MATH 566: Lecture 6 (09/05/2024)

Today: * Network transformations
 – arc reversal, removing upper bounds, node splitting
 * complexity of algorithms

---

② Arc reversal

Can be used to handle negative c_{ij}'s.

$$\begin{tikzcd} \overset{b(i)}{\textcircled{i}} \arrow[r, "c_{ij},\ u_{ij}", "x_{ij}"'] & \overset{b(j)}{\textcircled{j}} \end{tikzcd} \quad \text{changes to} \quad \begin{tikzcd} \overset{b(i)-u_{ij}}{\textcircled{i}} & \overset{b(j)+u_{ij}}{\textcircled{j}} \arrow[l, "-c_{ij},\ u_{ij}"', "x_{ji}"] \end{tikzcd} \qquad x_{ij} = u_{ij} - x_{ji}$$

Logic: First send u_{ij} from i to j to saturate (i,j). Modify $b(i)$ and $b(j)$ to reflect this change. Then we pull back x_{ji} units from j to i. Since u_{ij} is constant, the only variable contribution to the cost will be from x_{ji}, which flows in the opposite direction.

Contribution to total cost: $c_{ij}x_{ij}$

$$= c_{ij}(u_{ij} - x_{ji}) = c_{ij}u_{ij} \underbrace{-c_{ij}}_{c_{ji} := -c_{ij}} x_{ji}$$

$$= \text{constant} + c_{ji}x_{ji}$$

By setting $c_{ji} = -c_{ij}$, we get a positive cost when $c_{ij} < 0$.

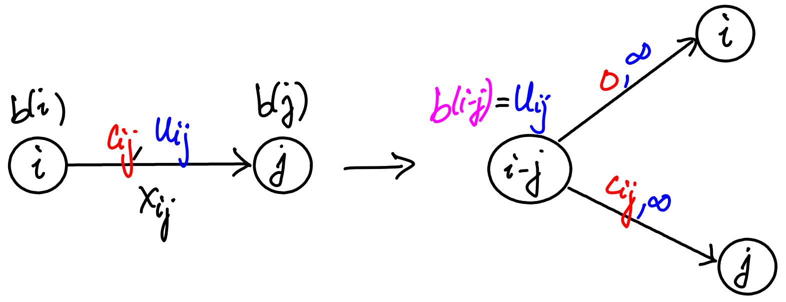
The motivation for arc reversal is the possibility of applying an algorithm which assumes all c_{ij} are non-negative.

Q: What if all c_{ij} are < 0 to start with? Could we add a constant value to all c_{ij}'s so that they all become ≥ 0?

This transformation is a bit tricky! Think about it 💡 ?!

③ Removing upper bounds ($l_{ij}=0 \ \forall \ (i,j)$ is assumed)

Want all $u_{ij}=+\infty$.



We present a complementary but equivalent formulation to that given in AMO.

Intuitively, each node i has a flow balance equation with b(i) in the rhs. Arc (i,j) has an upper bound that appears as $\leq u_{ij}$. Hence to remove this bound, we equivalently add an extra node i-j and set $b(i\text{-}j)=u_{ij}$.
"i-hyphen-j"

Let's concentrate on x_{ij} alone in the model:

$$x_{ij} = b(i) \quad \text{———} (1)$$
$$-x_{ij} = b(j) \quad \text{———} (2)$$
$$x_{ij} \leq u_{ij}$$
$$\Rightarrow x_{ij} + s_{ij} = u_{ij} \quad \text{———} (3)$$

"slack"

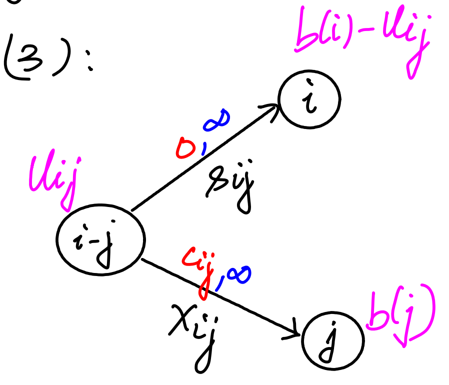
But now, s_{ij} appears only once in system (1), (2), (3), and x_{ij} appears 3 times. We want each flow appearing twice, with +1 and -1.

So we do (1)−(3) to get (1'): $-s_{ij} = b(i) - u_{ij}$ ——— (1')

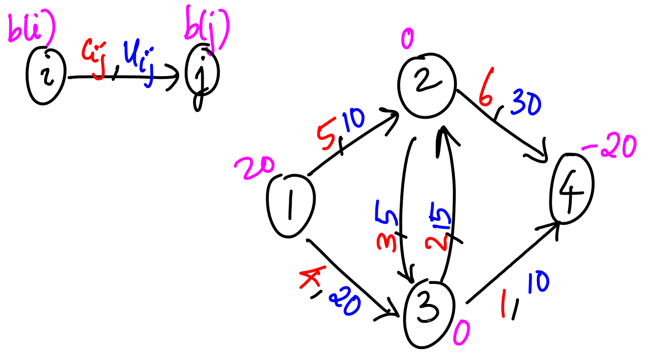
We consider the equivalent system (1'),(2),(3):

$$-s_{ij} = b(i) - u_{ij} \quad \text{———} (1')$$
$$-x_{ij} = b(j) \quad \text{———} (2)$$
$$\underbrace{x_{ij} + s_{ij} = u_{ij}} \quad \text{———} (3)$$

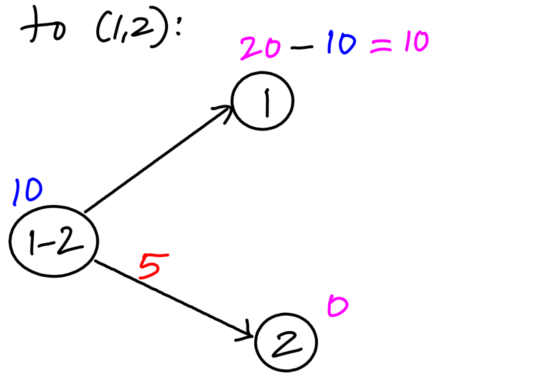
x_{ij} and s_{ij} are outflows from a new node with $b(\cdot)=u_{ij}$; and they flow into j and i, respectively.



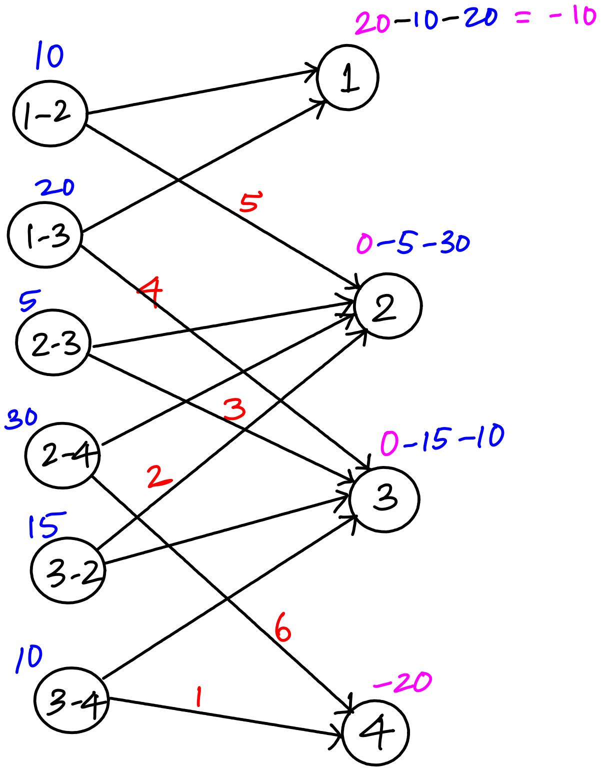
We could apply this transformation on multiple arcs together.
Consider the following network:



Here is the transformation applied to (1,2):



We aggregate the transformations to each arc to get the final network. Unmentioned costs are 0 (and, of course, all upper bounds are $+\infty$).



Here are some points to note:

* The overall change to b(i) is
$$b(i) \leftarrow b(i) - \sum_{(i,j)\in A} u_{ij}$$ ← all outarcs of node i

* The new network is bipartite, with the original nodes all in N_2 and the new nodes (modeling the arc upper bounds) being put in N_1. This result holds in general as long as all original u_{ij} are finite to start with.

* The flow $x_{i\text{-}j,j}$ in the new network is equal to the flow x_{ij} in the original network. Further notice that the contribution to total cost from flow on (i,j) in the original network is captured still as $c_{ij}x_{ij}$ in the new network.

One could use this class of transformations to obtain a bipartite structure. The motivation is the possible design/application of algorithms that work better on bipartite graphs.

At the same time, we might not apply this transformation if directed paths from i to j are of critical interest — notice that there is no directed path from any node $i \in N_2$ to any other node $j \in N_2$ in the new network.

4. <u>Node splitting</u>

~~Split~~ Replace node i into two: i'', i', and add arc (i'', i').

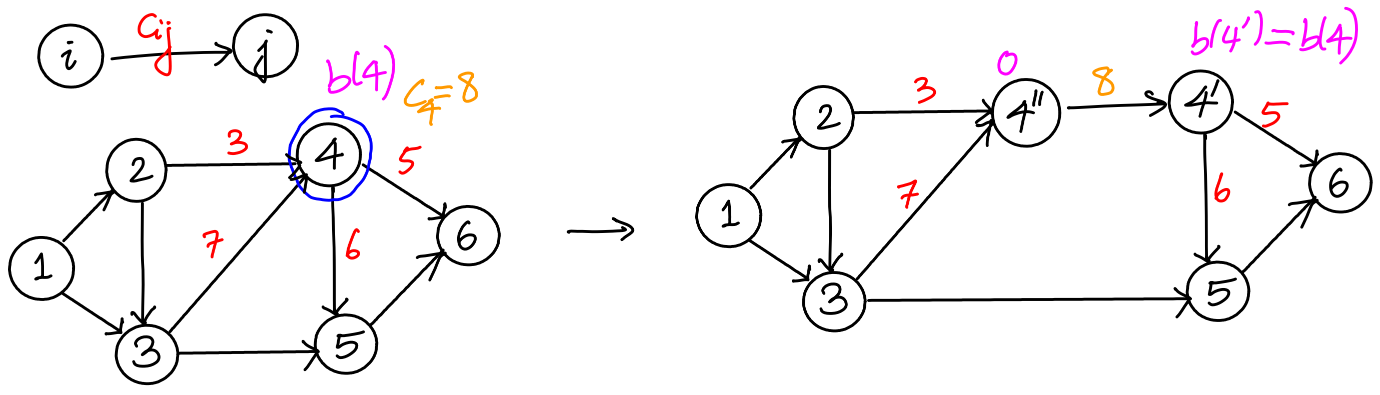
<u>Steps</u> Replace node i with two nodes i'' and i', add arc (i'', i').

Replace arc (j, i) with (j, i'') $\longrightarrow$ inarc of node i

Replace arc (i, k) with (i', k) $\longrightarrow$ outarc of node i

For the arc replacements, we transfer the arc parameters (cost, capacities) to the new arcs unchanged.

We illustrate the transformation on a small example. Say node 4 has a unit cost $c_4 = 8$ (for flow through the node). We split node 4 as shown below.



We assume that $b(4)$ does not incur cost c_4. If it were that case, we set $b(4'') = b(4)$ and $b(4') = 0$.

Intuition: Consider traffic flow through a road network modeled as a directed network. There could be a cost for going through the node modeling a city with downtown, for instance, but there is no cost for bypassing the city. Similarly, there might be a limit on how much traffic could go through downtown.

We summarize the various classes of network flow problems as well as network transformations we have seen so far.

Problem classes	Transformations
* shortest path	* adding nodes/arcs
* max flow	* removing lower bounds
* min-cost flow — most general	* arc reversal
* circulation	* removing upper bounds
* transportation	* node splitting
* assignment	

We will see minimum spanning trees as a separate class of problems later on. Similarly, matching (in general setting) is a different problem class.

All transformations were defined for the min-cost flow problem.

Computational Complexity (for dummies)

Q. How do we measure efficiency of an algorithm?

We do "worst case analysis", i.e., analyze the worst possible performance, so that we can **guarantee** it will perform at least as well on all instances.

Example: Add two $m \times n$ matrices A, B to get matrix C.

```
for i = 1 to m
    for j = 1 to n
        $C_{ij} := A_{ij} + B_{ij}$;  → assignment
    end
end
```

types of operations:
addition (+)
assignment (:=)
comparisons (for i = ...
j = ...)

What is the "running time" of this algorithm?

→ The number of steps (or operations) as a function of m & n.

→ Assume each operation takes a constant amount of time.

The exact time taken for an operation might vary from one computer to another. Hence we want to estimate the number of operations, and ignore the actual "time" as a constant multiplier. Further, our goal is to study how the algorithm performs as the size of the problem grows. As such, we want to compare the performance of the algorithm on different instances on the same machine, i.e., we do not want to be confused by the relative efficiencies of different machines.

Here, the running time includes

$$\left.\begin{array}{l} * \ \ c_1 mn \text{ for additions} \\ * \ \ c_2 mn \text{ for assignments} \\ * \ \ c_3 mn \text{ for comparisons} \end{array}\right\}$$

we want to ignore the constants c_1, c_2, c_3, c_4

i.e., $c_4 mn$ overall time.

"big-O" asymptotic upper bound

We say the algorithm runs in $O(mn)$ time.

(more in the next lecture...)