MATH 566: Lecture 22 (11/05/2026)

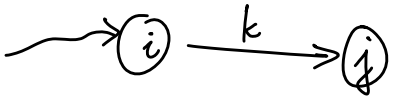
Today: * FIFO Preflow-Push algorithm
* Excess Scaling algorithm

Implementation of SAP algorithm

May be easier to work with x_{ij}'s in G itself, rather than $G(\bar{x})$.

Start with $\bar{x}=\bar{0}$;

Use arc-predecessor indices (apart from usual pred). Here, predA(j)=k means the arc leading into node j in your path is $k \in \{1,2,\ldots,m\}$.

So $H(k)=j$ (head of arc k is j). 

When working with G (and $\bar{x}$), we need to use both A{i} and AI{i} for candidate admissible arcs out of node i. Check them separately.

```
jj = A{i};
admjj = jj(find (x(jj) < u(jj)));
admjj = admjj(find (d(H(admjj)) + 1 == d(i)));
```
} outarcs

```
ji = AI{i};
admji = ji ( find ( x(ji) > 0 ));
admji = admji(find ( d(T(admji)) + 1 = d(i)));
```
} inarcs

[admjj, admji] could be the collection of admissible arcs. You can pick any one from this list (assuming there are many).

```
admarcs = [admjj, admji]; k = admarcs(1);
```
← picking the first admissible arc from the list

```
i = H(k);
```
→ advance operation

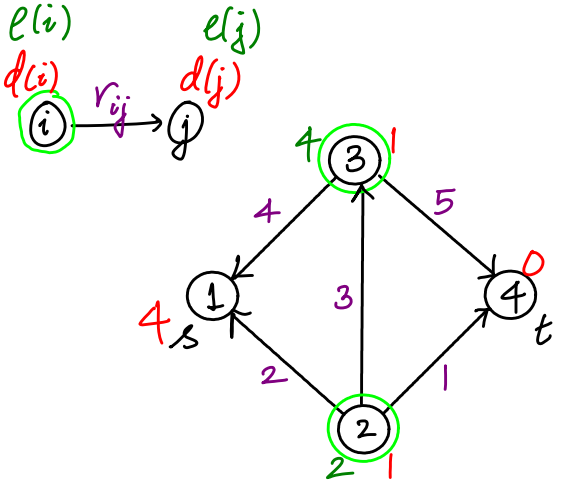
```
predA(i) = k;
```
→ arc predecessor

FIFO Preflow-Push Algorithm

* Whenever you select an active node i, keep pushing flow until $e(i)$ becomes zero (i.e., i is inactive), or i is relabeled. This whole process is called a **node examination.**
* Examine active nodes in a FIFO order.
  - maintain a LIST of active nodes, remove the first node from the front of LIST to examine.
  - add newly active nodes to the back of LIST.
* If you relabel node i, add it to the back of LIST.
  So, do not push from node i immediately after relabeling it.
  $\rightarrow d(i) = \min\{d(j)+1 \mid (i,j) \in G(\bar{x})\}$
* Complexity : $O(n^3)$ (AMO Theorem 7.17).

Back to our Example

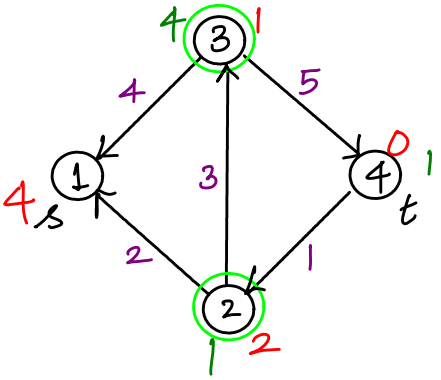
After preprocessing!



LIST = [~~2~~ 3]

examine 2 : push $\delta = \min\{e(2), r_{24}\}$

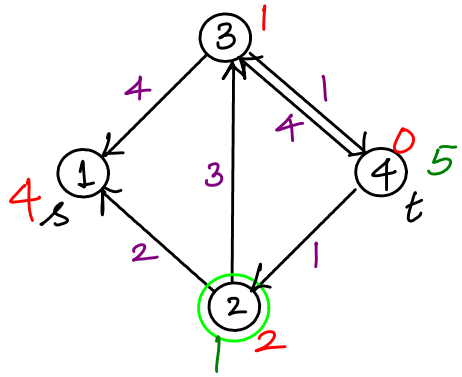
$= \min\{2, 1\} = 1.$



2 is still active, but no admissible arcs exist. So relabel, and add 2 to the back of LIST.

LIST = [~~3~~ 2]

Node examination of 3 now:
(3,4) is admissible. Push
$\delta = \min\{e(3), r_{34}\} = \min\{4, 5\} = 4$.



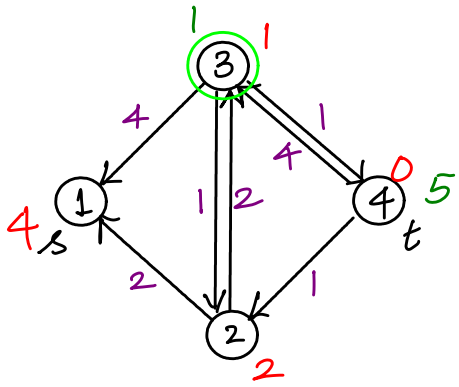
LIST = [~~2~~]

Examine node 2.
Push $\delta = \min\{e(2), r_{23}\} = 1$ along (2,3). 2 becomes inactive, but 3 is active — added to LIST.

LIST = [~~3~~]

Examine node 3.

Push $\delta = \min\{e(3), r_{34}\} = \min\{1, 1\} = 1$ along (3,4).



LIST is empty now, so flow is maximum.

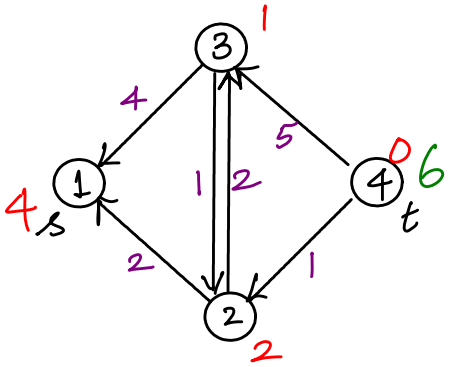
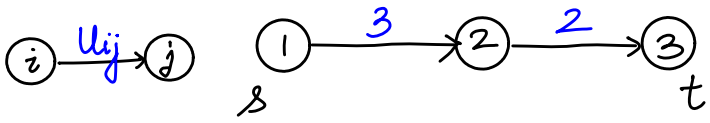
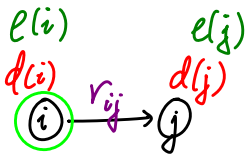
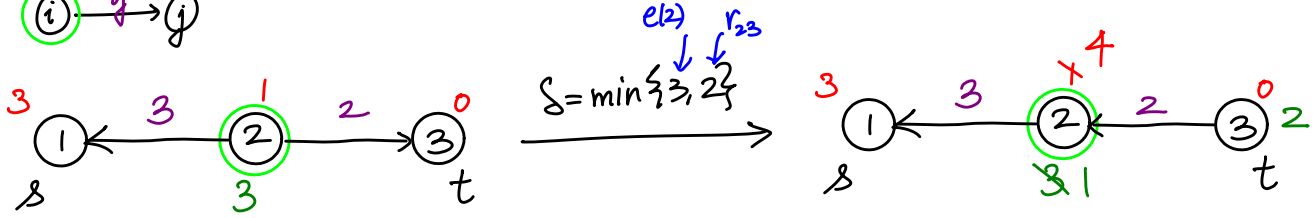


Illustration of pushing flow back to s

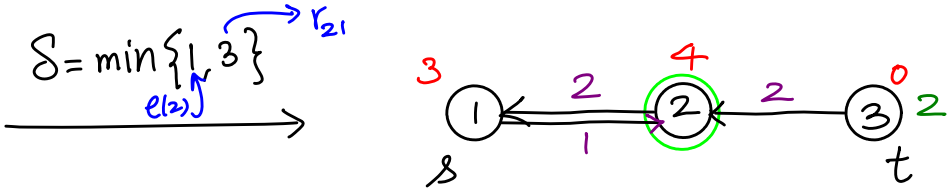
In the previous examples, we did not push any flow back to the source. But that sort of push-back happens routinely in more general networks. Here is a minimal example.







relabel 2, then push 1 unit back to s=1



Excess Scaling Algorithm

→ Idea: Have to get rid of all excess. Might as well push from large excess nodes. Similar to capacity scaling, but applied to preflow-push.

Let $e_{max} = \max_{i \in N \setminus \{s,t\}} \{e(i)\}$.

Start with large e_{max}, push from "large excess" nodes, while systematically reducing e_{max} to 0.

Def For a given Δ, **large excess nodes** are $j \in N \setminus \{s,t\}$ with $e(j) \geq \frac{\Delta}{2}$.

IDEA: Push flow from large excess nodes, but not too much!

Push $\delta = \min\{e(i), r_{ij}, \Delta - e(j)\}$ along admissible arc (i,j).

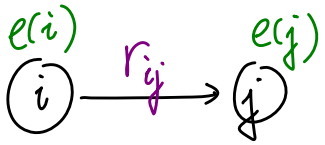
$e(j)$ stays $\leq \Delta$ after push!

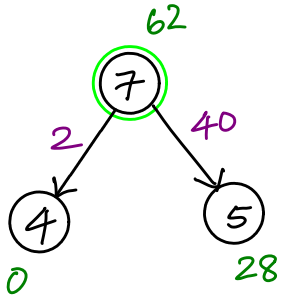
```
algorithm excess scaling;
begin
    preprocess;                                   → same as in preflow push
    Δ := 2^⌈log U⌉;                               → need Δ ≥ U at start
    while Δ ≥ 1 do
    begin (Δ-scaling phase)
        while the network contains a node i with a large excess do   e(i) ≥ Δ/2
            begin
                among all nodes with a large excess, select a node i with
                the smallest distance label;      → pick node closest to t
                perform push/relabel(i) while ensuring that no node excess exceeds Δ;
            end;                                  → δ = min{e(i), r_ij, Δ - e(j)}
        Δ := Δ/2;
    end;
end;
```

Figure 7.18 Excess scaling algorithm.

We want to push from a large excess node with the smallest $d(i)$ so as to get as much flow toward t as quickly as possible. At the same time, we do not want to just "kick the can down the road", i.e., push all excess from a node to another node just down the hill. The idea is to keep all the excesses in check, while still working on the large excess nodes.

Illustration of Δ-scaling phase



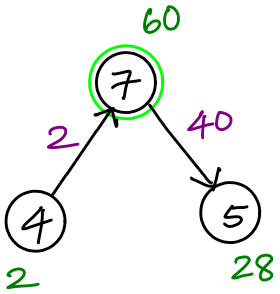


$\Delta = 64$ (64-scaling phase)

7 is a large excess node. We push along (7, 4) first. We push

$\delta = \min\{62, 2, 64\}$. ($64 = \Delta - e(4)$)

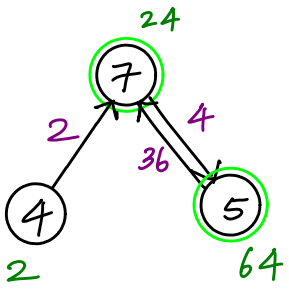
$e(7), r_{74}, \Delta - e(4)$



Then push $\delta = \min\{e(7), r_{75}, \Delta - e(5)\}$

$= \min\{60, 40, 36\}$ → $64-28$

$= 36$ along $(7,5)$.



Node 5 is now a large excess node (in the 64-scaling phase), but Node 7 is not.

In another variation, we choose a large excess node with the largest (instead of smallest) distance label. Intuitively, the idea is that we have to push flow out of excess nodes any way, and have to deal with nodes at all height levels, i.e., distance labels, eventually. We might as well start farther away from t.

The book has many more variants of these algorithms, along with details of implementation. We talk next about min-cost flow...