Assignment 9 Solution

Question 1 (10pt) Consider the following variant of the Knapsack problem. The input consists of

- a set of items with associated weights and values, just as before:

$$S = \{(w_1, v_1), (w_2, v_2), \dots, (w_n, v_n)\},$$

- a target value V,
- an upper bound W,
- and a "relax" factor ϵ.

Furthermore, the set S is guaranteed to contain a subset of items whose total weight is $\le W$ and whose total value is *exactly* V. The problem is to compute a subset of S whose total value is *at least* V, and whose total weight is $\le (1+\epsilon)W$ (so it can be a bit more than W).

Give a dynamic programming algorithm for solving this problem. Your algorithm must run in time polynomial in n and $\frac{1}{\epsilon}$. Prove the correctness of your algorithm and analyze its running time.

Solution We follow the approximation algorithm for Knapsack given in lecture, and consider the following "scaled-down" weights:

$$w'_i = \lfloor \frac{w_i}{b(\epsilon)} \rfloor$$

and $W' = \lceil \frac{W}{b(\epsilon)} \rceil$, for some parameter $b(\epsilon)$ to be determined below. The idea is to run the first dynamic programming algorithm for Knapsack on this new input. (That is to run the algorithm where subproblems are defined by i—which specifies the set of items $\{1, 2, \dots, i\}$, and T: the upper bound on the total weight.) We prove that this algorithm returns a subset of items with total value $\ge V$ and total weight $\le (1+\epsilon)W$.

Note that the algorithm is optimal for the new weight function. Note also that if a subset of S has total weight at most W, then its total new weights is at most W'. To see this, suppose that $\{j_1, j_2, \dots, j_\ell\}$ is a subset of items with total weight $\le W$:

$$w_{j_1} + w_{j_2} + \dots + w_{j_\ell} \le W.$$

Then, because $w'_i = \lfloor \frac{w_i}{b(\epsilon)} \rfloor \le \frac{w_i}{b(\epsilon)}$:

$$\sum_{t=1}^{\ell} w'_{j_t} \le \sum_{t=1}^{\ell} \frac{w_i}{b(\epsilon)} = \frac{\sum_{t=1}^{\ell} w_{j_t}}{b(\epsilon)} \le \frac{W}{b(\epsilon)} \le \lceil \frac{W}{b(\epsilon)} \rceil = W'$$

As a result, the output of the algorithm has total value at least V. It remains to show that, for the appropriate choice of the parameter $b(\epsilon)$, the total weight of the output is at most $\le (1+\epsilon)W$. So let $\{i_1, i_2, \dots, i_k\}$ denote the output of the algorithm. We know that

$$\sum_{t=1}^{k} w'_{i_t} \le W'$$

(by the correctness of the algorithm for Knapsack with the new weight function). The total weight of our output is

$$\sum_{t=1}^{k} w_{i_t} < \sum_{t=1}^{k}(1 + w'_{i_t})b(\epsilon)$$

$$= (k + \sum_{t=1}^{k} w'_{i_t})b(\epsilon)$$

$$\leq (k + W')b(\epsilon)$$

We also know that $W' < 1 + \frac{W}{b(\epsilon)}$, i.e., $(W' - 1)b(\epsilon) < W$. So rewrite $(k + W')b(\epsilon)$ as

$$(k + 1 + (W' - 1))b(\epsilon)$$

Then we want to guarantee that $k + 1 \leq \epsilon(W' - 1)$, because this will give us

$$\sum_{t=1}^{k} w_{i_t} < (1 + \epsilon)(W' - 1)b(\epsilon) < (1 + \epsilon)W$$

as desired.

Thus we will have $b(\epsilon)$ such that $n+1 \leq \epsilon(W' - 1)$. This is guaranteed by taking $b(\epsilon)$ such that $n + 1 \leq \epsilon(\frac{W}{b(\epsilon)} - 1)$ (because $\frac{W}{b(\epsilon)} \leq W'$). That is,

$$b(\epsilon) \leq \frac{W}{\frac{n+1}{\epsilon} + 1}$$

Finally, the running time of our algorithm is

$$\mathcal{O}(nW') = \mathcal{O}(n\frac{W}{b(\epsilon)})$$

So we want to make $b(\epsilon)$ as large as possible. In short, we will take $b(\epsilon) = \frac{W}{\frac{n+1}{\epsilon}+1}$. With this setting, the running time of the algorithm is

$$\mathcal{O}(n(\frac{n+1}{\epsilon} + 1))$$

which is a polynomial in n and $\frac{1}{\epsilon}$.

Question 2 (10pt) Your friends are looking at n consecutive days of a given stock, at some point in the past. The days are numbered $1, 2, \ldots, n$. For each day i they have a price $p(i)$ per share for the stock on that day.

For a certain (possibly large) integer k your friends want to know what is the best return of a so-called *k-shot strategy*. Here a k-shot strategy is a collection of m pairs of days

$$(b_1, s_1), (b_2, s_2), \ldots, (b_m, s_m)$$

for some $m \leq k$ and $b_1 < s_1 < b_2 < s_2 < \ldots < b_m < s_m$. This can be viewed as a set of at most k non-overlapping intervals, during each of which your friends buy 1,000 shares of the stock (on day

b_t) and then sell it (on day s_t). The return of such a strategy is simply the profit of the transaction, i.e.,

$$1,000 \sum_{t=1}^{m} (p(s_t) - p(b_t))$$

You are asked to design an efficient algorithm to determine the best k-shot strategy.

Formally, the input to your algorithm consists of

- positive integers $p(1), p(2), \ldots, p(n)$,
- a positive integer $k \leq n/2$

The output is a sequence of m pairs

$$(b_1, s_1), (b_2, s_2), \ldots, (b_m, s_m)$$

as above, for some $m \leq k$, with maximum possible return.

Your algorithm must run in time polynomial in n, k. Analyze its running time.

Solution First, let $P[i, j]$ denote the profit from buying on day i and selling on day j:

$$P[i, j] = 1000(p(j) - p(i))$$

Let $Q[i, j]$ denote the best profit from a single transaction (one buy then one sell) during the period from day i to day j (inclusive). We can build up an $n \times n$ table Q in time $\mathcal{O}(n^2)$ by a dynamic programming algorithm using the following formulas:

$$Q[i, i+1] = 1,000(p(i+1) - p(i))$$

and for $j - i \geq 2$:

$$Q[i, j] = max\{1000(p(j) - p(i)), Q[i+1, j], Q[i, j-1]\}$$

The program for computing Q: In the main for-loop (line 3) we run over all difference $\ell = j - i$.

1. Let Q be an $n \times n$ array
2. for i from 1 to $n - 1$ do $Q[i, i+1] \leftarrow 1000(p(i+1) - p(i))$ end for
3. for ℓ from 2 to $n - 1$ do
4.   for i from 1 to $n - \ell$ do
5. $j \leftarrow i + \ell$
6. $Q[i, j] \leftarrow max\{1000(p(j) - p(i)), Q[i+1, j], Q[i, j-1]\}$
7.   end for
8. end for

Let $M[m, d]$ denote the maximum return obtained by an m-shot strategy on days $1, 2, \ldots, d$, for $1 \leq d \leq n$ and $1 \leq m \leq k$. Then we have

$$M[1, d] = Q[1, d]$$

and for $1 \leq m \leq k - 1$:

$$M[m + 1, d] = max\{M[m, d], max_{1 \leq i < j \leq d}\{Q[i, j] + M[m, i - 1]\}\}$$

This recurrence comes from the fact that the optimal $(m + 1)$-shot trategy is either an m-shot strategy, or an m-shot strategy together with one more transaction (i, j). (Here $M[m, 0] = 0$.)

Program for computing M:

1. Let M be an $k \times n$ array
2. $M[1, 1] \leftarrow 0$
3. for d from 2 to n do $M[1, d] \leftarrow Q[1, d]$ end for
4. for m from 1 to k do $M[m, 0] \leftarrow 0$
5. for m from 1 to $k - 1$ do
6.   for d from 1 to n do
7. $M[m + 1, d] \leftarrow M[m, d]$
8.   for i from 1 to $d - 1$ do
9.   for j from $i + 1$ to d do
10.   if $M[m + 1, d] < Q[i, j] + M[m, i - 1]$
11. $M[m + 1, d] \leftarrow Q[i, j] + M[m, i - 1]$
12.   end if
13.   end for
14.   end for
15.   end for
16. end for

Program for computing the best k-shot strategy: To compute the best k-shot strategy we trace the computation of $M[k, n]$ to find out (at most) k pairs (b_i, s_i) in the strategy. Initialize $d = n$ and $m = k - 1$, at each step, if $M[m + 1, n] = M[m, n]$ then decrease m by 1. Otherwise find the pair (i, j) such that $M[m + 1, n] = M[m, i - 1] + Q[i, j]$. Add this pair to the solution. Then set $m \leftarrow m - 1$ and $d \leftarrow i - 1$, and continue.

1. P: empty sequence (this is out solution)

2. $d \leftarrow n$, $m \leftarrow k-1$

3. while $m > 0$ do

4.   if $M[m+1,d] = M[m,d]$ then $m \leftarrow m-1$

5.   else

6.   for i from 1 to $d-1$ do

7.   for j from $i+1$ to d do

8.   if $M[m+1,d] = Q[i,j] + M[m,i-1]$

9.   add (i,j) to P

10. $m \leftarrow m-1$, $d \leftarrow i-1$

11.   end if

12.   end for

13.   end for

14.   end if

15. end while

16. add $Q[1,d]$ to P

Analysis: Computing Q takes time $\mathcal{O}(n^2)$. Computing M takes time $\mathcal{O}(kn^3)$ (the for-loops on lines 6,8,9 have at most n loops each). Computing P from M takes time $\mathcal{O}(kn^2)$. So, overall, the running time is $\mathcal{O}(kn^3)$.