USACO NOV10 Problem 'feed' Analysis

by Jelle van den Hooff

Buying Feed is, at its core, a dynamic programming problem. The state is Farmer John's location and the amount of feed he currently has; for each state we are minimizing the number of cents spent. As we only want to process each store once, a natural way to store the location is by store number, with the stores sorted by x-coordinates.

The naive dynamic programming algorithm then gives an algorithm with complexity $O(NK^2)$, as at each state we can buy (in the worst case) any amount of feed. This is not fast enough, so we'll need an improvement somewhere. Because the cost of driving grows quadratically with the amount of feed FJ has in his truck, dynamic programming seems to be the only feasible option. There is also no obvious way to reduce the number of states substantially. That leaves only one choice: reducing the amount of calculations per state.

The easiest way to "discover" the optimization we are going to use is to experiment with the data and to work out some cases by hand. Let's say the we have a table with the best prices for every quantity to end up at store i without buying anything there yet, and store i has 2 pounds which it sells for 2 cents per pound.

Feed	Previous price	Price to end with 0 feed	1	2	3	4	Function	Last possible q
0	0	0	2	4	-	-	2q	2
1	3	-	3	5	7	-	2q+1	3
2	5	-	-	5	7	9	2q+1	4
3	8	-	-	-	8	10	2q+2	5
4	11	-	-	-	-	11	2q+3	6

Any previous quantity defines a linear function for possible new values, and all functions have the same slope. To get the minimum value of a bunch of these functions for a given q, we only need to pick the function with minimal constant term (or y-intercept). Now we have a way to solve the problem efficiently: for any given store, we loop through the current feed quantities, compute the function and its last valid q. We then store them as pairs in a priority queue. For each new q we retrieve the smallest valid function from the priority queue (removing smaller invalid functions) and evaluate it to get the best price. This gives a fast enough O(N K log K) solution.

Other data structures (such as a "min-queue") can be used in place of the priority queue to get O(NK). In addition, a binary search can get a different O(NK log K) solution; however, this did not quite run fast enough on the data. My implementation in C++:

```
#include <cstdio>
#include <algorithm>
#include <deque>
#include <iostream>
```

```
using namespace std;

typedef long long ll;

FILE *fin = freopen ("feed.in", "r", stdin),
     *fout = freopen ("feed.out", "w", stdout);

const int MAXK = 10005, MAXN = 505;

int K, E, N;

ll previous_bests[MAXK], bests[MAXK];

struct store {
  int X, F;
  ll C;
  bool operator< (const store& o) const {
    // smallest x first in stores
    return X < o.X;
  }
} stores[MAXN];

struct price {
  price (ll base_price, int valid_until) :
    base_price(base_price), valid_until (valid_until) {}
  ll base_price;
  int valid_until;
};

int main () {
  ios_base::sync_with_stdio (false);

  // read input
  cin >> K >> E >> N;
  for (int i = 0; i < N; i++)
    cin >> stores[i].X >> stores[i].F >> stores[i].C;

  // order stores by location
  sort(stores, stores + N);

  // dynamic programming: bests[q] stores best price
  //   for q pounds of feed at current location
  fill (previous_bests, previous_bests + K + 1, -1);
  previous_bests[0] = 0;
  int previous_x = 0;

  for (int i = 0; i < N; i++) {
    ll distance = stores[i].X - previous_x;
    fill(bests, bests + K + 1, -1);

    // prices stores functions with increasing base_prices
    //   and increasing valid_untils
    deque< price > prices;

    for (int q = 0; q <= K; q++) {
      if (previous_bests[q] != -1) {
        // compute new function
```

```
        ll base_price = (previous_bests[q] + distance * q * q)
          - q * stores[i].C;
        int valid_until = q + stores[i].F;

        // remove worse functions
        while (!prices.empty() && prices.back().base_price > base_price)
          prices.pop_back();
        prices.push_back(price(base_price, valid_until));
      }

      // remove invalid functions
      while (!prices.empty() && prices.front ().valid_until < q)
        prices.pop_front ();

      // compute best value
      if (!prices.empty ())
        bests[q] = prices.front().base_price + q * stores[i].C;
    }

    copy(bests, bests + K + 1, previous_bests);
    previous_x = stores[i].X;
  }

  cout << previous_bests[K] + K * K * (E - previous_x) << "\n";
}
```