.1 Knapsack problem

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

from __future__ import (unicode_literals, division, print_function
    ↪ )

from collections import namedtuple
from random import choice, random

Item = namedtuple('Item', ['name', 'weight', 'value'])
KNAPSACK = tuple([Item('map', 9, 150),
                  Item('compass', 13, 35),
                  Item('water', 153, 200),
                  Item('sandwich', 50, 160),
                  Item('glucose', 15, 60),
                  Item('tin', 68, 45),
                  Item('banana', 27, 60),
                  Item('apple', 39, 40),
                  Item('cheese', 23, 30),
                  Item('beer', 52, 10),
                  Item('suntan cream', 11, 70),
                  Item('camera', 32, 30),
                  Item('t-shirt', 24, 15),
                  Item('trousers', 48, 10),
                  Item('umbrella', 73, 40),
                  Item('waterproof trousers', 42, 70),
                  Item('waterproof overclothes', 43, 75),
                  Item('note-case', 22, 80),
                  Item('sunglasses', 7, 20),
                  Item('towel', 18, 12),
```

```
                    Item('socks', 4, 50),
                    Item('book', 30, 10),
                  ])


class Individual(object):
    def __init__(self, chromosome=None, size=None):
        assert chromosome or size, 'You must provide `chromosome`
           ↪ or `size`'
        if chromosome:
            self.chromosome = chromosome
        else:
            self.chromosome = self._newchromosome(size)
        self.score = None
        self.fitness = None

    def _newchromosome(self, size):
        return [Genes.randomgene() for _ in xrange(size)]

    def __repr__(self):
        return ', '.join(KNAPSACK[i].name
                         for i, v in enumerate(self.chromosome) if
                            ↪  v)

    def __gt__(self, other):
        if not other:
            return True
        return self.score > other.score


class Genes(object):
    @staticmethod
    def randomgene():
        return choice([True, False])


class Population(list):
    def new(self, searchsize, popsize):
        for _ in xrange(popsize):
            self.append(Individual(size=searchsize))

    @property
    def best(self):
        return sorted(self, key=lambda ind:ind.score)[-1]


class Evaluator(object):
    def __init__(self, searchvalue, population):
        self.searchvalue = searchvalue
        self.population = population
```

```
76
        def evaluate(self):
78          for indiv in self.population:
                indiv.score = self._score(indiv)
80          self._fitness()

82      def _score(self, indiv):
            value = 0
84          weight = 0
            for ind, item in enumerate(indiv.chromosome):
86              if not item:
                    continue
88              value += KNAPSACK[ind].value
                weight += KNAPSACK[ind].weight
90          if weight > self.searchvalue:
                return 0
92          else:
                return value
94
        def _fitness(self):
96          total = sum(indiv.score for indiv in self.population)
            for indiv in self.population:
98              indiv.fitness = indiv.score/total * len(self.
                    ↪ population)

100
    class GeneticAlgo(object):
102     def __init__(self, generations, population, searchvalue,
            ↪ mutationrate):
            self.generations = generations
104         self.population = population
            self.searchvalue = searchvalue
106         self.mutationrate = mutationrate
            self.crossover = Crossover(mutationrate)
108         self.bestever = None

110     def run(self):
            for generation in xrange(self.generations):
112             Evaluator(self.searchvalue, self.population).evaluate
                    ↪ ()
                best = self.population.best
114             if not self.bestever or self.bestever < best:
                    self.bestever = best
116             self.display(generation, best)
                if best.chromosome == self.searchvalue:
118                 break
                self.nextgeneration()
120         return self.bestever
```

```
    def display(self, generation, best):
        print('-' * 40)
        print('Gen: {}'.format(generation))
        print('Best: {}'.format(best.score))

    def nextgeneration(self):
        pool = MatingPool(self.population)
        newpopulation = []
        for _ in xrange(len(self.population)):
            newpopulation.append(self.crossover.uniform(pool.
                ↪ random(),
                                                            pool.
                                                            ↪ random
                                                            ↪ ())
                                                            ↪ )
        self.population = Population(newpopulation)


class MatingPool(object):
    def __init__(self, population):
        self.matingpool = []
        for indiv in population:
            for _ in xrange(int(indiv.fitness)):
                self.matingpool.append(indiv)
            if (float(indiv.fitness) - int(indiv.fitness)) >=
                ↪ random():
                self.matingpool.append(indiv)

    def random(self):
        return choice(self.matingpool)


class Crossover(object):
    def __init__(self, mutationrate):
        self.rate = mutationrate
        self.mutator = Mutator(mutationrate)

    def uniform(self, parent0, parent1):
        chromosome = []
        for i in xrange(len(parent0.chromosome)):
            if random() >= 0.5:
                chromosome.append(parent0.chromosome[i])
            else:
                chromosome.append(parent1.chromosome[i])
        self.mutator.mutate(chromosome)
        return Individual(chromosome=chromosome)


class Mutator(object):
```

```
166     def __init__(self, mutationrate):
            self.rate = mutationrate
168
        def mutate(self, chromosome):
170         new = []
            for c in chromosome:
172             new.append(c if random() > self.rate else not c)
            return new
174

176 if __name__ == '__main__':
        generations = 100
178     population = Population()
        capacity = 400
180     population.new(len(KNAPSACK), 1000)
        mutation = 0.01
182     algo = GeneticAlgo(generations, population, capacity, mutation
           ↪ )
        best = algo.run()
184     print(best)
        print(best.score)
186     print(sum(KNAPSACK[ind].weight for ind, item in enumerate(best
           ↪ .chromosome) if
                item))
188     for index, item in enumerate(KNAPSACK):
            if not best.chromosome[index]:
190             print(item)
```