COMPSCI 242: Data Structures and Algorithms

Knapsack Problem

The 0/1 Knapsack Problem

Suppose we have n items with sizes $s_i, 1 \leq i \leq n$. We assume that each value is an integer greater than zero. We want to find a subset of items that fits the knapsack exactly.

Let $P(i, S)$ be the the problem with the first i items and a knapsack of size S, where $S \leq K$. The following shows a recursive solution.

Base case. $P(1, k)$, $0 \leq k \leq K$.

$k = 0$ Then there is always a solution by not choosing any item
$k > 0$ Then there is a solution iff size of the first item is k ($s_1 = k$)

Hypotheses. Assume that we know how to solve $P(n-1, k)$ for $0 \leq k \leq K$.

Inductive Step. We want to solve $P(n, K)$. There are two cases.

1. There is a solution with the first $n-1$ items. Then we don't choose the nth item and use the solution from $P(n-1, K)$.
2. There is no solution with the first $n-1$ items. The we choose the nth item with size s_n and check if there is solution for $P(n-1, K-s_n)$

There are nK possible subproblems. To implement the algorithm we can store all the known results in an $n \times K$ table. Each entry in this table is computed from two entries in the row above that entry. The algorithm is shown on the next page.

```
KNAPSACK(S, K)
//Input:    S[1..n]: an array containing the sizes of the items.
//          K: size of the knapsack.
//
//Output:   P[0..n,0..K]:
//          P[i,k].exist = true implies that there is a solution to the knapsack
//                         of size k with the first i items.
//          P[i,k].belong = true implies that the solution includes the ith item.
1.    P[0,0].exist ← TRUE
2.    for k ← 1 to K
3.    do    P[0,k].exist ← FALSE
      //No need to initialize P[i,0], for i > 0,
      //as it will be computed from P[0,0]
4.    for i ← 1 to n
5.    do    for k ← 0 to K
6.          do    P[i,k].exist ← false //default value
                  // check if there is a solution with the first i-1 items
7.                if P[i-1,k].exist
8.                then  P[i,k].exist ← TRUE
9.                      P[i,k].belong ← FALSE
10.               else if (k-S[i])≥ 0
                  // there is no solution with the first i-1 items
                  //check if there is one if we choose the ith item
11.               then  if P[i-1,k-S[i]].exist
12.                     then  P[i,k].exist ← TRUE
13.                           P[i,k].belong ← TRUE
```

The worst-case run time is $\Theta(nK)$ and the space utilization is $\Theta(nK)$.

Some questions to ponder?

- How do we recover the actual solution from the table?

  The following code does it in $\Theta(n + K)$ time. It basically traces back the belong flag in the table.

```
1. if P[n,K].exist
2. then  x ← K
3.       for i ← n downto 1
4.       do    if P[i,x].exist
5.             then  if P[i,x].belong
6.                   then  print i
7.                         x ← x - s[i]
```

- What are the characteristics of the solution obtained from the knapsack algorithm? *It chooses the solution with the smallest possible indices.*
- What if we prefer to take the ith item over $(i-1)$st if a solution exists using both instead of preferring the $(i-1)$st item (as our algorithm above does)? *Then we will choose the solution with the largest possible indices. It will also run slower since it will skip over earlier solutions to get to the one with items with the largest possible indices.*
- Can we improve the space utilization to $\Theta(n)$? *Left as an exercise for the reader.*
- What if we want to recover all possible solutions to the 0/1 knapsack problem? *Keep track of both flags: for the solution including the ith item as well as for the solution with the first $i-1$ items. Then while tracing back explore both possibilities.*
- What if we have unlimited quantities of each item? *Left as an exercise for the reader.*
- What if each item has an associated value and we want to obtain the maximum value among all ways of packing the knapsack? *Discussed in the next section.*
- What if we are allowed to take a fractional part of an item? What if we are allowed to take a fractional part and we want to maximize the value? *Left as an exercise for the reader.*

The Knapsack problem with values associated with items

Consider the knapsack problem with n items with sizes s_i, $1 \leq i \leq n$, and associated values v_i, $1 \leq i \leq n$. We assume that each value is an integer greater than zero. Among all the candidate solutions to the knapsack problem, we want to find the one with the maximal total value of the items. A candidate solution is a subset of items that fits the knapsack exactly.

The algorithm is a modification of the original algorithm for the knapsack problem. In the two-dimensional matrix P, we will also keep track of the total value of a given solution in addition to the *belong* and *exist* flags. Thus $P[i,k].value$ would be the maximal value among all exact solutions to the knapsack of size k using the first i items. If there is no solution to the $P[i,k]$ knapsack problem, then $P[i,k].value$ value will be zero.

In order to compute the maximal value, we must examine the following cases:

1. If there is a solution with the first $i-1$ items, then there are two sub cases:
   (a) If there is also a solution using the ith item, then we must check to see which one has higher value.
   (b) If there is no solution using the ith item, then we simply use the value of the solution using the first $i-1$ items as the best value so far.
2. If there is no solution with the first $i-1$ items, then there are again two sub cases.
   (a) If there is a solution using the ith item, then we simply use its value as the best so far.
   (b) If there is no solution using the ith item, then there is no solution for the $P[i,k]$ knapsack problem.

The complete algorithm is on the next page.

```
KNAPSACK(S, K)
//Input:    S[1..n]: an array containing the sizes of the items.
//          V[1..n]: an array containing the values associated with the items.
//          K: size of the knapsack.
//
//Output:   P[0..n,0..K]:
//          P[i,k].exist = true implies that there is a solution to the knapsack
//                         of size k with the first i items.
//          P[i,k].belong = true implies that the solution includes the ith item.
//          P[i,k].value = the maximal value over all possible solutions
//                         to the knapsack of size k with the first i items.
1.    P[0,0].exist ← TRUE
2.    P[0,0].value ← 0
3.    for k ← 1 to K
4.    do    P[0,k].exist ← FALSE

5.    for i ← 1 to n
6.    do    for k ← 0 to K
7.          do    P[i,k].exist ← false //default value
8.                P[i,k].value ← 0 // default value
                  // check if there is a solution with the first i-1 items
9.                if P[i-1,k].exist
                  //There is a solution with the first i-1 items. Assume
                  //this is the better solution for now and we will update
                  //this information if it is not
10.               then  P[i,k].exist ← TRUE
11.                     P[i,k].belong ← FALSE
12.                           P[i,k].value ← P[i-1,k].value
                        // Now check if there is a better solution by choosing the
                        // ith item to be in the knapsack
13.                     if (k-S[i]) ≥ 0
14.                     then  if P[i-1,k-S[i]].exist
15.                           then  if ((P[i-1,k-S[i]].value + V[i]) > P[i-1,k].value)
16.                                 then  P[i,k].belong ← TRUE
17.                                       P[i,k].value ← P[i-1,k-S[i]].value+V[i]
                  //there is no solution with the first i-1 items
                  //check if there is one if we choose the ith item
18.               else  if (k-S[i]) ≥ 0
19.                     then  if P[i-1,k-S[i]].exist
20.                           then  P[i,k].exist ← TRUE
21.                                 P[i,k].belong ← TRUE
22.                                 P[i,k].value ← P[i-1,k-S[i]].value+V[i]
```