MAXIMUM FLOW: THE PREFLOW/PUSH METHOD

Goldberg and Tarjan (87)

Motivation

- Find a maximal flow over a directed graph
- Source and sink vertices are given

Some definitions (just a reminder)

- A flow network ➔ G = (V,E) a directed graph
- Two vertices {s, t} the source and the sink
- Each edge $(u,v) \in E$ has some positive capacity c(u,v), if $(u,v) \notin E$ $c(u,v) = 0$.
- The flow function *f* maps a value for each edge where:
  - $f(u,v) \le c(u,v)$
  - $f(v,u) = - f(u,v)$ (skew symmetry)
- Saturated edge (u,v) ⇔ $c(u,v) = f(u,v)$

Some definitions, contd.

- $r(u, v) = c(u,v) – f(u,v)$
- Residual graph R(V, E`) where E` is all the edges (u,v) where $r(u,v) \ge 0$
- Augmenting path p is a path from to the source to the sink over the residual graph
- *f* is a maxflow ⇔ there is no augmenting path

Just as Dinic but…

- We use the residual network
- We don't look for augmenting paths
- Instead we saturate all outgoing edges of the source and strive to make this "preflow" reach the sink
- Otherwise we'll have to flow it back.

Preflow

- **Flow constrains**:

$\forall (u,v) \in V \qquad f(u,v) \le c(u,v)$

$\forall (u,v) \in V \qquad f(v,u) = -f(v,u)$

$\forall v \in V-\{s\} \qquad \sum f(u,v) \ge 0$

- Every vertex *v* may keep some "excess" flow *e(v)* inside the vertex

Excess handeling

- We strive to push this excess toward the sink
- If the sink is not reachable on the residual network the algorithm pushes the excess toward the source
- When no vertices with $e(v) > 0$ are left the algorithm halts, and the resulting flow (!) is the max –flow

Valid distance labeling

- A mapping function $d(v) \rightarrow \mathbb{N} + \{ \infty \}$
- $d(s) = n$, $d(t) = 0$
- $r(u,v) > 0 \rightarrow d(u) \le d(v)+1$
- $d(v) < n \rightarrow$ d(v) is the lower bound on the distance from v to the sink (residual graph)
  - Let p= $v, v_1, v_2, v_{3\ldots} v_k, t$ be the s.p $v \rightarrow t$
  - $d(v) \le d(v_1) + 1 \le d(v_2) + 2 \ldots\ldots \le d(t) + k = k$
- Same way $d(v) \ge n \rightarrow$ d(v) -n is the lower bound on the distance from v to the source

Active vertex

- Active vertex:
  - $v \in V$-{s,t} is active if
    - d(v) < ∞
    - e(v) > 0
      - Eventually, I'll show that d(v) is always finite and therefore only the e(v) > 0 part is relevant

Basic operations

- Applied on active vertices only

- Push (u,v)
  - Requires: r(u,v) >0, d(u)=d(v)+1
  - Action:
    - δ = min(e(v) , r(u,v))
    - f(u,v) += δ, f(v,u) -= δ
    - e(u) -= δ , e(v) += δ

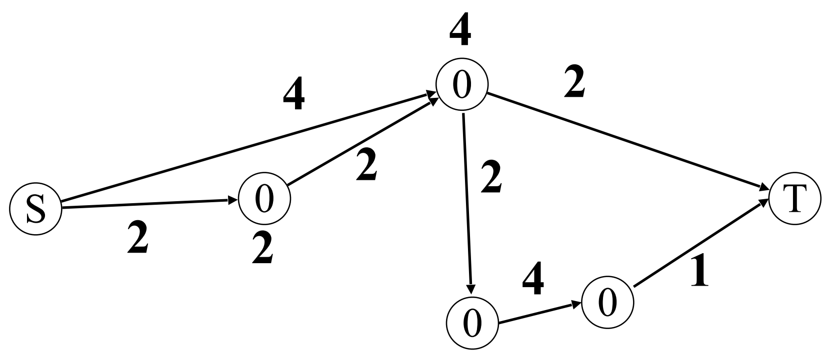
Basic operations , contd.

- Relabel (u)
  - Requires: $\forall (u,v) \in V\ r(u,v) > 0 \rightarrow \quad d(u) \le d(v)$
  - Action:
    - $d(u) = \min \{ d(v) + 1 \mid r(u,v) > 0 \}$
- One of the basic operations is applicable on a active vertex:
  - PUSH: Any residual edge (u,v) with $d(u) = d(v) + 1$
  - Otherwise: $d(u) \le d(v)$ for all residual edges, allows relabel

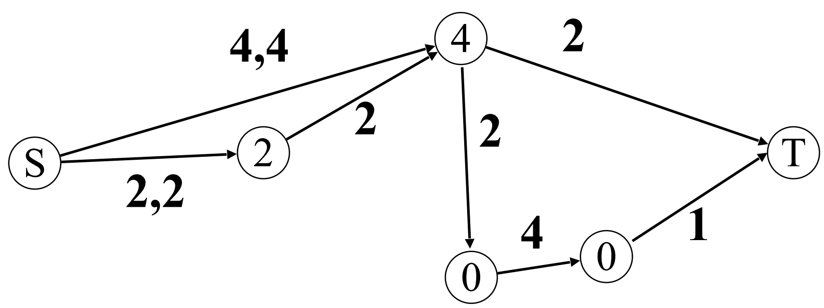
The algorithm

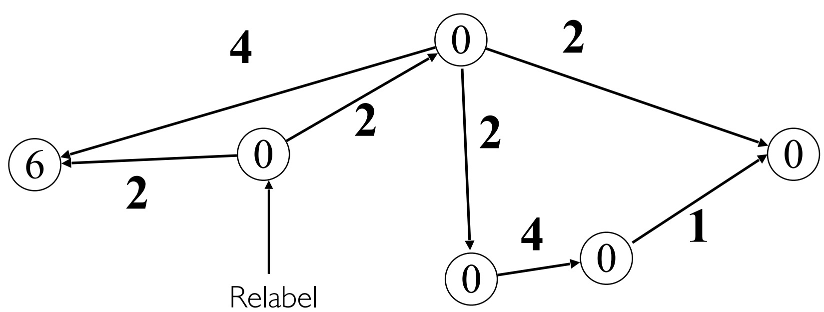
- Initialize: $d(s) = n$, $v \in V$-{s} $d(v) = 0$
- Saturate the outgoing edges of s
- While there are active vertices apply one of the basic actions on the vertex
- Simple, isn't it?
- Let's see an example

Example - Saturate all source edges

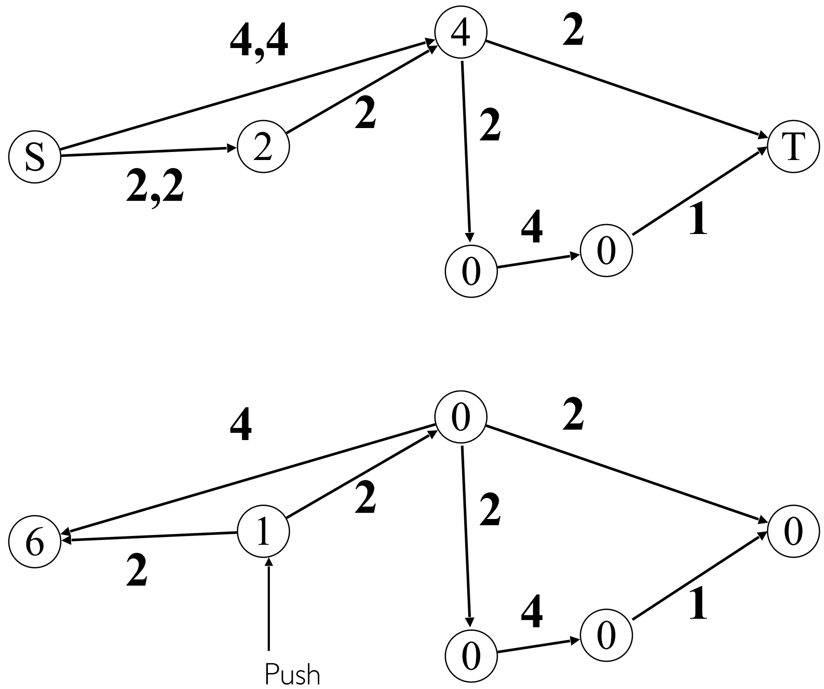


Example – contd.

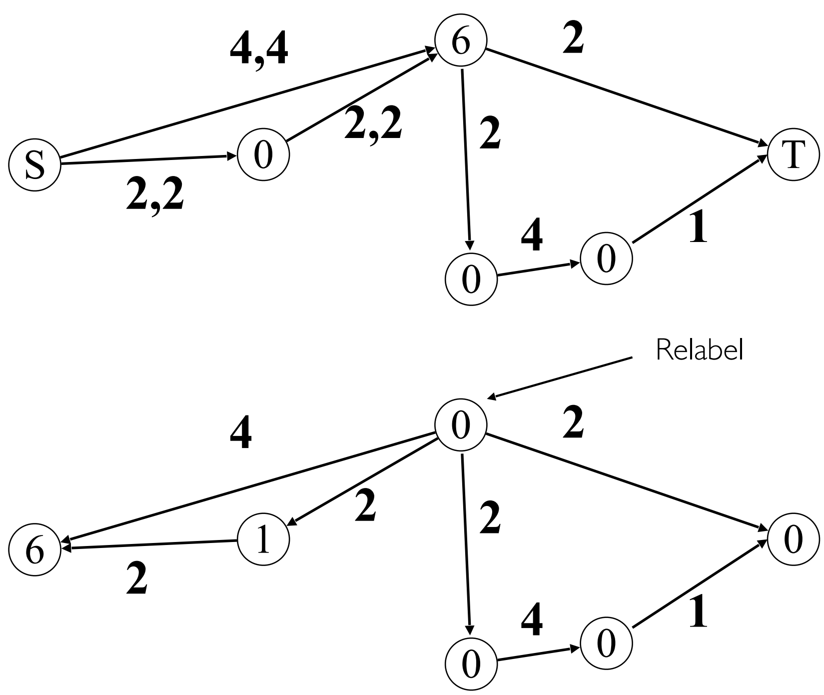




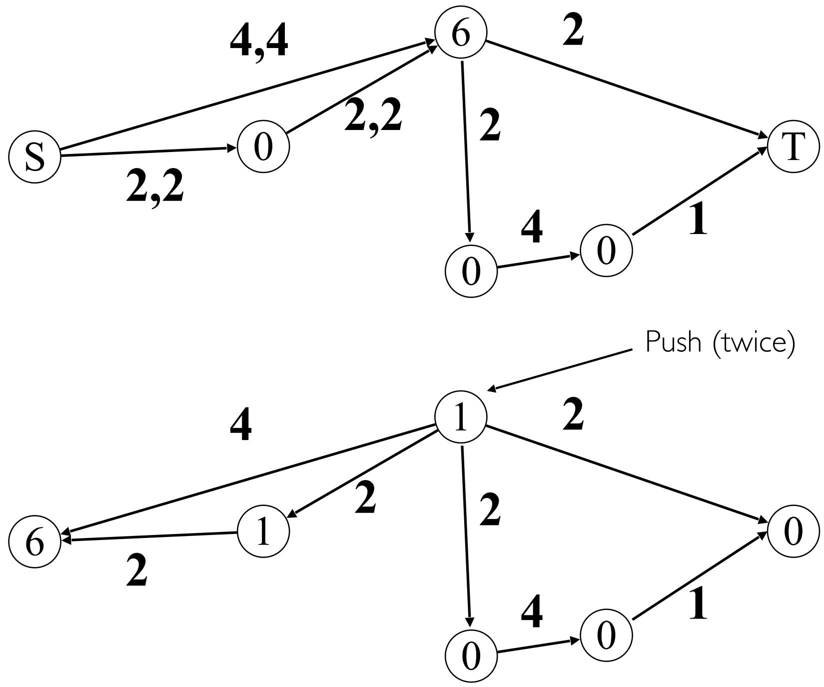
Example – contd.



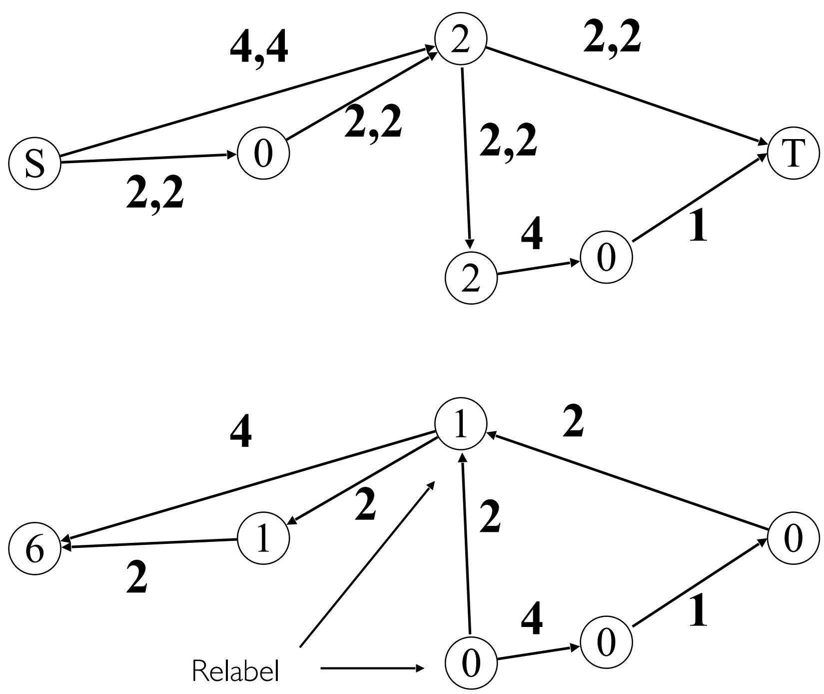
Example – contd.



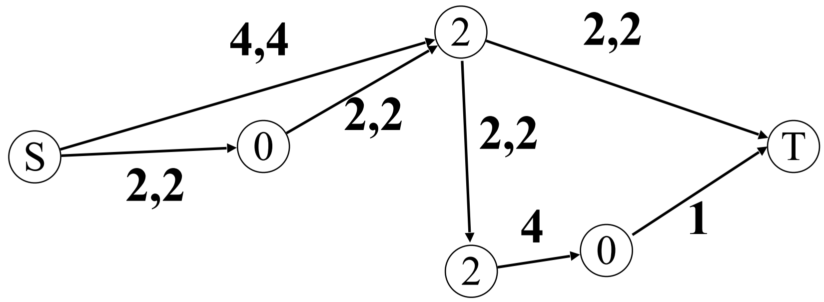
Example – contd.

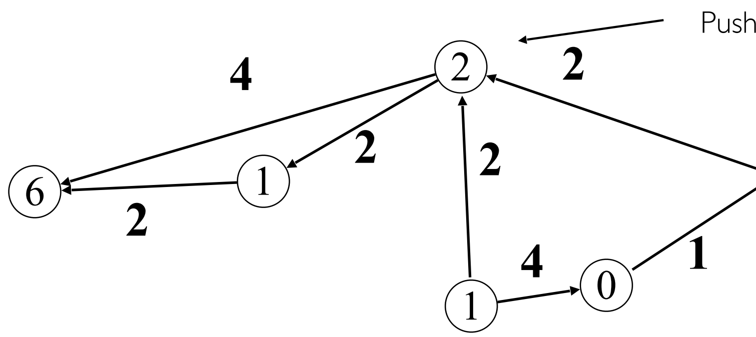


Example – contd.

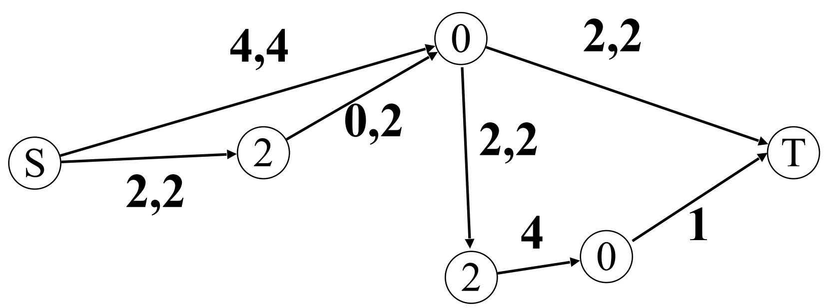


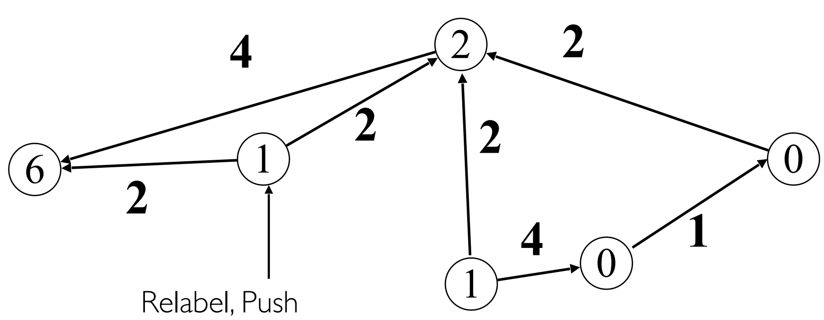
Example – contd.



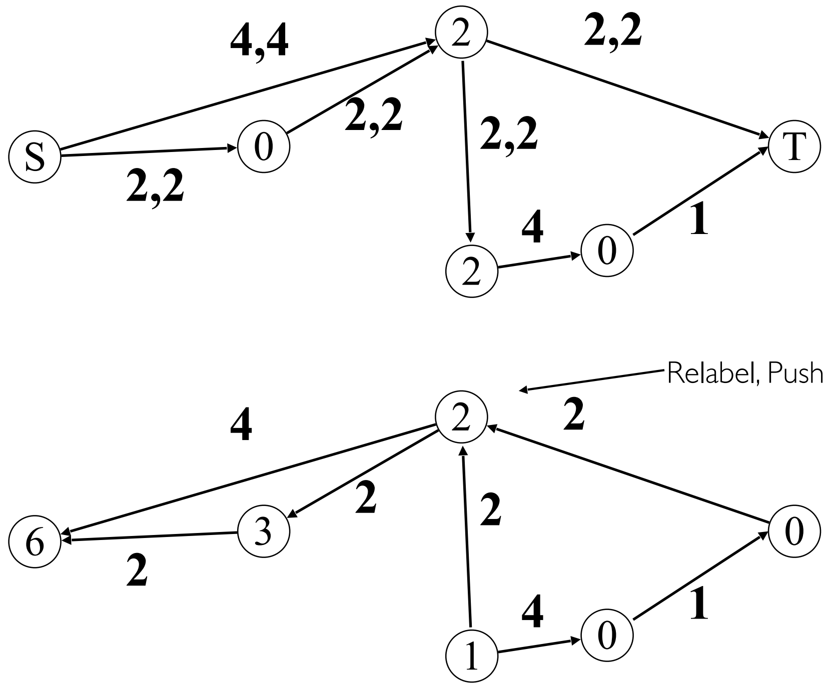


Example – contd.

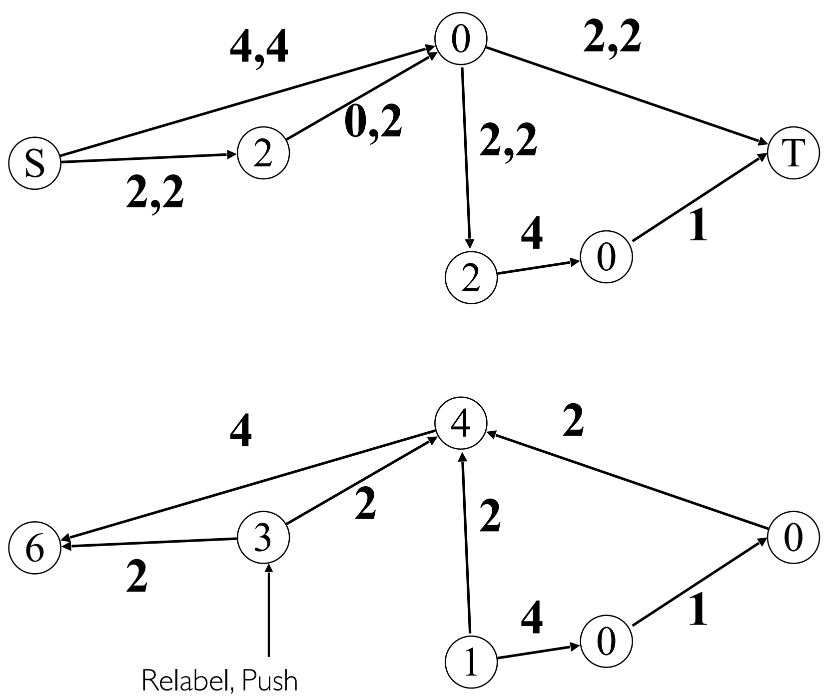




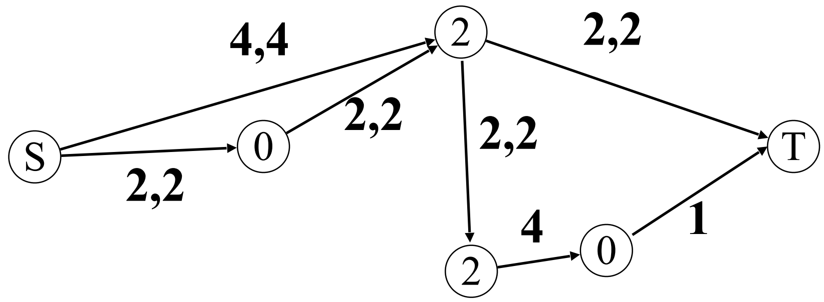
Example – contd.

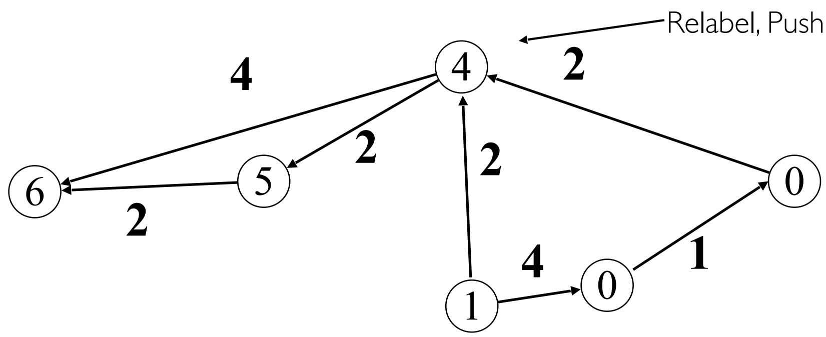


Example – contd.

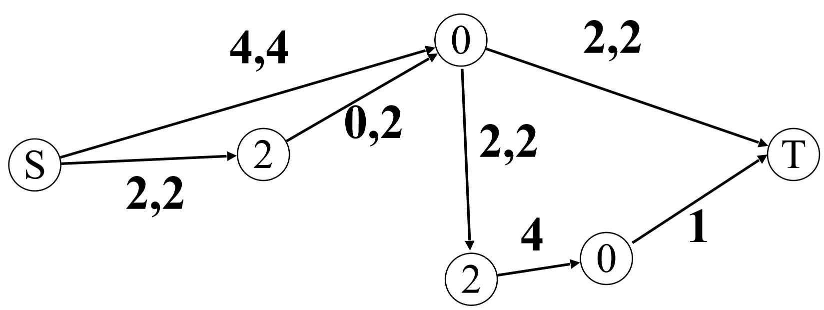


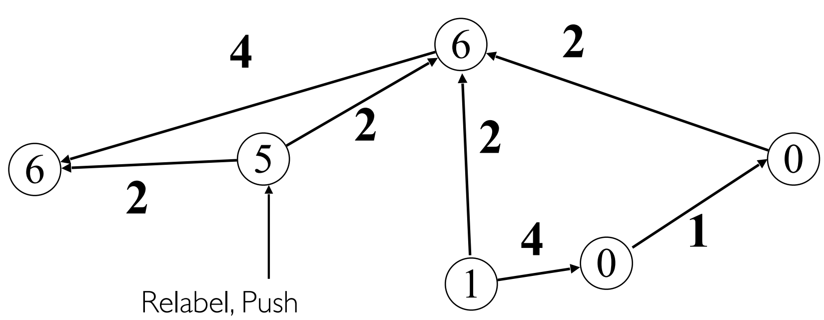
Example – contd.



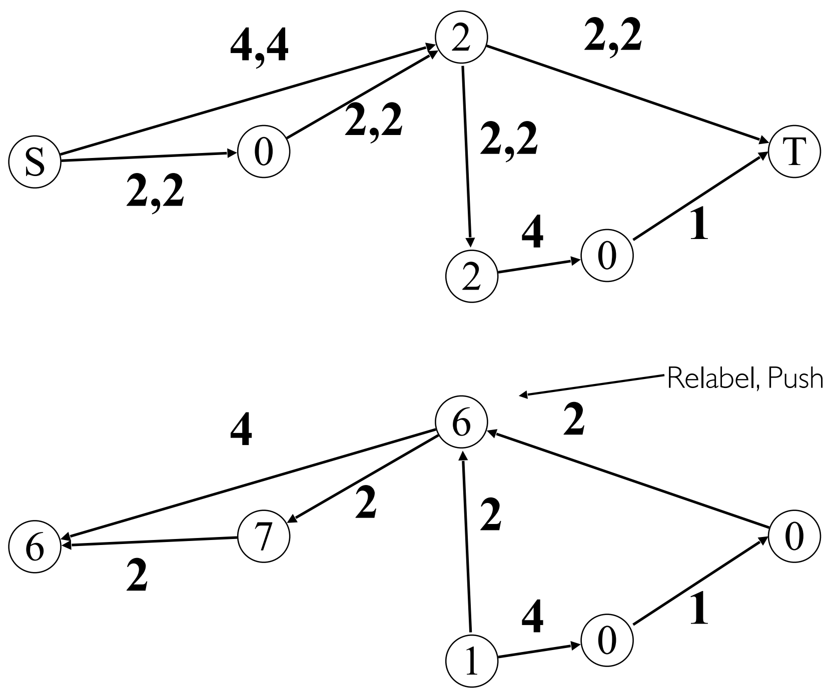


Example – contd.

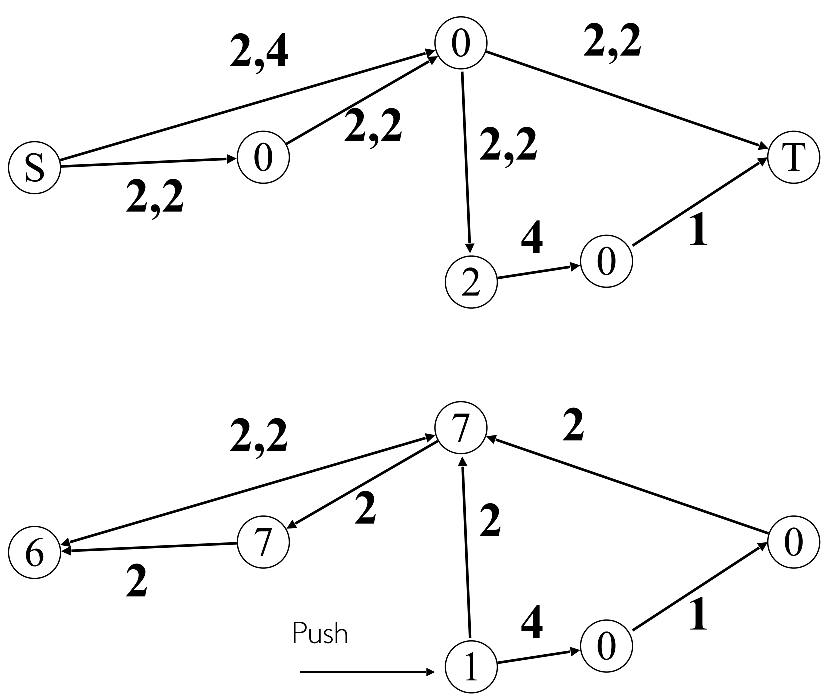




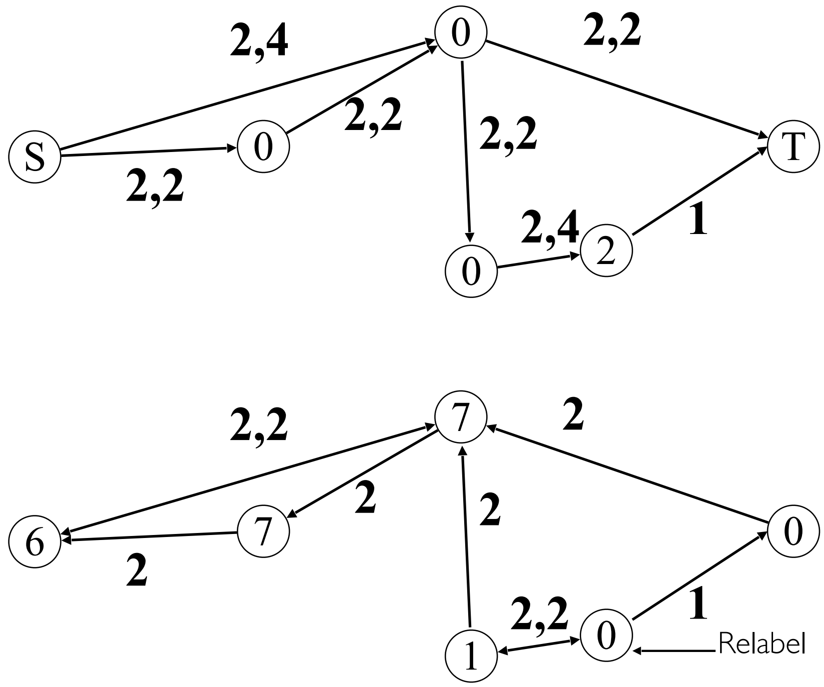
Example – contd.



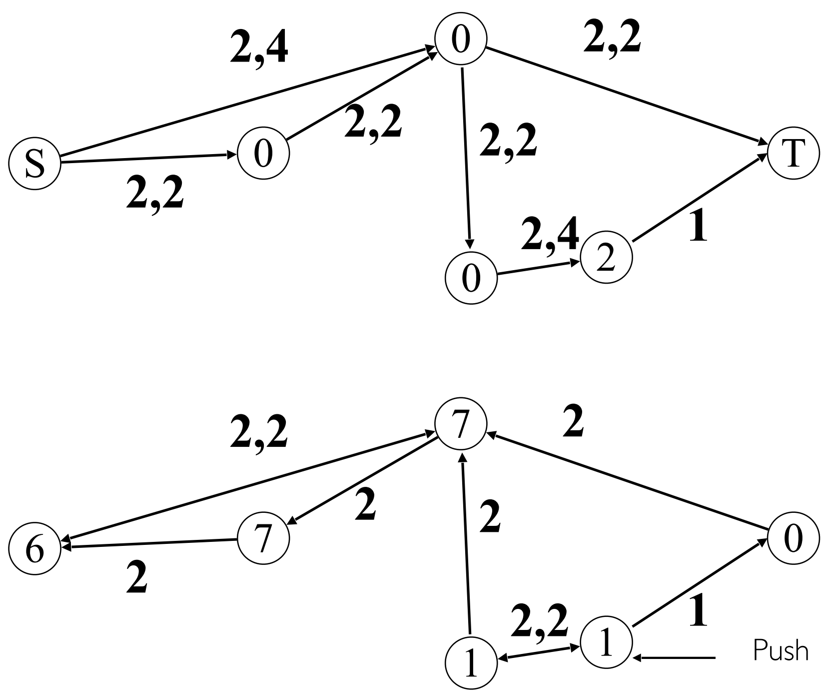
Example – contd.



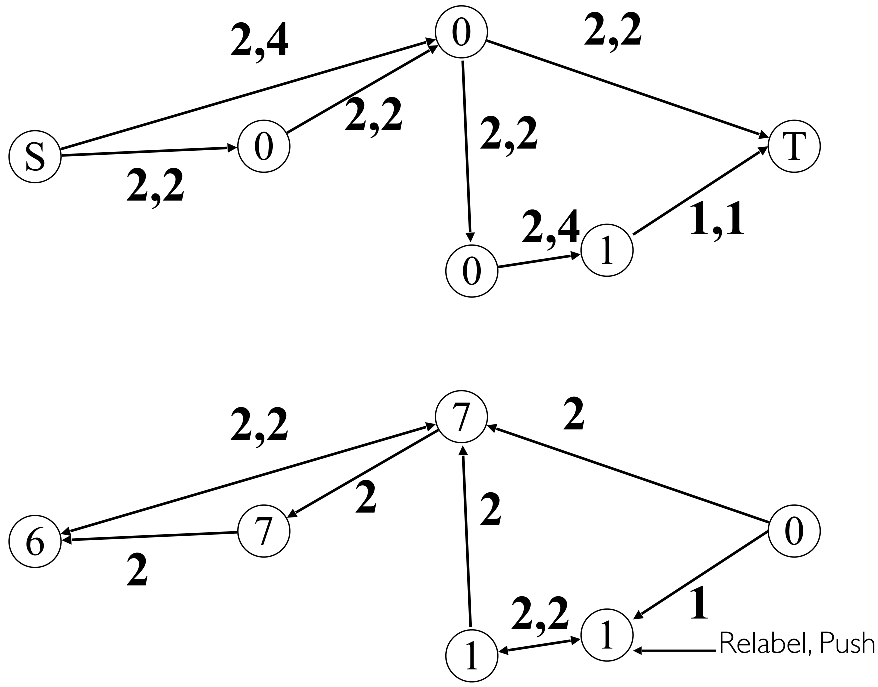
Example – contd.



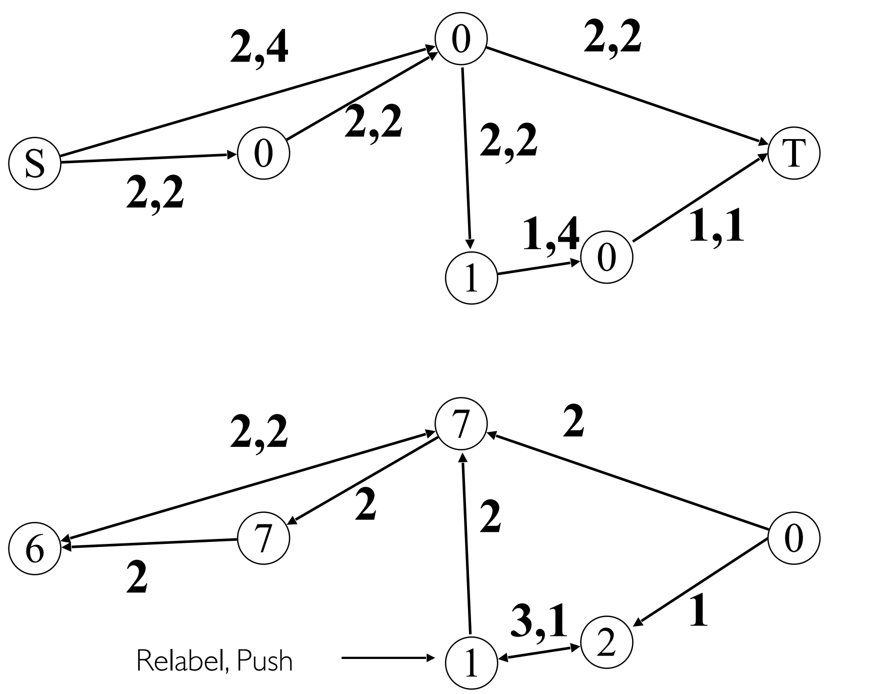
Example – contd.



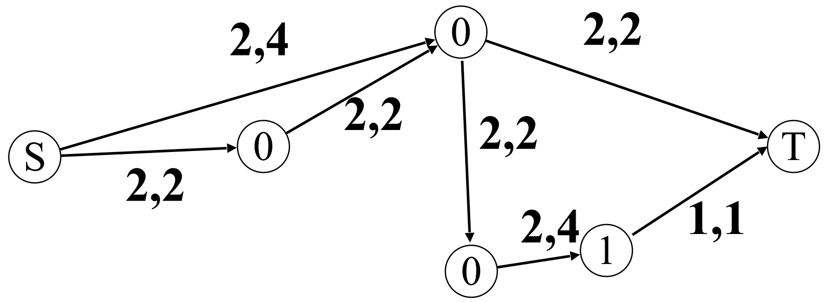
Example – contd.

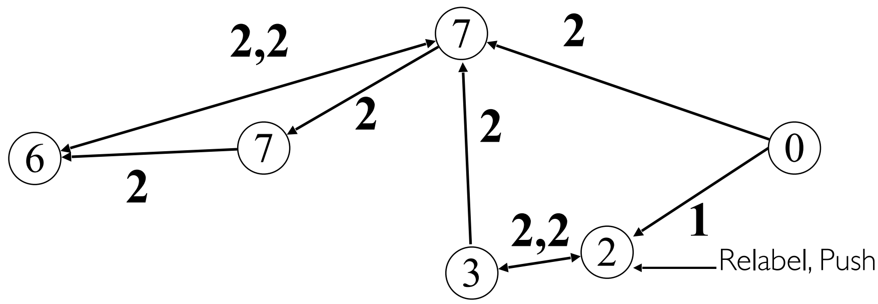


Example – contd.

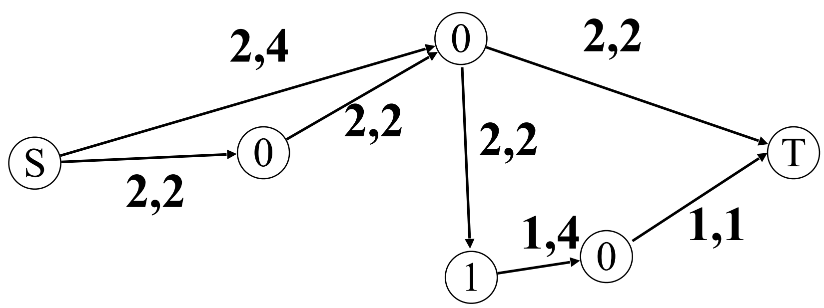


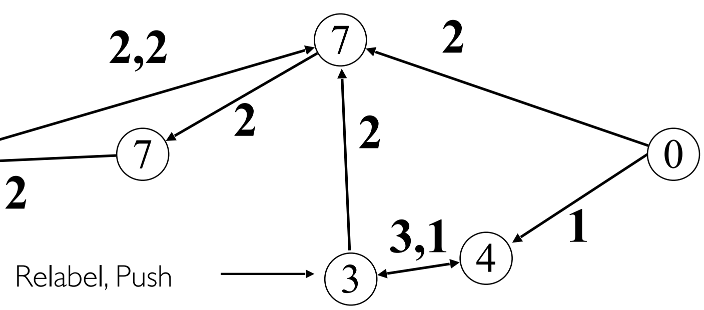
Example – contd.



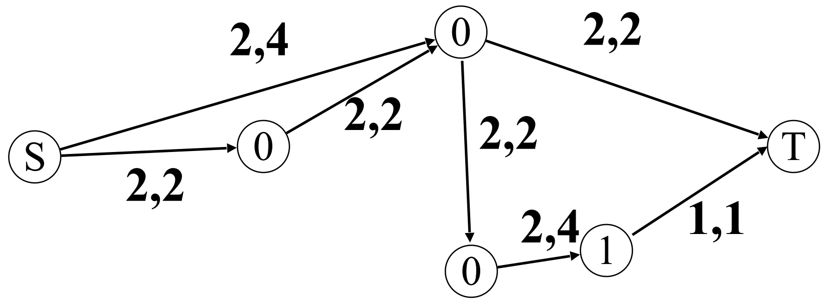


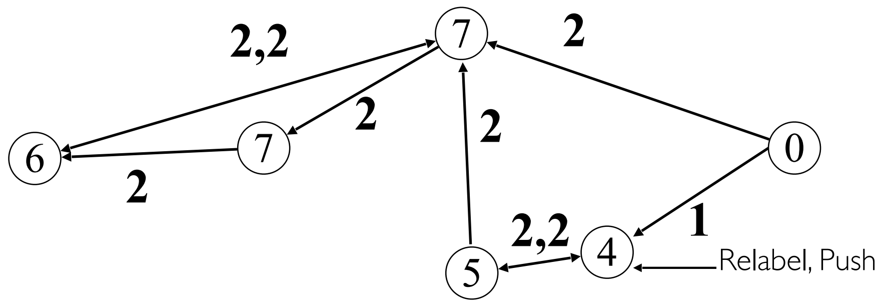
Example – contd.



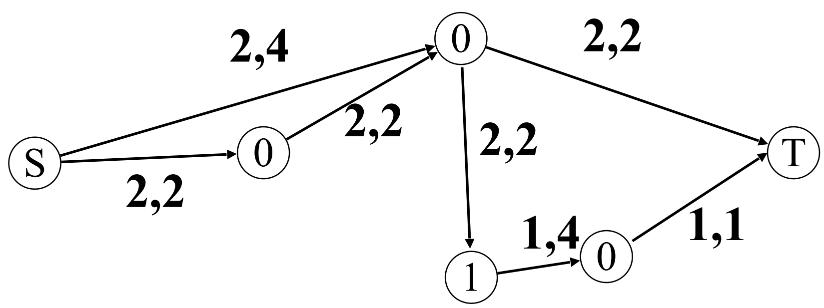


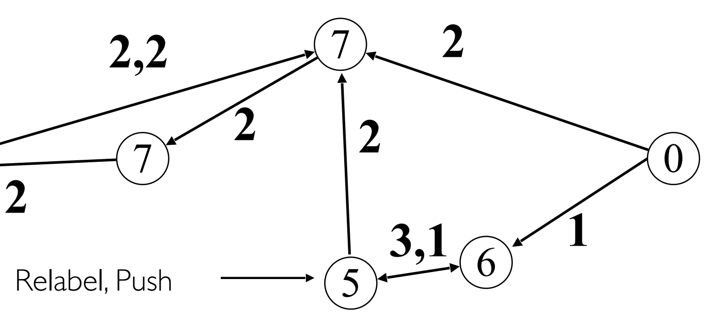
Example – contd.



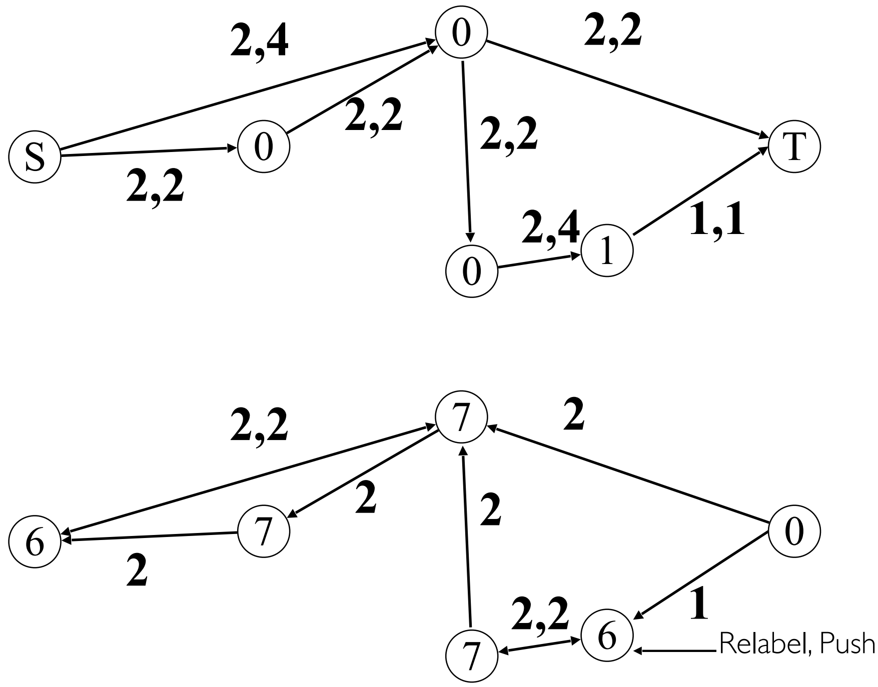


Example – contd.

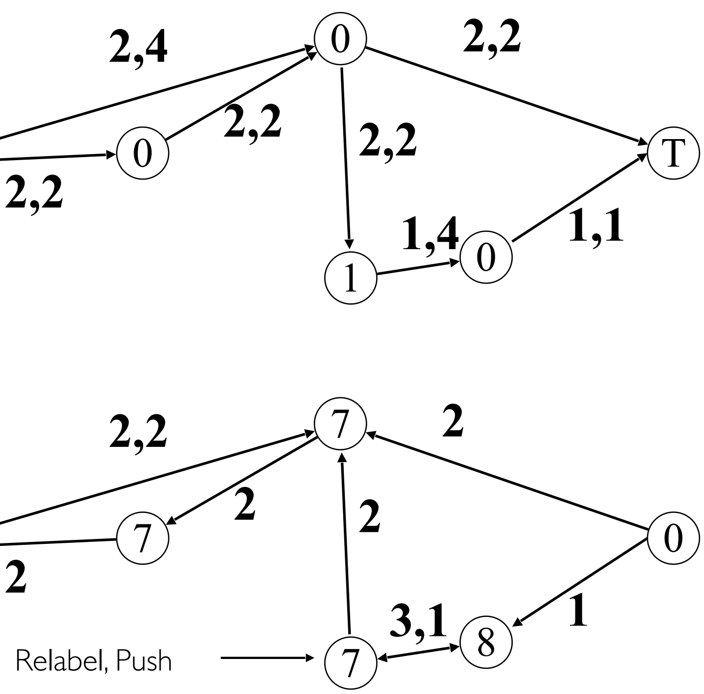




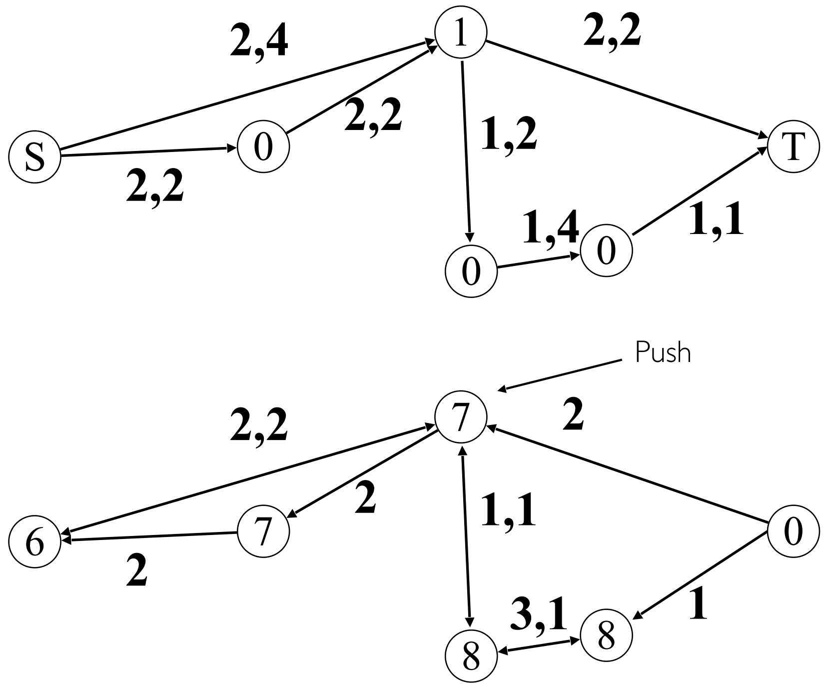
Example – contd.



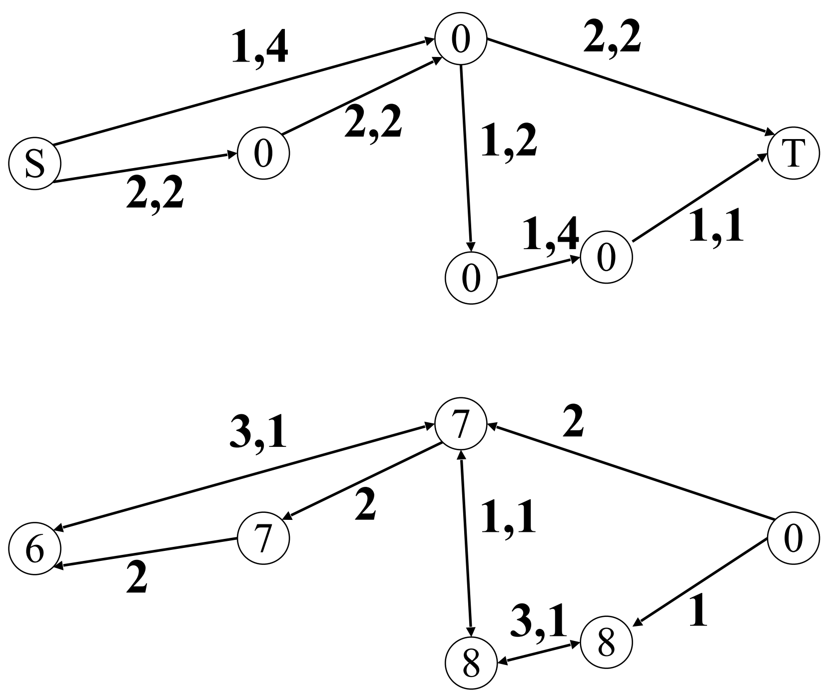
Example – contd.



Example – contd.



Example – contd.



Correctness

- For an active vertex v, there must be a residual path v→…→s
  - Otherwise, no flow enters v, and it is clearly not active
- So, every active vertex v has an outgoing edge
  - And this means, that if the distance labels are valid, v can be either relabled or pushed

Correctness of d(v)

- $r(u,v) > 0 \rightarrow \quad d(u) \le d(v)+1$
- By induction on the basic operations
- We begin with a valid labeling
- Relabel keeps the invariant
  - By definition for the outgoing edges
  - Only grows, so holds for all the incoming ones
- Push
  - Can only introduce (v,u) – back edge, but since $d(u) = d(v)+1$ the correctness is kept

Correctness of d(v) – contd.

- For any active vertex v, $d(v) < 2n$
  - Let $p = v, v_1, v_2, v_{3\ldots}\ v_k, s$ be a path $v \rightarrow s$
  - $d(v) \le d(v_1) + 1 \le d(v_2) + 2 \ldots\ldots \le d(s) + k = n+k$
  - The length of the path is $\le n-1$, so $k \le n-1$
  - ➔ $d(v) \le 2n-1$
- For a non active, it is kept when the vertex is active, or it is 0.
- ➔ d(v) is finite for any v during the run of the algorithm

Correctness contd.

- At the end, for all the vertices besides {s,t} no excess is left in the vertices
  - ➔ Our preflow is a flow
- The sink is not reachable from the source on the augmenting graph
  - Let $p = s, v_1, v_2, v_{3\ldots}\ v_k, t$ be a path $s \rightarrow t$
  - Notice $k \le n-2$
  - $n = d(s) \le d(v_1) + 1 \le d(v_2) + 2 \ldots \le d(t) + k+1 = k+1$
  - Implies that $n \le k+1$ in contradiction to above

Complexity analysis

- $d(v) \le 2n-1$, and can only grow during the execution, and only by relabel operation
- n-2 vertices are relabeled
  - → At most $(n-2)(2n-1) < 2n^2 = O(n^2)$ relabels.

Complexity analysis –

Saturating push

- First saturating push $1 \le d(u) + d(v)$
- Last saturating push $d(u) + d(v) \le 4n - 3$
- Must grow by 2 between 2 adjutant pushes
- → 2n-1 saturating pushes on (u,v) [or (v,u)].
- →$m(2n-1) = O(nm)$ saturating pushes at all

Complexity analysis –

Non Saturating push

- $\Phi = \sum d(v) \mid v$ is active
  - Φ is 0 in the beginning and in the end
- A saturating push increases Φ by $\leq 2n-1$
  - All saturating pushes worth $O(mn^2)$
- All relabelings increase Φ by $\leq (2n-1)(n-2)$
- Each non saturating push decreases Φ by at least 1
- There are up to $O(mn^2)$ non saturating pushes

Complexity analysis

- Any reasonable sequential implementation will provide us a polynomial algorithm
  - How much a relabel operation cost?
  - How much a push operation cost?
  - How much cost to hold the active vertices?
- How will we improve this?

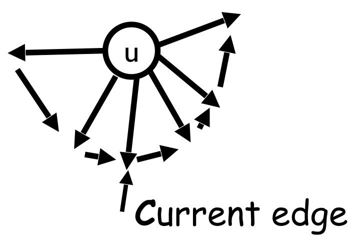
Implementation

- For an edge in {e = (u,v) | (u,v) ϵE or (v,u) ϵE } hold a struct of 3 values:
  - c(u,v) & c(v,u)
  - f(u,v)
- For a vertex v ϵV we hold a list of all incident edges in some fixed order
  - Each edge appears in two lists.
- We also hold an "current edge" pointer for each vertex

Implementation – contd.

Admissible arc in the residual graph

d(u) = d(v) + 1



- Push/relabel operation:
  - If the current edge is admissible perform push on the current edge and return
  - If the current edge is the last one, relabel the node and set the current edge to the first one in the list
  - Otherwise, just advance the current edge to the next one in line

Is this correct?

- When we relabel a node we'll have no admissible edges:
  - Any of the other edges (u,v) wasn't admissible before and d(v) can only grow
  - If it had $r(u,v) = 0$ before and now it is positive we had $d(u) = d(v) + 1$, and so $d(v) < d(u)$
- Hold a list of all active nodes – O(1) extra cost per push/relabel operation

And it costs

- Number of relabelings – 2n-1 per vertex
- Each relabeling causes a pass over all the edges of the vertex – m for all the vertices
- Besides that we have o(1) per push performed (recall $O(mn^2)$ non saturating pushes).

- Total – $O(mn + mn^2) = O(n^2m)$

Use FIFO ordering

- discharge(v) = perform push/relabel(v) until e(v) = 0 or the vertex is relabled
- Hold two queues – one is the active, the other is for the next iteration
- Iteration:
  - While the active queue is not empty
    - Discharge the vertex in the front
  - Any vertex that becomes active is inserted to the other queue

Use FIFO ordering - complexity

- Φ = max d(v) | v is active
  - Φ is 0 in the beginning and in the end
- A relabel during an iteration can increase Φ by the delta of the relabel or keep Φ.
- No relabel during an iteration will cause Φ to decrease by at least 1.
- There are up to $2n^2$ relabels during the run - $2n^2$ iteration of the first kind.

Use FIFO ordering - complexity

- As each node can add up to 2n-1 to Φ, Φ grows by up to (2n-1)(n-2) during the entire run
- $O(2n^2)$ iteration of the second kind as well
- $2n^2 + 2n^2$ iterations → $O(n^2)$ iterations
- Each iteration will have up to 1 non saturating push per vertex
- $O(n^3)$ non saturating pushes at all
- →$O(n^3)$ total run time

Dynamic tree operations

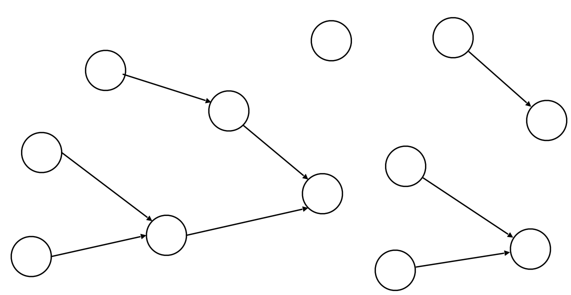
- FindRoot(v)
- FindSize(v)
- FindValue(v)
- FindMin(v)
- ChangeValue(v, delta)
- Link(v,w) – v becomes the child of w, must be a root before that.
- Cut(v) – cuts the link between v and its' parent

The algorithm using dynamic trees

- All that said before holds, but we also add dynamic trees
- Initially every vertex is a one node dynamic tree.
- The edges (u,v) that are eligible to be in the trees are those that hold
  - d(u) = d(v)+1 (admissible)
  - r(u,v) > 0
  - (u,v) is the "current edge" of the vertex u

The algorithm using dynamic trees

- Yet, not all eligible edges are tree edges
- If an edge (u,v) is in the tree v= p(u) and value(v) = r(u,v)
- For the roots of the trees value(v) = ∞



The Send operation

- Requires: u is active
- Action:
  - while FindRoot(u) != u && e(u) > 0
    - δ = min(e(u) , FindValue(FindMin(u)))
    - ChangeValue(u, -δ)
    - while FindValue(FindMin(u)) == 0
      - v = FindMin(u)
      - Cut(v)
      - ChangeValue(v, ∞)

Add the maximal possible flow on the path to the path to the root

Remove all the edges saturated by the addition

The Send operation – contd.

- The send operation will either cause e(v) become 0, or it will make it the root
- This implies v will not be active unless it is a root of a tree

And the algorithm is…

- As before – two queues etc.
- Discharge the vertex in the front
  - Use tree-Push/ Relabel instead of Push/ Relabel

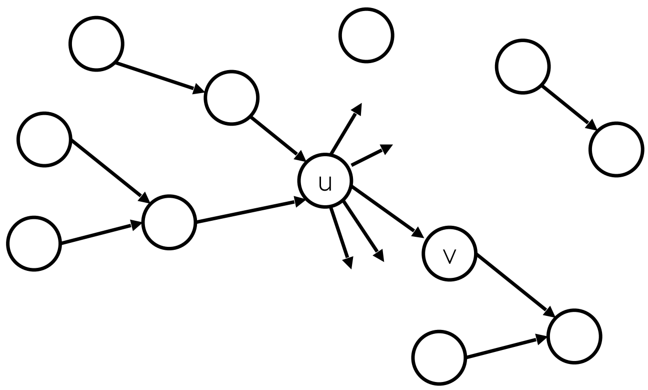
- We'll set some constant – k – to be the upper limit of the size of a tree during the algorithm execution

Tree-Push/Relabel operation

- Applied on an active vertex u
- If the current edge (u,v) is addmissible
  - If (FindSize(u) + FindSize(v) ≤ k)
    - Link (u,v), Send (u)
  - Else
    - Push (u,v), Send (v)
- Else
  - Advance the current edge
  - If (u,v) was the last one cut all the children of u & Relabel(u)

Tree-Push/Relabel operation – contd.

- The operation insures that all vertices with positive excess are the roots of some tree



Why is this correct?

- Since inside the tree the d values strictly growing no linking inside the tree can occur
- A vertex v will not have positive excess unless it is a root of a tree
  - Link operation is valid if required
- The rest is just as before

Complexity Tree-Push/Relabel

- Each dynamic tree operation is O(log(k))
- Each Tree-Push/Relabel operation takes
  - O(1) opearions
  - O(1) tree opearions
  - Relabeling time
  - O(1) tree operations per cut performed

Complexity – contd.

- The total relabeling time is O(mn)
- Total number of cut operations O(mn):
  - Due to relabeling – O(mn)
  - Due to saturating push – O(mn)
- Total number of link operations < Number of cut operations + n → O(mn)

- So we reach O(mn) tree operations + O(1) tree operations per vertex entering Q.

How many times will a vertex become active?

- Due to increase of d(v) – $O(n^2)$
- Due to Send operation, e(v) grows from 0
  - Any cut performed – total (mn)
  - One more per send operation
    - Link case – O(mn)
    - Push case - Need to split to saturating and not

    - There can be up to O(mn) such saturating pushes

Non saturating push analysis

- For a non saturating push (u,v) either Tu or Tv must be large - contain more than k/2 vertices
- For a single iteration, only 1 such push is possible per vertex
- Charge it to the link or cut creating the large tree if it did not exist at the beginning of the phase – O(mn)
- Otherwise charge it to the tree itself

Non saturating push analysis – contd.

- There are up to $2n/k$ large trees at the beginning of the iteration
- Total of $O(n^3/k)$ for all $O(n^2)$ iterations

- ➔ A vertex enters the Queue $O(mn + n^3/k)$ times due to a non saturating push

Total complexity

- Total of $O(mn + n^3/k)$ tree operations with tree size of k.
- We reach total of $O(\log(k)\ (mn + n^3/k))$ runtime complexity
- Choose $k = n^2/m$

- We reach $O(\log(n^2/m)\ (mn))$ runtime complexity

Conclusion

- We've seen an algorithm that finds a max flow over a network with $O(\log(n^2/m)\ (mn))$ runtime complexity
- The algorithm uses a different approach – a preflow instead of flow
- While providing same asymptotical result as Dinic, has better coefficients and therefore often used in time demanding applications

Questions?

- Thank you for listening