USACO JAN11 Problem 'bottleneck' Analysis

by Michael Cohen

The first key fact towards solving this problem is that the optimal strategy is to simply greedily move as much as possible from the up to field 1. This can be implemented simply: at each step, loop through the fields in a bottom-up order (i.e. each field is visited before the field it leads to) and transfer the maximum amount possible out (this is either the capacity of the path or the number of cows on the field). It is clearly optimal: by induction on subtrees, the flow on each path is maximal at each step, and it is always better for cows to be closer to field 1. However, implementing this algorithm directly will yield a solution that takes time O(N*T), where T is the latest time asked for. This is far too slow; it is not even polynomial time (in the sense of the size of the input in bits, since in B bits a time of 2^B can be described).

The next important fact is that the flow through each path is monotonically decreasing over the course of the algorithm. This can again be proven by induction on subtrees: it is clearly true for fields with nothing coming in. For other fields, the amount coming in to the field is monotonically decreasing by the induction hypothesis. As long as this amount is greater than the capacity of the path, the outward flow will be constantly equal to the capacity; as soon as it is not, the number of cows standing on the field is monotonically decreasing (as until there are no cows remaining, cows will be taken off the field to fill up the remaining capacity); the flow chosen by the greedy algorithm is thus monotonically decreasing as well.

Now, consider the total number of cows that moves out of each field by any given time T. This is clearly less than the smaller of the total number of cows beginning on or moving into that field (i.e. the starting cows plus total amount moving out of the field's "children") and M*T, where M is the capacity of the outward path. Somewhat less obviously, it is in fact equal to the smaller of these two quantities. This is because the flow into the field over time is decreasing, so the total number of cows arrived is a concave function; thus, if the total number of cows beginning on or moving into that field is greater than or equal to M*T then at time S it will be at least M*S and there will always be M cows available to send; otherwise one can send M cows until there are no longer M available and then just send all cows possible (ending with no cows on the field). Interestingly, this algorithm really corresponds to doing one big time step of the greedy algorithm. This algorithm is much better, running in time O(N*K); it is a polynomial time algorithm. Many competitors found this solution, and in fact nobody did better (perhaps because this was only a 3 hour contest). However, more is needed to get full credit.

The trick is to consider the state of the greedy algorithm over time. As long as the flows on edges are not changing, the number of cows on each field is a linear function of time. The flow on any edge is its capacity until there are too few cows on the field and moving in to sustain this; at this point, all the cows on the field disappear after 1 timestep and after this all the flow out of the field is equal to the flow in. This occurs when the linear function for that field's is about to hit 0.

The algorithm tracks linear functions on nodes. It maintains a priority queue of events corresponding to when the current linear function for fields will hit 0. However, when these events occur, a large number of other fields' linear functions could potentially change. However,

once a path is running below capacity, that path is no constraint at all. Thus, one could simply consider all the paths running into the field that just ran out to be directly running into its parent from then on (and since the remaining cows will move to the parent in the next timestep, they can be moved as well). To do this efficiently, one uses a union-find data structure to store groups of nodes whose paths are currently connected to the same node; when a node runs out, its set is merged with that of its parent (and the corresponding linear functions are added); the new time to run out is pushed into the priority queue. Finally, before each event all remaining times that occur before that event are evaluated using the linear function for field 1. The union-find data structure can be implemented very easily in conjunction with existing data structures of the problem; only path compression is needed as this guarantees a log N runtime of the operations. Overall, the simulation part of the algorithm is O(N log N) from the priority queue and the union find, and sorting the list of times is O(K log K).

Below is Michael's solution. -Neal

```
#include <fstream>
#include <queue>
#include <algorithm>
#define endl '\n'
using namespace std;

int N, K;
int P[100000];
long long M[100000];
long long current[100000];
long long incoming[100000];
long long when[100000];
bool alive[100000];

struct event {
	int n;
	int t;
};

event T[10000];
long long ans[10000];

bool operator<(event a, event b) {
	return a.t > b.t;
}

int find(int i) {
	if (alive[i]) return i;
	else return (P[i] = find(P[i]));
}

int kill(int i) {
	int p = find(P[i]);
	current[p] += current[i];
	incoming[p] += incoming[i]-M[i];
	alive[i] = false;
}
```

```
int main()
{
	ifstream inp("bottleneck.in");
	ofstream outp("bottleneck.out");

	inp >> N >> K;
	alive[0] = true;
	for (int i = 1; i < N; i++) {
		inp >> P[i] >> current[i] >> M[i];
		P[i]--;
		alive[i] = true;
		incoming[P[i]] += M[i];
	}

	for (int i = 0; i < K; i++) {
		inp >> T[i].t;
		T[i].n = i;
	}
	sort(T, T+K);

	priority_queue<event> pq;
	for (int i = 1; i < N; i++) {
		if (M[i] > incoming[i]) {
			when[i] = current[i]/(M[i]-incoming[i]);
			event e = { i, when[i] };
			pq.push(e);
		}
	}

	int ctime = K-1;
	while (!pq.empty()) {
		event f = pq.top();
		pq.pop();

		while (ctime >= 0 && f.t >= T[ctime].t) {
			ans[T[ctime].n] = current[0]+T[ctime].t*incoming[0];
			ctime--;
		}

		if (!alive[f.n] || f.t != when[f.n]) continue;
		kill(f.n);
		int p = find(f.n);
		if (M[p] > incoming[p]) {
			when[p] = (current[p]-1)/(M[p]-incoming[p]);
			event e = { p, when[p] };
			pq.push(e);
		}
	}
	while (ctime >= 0) {
		ans[T[ctime].n] = current[0]+T[ctime].t*incoming[0];
		ctime--;
	}
	for (int i = 0; i < K; i++) outp << ans[i] << endl;
	return 0;
}
```