Instructor:
Eleni Drinea
CSOR W4246–Fall, 2021

Student:
Vishweshwar Tyagi
vt2353@columbia.edu

Homework 3 Theoretical (110 points)

Out: Monday, November 8, 2021
Due: 11:59pm, Monday, November 22, 2021

Homework Problems

1. (30 points) A flow network with demands is a directed capacitated graph with potentially multiple sources and sinks, which may have incoming and outgoing edges respectively. In particular, each node $v \in V$ has an integer demand $d(v)$; if $d(v) > 0$, v is a sink, while if $d(v) < 0$, it is a source. Let S be the set of source nodes and T the set of sink nodes.

   A circulation with demands is a function $f : E \to \mathbb{R}^+$ that satisfies

   $$\begin{aligned}&(a) \text{ capacity constraints : For each } e \in E, 0 \le f(e) \le c(e)\\&(b) \text{ demand constraints: For each } v \in V,\ f^{\text{in}}(v) - f^{\text{out}}(v) = d(v).\end{aligned}$$

   We are now concerned with a decision problem rather than a maximization one: is there a circulation f with demands that meets both capacity and demand conditions?

   (i) (8 points) Derive a necessary condition for a feasible circulation with demands to exist.

   Solution: For a feasible circulation f to exist, we must have

   $$\begin{aligned}0 &= \sum_{e\in E} f_e - \sum_{e\in E} f_e\\&= \sum_{v\in V}\sum_{e \text{ into } v} f_e - \sum_{v\in V}\sum_{e \text{ out of } v} f_e\\&= \sum_{v\in V}\Big[\sum_{e \text{ into } v} f_e - \sum_{e \text{ out of } v} f_e\Big]\\&= \sum_{v\in V} d(v)\\&= \sum_{v:d(v)<0} -|d(v)| + \sum_{v:d(v)>0} |d(v)|\end{aligned}$$

   Hence,

   $$\begin{aligned}\sum_{v:d(v)<0} |d(v)| &= \sum_{v:d(v)>0} |d(v)|\\ \to \sum_{s\in S} |d(s)| &= \sum_{t\in T} |d(t)| \qquad (1)\end{aligned}$$

(ii) (25 points) Reduce the problem of finding a feasible circulation with demands to max-flow.

<u>Solution:</u> We are given a network $G = (V, E, c, d)$ with source node s and sink node t, capacity constraints c and demand constraints d. In order to reduce the problem of finding a feasible circulation with demands to max-flow, let us first transform given network $G = (V, E, c, d)$ to a modified version of it without the demand constraints, call it $G' = (V', E', c')$, with modified capacity constraints c' and new set of source node S' and new set of sink node T'

<u>Transformation:</u>

(a) Add a super source node s' and a super sink node t' to the network.

Hence, $V' = V \cup \{s', t'\}$, $S' = \{s'\}$ and $T' = \{t'\}$

(b) Set $c'(e) = c(e) \;\; \forall\, e \in E$

(c) Add edge (s', s) and set $c'(s', s) = |d(s)| \quad \forall\, s \in S$

(d) Add edge (t, t') and set $c'(t, t') = |d(t)| \quad\; \forall\, t \in T$.

Hence, $E' = E \cup \{(s', s) \mid s \in S\} \cup \{(t, t') \mid t \in T\}$

This transformation clearly runs in $O(|V| + |E|)$ time which is polynomial. This is because we can identify the set of source nodes (S) and the set of sink nodes (T) using their demand values in $O(|V|)$ time. Next, we can introduce super source s' and add edges outgoing from s' to s with capacity $|d(s)|$ for all $s \in S$ in $O(|V|)$ time. Again, we can add super sink node t' and add edges outgoing from each $t \in T$ to t' with capacity $|d(t)|$ in $O(|V|)$ time. We can also define the new capacity constraints in $O(|V| + |E|)$ time.

<u>Reduced problem (Claim)</u>: There exists f, a feasible circulation in G if and only if the size of max-flow in G' is $\sum_{s \in S} |d(s)|$, or equivalently $\sum_{t \in T} |d(t)|$

<u>Inputs for the problems:</u> The input for the problem of finding if a feasible circulation exists is $G = (V, E, c, d)$ with capacity constraints c and demands d. The input for the problem that checks if the size of of max-flow equals $\sum_{s \in S} |d(s)|$ is $G = (V', E', c')$ with capacity constraints c' and source node s', sink node t'.

We now argue the equivalence of these two problems.

<u>Equivalence</u> To establish equivalence, we must show that

(I) if f is a feasible circulation in G, then the size of the max-flow in G' is $\sum_{s \in S} |d(s)|$

(II) if the size of the max-flow in G' is $\sum_{s \in S} |d(s)|$ then there exists a feasible circulation f in G.

(I) Suppose f is feasible in G. We define $g : E' \to \mathbb{R}^+$ by letting

(e) $g(s', s) = |d(s)| \ \big(= c'(s', s) \big) \ \ \forall\, s \in S$

(f) $g(t, t') = |d(t)| \ \big(= c'(t, t') \big) \ \ \forall\, t \in T$

(g) $g(e) = f(e) \ \ \forall e \in E$

Note that we can clearly define this mapping and let it run on G' in $O(|V| + |E|)$ time, which is polynomial. If we can show that g is max-flow on G' of size $\sum\limits_{s \in S} |d(s)|$, we'll be done.

Because f is feasible in G, we have

$$\forall e \in E \quad 0 \leq f(e)\big(= g(e)\big) \leq c(e)\big(= c'(e)\big) \text{ and,} \tag{2}$$

$$\forall u \in V \setminus (S \cup T) \quad g^{\text{in}}(u) = f^{\text{in}}(u) = f^{\text{out}}(u) = g^{\text{out}}(u) \tag{3}$$

Because $\forall\, s \in S,\ \ f^{\text{out}}(s) - f^{\text{in}}(s) = |d(s)|$, from (e), we get,

$$\begin{aligned} &\forall s \in S \quad f^{\text{out}}(s) - (f^{\text{in}}(s) + |d(s)|) = |d(s)| - |d(s)| \\ \to\ &\forall s \in S \quad g^{\text{out}}(s) - g^{\text{in}}(s) = 0 \\ \to\ &\forall s \in S \quad g^{\text{in}}(s) = g^{\text{out}}(s) \end{aligned} \tag{4}$$

Similarly, because $\forall\, t \in T,\ \ f^{\text{in}}(t) - f^{\text{out}}(t) = |d(t)|$, from (f), we get,

$$\begin{aligned} &\forall\, t \in T \quad f^{\text{in}}(t) - (f^{\text{out}}(t) + |d(t)|) = |d(t)| - |d(t)| \\ \to\ &\forall t \in T \quad g^{\text{in}}(t) = g^{\text{out}}(t) \end{aligned} \tag{5}$$

Ofcourse,

$$\forall\ (s', s) \in E' \ \ 0 \leq f(s', s)\big(= |d(s)|\big) \leq c'(s', s) \text{ and} \tag{6}$$

$$\forall\ (t, t') \in E' \ \ 0 \leq f(t, t')\big(= |d(t)|\big) \leq c'(t, t') \tag{7}$$

Therefore, using $(2) - (7)$, we have $g : E' \to \mathbb{R}^+$ such that

$$\forall\ e \in E' \ \ 0 \leq g(e) \leq c'(e)$$

$$\forall u \in V' \setminus \{s', t'\} \ \ g^{\text{in}}(u) = g^{\text{out}}(u)$$

Therefore, g is a flow in G' and clearly,

$$\begin{aligned} |g| &= \sum_{e \text{ out of } s'} g(e) \\ &= \sum_{s \in S} |d(s)| \end{aligned} \tag{8}$$

(8) proves that we found a flow g in G' of size $\sum_{s\in S} |d(s)|$. It is easy to see that this is also the maximum size of a flow in G' because $(\{s'\}, V' \setminus \{s'\})$ defines a cut of size $\sum_{e \text{ out of } s'} c'(e) = \sum_{s\in S} |d(s)|$.

(II) Suppose $\exists g$ a max-flow in G' of size $\sum_{s\in S} |d(s)| \big(= \sum_{t\in T} |d(t)| \big)$.

This is possible only if

$g(s', s) = |d(s)| \;\; \forall\, s \in S$ because $c'(s', s) = |d(s)|$, and the fact that $|g| = \sum_{s\in S} g(s', s)$

$g(t, t') = |d(t)| \;\; \forall\, t \in T$ because $c'(t, t') = |d(t)|$, and the fact that $|g| = \sum_{t\in T} g(t, t')$

We will alter g to get a feasible circulation f in G. To get this f, we simply remove the flow on edges $(s', s)\; \forall\, s \in S$ and $(t, t')\; \forall\, t \in T$. If we can show that f is feasible in G, we'll be done.

$f : E \to \mathbb{R}^+$ is such that $f(e) = g(e) \;\; \forall\, e \in E$

Clearly,

$$\forall\, e \in E \;\; 0 \le g(e)\big(= f(e)\big) \le c'(e)\big(= c(e)\big) \tag{9}$$

and because g is a flow in G',

$$\begin{aligned} &\forall\, u \in V \setminus (S \cup T) \;\; f^{\text{in}}(u) = g^{\text{in}}(u) = g^{\text{out}}(u) = g^{\text{in}}(u) \\ \to\, &\forall\, u \in V \setminus (S \cup T) \;\; f^{\text{in}}(u) - f^{\text{out}}(u) = 0 = d(u) \end{aligned} \tag{10}$$

Also, $\forall\, s \in S$ and $\forall\, t \in T$

$0 = g^{\text{in}}(s) - g^{\text{out}}(s) = (f^{\text{in}}(s) + g(s', s)) - f^{\text{out}}(s) = (f^{\text{in}}(s) + |d(s)|) - f^{\text{out}}(s)$

$0 = g^{\text{in}}(t) - g^{\text{out}}(t) = f^{\text{in}}(t) - (f^{\text{out}}(t) + g(t, t')) = f^{\text{in}}(t)) - (f^{\text{out}}(t) + |d(t)|)$

and hence $\forall\, s \in S$ and $\forall\, t \in T$

$$f^{\text{in}}(s) - f^{\text{out}}(s) = -|d(s)| = d(s) \tag{11}$$

$$f^{\text{in}}(t)) - f^{\text{out}}(t) = |d(t)| = d(t) \tag{12}$$

Hence, from (9) – (12), f is feasible circulation in G and we're done.

2. (55 points)

(a) (30 points) Given an unlimited supply of coins of denominations $c_1, c_2, \dots c_n$, you wish to make change for a value v; that is, you wish to find a set of coins whose total value is v. This might not be possible: for instance, if the denominations are 5 and 10, then we can make change for $15 = 10 + 5$ but not for 11.

Design an $O(nv)$ algorithm to determine whether it is possible to make change for v using coins of denominations $c_1, c_2, \dots c_n$. If the answer is yes, also output a way to make change for v.

Solution: Since we'll be working with $0-$indexed arrays, let us denote the coins by $c_0, c_1 \dots c_{n-1}$ and assume that $n \geq 1$, which means that we're given atleast one coin. We will also assume that $v \geq 0$ where $v = 0$ means that there is no money to change. We will use Dynamic Programming in order to reduce the given problem to smaller sub-problems and efficiently solve them.

Sub-problem: We define t(j) for $0 \leq j < (v + 1)$ to denote whether or not it possible to change the value j with infinite supply of coins $c_0, c_1 \dots c_{n-1}$. That is, for $0 \leq j < (v+1)$, we define

$$t(j) = \begin{cases} \text{True} & \text{if (infinite supply of) } c_0, c_1, \dots c_{n-1} \text{ can change the value } j \\ \text{False} & \text{otherwise} \end{cases} \tag{1}$$

Then, we'll be able to change v using an infinite supply of $c_0, c_1 \dots c_{n-1}$ if and only if $t(v) = \text{True}$

Boundary Condition: Since we can always change the value 0, we have,

$$t(0) = \text{True} \tag{2}$$

Recurrence: Suppose, for $1 \leq j < (v + 1)$, we are given infinite supply of the coins $c_0, c_1 \dots c_{n-1}$ and we are asked to change the value j.

Clearly, this is possible if and only if it is possible to change (using infinite supply of the coins) atleast one of $j - c_k$ where $k \in \{0, 1 \dots (n-1)\}$ such that $c_k \leq j$.

That is, to be able to change j, we should be able to choose atleast one coin c_k such that $c_k \leq j$ and that it is possible to change $j - c_k$ again using the infinite supply of the coins. If no such c_k exists, then it is not possible to change j. This is summarized on the next page:

Let $S_j = \{c_i \mid 0 \leq i < n,\ c_i \leq j\}$, then, for $1 \leq j < (v+1)$, we have,

$$t(j) = \begin{cases} \mathrm{OR}_{c_i \in S_j}\ t(j - c_i) & \text{if } |S_j| > 0 \\ \text{False} & \text{otherwise} \end{cases} \tag{3}$$

where OR is logical OR and $\mathrm{OR}_{c_i \in S_j}\ t(j - c_i)$ evaluates to True if atleast one of $t(j - c_i)$ for $c_i \in S_j$ is True, otherwise False.

Now, in order to present a valid sequence of coins that sum to j, for $1 \leq j < v + 1$ we define k_j to be the smallest index $0 \leq k_j < n$ such that $c_{k_j} \in S_j$ and it is possible to change $j - c_{k_j}$ using infinite supply of the coins. If such an index does not exist, we set $k_j = -1$. Then, we also define,

$$p(j) = \begin{cases} c_{k_j} & \text{if } k_j \neq -1 \\ -1 & \text{otherwise} \end{cases} \tag{4}$$

Basically, $p(j)$ for $1 \leq j < n$ keeps track of the earliest coin among ordered sequence $(c_0, c_1 \ldots, c_{n-1})$ that can be used to change j successfully. If there is no such coin, we keep $p(j) = -1$. We will populate p along with t, both in left to right direction

pseudocode The following procedure takes the input coins into c and the value in v. It returns False if v cannot be changed using an infinite supply of the coins, otherwise it returns True and along with it a queue that contains a valid sequence of coins that sum up to v.

(pseudocode on next page)

```
procedure change-value(c=[c[0], c[1], ... c[n-1]], v):

    Initialize boolean array of length (v+1), set all
     elements = False and call it t             // O(v)

    Initialize array of length (v+1), set all
      elements = (-1) and call it p            // O(v)

    // boundary condition
    t[0] = True

    for j from 1 to v:
        t[j] = False
        p[j] = -1
        for i from 0 to n-1:
            if c[i] < j+1:
                if t[j-c[i]] == True:
                    t[j] = True
                    p[j] = c[i]
                    break


    if t[v] == False:
        return False

    Initialize LIFO queue, call it q

    j = v
    while j != 0:           \\ O(v)
        Enqueue(q, p[j])
        j -= p[j]

    return True, q
```

Time-Complexity: Apart from the complexities already mentioned, we have $[10-18]$ which runs in $O(nv)$ time since while calculating $t[j]$, we already have the values for $t[j-c[i]]$ for all those coins such that $c[i] \leq j$, which is because of the order in which we fill t (left to right). Hence, the required time complexity is:

$$T(n, v) = O(nv)$$

Extra-Space Complexity: We need to maintain two arrays of $(1, v+1)$ shape. Thus, $O(v)$ extra space is required.

(b) (25 points) Consider the following variation of the above problem. You are only allowed to use each denomination at most once. For example, if the denominations are $1, 5$ and 10, then you can make change for $6 = 1 + 5$ but not for 20 since you cannot use 10 twice. Design an $O(nv)$ algorithm to determine whether it is possible to make change for v using each denomination $c_1, c_2, ..., c_n$ at most once.

Solution: Since we'll be working with $0-$indexed arrays, let us denote the coins by $c_0, c_1 \dots c_{n-1}$ and assume that $n \geq 1$, which means that we're given atleast one coin. We will also assume that $v \geq 0$ where $v = 0$ means that there is no money to change. We will use Dynamic Programming in order to reduce the given problem to smaller sub-problems and efficiently solve them.

Sub-problem: We define t(i, j) for $0 \leq i < n$ and $0 \leq j < (v+1)$ to denote whether or not it possible to change the value j with the first $(i+1)$ coins, which are $c_0, c_1 \dots c_i$ using each atmost once. That is, for $0 \leq i < n$ and $0 \leq j < (v+1)$, we define

$$t(i, j) = \begin{cases} \text{True} & \text{if } c_0, c_1, \dots c_i \text{ can change (using each atmost once) } j \\ \text{False} & \text{otherwise} \end{cases} \tag{1}$$

Then, we will be able to change v using the given coins (each atmost once) if and only if $t(n-1, v) = \text{True}$.

Boundary Condition: Since we can always change the value 0, we have,

$$t(i, 0) = \text{True} \quad \forall\, 0 \leq i < n \tag{2}$$

Also, it is easy to see that $\forall\, 1 \leq j < (v+1)$

$$t(0, j) = \begin{cases} \text{True} & \text{if } j = c_0 \\ \text{False} & \text{otherwise} \end{cases} \tag{3}$$

This is because the only value (except for 0) we can change when only given the coin c_0 is c_0 itself provided we're allowed to use it atmost once.

Recurrence: Suppose, for $1 \leq i < n$ and $1 \leq j < (v+1)$, we are given the coins $c_0, c_1 \ldots c_i$ and we are asked to change the value j. The possibility of doing so is denoted by $t(i, j)$, as defined above.

Now, there are two mutually exclusive exhaustive cases:

(1) $c_i < j + 1$

(2) $c_i > j$

Case (2) is simpler, we will discuss this first. $c_i > j$ means that we can never use the coin c_i to change j. Hence, in this case, we'll be able to change j using the given coins if and only if we can change j using the coins $c_0, c_1, \ldots c_{i-1}$. Hence, $t(i, j)$ is same as $t(i-1, j)$ when $c_i > j$

In case (1), we can either decide to use the coin c_i or not use it in order to change j. In case we decide to use it, we'll be able to change j using the given coins if and only if we can change the value $(j - c_i)$ with coins $c_0, c_1 \ldots c_{i-1}$. Hence, if we do decide to use c_i, then, $t(i, j)$ will be same as $t(i-1, j-c_i)$.

However, if we decide not to use it, we'll be able to change j with given coins if and only if we can change the value j with coins $c_0, c_1 \ldots c_{i-1}$. Hence, in this case, $t(i, j)$ will be same as $t(i-1, j)$.

Now, realize that for case (1), it is sufficient for either of the sub-cases to yield a valid change for j. Therefore, for $1 \leq 1 < n$ and $1 \leq j < (v+1)$, we have

$$t(i,j) = \begin{cases} t(i-1,j) \text{ OR } t(i-1,j-c_i) & \text{if } c_i < j+1 \\ t(i-1,j) & \text{otherwise} \end{cases} \tag{4}$$

where OR is logical OR.

Order of filling: We will fill t in our algorithm which is $n \times (v+1)$ matrix. After defining $t[0:n, 0]$ and $t[0, 0:v+1]$ using the boundary conditions in (2) and (3), we will start filling $t[1:n, 1:v+1]$ in the order: $t[1,1], t[1,2] \ldots t[1,v], t[2,1], t[2,2], \ldots t[2,v], \ldots$ $t[n-1,1], t[n-1,2], \ldots t[n-1,v]$, that is, left to right and moving top to bottom.

pseudocode

The following procedure takes coins input in the array c and value input in v and returns True if it is possible to change v using the given coins, else False.

```
1  procedure change-value(c=[c[0], c[1], ... c[n-1]], v):
2
3      Initialize 2D boolean array of size (n, v+1), set all
          elements = False and call it t              // O(nv)
4
5      // boundary condition
6      for i from 0 to n-1:                // O(n)
7          t[i, 0] = True
8
9      for j from 1 to v:                  // O(v)
10         if j == c[0]:
11             t[0, j] = True
12         else:
13             t[0, j] = False
14
15     for i from 1 to n-1:
16         for j from 1 to v:
17             if c[i] < j+1:
18                 t[i, j] = t[i-1, j] || t[i-1, j-c[i]]
19             else:
20                 t[i, j] = t[i-1, j]
21
22     return t[n-1, v]
```

Time-Complexity Apart from the time-complexities already mentioned in pseudocode, we note that Equation (4) is constant time since we already know $t(i-1, j)$ and $t(i-1, j-c_i)$ (whenever $c_i \leq j$) due the order in which we fill t.
Hence, $[15-20]$ is $O(nv)$. Therefore, the time-complexity is given by:

$$T(n, v) = O(nv)$$

Extra-Space We need to store 2D array t of size $(n, v+1)$, hence, $O(nv)$ extra space is required.

Note that our recurrence only uses entries from the $(i-1)^{\text{th}}$ row when filling $t(i, j)$ for $1 \leq i < n$ and $1 \leq j < v+1$. Using this observation, we can further optimize extra-space using the following pseudocode.

```
procedure change-value-2(c=[c[0], c[1], ... c[n-1]], v):

    Initialize 2D boolean array of size (2, v+1), set all
       elements = False and call it t            // O(nv)

    // boundary condition
    for i from 0 to 1:             // O(n)
        t[i, 0] = True

    for j from 1 to v:               // O(v)
        if j == c[0]:
            t[0, j] = True
        else:
            t[0, j] = False

    for i from 1 to n-1:
        for j from 1 to v:
            if c[i] < j+1:
                t[i%2, j] =
                   t[(i-1)%2, j] || t[(i-1)%2, j-c[i]]
            else:
                t[i%2, j] = t[(i-1)%2, j]

    return t[(n-1)%2, v]
```

Here, the recurrence we're using is same as earlier, it is just that we are optimizing on space. Time-complexity still remains the same, $O(nv)$. We only need a $(2, v+1)$ shape array, so optimized extra-space is $O(v)$.

3. (22 points) Similarly to a flow network with demands, we can define a flow network with supplies where each node $v \in V$ now has an integer supply s_v, so that if $s_v > 0$, v is a source and if $s_v < 0$, it is a sink, and the supply constraint for every $v \in V$ is $f^{\text{out}}(v) - f^{\text{in}}(v) = s_v$.

   In a min-cost flow problem, the input is a flow network with supplies where each edge $(i, j) \in E$ also has a cost a_{ij}, that is, sending one unit of flow on (i, j) costs a_{ij}. Given a flow network with supplies and costs, the goal is to find a feasible flow $f : E \to \mathbb{R}^+$, that is, a flow satisfying edge capacity constraints and node supplies, that minimizes the total cost of the flow.

   Show that the max flow problem can be formulated as a min-cost flow problem.

<u>Solution:</u> Suppose that we're given a network $G = (V, E, c)$ with source node s and sink node t, capacity constraints c to be solved for max-flow. We perform the following transformation to get a modified network $G' = (V', E', c', S, a)$ with capacity constraints c', supply constraints S, and cost a

<u>Transformation:</u>

(a) Set $V' = V$

(b) Add edge (t, s), i.e, $E' = E \cup \{(t, s)\}$

(c) Set $c'(e) = c(e) \ \ \forall\, e \in E$ and $c'(t, s) = \infty$. This defines the capacity constraint on G'.

(d) Set $a(t, s) = -1$ and $a(e) = 0 \ \ \forall\, e \in E$. This defines the cost of sending unit flow along each edge in G'.

(e) Set $S(u) = 0 \ \ \forall\, u \in V$

Clearly, this transformation can be done in $O(|V| + |E|)$ which is polynomial running time. This is because, in each step, we're either working through the set of nodes in V or the edges in E.

<u>Natural Projection and Lift:</u> Let f be feasible flow in G', then $f_{|_G} : E \to \mathbb{R}^+$ is the projection of f defined as

$$f_{|_G}(e) = f(e) \quad \forall\, e \in E$$

Clearly, $0 \leq f_{|_G}(e)\big(= f(e)\big) \leq c(e)\big(= c'(e)\big) \ \ \forall e \in E$

Note that, we have $S(u) = 0 \ \ \forall\, u \in V$. This means that, $\forall\, u \in V$, $f^{\text{in}}(u) = f^{\text{out}}(u)$. Hence,

$$f_{|_G}{}^{\text{in}}(u) = f^{\text{in}}(u) = f^{\text{out}}(u) = f_{|_G}{}^{\text{out}}(u) \ \ \forall u \in V \setminus \{s, t\}$$

Hence, $f_{|_G}$ defines a flow on G and can be obtained in $O(|E|)$ polynomial time.

For a feasible flow f in G', when we speak of f in G, we will mean $f_{|_G}$.

Now, suppose f is a flow in G. We define the lift of f on G', denoted as $f^{|^{G'}} : E' \to \mathbb{R}^+$ as,

$$f^{|^{G'}}(e) = \begin{cases} f(e) & \text{if } e \in E \\ |f| & \text{if } e = (t, s) \end{cases}$$

Clearly, $0 \leq f^{|^{G'}}(e) \leq c'(e) \ \ \forall\, e \in E'$. Also in G', $S(u) = 0 \ \ \forall u \in V' \setminus \{s, t\}$ because f is a flow in G and for each node except for the source and sink, flow that enters this node equals the flow that exits this node.

Moreover, $S(s) = f^{|G'}{}^{\text{out}}(s) - f^{|G'}{}^{\text{in}}(s) = f^{\text{out}}(s) - |f| = 0$

and similarly, $S(t) = f^{|G'}{}^{\text{out}}(t) - f^{|G'}{}^{\text{in}}(t) = |f| - f^{\text{in}}(t) = 0$

Therefore, $S(u) = 0 \;\; \forall u \in V$ and all supply constraints of G' are satisfied.

Thus, $f^{|G'}$ is feasible in G' and can also be obtained in $O(|E|)$ time.

For a flow f in G, when we speak of f in G', we will mean $f^{|G'}$.

Reduced problem (Claim): f is max-flow in G if and only if f is min-cost flow in G'.

Inputs for the problems: The input for the problem that finds max-flow is $G = (V, E, c)$ with source node s and sink node t, capacity constraints c. The input for the problem that evaluates min-cost flow in is $G' = (V', E', c', S, a)$ with capacity constraints c', supply constraints S, and cost a.

Equivalence: To prove equivalence, we will argue that

(I) If f is min-cost flow in G' then f is max-flow in G

(II) If f is max-flow in G, then f is min-cost flow in G'

In case of (I), we know that $a(t, s) = -1$ and $a(e) = 0 \;\; \forall \; e \in E$. This ensures that the min-cost flow will try to send as much flow from t to s via (t, s) because (t, s) is the only negative cost edge and this is possible due to the infinite capacity of this edge. Hence, f has maximum possible flow travelling through (t, s) in G', i.e, maximum-flow travels from t and enters s. Now, since $f^{\text{in}}(s) = f^{\text{out}}(s)$ and $f^{\text{in}}(t) = f^{\text{out}}(t)$ due to the supply constraints, the same flow will exit s and enter t through the network G, but this flow is precisely given by $f_{|G}$. Thus, f is max-flow in G.

In case of (II), suppose that f is max-flow in G. We know any flow g in G' will be min-cost if and only if maximum possible flow travels via (t, s), again because (t, s) is the only negative edge and this is possible because of its infinite capacity. This is precisely what happens for $f^{|G'}$ that has $f^{|G'}(t, s) = |f|$, the maximum possible flow that can travel (t, s). Note that $|f|$ is the maximum possible flow that can travel (t, s) because if there exists a feasible flow g in G' with $g(t, s) > |f|$, then $g_{|G}$ will have more flow in G than f which is not possible. Hence, f is min-cost flow in G'.

Hence, we have argued the reduction correctly.

Remarks:

- $[x-y]$ or $[x, y]$ refers to code lines starting at x and ending at y for referenced procedure
- $\{x, y, z\}$ refers to code lines x, y and z for referenced procedure and so forth.
- Unless mentioned otherwise, 0-indexing has been used for most data structures.
- Algorithms studied in lectures have been used directly.