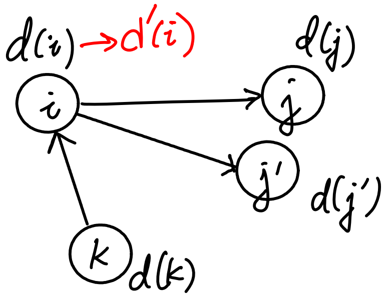
MATH 566: Lecture 20 (10/24/2024)

Today: * complexity of SAP algorithm
* Capacity scaling algorithm

Correctness of SAP Algorithm (continued)

* We saw that $d(i)$'s remain valid after augmentations.
* Relabeling

We relabel when there is no admissible arc (i,j).



$$\Rightarrow \quad d(i) \neq d(j)+1 \quad \forall (i,j) \in G(\bar{x})$$

$$\Rightarrow d(i) < d(j)+1 \quad \forall (i,j) \in G(\bar{x}),$$ due to validity of $d(i)$

$$\Rightarrow \quad d(i) < \min_j \{ d(j)+1 \mid (i,j) \in G(\bar{x}) \}.$$

Relabel: $d(i) \leftarrow d'(i) = \min_j \{ d(j)+1 \mid (i,j) \in G(\bar{x}) \}$.

Hence the relabeling strictly increases $d(i)$, and $d'(i) \leq d(j)+1$ still holds for all $(i,j) \in G(\bar{x})$.

Now Consider $(k,i) \in G(\bar{x})$. $d(k) \leq \overset{d'(i)}{d(i)}+1$ previously.

Since we strictly increased $d(i)$ by relabeling, $d(k) \leq d'(i)+1$ holds as well. □

Note that while we start with exact distance labels, we do not strive to maintain exactness at each step — we make sure labels remain valid after each iteration. In particular, we do not recalculate exact distance labels after each augmentation. The exactness of distance labels would hold at the end, though.

Complexity of Shortest Augmenting Path Algorithm

Outline (Corresponding results from AMO listed alongside)

1. Number of relabels $= O(n^2)$ Lemma 7.9 (a)

   Time for relabels $= O(nm)$ Property 7.7

2. Number of augmentations $= O(nm)$ Lemma 7.9(b) $\hookrightarrow$ uses Lemma 7.8

   Time for each augmentation $= O(n)$

   $\hookrightarrow$ each augmenting path has at most $n-1$ arcs

$\Rightarrow$ Total time for augmentations $= O(n^2m)$ Theorem 7.10

Recall that we might advance and retreat several times before finding each augmenting path. We need to count all that effort!

3. Time Spent looking for augmentations

   Total number of retreat operations $= O(n^2)$ (= number of relabels)

$\Rightarrow$ Total number of advance operations $= O(n^2 + n^2m)$

Each retreat operation cancels out a previous advance operation. We end back at $i = s$ (when $d(s) \geq n$).

We will now discuss details of each result.

Details of Complexity Analysis of SAP Algorithm

1. $d(i) \le n-1 \;\; \forall i \in N$.

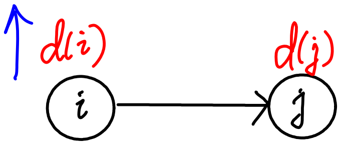
   After each relabel, $d(i)$ strictly increases. Hence we relabel node i at most n times.

   $\Rightarrow$ total # relabels $= O(n^2)$.

   n nodes, $\le n$ relabels per node

Each arc in $A(i)$ is examined once per relabel.

If (i,j) is inadmissible, $d(i) < d(j)+1$, and it remains inadmissible until i is relabeled.



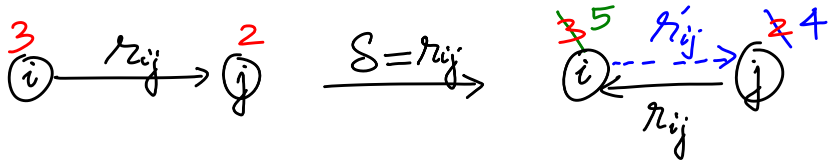
Total time for all relabels $= O(n \overset{\sum_i |A(i)|}{m})$.

We might conclude that the time for all relabels is $O(n^2m)$, counting it as (#nodes) × (max #relabels) × (# arcs per relabel) $= O(n \times n \times m)$. But we can be more careful in counting. Let r_i be the number of times node i is relabeled. For each relabel, the # arcs involved is $|A(i)|$.
Hence the total time for all relabels is

$$O\Big(\sum_{i \in N} r_i |A(i)|\Big) = O\Big(\sum_{i \in N} n |A(i)|\Big) = O\Big(n \sum_{i \in N} |A(i)|\Big) = O(nm).$$

This result is given as AMO Property 7.7 with $r_i = k$.

2. Consider the following example.



Initially, $d(i)=3=d(j)+1=2+1$, and (i,j) is admissible. We then have to push flow backward, i.e., along (j,i), for which $d(j)$ has to be increased to $d(i)+1=4$. Subsequently, $d(i)$ is increased to the new $d(j)+1=4+1=5$, so that (i,j) can be saturated again.

Let the algorithm saturate (i,j) in one push. We could saturate (i,j) again in a subsequent augmentation. But for this step to happen, both $d(i)$ and $d(j)$ must have increased by 2 each. We know

$d(i) \leq n$ ($n-1$ to be exact), and hence

(i,j) can be saturated at most $\frac{n}{2}$ times, i.e., $O(n)$ times.

There are m arcs in total, and each augmentation will saturate at least one arc.

$$\Rightarrow \text{total \# augmentations} = O(\overset{\#\text{arcs}}{m}\underset{\#\text{ saturations of one arc}}{n})$$

Time for each augmentation $= O(n)$ → $\leq n-1$ arcs per augmenting path

$$\Rightarrow \text{Total time for all augmentations} = O(n \times mn) = O(n^2 m).$$

3. Each retreat operation removes one arc from a partially admissible path, while each advance operation adds an arc to such a partial s-t path.

Total # retreats $= O(n^2)$ = # relabels

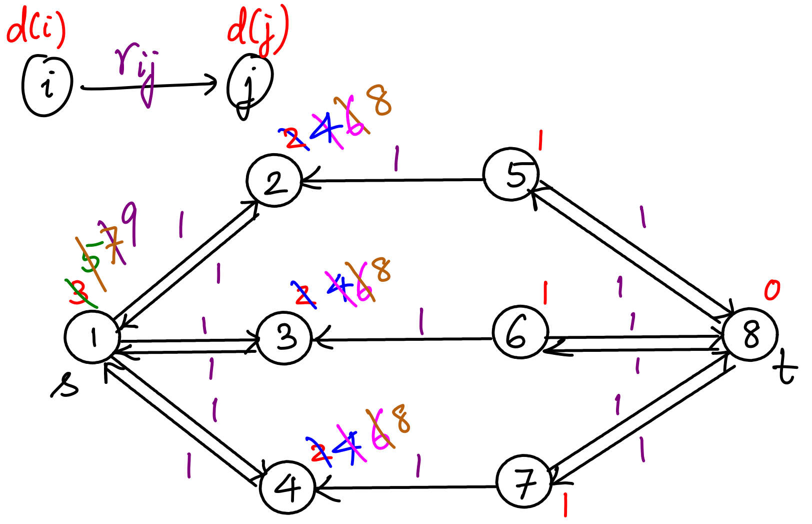
$$\text{Total \# arcs in all augmentations} = O(\underbrace{n^2 m}_{O(nm) \text{ augmentations} \times O(n) \text{ arcs per s-t path}})$$

$$\Rightarrow \text{Total \# advances} = O(\underbrace{n^2}_{\#\text{retreats}} + \underbrace{n^2 m}_{\#\text{arcs in augmentations}}) = O(n^2 m)$$

Notice that we have to make advances first before we could retreat. In the case of augmentations, we can count the advances alone, as there are no retreats in this process. In other words, we make as many advances as the number of retreats, in addition to the advances associated with augmentations.

$\Rightarrow$ Overall running time $= O(n^2 m)$. □

We could implement practical improvements. Consider the instance from the last lecture. Ideally, the algorithm could stop after the three augmentations, as the flow is already maximum. But it performs a number of relabels before terminating.



Here, we could try to identify a min cut $[S, \bar{S}]$, with $S = \{1,2,3,4\}$ as soon as $d(4)$ is relabeled from 2 to 4.

Capacity Scaling Algorithm

IDEA: Motivated by the "largest capacity" augmenting paths example, we augment along paths with "sufficiently large" capacity, rather than finding the largest capacity path.

Def (**Δ-residual network**) For any fixed $\Delta \geq 0$, we let

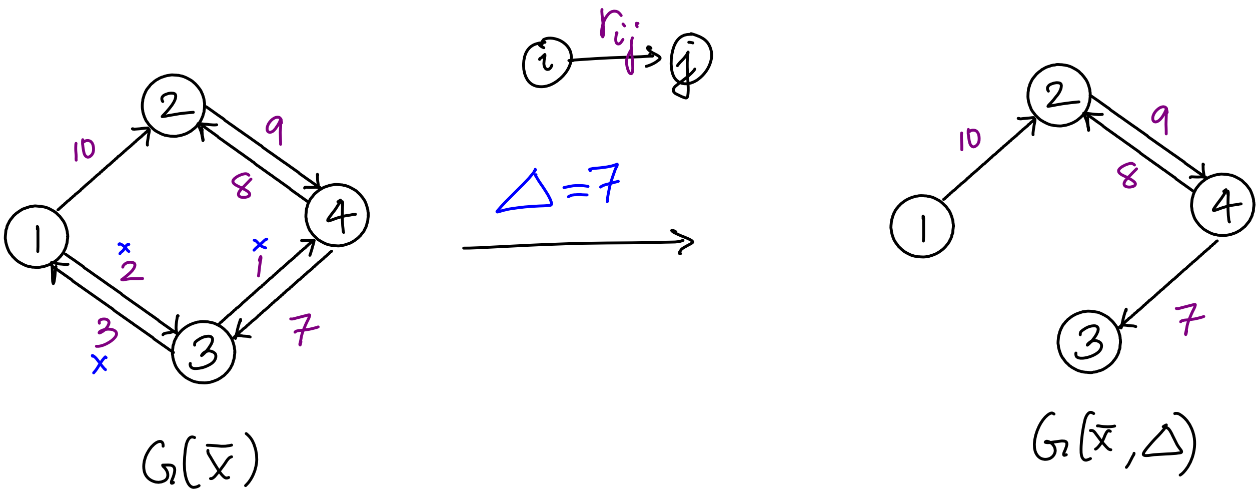
$$G(\bar{x},\Delta) = \{(i,j) \in G(\bar{x}) \mid r_{ij} \geq \Delta\}.$$

→ arcs in $G(\bar{x})$ with residual capacity $\geq \Delta$.

Def A flow $\bar{x}$ is **Δ-maximum** if $G(\bar{x},\Delta)$ has no s-t path, i.e., $G(\bar{x})$ has no augmenting path with capacity Δ or more.

Note that a flow $\bar{x}$ is maximum if it is 1-maximum, i.e., for $\Delta=1$.

Here is an example.



The idea is to push flow along paths with capacity at least Δ for a large value of Δ, and then scale Δ by ½ repeatedly, until we get to $\Delta=1$.

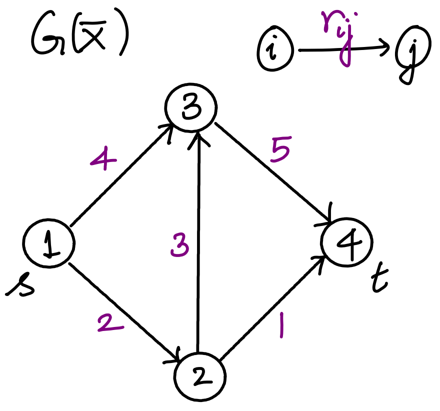
Here is the algorithm.

```
algorithm capacity scaling;
begin
    x : = 0;
    Δ : = 2^⌊log U⌋;          → largest power of 2 ≤ U
    while Δ ≥ 1 do
    begin
        while G(x, Δ) contains a path from node s to node t do   }
        begin                                                      }
            identify a path P in G(x, Δ);                          }  Δ-phase (one scaling phase)
            δ : = min{r_ij : (i, j) ∈ P};                          }
            augment δ units of flow along P and update G(x, Δ);    }
        end;
        Δ : = Δ/2;
    end;
end;
```

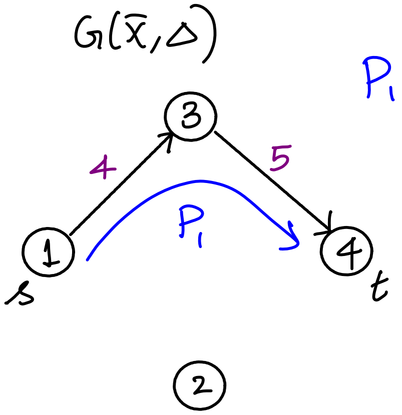
Figure 7.3 Capacity scaling algorithm.

Example

We start with $\bar{x} = \bar{0}$.



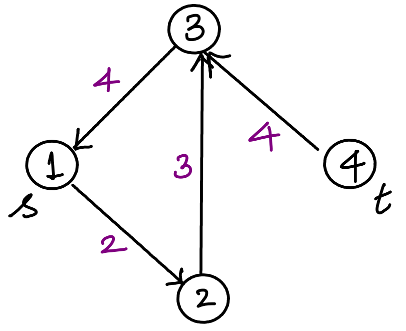
$\xrightarrow{\Delta=4}$



$P_1 = 1\text{-}3\text{-}4$, $\delta(P_1) = 4$

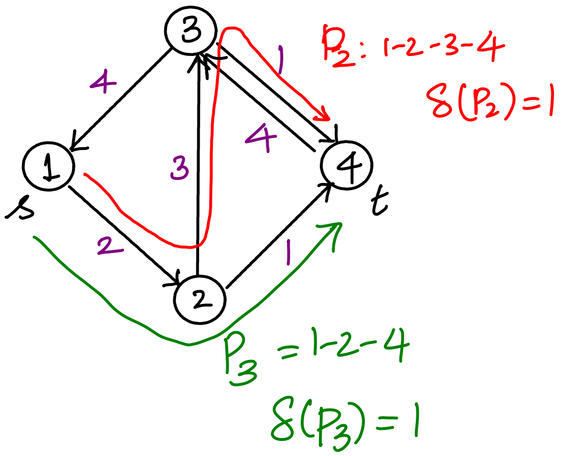
No more s-t paths. ⇒ $\Delta \leftarrow \frac{\Delta}{2}$, i.e., $\Delta = 2$.

$\xrightarrow{\Delta=2}$



There is no s-t path in $G(\bar{x},\Delta)$, so $\Delta \leftarrow \frac{\Delta}{2}$, i.e., $\Delta = 1$.

$\xrightarrow{\Delta=1}$



We find two s-t paths, and get no more s-t paths after augmenting along P_2 and P_3, i.e., we have a max-flow.