The Knapsack problem

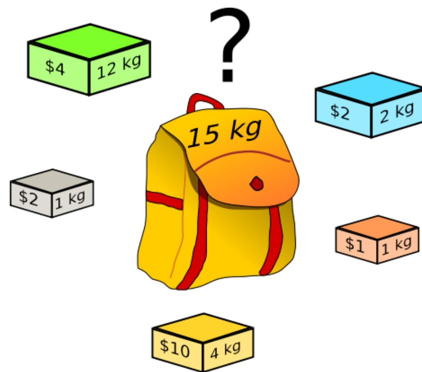
Contents

1 The knapsack problem (Sec. 1.3) **1**
1.1 A typical optimization problem 2

2 Backtracking Algorithms (Sec. 4.1) **2**
2.1 A naive approach to the knapsack problem 2
2.2 First optimization: Pruning the infeasible solutions 3

1 The knapsack problem (Sec. 1.3)

The knapsack problem is a key problem in combinatorial optimization. You have a set of objects, and to each is associated a weight, and a value. You are about to set out travelling, and you cannot carry more than a certain weight, say 15kg. The total weight of all of your objects is more than this. How do you choose a subset of your objects so that their total value is maximized, and their total weight does not exceed 15kg?



This problem appears in many different formats and variations in resource allocation, cryptography, combinatorics, complexity theory..

There are several variants of the problem. For each of the following, assume there are n items, that the weight of item i is w_i, and that the value of item i is v_i. The capacity of the backpack is M. We represent a backpack configuration by an n-tuple $[x_1, x_2, \ldots, x_n] \in \{0, 1\}^n$ such that $x_i = 1$ if and only if item i is in the backpack.

Decision Given target value V, *does there exist* a configuration $[x_1, x_2, \ldots, x_n] \in \{0, 1\}^n$ such that

$$\sum_{i=1}^{n} v_i x_i \geq V$$

and

$$\sum_{i=1}^{n} w_i x_i \leq M?$$

Search Given target value V *find a configuration* $[x_1, x_2, \ldots, x_n] \in \{0, 1\}^n$ such that

$$\sum_{i=1}^{n} v_i x_i \geq V$$

subject to

$$\sum_{i=1}^{n} w_i x_i \leq M.$$

Optimal Value Find *the maximum value* V such that

$$\sum_{i=1}^{n} v_i x_i \geq V$$

and

$$\sum_{i=1}^{n} w_i x_i \leq M.$$

Optimization Find the configuration $[x_1, x_2, \ldots, x_n] \in \{0,1\}^n$ such that

$$\sum_{i=1}^{n} v_i x_i \geq V$$

is optimized subject to

$$\sum_{i=1}^{n} w_i x_i \leq M.$$

1.1 A typical optimization problem

This example bears many of the typical features of an optimization problem. There are *constraints* to be satisfied. Any n-tuple which satisfies them is a *feasible solution*. Associated to each feasible solution is an *objective function*, which takes the value of an integer or real number typically thought of as either the cost or profit. The object of an optimization problem is to find a feasible solution that attains the maximum possible profit, or incurs the minimum possible cost.

2 Backtracking Algorithms (Sec. 4.1)

A *backtracking algorithm* is a recursive method to build up feasible solutions to a combinatorial optimization problem one step at a time. You can view a backtracking algorithm as an exhaustive search amongst all feasible solutions to find the optimal solution. Very often the set of feasible solutions is very large, and hence we would like to avoid considering feasible solutions that are clearly not optimal. We will do this by *pruning* the set of solutions we consider, and this will be the first strategy. Let us see how it works in practice.

2.1 A naive approach to the knapsack problem

We are pros at listing all binary strings at this point. We could simply list all binary strings and iterate through all possible 2^n binary strings of length n, each time computing the constraint $\sum_{i=1}^{n} w_i x_i \leq M$ to see if it is feasible and then compute the value of the objective function $\sum_{i=1}^{n} v_i x_i$ and compare to the best so far.

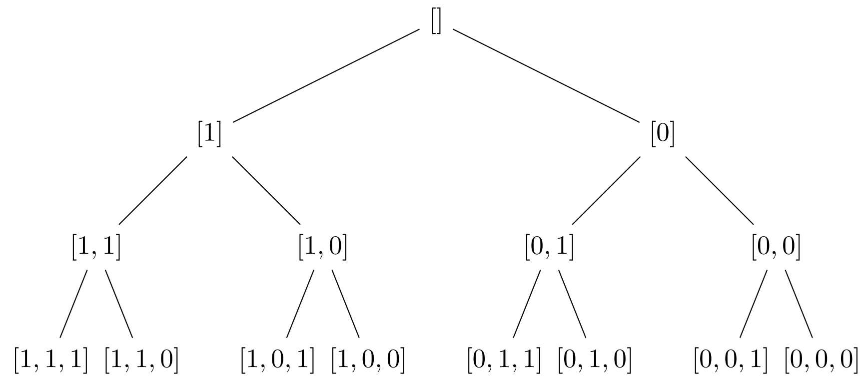
Naive Knapsack solution

```
knapsac:= proc(m)
global x      // current feasible solution built up one digit at a time
         optX // current optimum solution
         optV // current optimum value
local   curP // current value

if m=n then // determine the current value against optimum value
        if add(wi*xi, i=1..n) <= M   // current x is feasible
           curV:= add(vi*xi, i=1..n)
           if curV > optV    // a more optimal solution!
                optV:= curV
                optX:=x
else // we are only partway through processing the string
         xm:= 1
         knapsac(m+1)
         xm:= 0
         knapsac(m+1)
end proc;
```

The algorithm is invoked with `knapsack(0)` and at the end of the run the global variable `optX` contains the optimum solution and `optV` will have the optimum value of the objective function.

The *state space* of the algorithm is all of the values that are generated. For this case, we represent it with a state space tree which is traversed in a depth first search in the course of the algorithm:



This algorithm checks all of the strings and takes $\Theta(n)$ time to process each string, so the total run time is $\Theta(n2^n)$, which is exponential, and not suitable for large n.

2.2 First optimization: Pruning the infeasible solutions

This algorithm is calling out for at least a simple modification. We would like to test feasibility of some inner node, before we recurse on its children. For example, if we have partially built a solution, say $[x_1, \ldots, x_m]$ such that $m < n$, and we know already that $\sum_{i=1}^{m} w_i x_i > M$, we do not need to continue to add elements and test configurations; the bag is already too heavy! We assume in this example that items have a non-negative weight, so there is no item we can add that will bring the weight back down. We *prune* the search tree and do not consider descendants of this element.

```
Optimized Knapsack solution
oknapsac:= proc(m, curW)
global x      // current feasible solution built up one digit at a time
          optX // current optimum solution
          optV // current optimum value
          Cm// value of the first m elements in x
local   curV // current value

if m=n+1 then // determine the current value against optimum value
        if add(wi*xi, i=1..n) <= M   // current x is feasible
           curV:= add(vi*xi, i=1..n)
           if curV > optV    // a more optimal solution!
                optV:= curV
                optX:=x
        Cm:={} // we are done, and the recursive call below will be
        empty.
else // we are only partway through processing the string
        if curW+wm <= M then // still feasible!
            Cm:= {1,0}   // check both children in the search tree
        else
            Cm:= {0}
for d in Cm do
            xm:= d
            oknapsac(m+1, curW+wmxm)
end proc;
```

This is serves as a general template for an optimization problem. Our search space may be bigger than simply binary strings, and it is left as an exercise to see how to write a generic algorithm. (See Algorithm 4.2 in the Kreher and Stinson *Combinatorial Algorithms*).