Exercise: Dynamic Programming for the Knapsack Problem

Introduction to Optimization lecture
at Université Paris-Saclay

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

September 25, 2015

Abstract

In the lecture, we have seen the general concept of *dynamic programming* and it is the purpose of this exercise to apply it to the knapsack problem. We are going to not only formally define the algorithm but also implement it. The choice of the programming language you use is up to you, but please make sure that we can read and understand your code. To this end, we recommend to use a standard programming language such as python (preferred), MATLAB, R, C/C++, or Java whereas the former are preferred because of their easier interactive handling when developing and testing in a scientific environment. Please refrain from using exotic languages such as Assembler or Postscript.

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 in the same manner.

2 Part II: Dynamic Programming for the Knapsack Problem

In the second part of the exercise, we want to develop and implement an exact algorithm for the knapsack problem based on the idea of dynamic programming. Before you actually implement the algorithm, answer the following questions about the dynamic programming formulation of the problem first:

a) What are potential subproblems here?

b) How do you solve the smallest problems (initialization)?

c) How do you construct larger subproblems from already solved smaller subproblems? Write down the Bellman equation.

Finally, implement the algorithm for the knapsack problem and test it. To this end, follow the tasks below.

d) Implement your dynamic programming algorithm and test it on a few example instances which you can find via the lecture's web page at `researchers.lille.inria.fr/~brockhof/optimizationSaclay/knapsackinstances/`.

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

f) Compare the output of the two algorithms on the knapsack instances provided on the lecture's web page. In particular check whether both the brute-force and the dynamic programming approach result in the same optima (and in particular the same optimal values). Why is the latter more important to test?

g) Finally also compare the times, the algorithms take to solve the provided instances. When looking at the influence of the problem dimension (i.e. the number of items), can you make predictions about larger, yet un-tested instances? See also Part III.b).

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.