Exercise: A Greedy Algorithm for the Knapsack Problem

Introduction to Optimization lecture
at Ecole Centrale Paris / ESSEC Business School

Dimo Brockhoff
firstname.lastname@inria.fr

October 28, 2016

Abstract

In the lecture, the general concept of *greedy algorithms* has been introduced in which locally optimal choices are made. In this exercise, we apply this idea to the knapsack problem. We are not only going to formally define the algorithm but also to implement it. The programming language should thereby be python (exceptionally, also MATLAB/Octave is okay).

1 Part I: Implementing the Knapsack Problem

Given a set of n items with weights $w_i \in \mathbb{R}$ and profit $p_i \in \mathbb{R}$ $(1 \leq i \leq n)$ and a weight restriction $W \in \mathbb{R}$, the knapsack problem asks for a packing of items into the knapsack which (a) total weight does not exceed the weight restriction and (b) has the maximum profit. Here, we are focusing on the 0-1 knapsack problem variant where each item is allowed only once (or not at all) in the knapsack:

$$\text{max. } f(x) = \sum_{j=1}^{n} p_j x_j \text{ with } x_j \in \{0, 1\}$$

$$\text{s.t.} \sum_{j=1}^{n} w_j x_j \leq W$$

Questions and Tasks

a) Start with implementing the objective function. Given a 0-1 vector of length n, it shall give back the f-value for a given knapsack problem instance, specified in a text file.

b) To this end, write the code which initializes the objective function by reading in the weights and profits of the items from a file of the format:

```
n = 100 # number of items
W = 78  # maximum weight of knapsack (capacity)
w_1 p_1
w_2 p_2
.
.
.
w_n p_n
```

with the `w_i` and `p_i` being the weights and profits of the item `i` respectively. The separators between weights and profits can be assumed to be blanks.

c) Then write a function for the constraint violation in the same manner.

2 Part II: A Greedy Algorithm for the Knapsack Problem

In the second part of the exercise, we want to develop and implement a greedy algorithm for the knapsack problem. We cannot expect that the greedy approach will be able to find the optimal function value reliably[1].

[1]Otherwise, a whole bunch of brilliant scientists would have been wrong for quite some time and/or we would be already rich: `https://en.wikipedia.org/wiki/P_versus_NP_problem`

Instead, we want to investigate experimentally how far the quality of the produced solution of the greedy algorithm is in terms of the true optimal function value. To compute the optimal function value (for small instances at least), we also implement an exact brute-force algorithm which enumerates all potential solutions.

d) Think about which greedy choice you can make when you have to come up with a solution for the knapsack problem.

e) Implement your greedy algorithm and test it on a few example instances which you can find at `researchers.lille.inria.fr/~brockhof/introoptimization/knapsackinstances/`.

f) In order to double-check that your algorithm is doing the right thing, write a simple brute-force algorithm which enumerates all solutions of the search space and returns the best (feasible) solution it has seen.

g) Compare the output of the two algorithms on the knapsack instances provided at the above link. In particular check how often the greedy algorithm finds the optimal solution of the brute-force approach.

3 Part III: Optional

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

a) Write a random search algorithm which randomly picks a new assignment of items to the knapsack at each step and keeps track of the best-so-far f-value. It should have the number of iterations (or the number of times, it samples the objective function) as an input parameter.

b) Compare all algorithms on instances with increasing difficulties in order to see the scaling with the input length. For example, create random instances with different numbers of items and plot the runtime to reach the optimal solution over this "measure" of problem difficulty. Do you observe differences between runs on the same instance? How large are the variances between instances of the same dimension? How large between different dimensions? In case you observe differences, think about what you actually display best to keep the most information.