Subset-Sum and Knapsack problems

Notation

We will consider vectors of the form $x_0, \cdots, x_{n-1}$ and sets of the form $S \subset \{0, \cdots, n-1\}$. We will use notation

$$x(S) = \sum_{i \in S} x_i$$

Usage:
values of elements $v_0, \cdots, v_{n-1}$, value of a set $v(S)$
weights of elements $w_0, \cdots, w_{n-1}$, weight of a set $w(S)$

Subset-Sum Problem

Input: weights $w_0, \ldots, w_{n-1}$, target weight W
Question: is there $S \subset \{0, \cdots, n-1\}$ such that $w(S) = W$?

a. Subproblems: for $U \le W$ and $k \le n$, $A(U, k)$ is the answer, true or false, to the question
"is there $S \subset \{0, \cdots, k-1\}$ such that $w(S) = U$?"

b. Recursive solution
Basic cases: $A(0,0) = \textbf{true}$, $A(U,0) = \textbf{false}$ for $U \neq 0$.
Recurence relation:

$$A(U, k+1) = A(U, k) \textbf{ or } A(U - w_k, k) \tag{1}$$

c. Pseudocode. We will use array `A[W+1]` for $A(U, k)$, so we will initialize it with $A(\bullet, 0)$ and in iteration k we will change the content to $A(\bullet, k+1)$:

```
// initialization
A[0] = 1;
for (i = 1;  i <= W;  i++)
   A[i] = 0;
// iterations
for (k = 0;  k < n;  k++)
   for (i = W;  i >= w[k];  i--)
      A[i] ||= A[i-w[k]];
// result
return A[W];
```

Clearly, the running time is $O(nW)$. Note that in the iteration we have to update array `A[W]` from the largest to the smallest, so the assignment

```
A[i] ||= A[i-w[k]];
```

has at its right-hand side values that are not updated yet, so it is indeed equivalent to the recurrence (1).

Note.

this algorithm is efficient only when W is small. Using "small" number of bits we can describe "large numbers". When weights are large, the problem is NP-complete and thus we believe that no efficient algorithm exists. The best known algorithm performs roughly $\Theta(2^{n/2})$ operations.

Knapsack Problem

Input: weights $w_0, \ldots, w_{n-1}$, values $v_0, \ldots, v_{n-1}$, weight limit W
Task: find $S \subset \{0, \cdots, n-1\}$ such that $w(S) \leq W$ while $v(S)$ is maximal.

Let V be the largest possible $v(S)$. We will show two very algorithms very similar to the previous Subset Sum algorithm, one running in time $O(nW)$, the other, in time $O(nV)$.

Algorithm for small weights

a. Subproblems: for $U \leq W$ and $k \leq n$, $LV(U,k)$ is the largest $v(S)$ such that such that $w(S) = U$.

b. Recursive solution
Basic cases: $LV(0,0) = 0$, $LV(U,0) = -\mathit{infty}$ for $U \neq 0$.
Recurence relation:

$$LV(U,k+1) = \max(LV(U,k), LV(U-w_k,k)+v_k) \tag{1}$$

c. Pseudocode. We will use array `LV[W+1]` for $LV(U,k)$, so we will initialize it with $LV(\bullet,0)$ and in iteration k we will change the content to $LV(\bullet,k+1)$:

```
// initialization
LV[0] = 0;
for (i = 1;  i <= W;  i++)
  LV[i] = -1;
// iterations
for (k = 0;  k < n;  k++)
  for (i = W;  i >= w[k];  i--)
     if (x = LV[i-w[k]], x >= 0 && x+v[k] > LV[i])
        LV[i] = x+v[k];
// result
largestV = 0;
for (i = 0;  i <= W;  i++)
  if (LV[i] > largestV)
     largestV = LV[i];
return largestV;
```

Algorithm for small values

a. Subproblems: for $U \le V$ and $k \le n$, $SW(U,k)$ is the smallest $w(S)$ such that such that $v(S) = U$.

b. Recursive solution
Basic cases: $SW(0,0) = 0$, $SW(U,0) = \mathit{infty}$ for $U \ne 0$.
Recurence relation:

$$SW(U,k+1) = \min(SW(U,k), SW(U-v_k,k)+w_k) \qquad (1)$$

c. Pseudocode. We will use array `SW[W+1]` for $SW(U,k)$, so we will initialize it with $SW(\bullet,0)$ and in iteration k we will change the content to $SW(\bullet,k+1)$:

```
// initialization
SW[0] = 0;
for (i = 1;  i <= W;  i++)
  SW[i] = -1;
// iterations
for (k = 0;  k < n;  k++)
  for (i = V;  i >= v[k];  i--)
     if (x = SW[i-w[k]], x >= 0 && x+w[k] < SW[i])
        SW[i] = x+w[k];
// result
largestV = 0;
for (i = 0;  i <= V;  i++)
  if (SW[i] >= 0)
     largestV = i;
return largestV;
```

Knapsack Problem — approximate solution

How can we know that V is small? Because very quickly we can find V' such that

(i) $V' = v(S)$ for some $S \subset \{0, \cdots, n-1\}$

(ii) $V' \geq V/2$

This is an approximate solution with ratio 2.

It can be obtained with a greedy algorithm:

sort i's according to unit values v_i/w_i, so

$$\frac{\mathtt{v[I[0]]}}{\mathtt{w[I[0]]}} \leq \frac{\mathtt{v[I[1]]}}{\mathtt{w[I[1]]}} \leq \cdots \leq \frac{\mathtt{v[I[n-1]]}}{\mathtt{w[I[n-1]]}}$$

```
for (j = V′ = sw = 0;  sw <= W;  j++)
      lastv = v[I[j]],
      V′ += lastv,
      sw += w[I[j]];
V′ = max(V′-lastv,lastv)
```

One can easily show that V′ that results from the iteration is at least as large as V, and thus the final result is at least V/2.

To see that, we can represent item i as a rectangle of width w_i and height v_i/w_i. The value is equal to the area. A solution can be depicted by lining the items so that they touch each other while "standing" on the X axis, ordrered by height. The first rectangle touches Y axis.

In the greedy algorithm, we use the ordering of all rectangles. If we remove one, we shift the rectangles that were to the right, hence equal in height or smaller, to the left. This can shift the upper contour, "the skyline", down, and surely cannot shift the skyline up. Thus the skyline of the greedy solution is the highest possible.

The only reason that the greedy solution is not as good as the optimum is that the last item crosses the line $x = W$. But with this item, the skyline of the greedy solution contains the skyline of any other solution.

Try to draw a picture to illustrate this reasoning.

Remark: if we select correctly items with $w_i > W/k$ and add small items using this greedy method, the error is not larger than V/k.

Greedy algorithms

Greedy algorithms can be viewed as a special version of Decrease-and-Conquer approach.

We have a problem instance, and forming a solution can be viewed as making a sequence of decisions. We have some small number of choices for the decision, and each decision allows to:

change the problem instance I to a smaller instance I'

change each solution I' to the solution of I'

If a greedy algorithm we have an easy to compute criterion how to make the first decision.

The analysis of a greedy algorithm usually shows that there exists a solution of I that is consistent with our decision. Then, because we reduced the problem instance, each decision is the first, until the instance is so reduced that it has only one possible decision.

The analysis of the approximation of Knapsack Problem is not typical. More typical is forcing our decision into an arbitrary solution. We consider a hypothetical solution and if it is inconsistent, we modify it.

Greedy algorithms example: a scheduling problem

We will consider the following problem: we have a set J of possible scheduling entries of the form (i, s, t, p) where
i is the activity (or customer, or a job),
s is the start,
t is the termination or end,
p is the profit.

Two entries (i, s, t), (i', s', t') are in conflict if either

they overlap at some time point x, i.e. $s, s' \leq x \leq t, t'$ (we should not schedule two activities in the same time), or

$i = i'$ (we can schedule a certain activity only once)

A schedule is a set of scheduling entries without conflict. Our goal is to find a schedule S with the largest sum of profits.

Version 1: every profit equals 1, no two scheduling entries have the same i. We skip p from the description of scheduling entries.

Decision: select a scheduling entry (i, s, t).

Problem reduction: remove from J every (i', s', t') such that intervals $[s, t]$ and $[s', t']$ overlap. This creates J'.

Obtaining the solution for J from a solution for J': add (i, s, t).

To show that we made a good choice it suffices to show that for every schedule S we have $|S \cap J'| \geq |S| - 1$: by inductive hypothesis, we will find a schedule of size at least $|S| - 1$ for J', then we will the entry we selected already, and our schedule will have size at least $|S|$.

Good choice: select (i, s, t) with the minimum t.

The claim that (i, s, t) is a good choice is equivalent to "in every schedule S there is at most one entry in conflict with (i, s, t).

Because J contains only one scheduling entries with i, the conflict can arise only if $[s, t]$ overlaps $[s', t']$. Because $t' \geq t$, the overlap implies that $s' \leq t$. Suppose there is another entry in S that is in conflict with (i, s, t), say (i'', s'', t''). Then $s', s'' \leq t \leq t', t''$), so (i', s', t') is in conflict with (i'', s'', t''), a contradiction, because they are in the same schedule.

Conclusion: our Good Choice, Problem Reduction and "Obtaining the solution" define a greedy algorithm that solves exactly Version 1.

Version 2: every profit is 1, but we allow multiple entries with the same i.

We use the same algorithm —: the choice, the problem reduction etc.

Now it is not true that (i, s, t) with minimum t can be in conflict with only one scheduling entry from a valid schedule S. Our prove shows that there can be only one conflict caused by overlapping time interval, but there can be another, cause by $i = i'$.

But there can be no other conflicts. Thus when we reduce the problem to J', we secure one scheduling entry, while the size of the remaining optimum schedule goes down by at most 2. This proves that we will obtain a schedule S with $|S| \geq |S*|/2$, where $S*$ is the best schedule.

Version 3: we have variable profits, and only one scheduling entry for each i.

Decision: use some entry (i, s, t, p) to reduce the problem.

Problem reduction: consider every entry (i', s', t', p') that is in conflict with (i, s, t, p); if $p' \leq p$, remove that entry, otherwise, replace it with $(i', s', t', p' - p)$. This creates a reduced instance J'.

Obtaining the solution for J from a solution S' for J': if S' contains no entries in conflict with (i, s, t, p), add (i, s, t, p). Otherwise, for every such entry (i', s', t', p') replace the profit with $p + p'$.

How it works: if (i, s, t, p) is in conflict with k entries from the optimum schedule, we decrease the profit of that schedule by at most kp, and later, when we modify the solution, we increase by p.

We use the same choice as before: (i, s, t, p) with the minimum t. We get $k = 1$ (as we have shown for Version 1), so we get the maximum profit.

Version 4: variable profits, and multimple scheduling entries for each i.

We can use the same solution as for Version 3, but now we have $k = 2$, (as we have shown for Version 2), so we will obtain schedule S with $p(S) \geq p(S*)/2$.

Remark 1: We basically have two algorithms, a greedy for Version 1 and 2, and a ‘‘close to greedy’’ for Version 3 and 4.

Remark 2: Both algorithms can be implemented in time $O(n \log n)$.