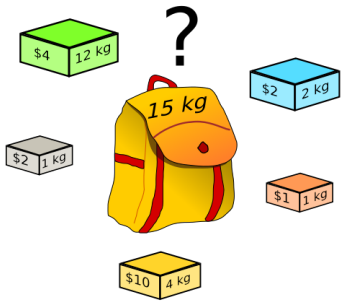
The Knapsack problem

Contents

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 $X = [x_1, x_2, \ldots, x_n] \in \{0,1\}^n$ such that $x_i = 1$ if and only if item i is in the backpack. Each x_i is called an indicator variable, since its range is $\{0,1\}$ and its value indicates whether i has been selected in the current backpack configuration.

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, for some configuration $[x_1, x_2, \ldots, x_n] \in \{0,1\}^n$,

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

and

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

Optimization Find a configuration $[x_1, x_2, \ldots, x_n] \in \{0,1\}^n$ such that the value $\sum_{i=1}^{n} v_i x_i$ is maximum,

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.

2 Backtracking Algorithms (Sec. 4.1)

A backtracking algorithm is a recursive method to generate feasible solutions $[x_1, x_2, \ldots, x_n]$ to a combinatorial optimization problem one component x_i at a time. At all times, we have a partial configuration $X = [x_1, x_2, \ldots, x_{m-1}]$ for some m with $1 \leq m \leq n$. The vector X represents a node at level $m-1$ of a search tree whose root represents $X = [\,]$. We call the algorithm with input m to generate some or all of the children of node X and recursively call the algorithm with input $m+1$ for each of these children.

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 with the knapsack problem

2.1 A naive approach to the knapsack problem

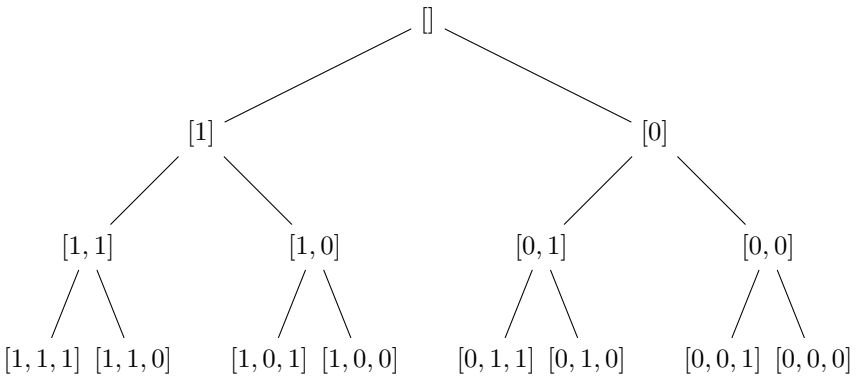
We have already seen a couple of ways listing all binary strings (lexicographically, minimal change orders). We can simply recursively generate all 2^n possible n-bit indicator vectors $X = [x_1, x_2, \ldots, x_n]$, each time testing whether $\sum_{i=1}^{n} w_i x_i \leq M$ to see if it is feasible. If X is feasible, then we compare the value of the objective function $\sum_{i=1}^{n} v_i x_i$ to the highest value found so far and record X if it is the most valuable feasible configuration found so far.

Naive Knapsack solution

```
knapsac := proc(m)
global w[1],w[2],...,w[n]     // weights of objects
       v[1],v[2],...,v[n]     // values of objects
       [x[1],x[2],...,x[n]] // partial configuration is X = [x[1],x[2],...,x[m]]
       optX                   // current optimum solution
       optV                   // current optimum value
if m=n                        // compare the current configuration value to optV
   if add(w[i]*x[i], i=1..n) <= M     // current x is feasible
      curV:= add(v[i]*x[i], i=1..n)
      if curV > optV                  // a better solution!
           optV := curV
           optX :=X
else                 // Branch the search on both choices of whether to include item m
    x[m+1]:= 1
    knapsac(m+1)
    x[m+1]:= 0
    knapsac(m+1)
```

For $m = 1, \ldots, n$ the function call `knapsac(m)` assumes that the partial configuration $X = [x_1, x_2, \ldots, x_m]$ indicates the current subset of items in $\{1, 2, \ldots, m\}$ in the backpack. If $m = n$ then X indicates a complete backpack configuration, which we test for optimality. If $m < n$ then we invoke `knapsack(m+1)` after setting $x_{m+1} = 1$ and again with $x_{m+1} = 0$. The algorithm is started with `knapsac(0)`. In the end, the global variable `optX` will represent an optimum solution and `optV` will equal optimum value of the objective function.

The state space of the algorithm is the set of all partial configurations X that are generated. The state space is the set of nodes of a search 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 the feasibility of each inner node X before we recurse on its children. For example, suppose we have partially built a solution, say $X = [x_1, \ldots, x_m]$ for some $m < n$, and we are considering whether to search the child node $[x_1, \ldots, x_{m+1}]$. If we find that $\sum_{i=1}^{m+1} w_i x_i > M$, then we do not need to search the node $[x_1, \ldots, x_{m+1}]$ or any of its decendants; adding item $m + 1$ makes the bag too heavy! (We are assuming that each item has a non-negative weight, so there is no item we can add that will bring the weight back down.) We have pruned the subtree rooted at the node $[x_1, \ldots, x_{m+1}]$.

While executing the search at level m, we generate a set $C(m)$ which lists all of the children of X which pass the above pruning test. Then, one at a time, we set x_{m+1} equal to an element of $C(m)$ and call the search recursively at level $m+1$. When the procedure is called, the current partial configuration is guaranteed to be feasible.

As a further optimization, we pass the current value and current weight of X to the procedure. This saves two $O(n)$ summations and constant-time for each procedure call .

Optimized Knapsack solution

```
oknapsac := proc(m, curW, curV)
arguments m              // Level of recursion
          curW           // weight of current partial solution X
          curV           // value of current X
global w[1],...,w[n]  // weights of objects
       v[1],...,v[n]  // values of objects
       x[1],...,x[n]  // X = [x[1],x[2],...,x[m]] is a feasible partial configuration
       C[0],...,C[n]  // C[k] is the choices set of subtrees for level k, k=0,1,...,m
       optX           // current optimum solution
       optV           // current optimum value

if m=0                       // Should we initialize?
    optV := 0
    optX := []

if      m=n                  // Is X is a complete (feasible) configuration?
           C[m]:={}              // X has no children to recurse on
           if curV > optV        // X is a better solution
              optV := curV
              optX := X
else if curW + w[m+1] <= M  // Does adding item m+1 retain feasibility?
           C[m] := {1,0}         // Both children are feasible
else
           C[m] := {0}           // Prune the child [x[1],x[2],...,x[m], 1]

for x[m+1] in C[m]           // Recurse on unpruned children
    oknapsac( m+1, curW + w[m+1]*x[m+1], curV + v[m+1]*x[m+1] )
```

We start the search with `oknamsac(0,0,0)`. When finished, `optX` will be an optimal solution with value `optV`.

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*).