Exercise: An Evolutionary Algorithm for the Knapsack Problem

Introduction to Optimization lecture
at Université Paris-Saclay

Anne Auger and Dimo Brockhoff
firstname.lastname@inria.fr

October 2, 2015

Abstract

Evolutionary algorithms (EAs) are stochastic optimization algorithms which are based on the idea of artificial evolution of solutions towards the optimum. In this exercise, we will implement a basic EA for the knapsack problem in order to show how easy their application can be. But at the same time, we will also illustrate that the choice of their individual components and the tuning of their parameters is crucial to their success.

1 Part I: Designing the algorithm

Designing the operators of an evolutionary algorithm is often already half the battle, when solving a difficult optimization problem. However, parameter tuning might be equally important to further improve the algorithm once it is implemented. Spend therefore first some time on the following questions before starting the actual implementation in Part II of the exercise.

a) How would you define the chromosomes of the algorithm? In other words, how do you represent a solution in your algorithm?

b) What is the genotype/phenotype mapping in your case? In particular, how would you encode the constraint and make sure that you compute

feasible solutions? Remember the different ways to do that from the lecture slides.

c) Think about the most basic algorithm. In particular do not use any crossover to simplify the implementation. Which population initialization would you use? Which mutation operator(s)? Which mating and environmental selection strategies?

2 Part II: Implementing a very basic Evolutionary Algorithm

We will now implement a very basic evolutionary algorithm based on the considerations of Part I. In order to keep the implementation simple, please follow the instructions below.

a) Implement a basic 1-bit flip mutation operator which takes a solution (set) and flips a single bit, uniformly chosen at random, in each solution. Assume thereby that a solution is coded as an array of bits (i.e., each bit can be either 0 or 1 or, depending on your programming language, FALSE or TRUE).

b) Do you think that relying solely on 1-bit flips is a reasonable operator for the knapsack problem? What do you in particular expect to happen in the end of the optimization when you only use 1-bit flips? What do you suggest to circumvent this behavior?

c) Implement also a 2-bit flip mutation operator.

d) Implement the (random) initialization of the algorithm's population with μ individuals. Choose μ in the range of $10 \ldots 20$ for now to keep later numerical experiments quick.

e) To deal with infeasible solutions, implement a greedy repair function which takes a solution (set) and removes items greedily in the order of their profit/weight ratio until the weight constraint is fulfilled. Think carefully about the order in which the items are removed.

f) Implement the whole algorithm now. To keep it simple, we assume that only a single parent is selected uniformly at random from the population

in each iteration. After mutation, the worst solution is replaced with the new offspring if it is better. The greedy repair function should be always used to only produce and compare feasible solutions.

g) Test your algorithm for some of the knapsack instances from `http://researchers.lille.inria.fr/~brockhof/optimizationSaclay/knapsackinstances/` by using the code of last week for evaluating the objective and constraint functions. In particular compare the algorithm variant which only uses 1-bit flip mutations vs. the one which uses only 2-bit flip mutations vs. a variant which chooses between the two mutations randomly. What do you observe in terms of the solution quality over time?

h) With respect to parameter settings: how does the performance of the algorithm change with the population size μ and the ratio of 1-bit flip to 2-bit flip mutations?

3 Part III: Optional

The following questions and tasks are optional but can again be taken as additional exercises to prepare for the exam.

a) Obviously it might be needed to exchange more than one item at the same time when performing a mutation. Implement the standard bit mutation in which each bit is flipped with a probability of $1/n$ with n being the bitstring length. Compare it experimentally with the mutations performing a fixed number of items.

b) Design and implement a crossover operator and decide at the same time about the ratio of when crossover and mutation and when only mutation is applied. Compare how much the algorithm is improved by using the crossover. The easiest way here is to plot the function values, averaged over > 2 independent runs, achieved by both algorithm variants over the number of function evaluations (the so-called "convergence plot").