CSC 373 - Algorithm Design, Analysis, and Complexity **Summer 2014**

Assignment 2: Due Wednesday June 25, Midnight

Please follow the instructions provided on the course website to submit your assignment. You may submit the assignments in pairs. Also, if you use **any** sources (textbooks, online notes, friends) please cite them for your own safety.
You can use those data-structures and algorithms discussed in CSC263 (e.g. merge-sort, heaps, etc.) and in the lectures by stating their name. You do not need to provide any explanation or pseudo-code for their implementation. You can also use their running time without proving them: for example, if you are using the merge-sort in your algorithm you can simply state that merge-sorts running time is $\mathcal{O}(n \log n)$.

Every time you are asked to design an efficient algorithm, you should provide both a short high level explanation of how your algorithms works in plain English, and the pseudo-code of your algorithm similar to what we've seen in class. State the running time of your algorithm with a brief argument supporting your claim. You must prove that your algorithm finds an optimal solution!

1 When I was your age...

Remember when we were kids, we played with marbles?[1] We used to also collect them and exchange them for different types. Clearly each type had a different value, you could trade two of these "ugly" ones for this pretty one.

Suppose you have a group of n friends. Each one of them collects one specific type of marbles. Suppose also that for each pair of friends $i \neq j$, you've computed a ratio r_{ij}: one marble from i gives you r_{ij} marbles from j. Notice that r_{ij} can be a fraction; that is if $r_{ij} = \frac{1}{2}$ then if you buy two marbles from friend i, you can exchange them for one marble from friend j.

A marbling cycle is a fixed order $x_1, x_2, ..., x_k$ where you successively exchange marbles from friend x_i with marbles from friend x_{i+1} for all $1 \leq i < k$, and you exchange marbles from x_k with marbles from x_1. More precisely, you buy your favourite type of marbles from the friend who collects them (that's x_1). You then perform a sequence of exchanges from x_1 to x_2, x_2 to x_3 etc until x_k to x_1. And you hope that by the end of your marbling cycle, you end up with more marbles of your favourite type!

This only happens if the product of the ratios along this marbling cycle is above 1! Give a poly-time algorithm that checks whether you will be able to end up with more marbles at the end of your cycle.

Solution:

A profitable marbling cycle occurs when the product of the ratios along the cycle is above 1. One way to deal with products in cases like this is to take logarithms, which causes them to become sums. So we build the following graph $G(V, E)$ where each friend x_i is represented as a node $v_i \in V$, and a directed edge (i, j) for each pair of friends. We assign a weight of $-\log r_{ij}$ for every edge (i, j).

[1] No? Just me? Maybe I'm too old ...

Now a marbling cycle $\mathcal{C}$ in G is good if an only if:

$$\prod_{(i,j)\in\mathcal{C}} r_{ij} > 1$$

In other words, taking logarithms of boths sides, we get:

$$\sum_{(i,j)\in\mathcal{C}} \log r_{ij} > 0$$

or

$$\sum_{(i,j)\in\mathcal{C}} -\log r_{ij} < 0$$

Therefore a marbling cycle $\mathcal{C}$ in G is good if and only if there exists a negative cycle in G. It thus suffices to find a negative cycle in G. We could either use the Bellman-Ford algorithm or Floyd-Warshall algorithm to detect such a cycle.

Recall the Bellman-Ford Algorithm, we showed in class that $n-1$ relaxation rounds were enough to compute the single source shortest paths in G given a source vertex s. That is, if for any $v \in V$, $d(v)$changes in the n^{th} relaxation round, then G has a negative cycle. The algorithm however takes s as a starting point, since we don't have a specified source, we could run the algorithm n times where $s = u_i$ for $1 \leq i \leq n$. So the algorithm becomes:

Algorithm 1 The Modified Bellman-Ford Algorithm

```
1:  for w ∈ V do    s ← w
3:      set d(s) = 0 and d(v) = ∞ for all v ≠ s
4:      set pred(v) = NULL for all v
5:      for i = 1 ... n-1 do                          ▷ Run Bellman-Ford as usual
6:          for every edge (u, v) in G do
7:              if d(v) > d(u) + w(u, v) then
8:                  d(v) = d(u) + w(u, v)
9:                  pred(v) = u
10:             end if
11:         end for
12:     end for
13:     for every edge (u, v) in G do                 ▷ Run one more relaxation round
14:         if d(v) > d(u) + w(u, v) then
15:             return True                           ▷ There is a negative cycle
16:         end if
17:     end for
18:     return False  =0
```

A more elegant way to solve the problem is to use Floyd-Warshall All Pairs Shortest Paths algorithm. When doing APSP using Floyd-Warshall's algorithm, we always set the distance from a vertex i to itself to be 0. Now if i is in a negative cycle, then there should be a path from i to i with length < 0. Therefore it suffices to run Floyd-Warshall's algorithm and then check the diagonal of our DP table. If there exists a vertex i such that $M[i, i, k] < 0$ for some k then G has a negative cycle.

Algorithm 2 APSP

```
for 1 ≤ i, j, ≤ n do
    M[i, j, 0] = w_ij
end for
for k = 1 ... n do
    for i = 1 ... n do
        for j = 1 ... n do
            M[i, j, k] = min{M[i, j, k − 1], M[i, k, k − 1] + M[k, j, k − 1]}
        end for
    end for
end for
for i = 1 ... n do
    if M[i, i, n] < 0 then
        return True                                   ▷ There is a negative cycle
    end if
end for
return False =0
```

If we use BF, we know that it takes $\mathcal{O}(mn)$ time, and the second loop takes $\mathcal{O}(m)$ time, so the running time of the algorithm is $\mathcal{O}(mn)$ time per vertex $s = u_i$. So in total we have a running time of $\mathcal{O}(mn^2)$. And FW takes $\mathcal{O}(n^3)$ time. Therefore both algorithms run in polynomial time.

2 Sucks to be broke!

You can assume you do not pay for the first move.

Summer is still here but you're broke already and are trying to save money for your next trip! You found a night job at a hospital: Saint Hell's Hospital (SHH). SHH operates two main clinics, one in Toronto (SHHT) and one in Ottawa (SHHO), and by law they have to always have k night guards in total between the two locations. They hired you to meet the law requirements, which makes your job more flexible (or not). For the n coming months, you have to work at either SHHT or SHHO and you have the freedom to chose the clinic you go to.

For every month i, you make T_i or O_i dollars depending where you work. However, if you decide to switch clinics from month i to month $i+1$, you incur moving expenses, a fixed price E for the move. For simplicity, suppose all your other expenses (rent, food...), except the move, are covered by some generous God. By the end of the n^{th} month, you want to maximize your savings.

Input : A list of values $T_1, T_2, ..., T_n, O_1, O_2, ..., O_n$ where T_i, O_i is your income on month i at $SHHT, SHHO$ respectively, and a value E representing the moving expense.
Output : A sequence S of n locations that maximizes $v(S)$, where location i represents the clinic you will work at during month i and $v(S)$ is the total value of your savings.

Devise an algorithm to solve this problem efficiently. Give a high-level description of how your algorithm works, state the time complexity of your algorithm, and prove that your algorithm is optimal.

Solution:

The first basic observation we make is that an optimal plan will either end in Toronto, or in Ottawa. If it ends in Toronto, we get paid T_n plus one of the following two quantities:

- The savings of the optimal plan on $n-1$ months, ending in Toronto **OR**
- The savings of the optimal plan on $n-1$ months, ending in Ottawa, **minus** the moving expense E.

An analogous observation holds if the optimal plan ends in Ottawa. Therefore if $OPT_T(j)$ denotes the maximum profit of a plan on months $1, ..., j$ ending in Toronto, and $OPT_O(j)$ denotes the maximum profit of a plan on months $1, ..., j$ ending in Ottawa, then:

$$OPT_T(n) = T_n + max(OPT_N(n-1), OPT_O(n-1) - E)$$
$$OPT_O(n) = O_n + max(OPT_O(n-1), OPT_T(n-1) - E)$$

We therefore translate this direct into the following algorithm:

Algorithm 3 Moving Plan

$OPT_T(1) = T$, $OPT_O(1) = O$
for i=1...n **do**
    We define two arrays aT and aO to trace back our locations
    We compute $OPT_T(i)$ first:
    if $OPT_T(i-1) > OPT_O(i-1) - E$ **then**
        $OPT_T(i) = T_i + OPT_T(i-1)$
        $aT[i] =$ "T''
    else
        $OPT_T(i) = T_i + OPT_O(i-1) - E$
        $aT[i] =$ "O''
    end if
    Similarly for $OPT_O(i)$:
    if $OPT_O(i-1) > OPT_T(i-1) - E$ **then**
        $OPT_O(i) = O_i + OPT_O(i-1)$
        $aO[i] =$ "O''
    else
        $OPT_O(i) = O_i + OPT_T(i-1) - E$
        $aO[i] =$ "T''
    end if
end for
if $OPT_T(n) > OPT_O(n)$ **then**
    $v_{max} = OPT_T(n)$
    $PickFrom = aT$ ▷ Which array to pick from
    $S \rightarrow$ "T'' ▷ We ended up in Toronto in the last month
else
    $v_{max} = OPT_O(n)$
    $PickFrom = aO$
    $S \rightarrow$ "O'' ▷ We ended up in Ottawa in the last month
end if
for i = n ... 2 **do**
    $S \rightarrow S \cup PickFrom[i]$
    if $PickFrom[i] ==$ "T'' **then**
        $PickFrom = aT$
    else
        $PickFrom = aO$
    end if
end for
return $\{v_{max}, S\}$

We now show that the algorithm returns the maximum saving we can make. We use induction on i. Let $M(i)$ denote the maximum profit we can make up to month i. Clearly $M(i)$ could either be $OPT_T(i)$ or $OPT_O(i)$.

For $i = 1$, we have $M(1) = \max(OPT_T(1), OPT_O(1))$ which is indeed what v_{max} computes, and thus $M(1)$ is optimal. Suppose $M(i)$ is optimal up to $i < k$, and let's consider $M(k)$.

We consider both possible cases. If $M(k) = OPT_T(k)$, then we ended up in Toronto at month k, and $OPT_T(k) = T_k + \max(OPT_T(k-1), OPT_O(k-1) - E)$. By induction hypothesis, we know that $OPT_T(k-1)$ and $OPT_O(k-1)$ are optimal. Since we are chosing the maximum of the two choices ($OPT_T(k-1)$ and $OPT_O(k-1) - E$), $OPT_T(k)$ is maximized and thus so is $M(k)$. A similar argument holds if $M(k) =$

$OPT_O(k)$. Finally since our algorithm picks the maximum of $\max(OPT_T(k), OPT_O(k))$, it follows that $M(k)$ is thus maximized at month k.

We loop n times, and each iteration takes constant time, so the algorithm takes $\mathcal{O}(n)$ time.

3 Piw Piw!

x_i is the number of infected patients you get to see and cure at minute i. Patients who showed up at minute i but were not cured die. You do not get to see them at minute $i+1$.

Working for SHH isn't so bad after all. It sure is quiet and boring at times and not much is happening on Reddit over night, but you manage to keep yourself busy. On a boring night however, you get a call from the emergency room. There was an outbreak and they need to administer the antidote to the affected patients. You were asked to come help [2].

You run upstairs where you are assigned a room, an antidote gun, and a list $\mathcal{L} = \{x_1, x_2, ..., x_n\}$ of number of patients per minute that you will get to *hopefully* save. This means, at minute i, you will have x_i infected patients and you want to save as many of them as possible. The gun you're given pumps the antidote from a heated container, and the whole process works as follows:

- If you let the antidote container heat for j minutes, then you get $f(j)$ dosages (shots) to cure $f(j)$ patients.
- If you use the gun at the i^{th} minute, and it has been j minutes since you last used it, then you will cure $\min(x_i, f(j))$ patients. Of course, if you use $\min(x_i, f(j))$ dosages, then the gun will be completely empty and you will need to wait for more dosages to be available.
- When you first got to the room, the gun was empty. This means, if you started using it at the j^{th} minute, then you will be able to cure up to $f(j)$ patients. In other words, if you first use the gun at minute j, then you will cure up to $f(j)$ patients.

Given the list $\mathcal{L} = \{x_1, x_2, ..., x_n\}$ and the function $f(\cdot)$, you want to select when you will use the antidote gun in order to cure as many patients as possible.

Construct an efficient algorithm that returns the maximum number of patients you can cure. Give a high-level description of how your algorithm works, state the time complexity of your algorithm, and prove that your algorithm is optimal.

Solution:

Let $OPT(j)$ be the maximum number of patients we can cure for the instance of the problem on $x_1, ..., x_j$. Clearly if the input ends at x_j, then there is no reason not to use the gun since you're you're not saving it for anything, so the choice is just when to last use the gun before step j. Therefore $OPT(j)$ is the best of these choices over all i:

$$OPT(j) = \max_{0 \le i < j} (OPT(i) + \min(x_j, f(j-i)))$$

[2] I don't know why they believe a night guard is capable of doing this...

where $OPT(0) = 0$. Essentially what we are doing is first saving the most patient we can save at minute j, that is $\min(x_j, f(j-i))$ where i is the last minute we used the gun before minute j. So then it suffices to select the i that maximizes $OPT(i) + \min(x_j, f(j-i))$. The full algorithm is then just:

Algorithm 4 SaveTheWorld

1: $OPT(0) = 0$
2: **for** j=1...n **do**
3: $\quad OPT(j) = \max_{0 \le i < j} (OPT(i) + \min(x_j, f(j-i)))$
4: **end for**
5: **return** $OPT(n)$

We now show that $OPT(n)$ as returned by the algorithm is the maximum number of patients we can save. At iteration n, let i be the index that satisfies $\max_{0 \le i < n} (OPT(i) + \min(x_n, f(n-i)))$.

Suppose $OPT(n)$ isn't the best we can do, and let O' be the solution returned by an optimal algorithm. Thus $O' > OPT(n)$. As we mentioned above, since the input ends at x_n, we have no reason not to use the gun since we're not saving it for anything. Therefore at $t = n$, for O', we saved $\min(x_n, f(n-k))$ for some $k \neq i$ (since $O' \neq OPT(n)$). And so $O' = \min(x_n, f(n-k))$ plus the best we can do at $t = k$.

However, since our recursive definition checks all possible indices and picks the maximum out of all possible choices, it must have encounted the scenario of activating the gun at $t = k$ as a possible solution. And thus $OPT(n)$ would have used the same gun activating schedule as O'. Therefore $OPT(n) = O'$.

Every iteration takes $\mathcal{O}(n)$, therefore the total run time of the algorithm is $\mathcal{O}(n^2)$.

4 Everybody deserves good health care!

You saved the most patients out of all the staff! [3] Unfortunately many patients passed away. There just weren't enough staff to help all the patients; so you decided to run for mayor elections again to change this situation. Everybody deserves good health services! This was your selling point at the town elections. You've been elected (Gee you're good at politics) and as a good politician *cough again* you want to improve everyone's access to hospitals and health services in general.

A good access to health services means two things: 1. A household should not travel further than s kilometers to get to a hospital, and 2. no hospital should be responsible for more than t households.

Before investing into building new hospitals, you want to check if the current setup of the town is good[4] (i.e. it meets the requirements above).

Input : A list A of n houses and a list B of m hospitals, where every house and hospital is represented by a pair (x, y) on the plane, and s and t.
Output : "Good" if each household has access to a hospital under the restrictions above, and "Eeek" otherwise.

Think of "Good" as an assignment of houses to hospitals that satisfies the constraints.

[3] Not bad for a night guard :D
[4] the breakout night was an exception..

Devise an algorithm to solve this problem. Give a high-level description, state the time complexity of your algorithm, and prove that it is optimal.

Solution:

We reduce this problem to the maximum flow problem. That is, we give a transformation that takes as input the houses, the hospitals and the distances from the houses to the hospitals, and produces a flow network (G, a, b, c)[5]. We construct the flow network (G, a, b, c) as follows: For every house i, we add a vertex u_i and for every hospital j we add a vertex v_j. We also add a source node a and a terminal node b to G. We add the following edges to the flow network:

- (a, u_i) with capacity $c(a, u_i) = 1$.
- (u_i, v_j) if house i is in the range of hospital j. Set the capacity to $c(u_i, v_j) = 1$.
- (v_j, b) with capacity $c(v_j, b) = t$.

It suffices to compute a maximum flow f on this network and check if every house has a flow to the terminal. Thus we claim that if $val(f) = n$, then every house can be assigned to a hospital while satisfying the conditions above. The algorithm is therefore:

Algorithm 5 HHAllocation

```
1: Construct (G, a, b, c) as described above
2: f = Ford-Fulkerson(G)
3: if val(f) = n then
4:     return Good
5: else
6:     return Eeek
7: end if
```

Claim : The algorithm returns "Good" if and only there is a valid assignment of houses to hospitals.

Proof. Suppose the algorithm returns "Good". Since $val(f) = n$, we know that every house is connected to some hospital in its range. This follows from the fact that an edge (u_i, v_j) exists iff house i is in the range of hospital j. Moreover $c(a, u_i) = 1$ and $val(f) = n$, therefore $f(a, u_i) = 1$ for all houses i, and by conservation of flow, $f(u_i, v_j) = 1$. Finally, we know that $c(v_j, b) = t$ and thus no hospital j sends more than t unit of flow to the terminal t. Again by conservation of flow, it follows that the incoming flow into every hospital does not exceed t. So every hospital is connected to at most t houses.

To prove the converse, we use the contrapositive: The algorithm returns "Eeek" $\implies$ there is no valid assignment of houses to hospitals.

Notice that if the algorithm returns "Eeek", then $val(f) < n$. This implies the existence of a vertex (house) u_i such that $c(a, u_i) = 1$ and $f(a, u_i) = 0$. By the correctness of Ford-Fulkerson, we know that f is maximized and thus there is no augmenting path in G. In particular, there is no augmenting path that sends a unit of flow down the edge (a, u_i). So $f(a, u_i) = 0$, and by conservation of flow, there is 0 flow from u_i to any v_j, and thus house i was not assignment to any hospital j. □

[5] I used s and t for the constraints in the question, so I will refer to the source as a, and the terminal as b.

This flow network has $|V| = n+m+2$ vertices and at most $|E| = n+mn+m$ edges, so G can be constructed in $\mathcal{O}(|V| + |E|)$ time. Using the Edmonds-Karp implementation of Ford-Fulkerson, we can compute the max-flow f in $\mathcal{O}(|V||E|^2)$ time, therefore the total run time of the algorithm is $\mathcal{O}(|V||E|^2)$.

5 Eeek!

It turned out the town needs one more hospital. You selected the location and hired all the necessary staff, n' people in total $p_1, p_2, ..., p'_n$. There are different tasks (k of them) to be done at the hospital, from surgery to mopping the floor etc. Suppose there are n tasks to be completed in a given day, and assume (clearly) that $n \geq k$ (might have to perform more than one surgery in a given day). Suppose that each task takes one hour to complete, and only one person can work on one instance of any task. Each person p_i works a total of h_i hours per day and can perform a subset S_i of the of task types.

Each job j_i is a pair (t_i, c_i) where t_i is the task type (surgery, etc.) and c_i is an integer representing the number of times we have to do that job in a given day.

Give an algorithm that checks whether it is possible to get all the jobs done in a given day. If it is possible, show how to extract the list of jobs each p_i will have to complete. Prove the correctness of your algorithm.

Example

Suppose the set of tasks is {mopping, surgery, reception work, checking blood pressure}, and suppose on a given day, the jobs are (mopping, 5), (surgery, 10), (checking blood pressure, 6) and (reception work, 2).

Now each staff is represented with (p_i, h_i, S_i) and suppose we have 4 people with:

(Alice, 6, (mopping, reception))
(Bob, 3, (mopping, checking blood pressure, reception))
(Sarah, 8, (surgery))
(Eve, 6, (surgery, checking blood pressure))

Now for this specific day, with these specific jobs and hours, it is possible to get all the jobs done as follows:
Alice will spend 4hrs mopping and 2hrs doing reception work.
Bob will do 1hr mopping and 2hrs checking blood pressure.
Sarah will spend 8hrs doing surgery.
Eve will do 2hrs of surgery and 4 hours checking blood pressure.

No person goes over their hours, and all the jobs get to be done.

Solution:

We reduce this problem to the maximum-flow problem. We give two transformations T_1 and T_2. T_1 will take an instance of our problem and transform it into a flow network (G, s, t, c) and T_2 transforms a maximum flow on G into a solution to our job assignment problem.

More precisely, we construct a flow network (G, s, t, c) as follows: For every person p_i, we add a vertex u_i to G, and for every job j_k we add a vertex v_k to G. We add the following edges to the flow network:

- (s, u_i) with capacity $c(s, u_i) = h_i$.
- (u_i, v_k) if the job j_k is in the set S_i of p_i. Set the capacity to $c(u_i, v_k) = \infty$.
- (v_k, t) with capacity $c(v_k, t) = c_k$.

We run Ford-Fulkerson on this network to compute the maximum flow f, and claim that $val(f) = \sum_k c_k$ iff all the jobs can be done in a given day. Transformation T_2 takes f and G and recovers the valid job assignment if one exists.

By construction of G, we know that no person p_i works more than h_i hours, and thus no u_i has more than h_i unit of incoming flow. Moreover, since an edge $(u_i, v_k) \in E$ iff p_i can perform job j_k, it follows that no person does a job not in their set S_i. So if $val(f) = \sum_k c_k$ then we've saturated all the capacities along the edges (v_k, t), and thus all the jobs have been completed, while satisfying the constraints above. Therefore, an integer flow corresponds to a valid job assignment to the hospital staff.

To recover the set of jobs each p_i performs, it suffices to look at the flow on the edges (u_i, v_k). If we have x unit of flow on the edge (u_i, v_k) then p_i will work x hours on job j_k.

G has $n' + n + 2$ vertices and $\mathcal{O}(n'n)$ edges, and can be computed in $\mathcal{O}(n'n)$ time by iterating over every p_i's set S_i. Running the Edmonds-Karp version of the max-flow algorithm will compute f in polynomial time. Given a maximum flow f, we can recover the set of jobs assigned to people in $\mathcal{O}(n'n)$ by looking at the flows on the edges (u_i, v_k). Therefore the total run time of the algorithm is polynomial.