# CSCI 570, Summer 2016 Homework 4 Solutions

For each of the following problems, you will use network flow. Each requires polynomial time, but does not require a more specific running time. You may (and should!) use "maximum flow" (or minimum cut) as a sub-routine without needing to explain it or its running time; when grading, we will assume that you call a polynomial time solution to network flow. Note that the graph you provide must still be polynomial in size.

For each question, explain how to construct the graph that will be used as input for the network flow algorithm, why that graph is polynomial in size, and how you will interpret the output of that algorithm to solve the problem.

1. Suppose we are given a set of tasks $S$ where each task $i \in S$ has an *arrival time* $a_i$, a *deadline* $d_i$, and a *workload* $w_i$. For each unit of time, we can do one unit of workload for the task, and we do not need to do it continuously (we can stop a task and resume it later). Our goal is to determine if we can complete all tasks by their deadline.

   (a) Use network flow techniques to design a polynomial-time algorithm that will determine if the tasks can all be done by their deadlines. Explain why your algorithm is polynomial time.

   Sort all arrivals and deadlines in one list. Each consecutive pair of times will form a time interval $t_i = (b_i, f_i)$, where $b_i$ and $f_i$ are arrival or deadlines (not necessarily one of each, nor in that order), but there are no arrivals or deadlines between those times. We can solve this by max-flow; create one vertex for each task - call it $s_i$ - and one for each time interval - call it $t_i$ . There is an edge from $s_i$ to $t_j$ if and only if $a_i \leq b_j \leq f_j \leq d_i$ - that is, if the time unit that $t_j$ represents is within the time that it is valid to do task $s_i$ . A unit of flow on this edge, once flow is computed, will represent that task $i$ has one unit of work being done at this time interval $j$. Since at most one job can be computed at each time step, we can enforce this constraint by having each time unit have an edge to the sink $t$, $e = (t_j, t)$ with capacity $f_j - b_j$ .
   We must, however, send the flow to each job so it can assign the flow to the times it is available. Create a source s and add edges $(s, s_i)$ for each $i$ with capacity $w_i$ . This means that, once flow is computed, the task will "get" one unit of flow for each time unit to partition to the time units in which it can be computed, sending at most one flow to each. We now can compute max flow. A schedule exists iff the max flow $= \sum_i w_i$

   (b) An **overworked period** is a pair of times $A$ and $B$ with the following property. If we let $S_{AB}$ be the set of tasks whose arrival time is $A$ or later and deadline is $B$ or earlier (i.e. $A \leq a_i \leq d_i \leq B$) then the total workload exceeds the time (i.e. $\Sigma_{i \in S_{AB}} w_i > B - A$). If there is an overworked period, then there is no way to complete all the tasks by their deadlines.

   Use network flow techniques to design a polynomial time algorithm that determines if there is an overworked period, and if so, what it is.

Suppose no such schedule exists. Then the max flow was less than $\sum_i w_i$ . There means there is some min-cut of the same cost that partitions $s$ from $t$, into sets $A$ and $B = V - A$. If we set the $(s_i, t_j)$ capacities above to be infinite, then we know they will not be cut, but the value of the max flow will not be changed.
Let's compute the cost of the cut. It is:

$$\sum_{i \in V-A} w_i + \sum_{x \in A} 1 < \sum_i w_i$$

Because

$$\sum_i w_i = \sum_{i \in V-A} w_i + \sum_{i \in A} w_i$$

we can subtract $\sum_{i \in V-A} w_i$ from both sides. We get:

$$\sum_{x \in A} 1 < \sum_{i \in A} w_i$$

If $A$ is contiguous, then $A$ must not include any task earlier than $a_i$ or later than $d_i$ . Otherwise, an infinite edge would have to be cut. So we're done if this is the case. If not, we can partition A into contiguous subsets of times, and use the above statement for each such segments.

2. Suppose we'd like to acquire a set of $n$ useful programs. Each program is sold by two companies, and the two versions are potentially of different quality. Suppose that company one's version of program $i$ has quality $x_i$ and company two's version has quality $y_i$. The obvious thing to do is to get the higher quality version for every program, but unfortunately the programs need to be *compatible* with one another and using incompatible versions (for example using company one's word processor and company two's web page editor) causes some amount of annoyance. Suppose we can evaluate this annoyance at $a_{(i,j)}$ for using company one's version of program $i$ and company two's version of program $j$. Our goal is to select sets $S_1$ and $S_2$ of programs to buy from company one and two respectively, such that $S_1 \cup S_2 = \{1, 2, ..., n\}$, $S_1 \cap S_2 = \emptyset$, and we maximize:

$$\sum_{i \in S_1} x_i + \sum_{j \in S_2} y_j - \sum_{i \in S_1, j \in S_2} a_{(i,j)}$$

Use network flow techniques to design a polynomial time algorithm for this problem.

*Hint: You can do this with only $n + 2$ vertices. This is not a matching problem.*

*Hint 2: Maximizing that quantity is equivalent to minimizing some other quantity.*

*Solution* We build a graph with a node for each of the $n$ programs. We add additional nodes $s$ and $t$. We insert edges from $s$ to $i$ with capacity $x_i$ and from $i$ to $t$ with capacity $y_i$, and from $i$ to $j$ with capacity $a_{(i,j)}$. We now compute a *minimum cut* in this graph. This is the partition $S, V - S$ which minimizes the weight of cut edges, which means minimizing:

$$C = \sum_{i \in S} y_i + \sum_{j \in V-S} x_j + \sum_{i \in S, j \in V-S} a_{(i,j)}$$

We note that the sum of *all quality values* is some fixed $Q = \sum_i (x_i + y_i)$. If we subtract $Q - C$, we are left with exactly the quantity we were asked to maximize. Since no decision we make can change $Q$, we know that minimizing $C$ and maximizing $Q - C$ are the same problem. So we return sets $S_1 = S$ and $S_2 = V - S$.

What? Only two problems? If you would like additional practice, the following questions from the Kleinberg & Tardos textbook are suggested. Do not submit them for credit; these are for your own practice.

For practice with network flow as a concept, try Chapter 7, problems 1, 2, 3, 4, 10, 23, 24 (if you struggle on 23, try doing 24 first). For practice with using network flow, try Chapter 7, problems 8, 9, 11, 13, 14, 16, 17, 18, 21, 23, 24, 51.

There are also some great questions in the textbook of Goodrich and Tamassia; check A-16.1 through A-16.7.