15-451/651 Algorithms, Spring 2018

Homework #4 **Due: March 6–8, 2018**

(25 pts) 1. **(Coffee, Anyone?)** The coffee chain ⋆⋆⋆$$$$ asks you to locate their $s \geq 1$ shops on the PA turnpike. You model the turnpike as a straight line. There are $n \geq s$ cities on the turnpike, which are located at positions $0 = a_1 < a_2 < \ldots < a_n$ on the line, with distances on the line modeling driving distances between the cities. You are allowed to locate the shops anywhere on the line, even on locations that are not cities. If you decide to locate the s shops at positions $b_1, b_2, \ldots, b_s$, the "pain" of this solution is the distance from an average city to its closest shop, i.e.,

$$\frac{1}{n}\sum_{i=1}^{n}\left(\min_{j=1}^{s}|a_i - b_j|\right).$$

Give a dynamic programming algorithm to find the solution (i.e., the location of shops) that incurs the least pain, and that runs in time $O(n^3)$. It should not depend on the values of a_i.

Hint: consider subproblems that correspond to a prefix $\{a_1, \ldots, a_i\}$ of the cities and a number of shops $s' \leq s$. One more useful fact is that we need only consider solutions that place shops at some subset of the cities $\{a_i\}$ (i.e., $\{b_j\} \subseteq \{a_i\}$); this can be seen most easily in the case of $s = 1$ where if the shop is between a_i and a_{i+1}, we can simply move it in whichever direction has more cities, or to either one if there is a tie. You may use this fact without proof, but convince yourself that it is true.

Solution: First of all, let's try to minimize the sum of the distances rather than the average, since this is just scaling by a factor of n that does not depend on the solution. We will call this the "total cost".

Let $T(i, s')$ be the total cost of the best solution just for the interval $\{a_1, \ldots, a_i\}$ when we open s' shops within these i locations, and allocate all these i locations to these shops. If we compute this for all $1 \leq i \leq n, 1 \leq s' \leq s$, we will be done, since we need to return $T(n, s)$.

To compute $T(i, s')$: since each gate will go to their nearest shop, the first few gates go to the first shop, the next few to the next shop, and the last few to the last shop. So consider at the "rightmost" of these s' locations to be opened. The gates which go to this shop will be $\{j^* + 1, \ldots, i\}$ for some value of j^*, and the other gates $\{1, \ldots, j^*\}$ will be served (optimally) by the other $s' - 1$ shops. Hence,

$$T(i, s') = \Big(T(j^*, s' - 1) + C(j^* + 1, i)\Big),$$

where $C(j+1, i)$ is the cost of opening *one* shop in the interval $\{j+1, \ldots, i\}$ and serving the gates optimally. (Of course we don't know this unknown "breakpoint" j^*.) Moreover, if we choose some other "breakpoint" j, open some solution of with $s' - 1$ locations in

$\{1, \ldots, j\}$ and a single shop in $\{j+1, \ldots, i\}$ on the right, this will cost more than the optimal solution (and also cost at most $T(j, s'-1) + C(j+1, i)$). So we

$$T(i, s') = \min_{j<i} \Big(T(j, s'-1) + C(\{j+1, i\}) \Big),$$

And

$$T(i, 1) = C(1, i),$$

and also

$$T(i, j) = 0 \text{ for } j \geq i.$$

For each $p \leq q$, the quantity $C(p, q)$ can be computed in time $O(q-p+1)^2$ easily: try all $q-p+1$ locations, compute their costs in time $O(q-p+1)$, and take the best one. However, it can also be done in linear time $O(q-p+1)$. Approach #1: we start off by computing the sum of distances from a_p to all $a_p, \ldots, a_q$. Then for each integer $r \in [p+1, q]$, we can compute the sum of distances of a_r to all $a_p, \ldots, a_q$ by taking this same quantity for a_{r-1}, adding $(r-p)|a_r - a_{r-1}|$ and subtract $(q-r+1)|a_r - a_{r-1}|$. Approach #2: the optimal single location will be at the median of $a_p, \ldots, a_q$. (Why?) Hence the computation of all the $C(\cdot, \cdot)$ values can be done in $O(n^3)$ time.

Now for the $T(\cdot, \cdot)$ values: there are $O(ns)$ problems (one for each i, s'). Each considers at most n subproblems within it, and hence takes $O(n)$ time to compute the minimum (assuming the C's are precomputed). That takes a total of $O(n^2 s)$ time more. Since $s \leq n$ the total time is $O(n^3)$.

Exercise: since $C(p, q)$ is the total distance from the median of $a_p, \ldots, a_q$ to these values, how can you compute the entire matrix $C(p, q)$ faster than $O(n^3)$?

(25 pts) 2. **(ClubTown.)** The n residents of ClubTown like belonging to clubs, but no one wants to be the chairperson of these clubs. Say there are m clubs, and S_i is the set of people in club i. For each non-empty club i, one of the people $j \in S_i$ must be the chairperson. Since people would rather not be chair, they want the work of being chairperson to be divided as fairly as possible. Your task in this problem is to give an algorithm to do this efficiently.

The fairness criterion is the following: Say that person p is in some k of the sets, which have sizes $n_1, n_2, \ldots, n_k$, respectively. Person p should really have to be the chair for $\frac{1}{n_1} + \frac{1}{n_2} + \cdots + \frac{1}{n_k}$ clubs, because this is the amount of "club-ness" resource that this person effectively uses. Of course this number may not be an integer, so let's round it up to an integer. This quantity $\lceil \frac{1}{n_1} + \frac{1}{n_2} + \cdots + \frac{1}{n_k} \rceil$ is called their *fair cost*. A *fair solution* is an assignment of chairpersons to clubs so that that each person is chair for no more clubs than their fair cost.

> Say the first club has Alice and Can, and the second has Alice, Tanvi, and Shalom. Alice's fair cost would be $\lceil 1/2 + 1/3 \rceil = 1$. So Alice chairing both clubs would not be fair. Any solution except that one is fair.

Note that it is not clear there even *exists* a fair solution. Give a polynomial-time algorithm that, given $S_1, S_2, \ldots, S_m$, always finds a fair solution. (This will also show that there always exists a fair solution.)

Solution: We will solve this problem using network flow, so the first step is to construct a flow-graph. We will have one vertex in the graph for each resident of ClubTown, and one for each club, as well as a source s, and sink t.

The source s will have an edge to each club with capacity 1. Each resident will have an edge to the sink t with capacity equal to the resident's fair cost ($\lceil \frac{1}{n_1} + ... + \frac{1}{n_k} \rceil$). For each resident p, who is a member of clubs $c_1, ..., c_k$, we will have an edge from c_i to p with capacity 1.

Claim: The max flow, F, in this graph is exactly m.

To prove the claim, we need to show that $F \leq m$ and $F \geq m$. Since F is the maximum flow pushed, we only need to find a cut whose capacity is m to prove that $F \leq m$, so consider the cut that has $S = \{s\}$ and $T = V \backslash \{s\}$. The capacity of this cut is m.

To show that $F \geq m$, we need to exhibit a flow that has value m, and it will follow that F is at least m. To do this, for each club c_i, which has n_i members, push 1 flow to c_i, and then for each member of c_i, push $\frac{1}{n_i}$ to the member into the sink. This satisfies the capacity constraints because for each resident p, the flow we push is the sum of $\frac{1}{n_i}$ for each club the resident is in, which is the fair cost without the ceiling.

Alternatively, a more complicated way to do this is to show that the min cut has capacity m. It helps for this if you make the c_i to p_j edges have a large capacity (say, $> m$), because then the min cut can't contain any $c_i \to p_j$ edges. (Note that this doesn't change the solution because the flow out of any c_i is still at most 1). The cut that includes all S to c_i edges has capacity m, and you can then show that any cut that includes a combination of S to c_i and c_i to T edges must have capacity $\geq m$ (this argument is like the Hall's theorem proof from recitation).

Now, because the max flow is m, and all edge weights are integral, Ford-Fulkerson runs in polynomial time and returns an integral flow whose value is m. For every person, p, if flow is sent from p to club c_i, then we will assign p to be the chair of club c_i. The number of clubs p is a chair for is equal to the flow pushed through p, which is less than p's fair cost. So, we will be able to assign all m clubs to a single resident in a fair way.

(25 pts) 3. **(Rental Carts.)** You start a new cart rental company Hürtz. You will locate cart rental offices in some subset S of n cities, to produce the most profit. Profit is the revenue minus cost.

~~The ith cell tower will cost a_i dollars to build,~~ Each city $i \in \{1, \ldots, n\}$ has an associated cost $a_i \geq 0$. Building a cart rental office at city i costs $a_i \geq 0$ dollars, so the total cost of building offices at some subset S of the n cities is $\sum_{i \in S} a_i$.

You get revenue from these rental offices too. People drive carts only between distinct cities. There are m of the $\binom{n}{2}$ pairs of cities such that people drive carts between them, and you can get some revenue for that. So, as input, we are also given m triples (x_i, y_i, r_i), such that the company gets revenue of $r_i \geq 0$ dollars if it builds on both locations x_i and y_i.

E.g., if the costs were $7, 9, 12, 14$ and the triples were $\{(1,2,14),(1,3,11),(2,3,22),(1,4,6)\}$, then building an office at just location 1 gets profit -7, building at $\{1,2\}$ gets profit $14-(7+9)=-2$, building at $\{1,2,3\}$ gets profit $(14+11+22)-(7+9+12)=19$, etc.

Give an efficient algorithm to find a subset S of these n locations the company should build in order to earn the most profit. (Hint: Use maximum s-t flow.)

Solution: Define $R = \sum_{i=1}^{m} r_i$ to be the total possible revenue in the system. If we build a set B of towers, and this satisfies set T of tuples (leaving $U = [1..m] \setminus T$ tuples unsatisfied), the profit is

$$\sum_{i \in T} r_i - \sum_{j \in B} c_j.$$

Define the *lost revenue* to be $R - \sum_{i \in T} r_i = \sum_{i \in U} r_i$. Maximizing profit is same as maximizing

$$\sum_{i \in T} r_i - \sum_{j \in B} c_j - R$$

(since we just subtracted off R, which is fixed for this instance). This is same as *minimizing*

$$\begin{aligned} &- \sum_{i \in T} r_i + \sum_{j \in B} c_j + R \\ &= \sum_{j \in B} c_j + \sum_{i \in U} r_j = \text{our cost } + \text{ lost revenue.} \end{aligned}$$

Now we'll solve this problem of minimizing our cost plus lost revenue using s-t min-cut.

Make a flow network with source s, sink t. Make a node P_j for each cell tower j, and one node Q_i for each tuple i. Add directed arcs (s, P_j) with capacity c_j (equal to cost), and arcs (Q_i, t) with capacity r_i (equal to revenue). Now if tuple i contains towers $j1, j2$ then add arcs $(P_{j1}, Q_i), (P_{j2}, Q_i)$ with infinite capacity.

Claim: The finite capacity s-t cuts in this network are in bijection with feasible solutions to the original problem, with the capacity of the s-t cut equal (cost plus lost revenue).

Proof: Take any solution building towers in B and *not* getting revenue for tuples T. Cut the edges for these towers and these tuples. Any s-t path contains a tower edge, an infinity edge, and a tuple edge: if the tuple edge is not cut then we must be getting revenue for that tuple, then the tower must be built, so we must have cut that edge. Conversely, take any finite capacity s,t-cut: it only cuts the tower/tuple edges. Build the towers for tower edges cut, and claim revenue for tuple edges *not* cut. This is feasible, because if the tuple edge is not cut, both tower edges must be cut (and those towers bought). Also, the cut capacity equals the (cost plus lost revenue) in both directions. ■

By this claim, finding the minimum s,t-cut will find the solution with minimum cost plus lost revenue; by the discussion above this maximizes profit (i.e., gained revenue minus cost).

(25 pts) 4. **(Delightful Dominos.)** In this programming problem you're given a rectangular board of square cells. Some of the cells are blocked. You are to compute the maximum number of dominos (1×2 tiles) that can be placed on the board. Each domino will be in either a vertical or horizontal orientation, and is placed on two neighboring squares, neither of which must be blocked. The time limit is 10 seconds. The your program should be called `dominos.c`, or the analogous name based on what language you use.

Input: The first line contains two space-separated integers: r and c. The next r lines are strings of length c comprised of the characters "`.`" and "`x`". The "`x`" characters denote cells of the board that are blocked. $1 \leq r, c \leq 30$.

Output: Print the maximum number of dominos that can be placed on the board satisfying these constraints. The format of the output is illustrated below.

Input 1:

```
2 3
...
...
```

Output 1:

```
3 dominos
```

Input 2:

```
5 4
.xx.
xx.x
x...
xx.x
.xx.
```

Output 2:

```
1 domino
```

Input 3:

```
2 5
..xx.
.....
```

Output 3:

```
4 dominos
```