Lecture 5. Max-Flow Algorithms

History of Worst-Case Running Times

Year	Discoverer	Method	Asymptotic Time
1951	Dantzig	Simplex	$m\,n^2\,C$ †
1955	Ford, Fulkerson	Augmenting path	$m\,n\,C$ †
1970	Edmonds-Karp	Shortest path	$m^2\,n$
1970	Edmonds-Karp	Fattest path	$m \log U\,(m \log n)$ †
1970	Dinitz	Improved shortest path	$m\,n^2$
1972	Edmonds-Karp, Dinitz	Capacity scaling	$m^2 \log C$ †
1973	Dinitz-Gabow	Improved capacity scaling	$m\,n \log C$ †
1974	Karzanov	Preflow-push	n^3
1983	Sleator-Tarjan	Dynamic trees	$m\,n \log n$
1986	Goldberg-Tarjan	FIFO preflow-push	$m\,n \log (n^2 / m)$
. . .	. . .	. . .	. . .
2013	Orlin + KTR	Contraction	mn

† Edge capacities are between 1 and C.

Assumptions

Flow network $D = (V, A;\ c)$

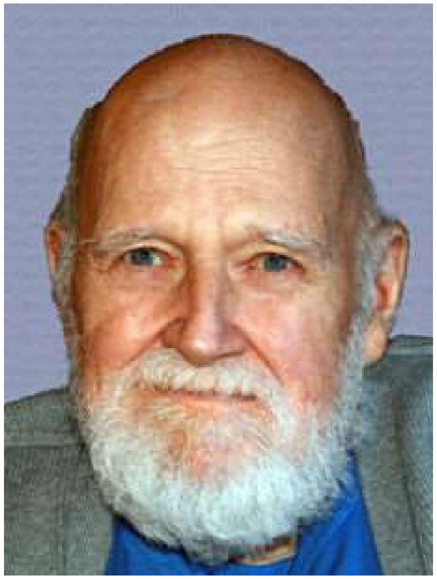
- Simple, bidirected
- c is nonnegative
- Every node is on an s-t path
- No uncapacitated s-t path

Outline

- Augmenting flow by single path-flow
- Augmenting flow by blocking-flow
- Preflow push on arcs

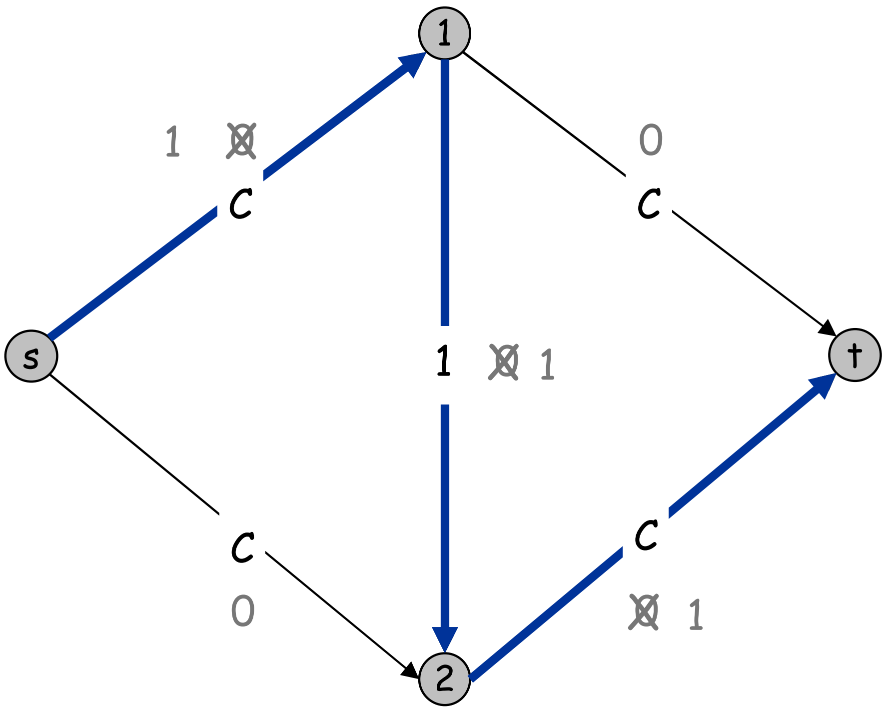
1. Augmenting Flow by Single Path-Flow

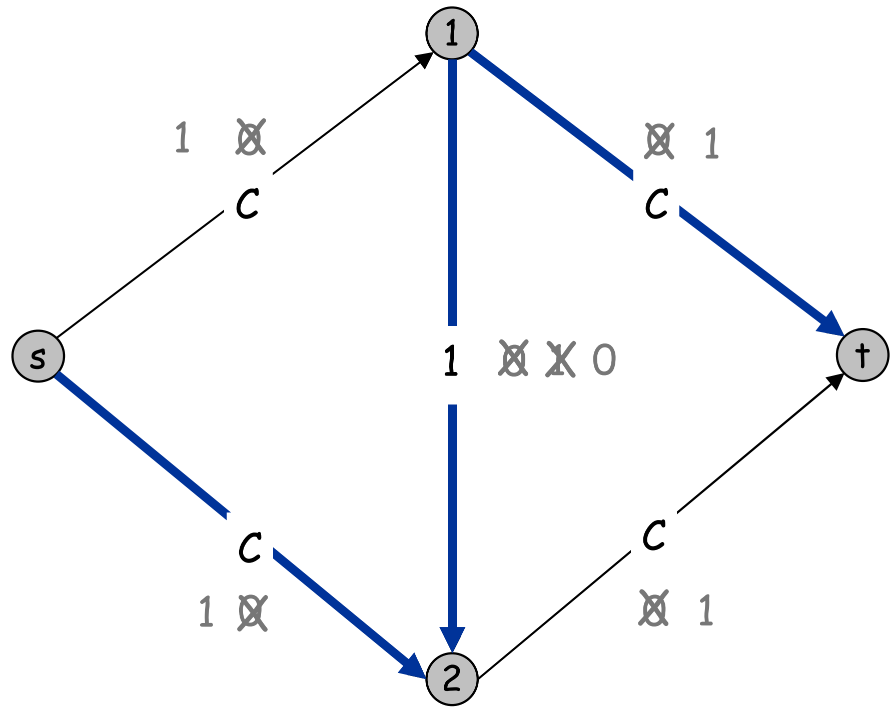
Ford-Fulkerson Method



Ford-Fulkerson Method

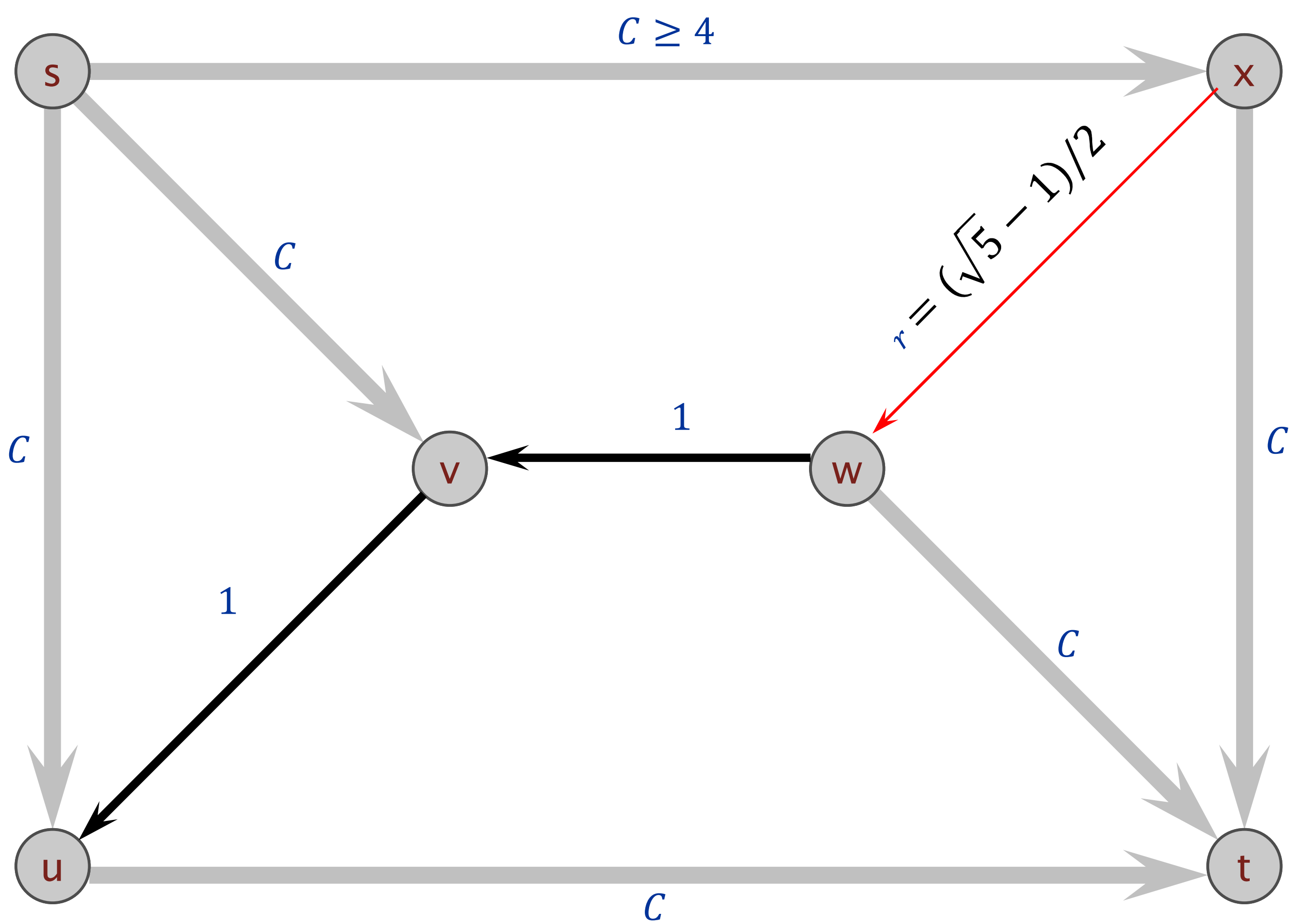
Ford-Fulkerson (D, s, t, c)
$f \leftarrow 0$
while (there exists f-augmenting path P) $f \leftarrow f \oplus P$
return f





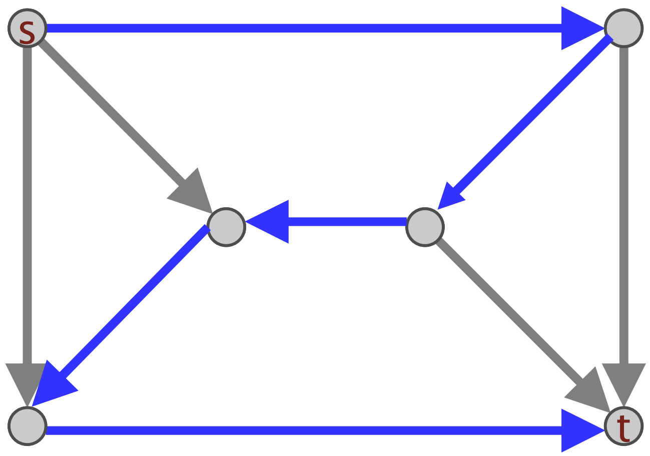
Zwick's flow network

Theorem. The Ford-Fulkerson algorithm may not terminate; moreover, it may converge to a value not equal to the value of the maximum flow.

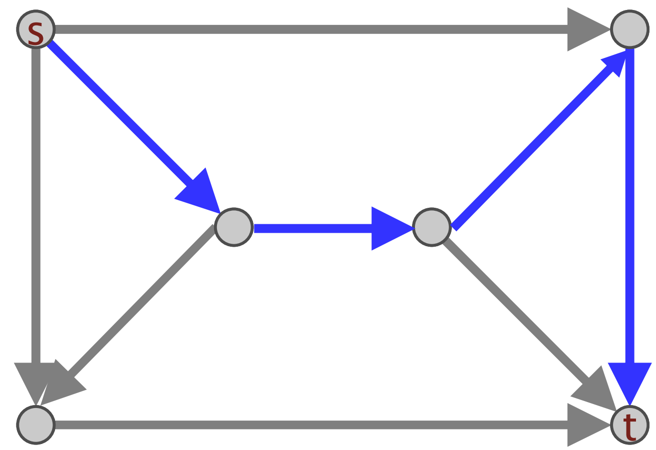


Valid Ford-Fulkerson sequence

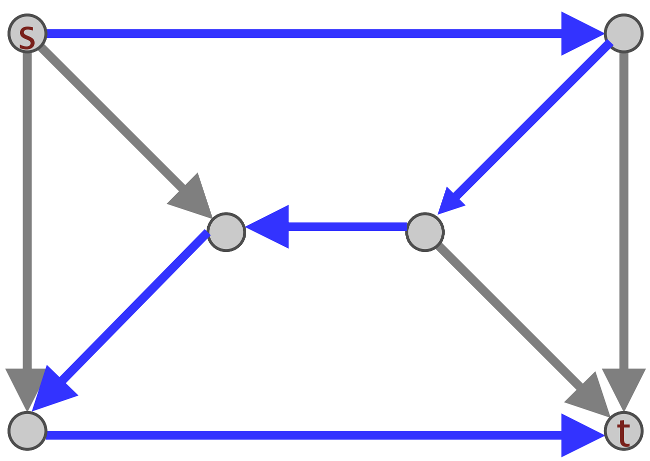
Iteration $4k-3$: r^{2k-1}



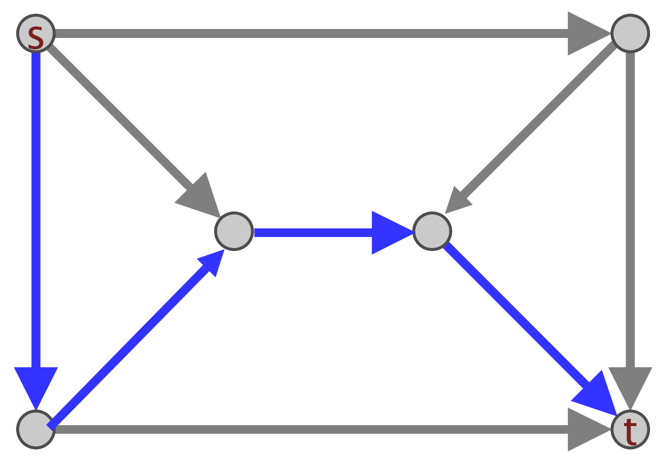
Iteration $4k-2$: r^{2k-1}



Iteration $4k-2$: r^{2k}



Iteration $4k$: r^{2k}



Choosing good augmenting paths

- Can find augmenting paths efficiently.
- Few iterations.

Choose augmenting paths with: [Edmonds-Karp 1972, Dinitz 1970]

- Max bottleneck capacity.
- Sufficiently large bottleneck capacity.
- Fewest number of edges.

(Nearly) Widest Augmenting Path

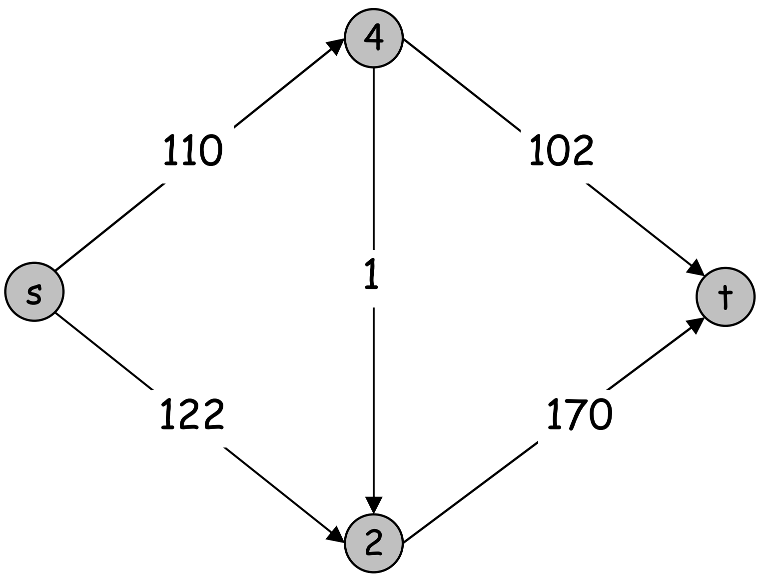
A greedy intuition

[Edmonds -Karp 1970,1972; Dinitz 1973]

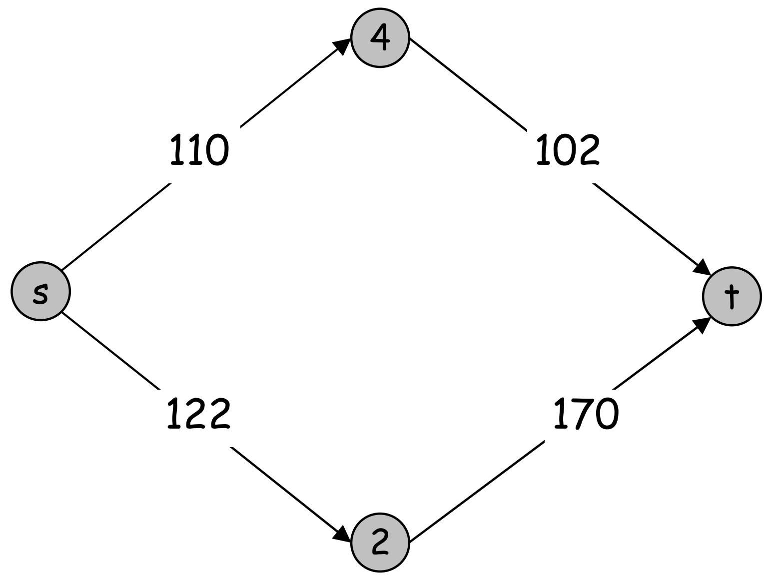
Widest augmenting path: max increase, but slow to find

Nearly widest augmenting path: nearly max increase in $O(m)$ time

- Maintain scaling parameter Δ.
- $D_f(\Delta)$: the subgraph of D_f consisting of only arcs with capacity $\geq \Delta$



D_f



$D_f(100)$

Capacity scaling

Assumption. All edge capacities are integers with absolute values at most C.

```
Scaling-Max-Flow (D, s, t, c)

   C ← max{c(a): a ∈ A}
   Δ ← smallest power of 2 greater than or equal to C

   f ← 0
   while (Δ ≥ 1)
            while (there exists augmenting path P in D_f(Δ)) do f ← f ⊕ P
            Δ ← Δ / 2
   return f
```

The outer while loop (scaling phase) repeats $1 + \lceil \log C \rceil$ times.

Correctness

Invariants.

- At the beginning of Δ-phase, $D_f(2\Delta)$ has no s-t path
- All flow and residual capacity values are integral.

Correctness. When the algorithm terminates, f is a max flow.

Pf.

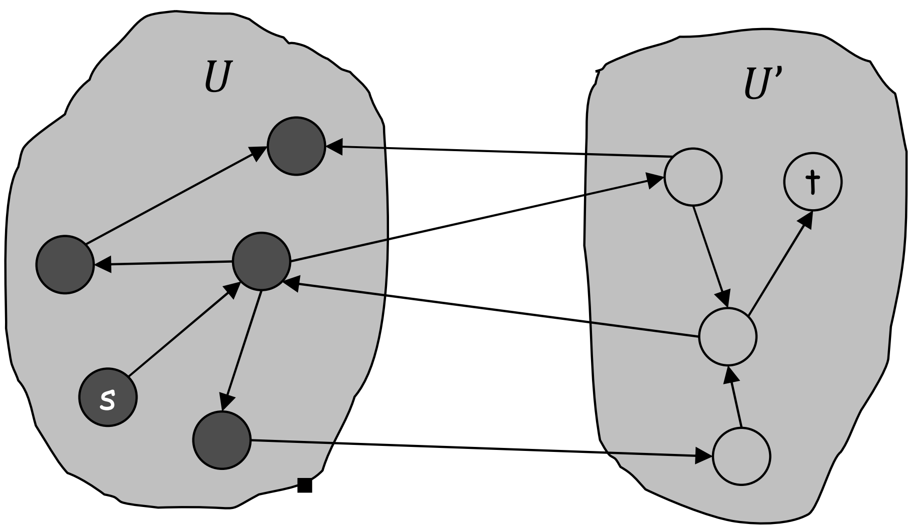
- By integrality invariant, when $\Delta = 1 \Rightarrow D_f(\Delta) = D_f$.
- Upon termination of $\Delta = 1$ phase, there are no augmenting paths. ▪

Optimality gap at the end of each phase

Claim. At the end of a Δ-scaling phase, $val(f^*) - val(f) < m\Delta$.

Pf.

- U: the set of nodes reachable from s in $D_f(\Delta)$
- $val(f^*) - val(f) \leq$ (residual) cut-capacity of U in D_f $<$ $m\Delta$



residual network

Running time

Claim. There are at most $2m$ augmentations per scaling phase.

Pf.

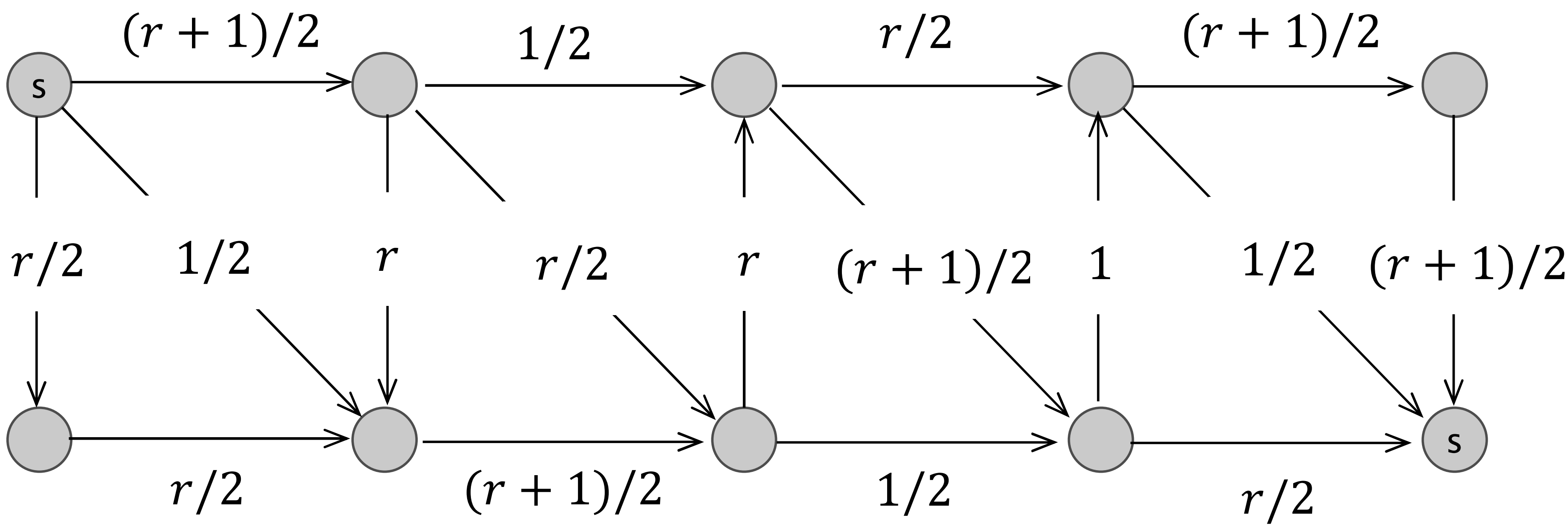
- At the beginning of Δ-phase, the optimality gap $< m(2\Delta) = 2m\Delta$.
- Each augmentation in a Δ-phase decreases the gap by at least Δ. ▪

Totally, $O(m \log C)$ augmentations.
Overall running time: $O(m^2 \log C)$ time.

Weakly polynomial, may not terminate for irrational capacities
[Queyranne 1980]

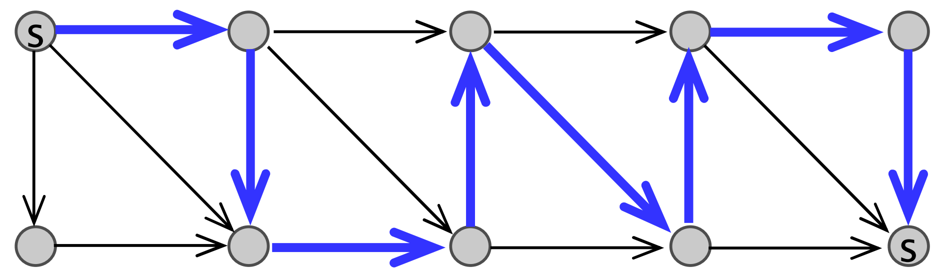
Queyranne's flow network

Theorem. The Edmonds-Karp algorithm may not terminate.

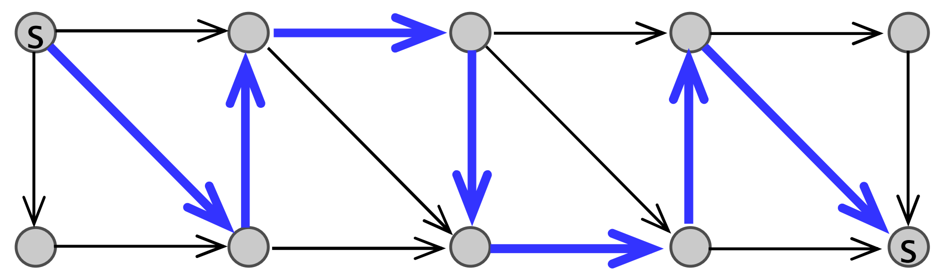


Valid Edmonds-Karp sequence

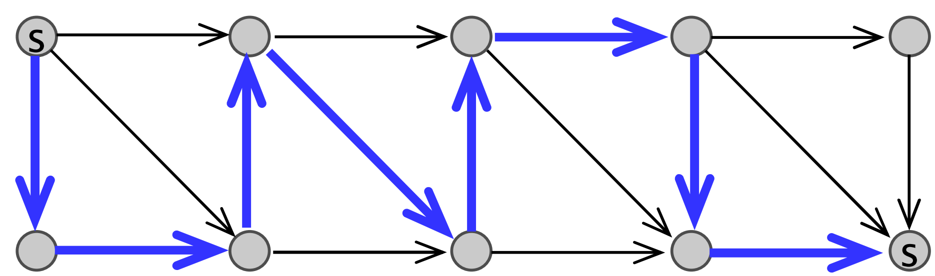
Iteration $3k-2$: r^{3k-2}



Iteration $3k-1$: r^{3k-1}



Iteration $3k$: r^{3k}



Shortest Augmenting Path

Shortest Augmenting Path

[Dinits 1970, Edmonds-Karp 1972]

Shortest-Augmenting-Path (D, s, t, c)

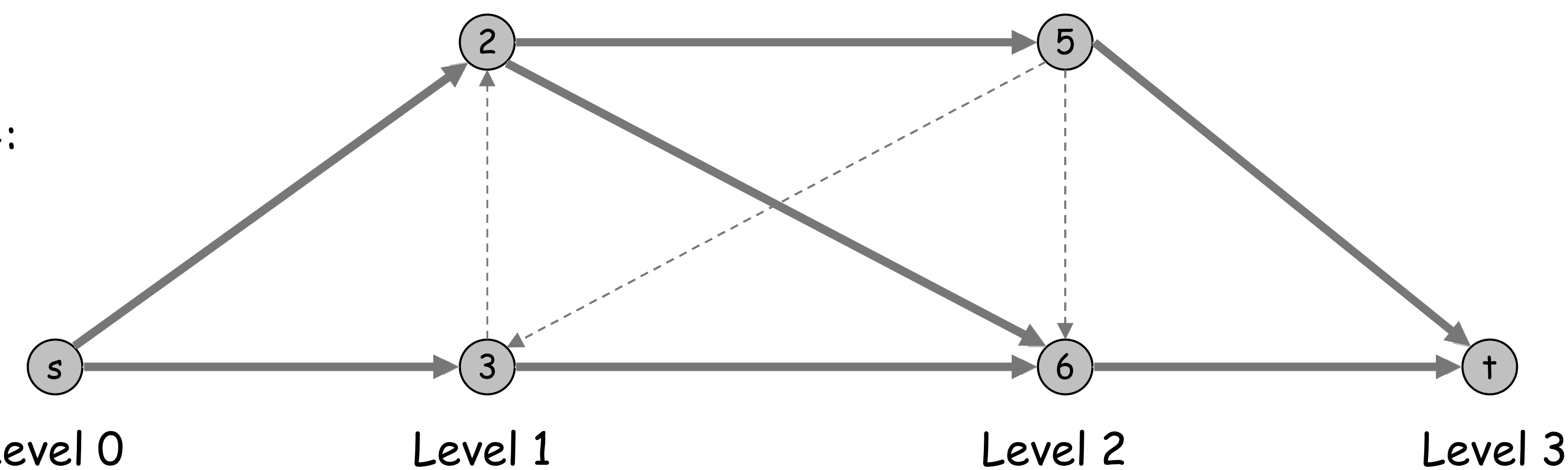
$f \leftarrow 0$
while (there exists an augmenting path)
    find such a shortest such path P using BFS
    $f \leftarrow f \oplus P$

return f

Level graph

- L_f : subgraph of D_f containing of all vertices and edges appearing in some shortest s-t path in D_f
  - Compute in $O(m+n)$ time using BFS by keeping only forward edges (deleting back and side edges).
- P is a shortest s-t path in D_f iff it is an s-t path L_f.

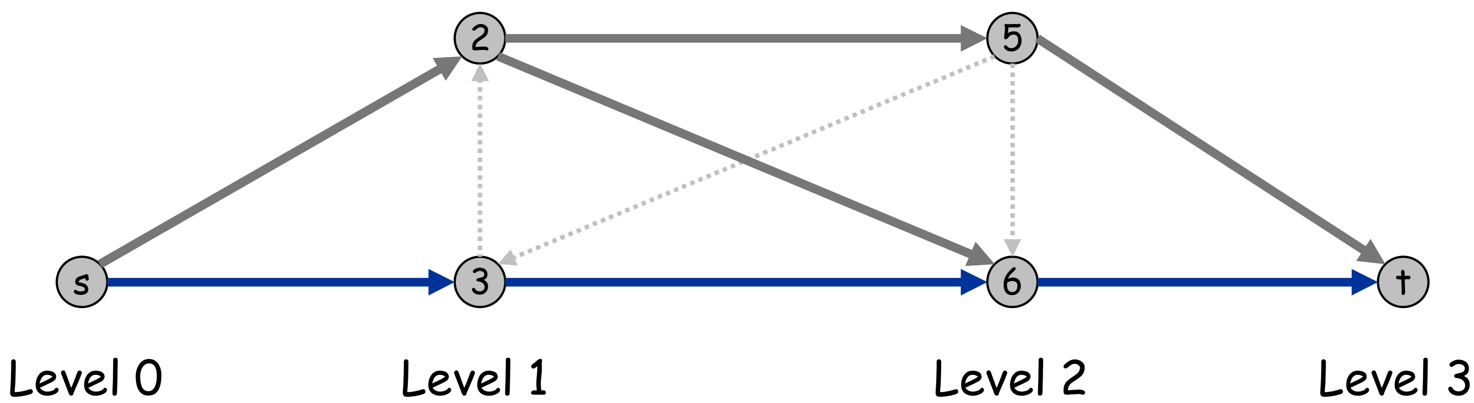
L_f:



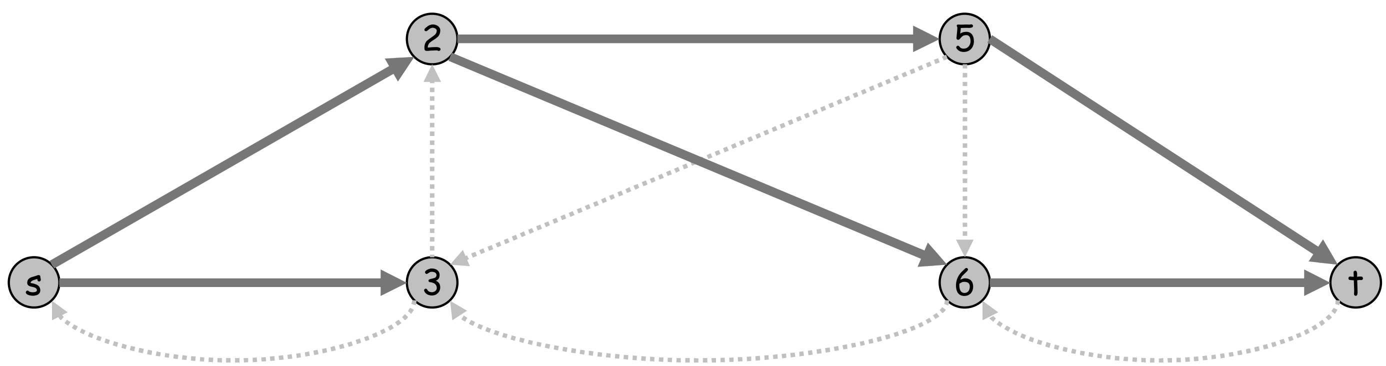
Evolution of level graphs

- Let f and f' be flow before and after a shortest path augmentation.
- Only back edges added to $D_{f'}$, and at least one edge (the bottleneck edge) in D_f (and L_f) is deleted from $D_{f'}$ (and $L_{f'}$)
- Path with back edge has length greater than previous length.

L_f:



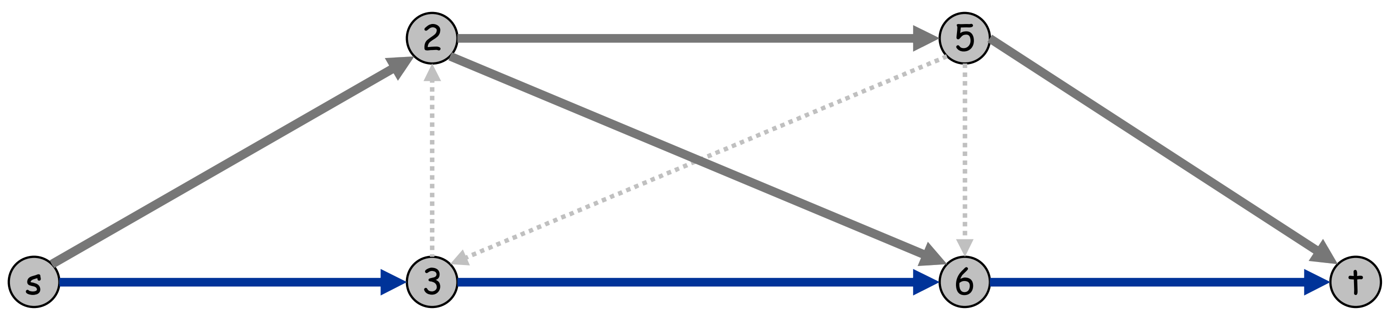
$L_{f'}$:



Evolution of level graphs

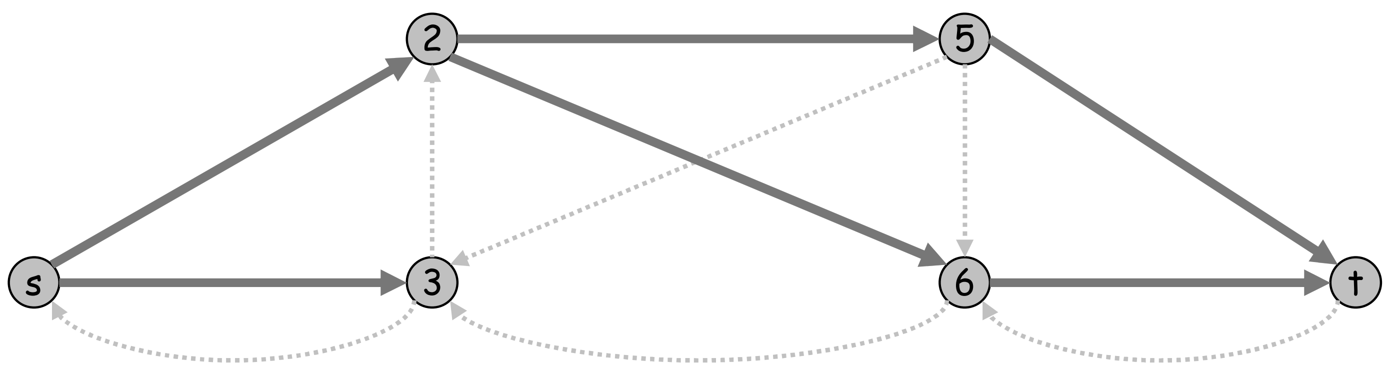
- The length of the shortest path never decreases.
- If the length of shortest s-t path in $D_{f'}$ does not increases, then the edge set of $L_{f'}$ strictly decreases.

L_f:



Level 0 Level 1 Level 2 Level 3

$L_{f'}$:



Running time

Phase: successive shortest path augmentations in which the shortest augmenting paths have the same length

Fact: at most n phases, and at most m augmentations per phase

Theorem. The shortest augmenting path algorithm performs at most $O(mn)$ augmentations. The overall running time is $O(m^2n)$.

2. Augmenting Flow by Blocking-Flow

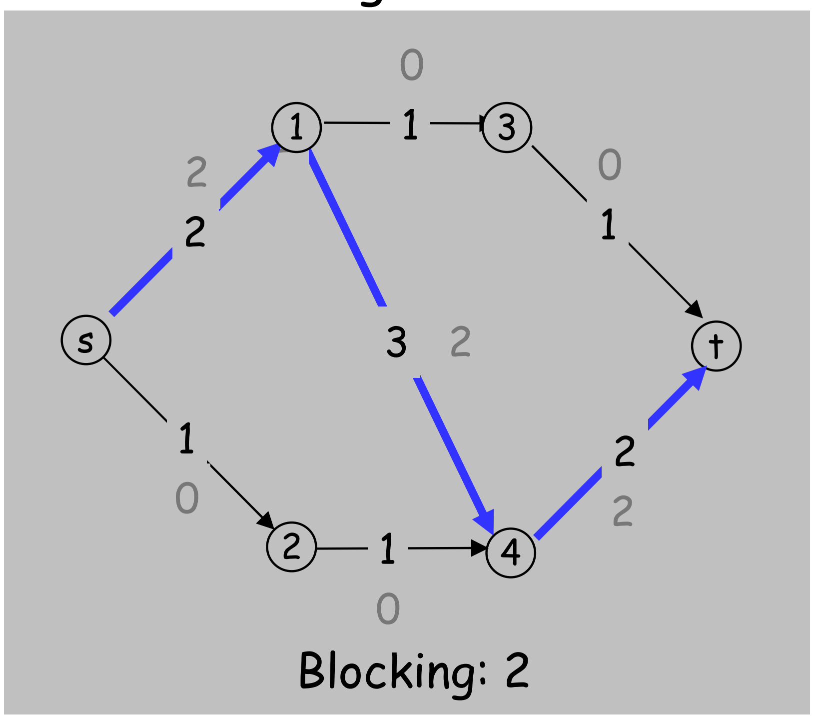
 

Blocking augmenting flow

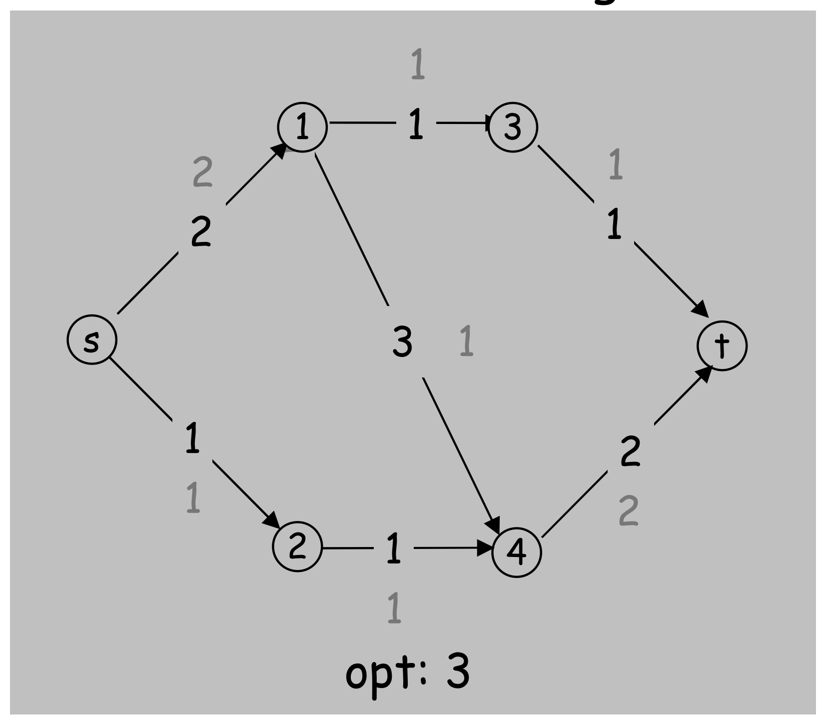
Analogy of maximal disjoint paths in the Hopcroft-Karp algorithm for maximum bipartite matching

Def. For a flow f in D, a flow g in L_f is blocking if all edges in L_f not saturated by g contain no s-t path

Blocking $\not\Rightarrow$ Maximum



Maximum $\Rightarrow$ Blocking



Dinitz's Method

Dinitz (D, s, t, c)

$f \leftarrow 0$
while (there exists f-augmenting path)
  $L_f \leftarrow$ the level graph of D_f
  $g \leftarrow$ a blocking flow in L_f
  $f \leftarrow f + g$
return f

Algorithms for finding block-flow:

[Dinitz 1970]: $O(mn)$

[Karzanov 1974]: $O(n^2)$

[Sleator-Tarjan 1983]: $O(m \log n)$

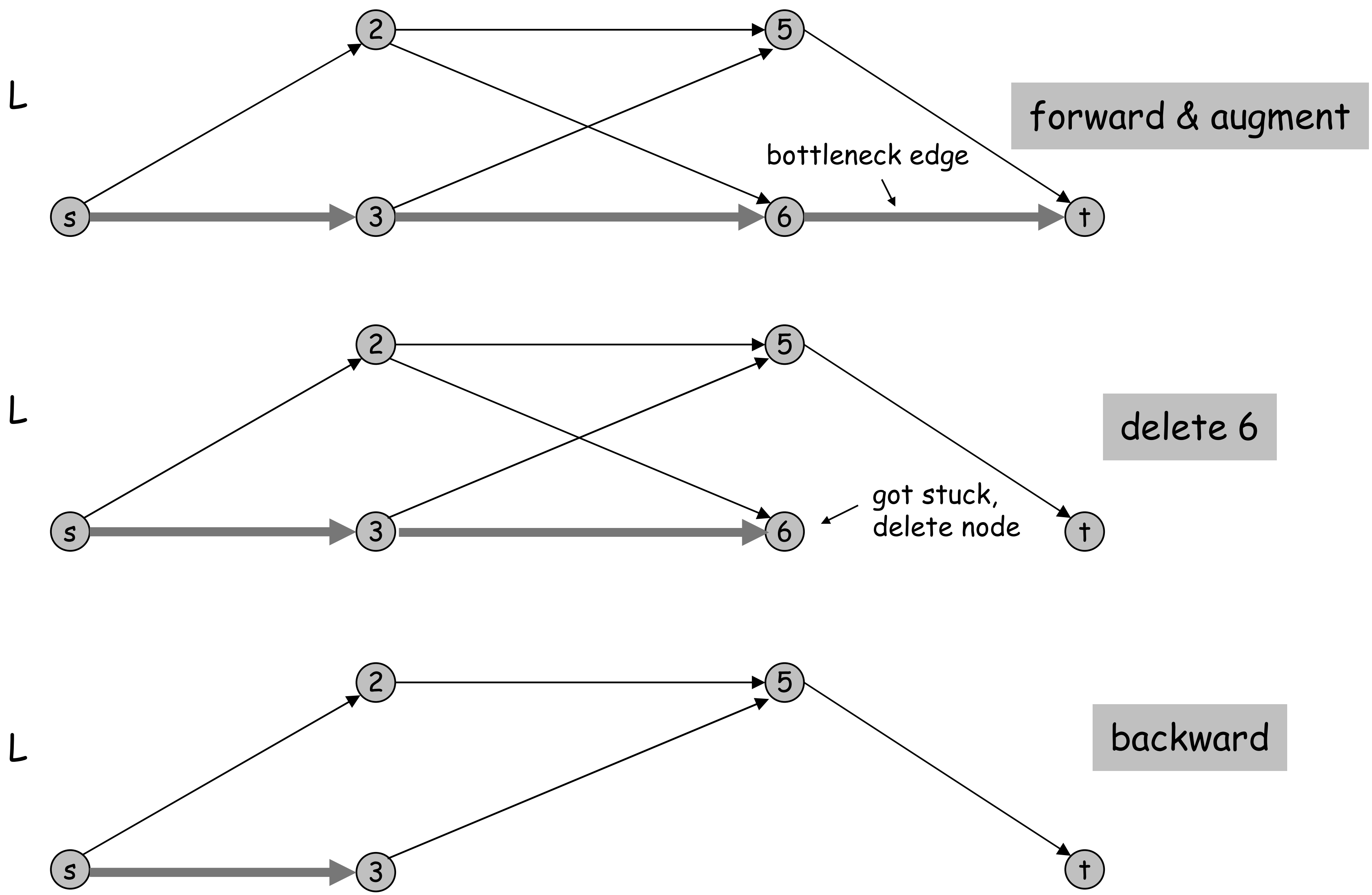
Finding blocking flow via DFS

[Dinitz 1970]

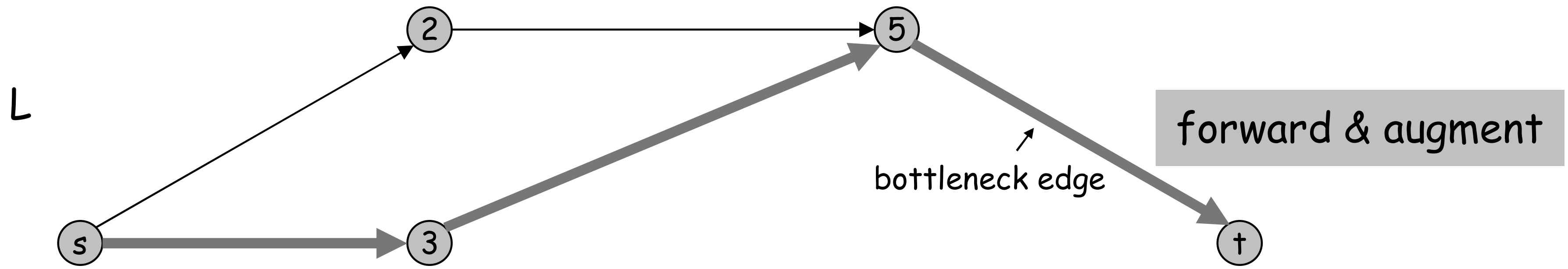
Each iteration starts at s, and acts at current vertex v as follows:

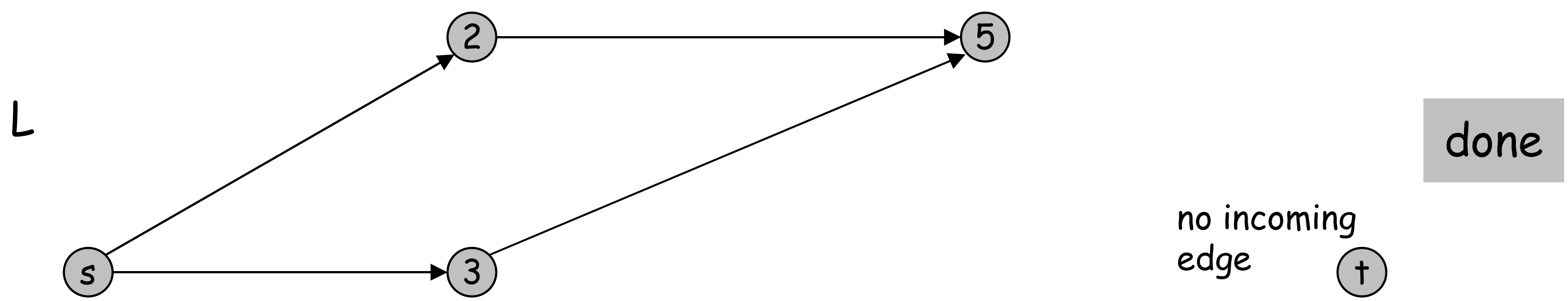
- Case 1: v has a forward edge. Move along a forward edge to the next node.
- Case 2: v has no forward edge.
  - subcase 2.1: $v = s$. Stop
  - subcase 2.2: $v = t$. Augment and delete bottleneck edges on path. If t has no incoming edge, stop; otherwise, move on to the next iteration
  - subcase 2.3: $v \neq s, t$. Delete v (and all its incident edges) and move backward to its predecessor.

Demo


L
s
2
3
5
6
t
forward & augment
bottleneck edge
L
s
2
3
5
6
t
delete 6
got stuck,
delete node
L
s
2
3
5
t
backward

Demo


L
2
5
forward & augment
bottleneck edge
s
3
t


L
2
5
done
no incoming
edge
s
3
t

Running time

- Each iteration, at least one edge is deleted. There are $O(m)$ iterations.
- Running time of each iteration: $O(n$ + number of edges deleted$)$
- Total running time of finding a blocking flow: $O(mn + m) = O(mn)$
- Total running time of finding maximum flow: $O(n^2 m)$

Finding blocking flow via preflow push/pull

Def. (residual) capacity of a vertex in L_f:

$$c(s) \coloneqq c\left(\delta^{out}(s)\right)$$

$$c(t) \coloneqq c\left(\delta^{in}(t)\right)$$

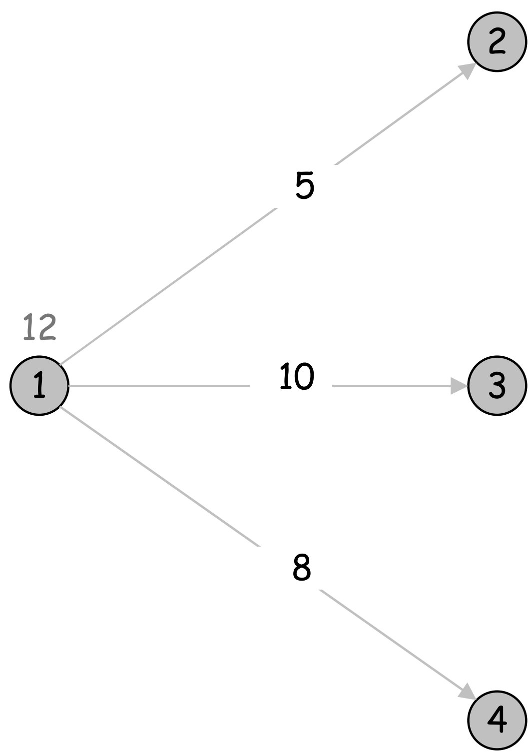
$$c(v) \coloneqq \min\{c(\delta^{in}(v)), c\left(\delta^{out}(v)\right)\}$$

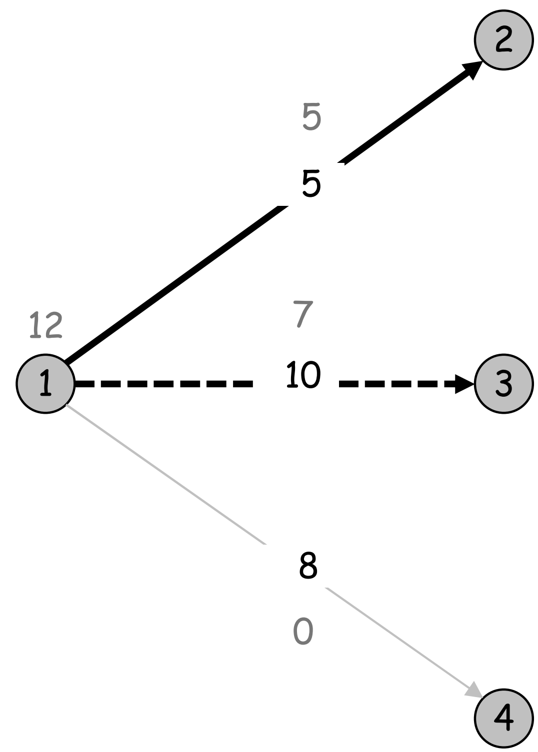
Each iteration:

- Compute a vertex v^* with minimum capacity ε.
- If $\varepsilon > 0$, then
    - Compute an $s - v^*$ flow of value ε by greedy pulling
    - Compute a $v^* - t$ flow of value ε by greedy pushing
- If $c(s)$ or $c(t)$ is ε, return g;
- Remove v^* and its incident edges, update its neighbors' capacity, and repeat.

Greedy pulling/pushing

- Greedy pulling (or pushing) backward (or forward) from v^* to s (or t) level-by-level
- At each vertex scan the incoming (or outgoing) edges one at a time:
  - fully saturate the edge before going on to the next one.
  - remove saturated edges
  - Update the capacities of itself and its neighbors.





Running time

- $O(n)$ iterations: at least one vertex is removed in each iteration.
- Running time of each iteration: $O(n$ + number of edges removed)
- Total running time of finding blocking flow: $O(n^2 + m) = O(n^2)$
- Total running time of finding maximum flow: $O(n^3)$

3. Preflow Push on Arcs

Overview

- Flow-augmenting approach: maintain a flow f and iteratively augment it until no s-t path in A_f (i.e. optimality)

- Preflow push/lift approach: maintain an s-preflow f without s-t path in A_f and modify it on an arc-by-arc basis until f is a flow, which is optimal

  - Initial s-preflow: the source s saturates all outgoing arcs

  - Subsequently, pick an excessive node $u \neq t$ to discharge its excess towards residual neighbors (including s possibly) "closer" to t

Recap: preflow

f: s-preflow

Fact. Every excessive node has at least one residual neighbor and can reach s in the residual graph.

Pf. The elementary decomposition of f has an s-v path

$$P \subseteq A^+(f) \subseteq A_f^{-1}$$

$P^{-1} \subseteq A_f$ is a v-s path in D_f.

Push operation

Condition: $u \neq t$ is excessive and $(u, v) \in A_f$

Push(u, v):
$\varepsilon \leftarrow \min\{f(\delta^{in}(u)), c_f(u, v)\}$; //maximal amount
$f(u, v) \leftarrow f(u, v) + \varepsilon$, $f(v, u) \leftarrow -f(u, v)$.

Classification: balancing if $\varepsilon = f(\delta^{in}(u))$, non-balancing otherwise

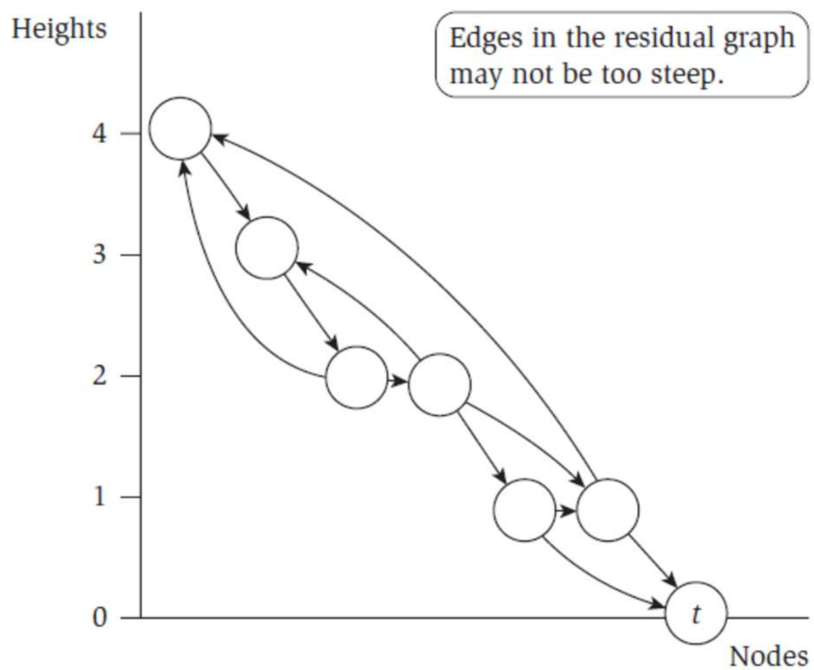
- balancing: u becomes balanced;
- non-balancing: (u, v) is removed from A_f

Decision: which u and which residual edge (u, v)?
Analogy: fluid naturally finds its way "downhill“.

Valid heights

$h: V \rightarrow Z_+$ with $h(s) = n$ (hill middle) and $h(t) = 0$ (hill bottom)

Def. h is valid for f if for each $(u, v) \in A_f$ then $h(u) - h(v) \leq 1$



Initial height for initital f: $h(s) = n$ and $h(v) = 0$ for all $v \neq s$.

Properties of valid height

Claim. There exists a valid h for $f \Leftrightarrow$ there is no s-t path in D_f.

Pf. ($\Rightarrow$) Otherwise, the s-t distance in D_f would be $\geq h(s) - h(t) = n$.

($\Leftarrow$) Put all nodes reachable from s at level n, and others at level 0.

Claim. For any valid h and any excessive node u, $h(u) < 2n$.

Pf.

$$n > u\text{-}s \text{ distance in } D_f \geq h(u) - h(s) = h(u) - n$$

So, $h(u) < 2n$.

The highest rule and downhill rule

- Choose a highest (excessive) u
- If u has a residual neighbor v with $h(v) = h(u) - 1$, push along (u, v)

Claim. **Push**(u, v) preserves the validity of h.
Pf. Easy verification at u and also at v.

- Otherwise,

$$\boxed{\textbf{Lift}(u)\text{: } h(u) \coloneqq h(u) + 1.}$$

Claim. **Lift**(u) preserves the validity of h.

Treatment on u

> **while** (u is not balanced)
>   **if** ($\exists (u, v) \in A_f$ with $h(v) = h(u) - 1$) **Push**(u, v);
>   **else** **Lift**(u).

The last push is balancing, and all others are non-balancing.

Push/Lift Algorithm

```
initialize f, h;
while (there is an excessive node other than t)
        pick a highest excessive node u ≠ t;
        treat u;
return f
```

- The number of lifts is $< 2n^2$
- The number of balancing pushes is O(n^3)
- The number of non-balancing push is O(mn)
- The total number of operations is O(n^3)

Simple O(n^3)-time implementation with linked lists and arrays.

Evolution of heights

- s and t have fixed heights

- Every other node u can never go down and its final height is $< 2n$
  - If u has never been lifted, its final height is 0.
  - Otherwise, it is excessive right after the last lift and hence its final height is $< 2n$.

- The number of lifts per node is $< 2n$
- The total number of lifts is $< 2n^2$.

Number of balancing pushes

Claim: $\leq n - 2$ balancing pushes between any two consecutive lifts,

Pf. Each of them makes one highest excessive node balanced.

The total number of balancing pushes is $O(n^3)$.

Number of non-balancing pushes

- $\forall(u, v) \in A$, a non-balancing **Push**(u, v) can occur at most n times.
- The number of non-balancing pushes is $O(mn)$.

Claim Between two consecutive non-balancing **Push**(u, v), the height of v increases by at least 2.

Pf. Right after the first non-balancing **Push**(u, v), u is above v, and (u, v) is no longer a residual edge.

Right before the second non-balancing **Push**(u, v), (u, v) must have become a residual edge again, which must be a consequence of some **Push**(v, u).

However, in order to make **Push**(v, u), we first need for v's height to increase by at least 2 so that v is above u.

Summary

- Augmenting flow by single path-flow
  - (Nearly) Widest Augmenting Path
  - Shortest Augmenting Path

- Augmenting flow by blocking-flow
  - via DFS
  - via preflow push/pull

- Preflow push on arcs: push/lift

- Still active research on faster weakly polynomial-time algorithm
- https://www.quantamagazine.org/researchers-achieve-absurdly-fast-algorithm-for-network-flow-20220608/