7.5 Analyzing the Ford-Fulkerson algorithm

- In what follows, we analyze the complexity of the introduced Ford-Fulkerson algorithm
- First of all, we will see that the correctness of the algorithm is limited to integer and rational capacity values
- However, in case of irrational capacity values, even termination and correctness of the procedure are not guaranteed anymore
- This result is somehow surprising since the procedure seems to be finite as every previously introduced algorithm

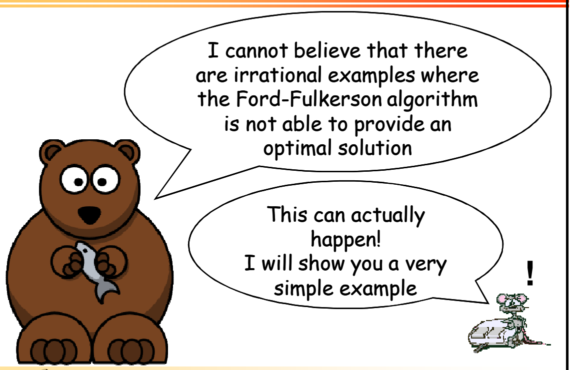
7.5.1 Correctness

- If capacities are integers, the termination of the algorithm follows directly from the fact that the flow is increased by at least one unit in each iteration
- Since, if the optimal flow has the total amount of f_{opt}, f_{opt} iterations (augmentations) are at most necessary
- Analogously, if all capacities are rational, we may put them over a common denominator D, scale by D, and apply the same argument.
- Hence, if the optimal flow has the total amount of f_{opt}, $f_{opt} \cdot D$ iterations (augmentations) are at most necessary (see Papadimitriou and Steiglitz (1982) pp.124)

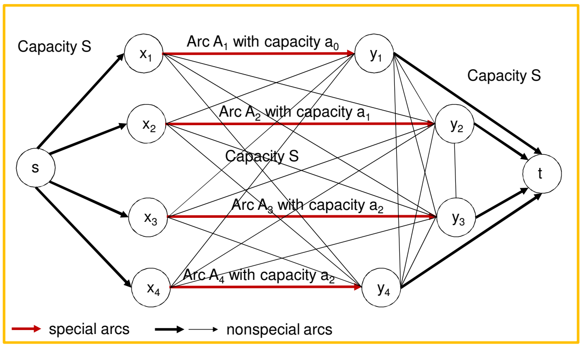
The pitfall – irrational case

- However, when the capacities are irrational, one can show that the method does not only fail to compute the optimal result but also converges to a flow strictly less than optimal
- In what follows, we shall introduce and illustrate an example originally given by Ford and Fulkerson (1962) and depicted in Papadimitriou and Steiglitz (1982)
- Edmonds and Karp (1972) proposed a modified labeling procedure and proved that this algorithm requires no more than $(n^3-n)/4$ augmentation iterations, regardless of the capacity values

Analyzing the problem in detail



The irrational case – the network



The irrational case – capacities

- Special arcs
  - These are the arcs A_1, A_2, A_3, and A_4
  - Capacity is a_0 for A_1, a_1 for A_2, a_2 for A_3, and a_2 for A_4
- Nonspecial arcs
  - All other arcs are nonspecial arcs, i.e., all arcs (s,x_i), (y_i, y_j), (y_i, x_j), (x_i, y_j), or (y_i, t) with $i \neq j$
  - Capacity is S
- We define

$$a_{n+2} = a_n - a_{n+1}$$

$$a_0 = 1,\ a_1 = \sigma = \frac{\sqrt{5}-1}{2} < 1,\ \frac{\sqrt{5}-1}{2} \approx 0.618033989, \text{ and } S = \frac{1}{\sigma}$$

The capacities of the special arcs

7.5.1.1 Lemma:

It holds that: $\forall n \geq 0 : \forall i \in \{1,\ldots,n\} : a_i = \sigma^i$

Proof:

We prove the proposition by induction:

$$i=0: a_i = a_0 = 1 = \sigma^0 = \sigma^i$$

$$i=1: a_i = a_1 = \frac{\sqrt{5}-1}{2} = \sigma = \sigma^1 = \sigma^i$$

$$i>1: a_i = a_{i-2} - a_{i-1} = \left(\frac{\sqrt{5}-1}{2}\right)^{i-2} - \left(\frac{\sqrt{5}-1}{2}\right)^{i-1} = \left(\frac{\sqrt{5}-1}{2}\right)^{i-2} \cdot \left(1 - \frac{\sqrt{5}-1}{2}\right)$$

$$= \left(\frac{\sqrt{5}-1}{2}\right)^{i-2} \cdot \left(\frac{2-\sqrt{5}+1}{2}\right) = \left(\frac{\sqrt{5}-1}{2}\right)^{i-2} \cdot \left(\frac{3-\sqrt{5}}{2}\right)$$

Proof of Lemma 7.5.1.1

- Since it holds that

$$\sigma^2 = \left(\frac{\sqrt{5}-1}{2}\right)^2 = \left(\frac{\sqrt{5}-1}{2}\right) \cdot \left(\frac{\sqrt{5}-1}{2}\right) = \frac{5-2\cdot\sqrt{5}+1}{4} = \frac{6-2\cdot\sqrt{5}}{4} = \frac{3-\sqrt{5}}{2}$$

- we obtain

$$i>1: a_i = a_{i-2} - a_{i-1} = \left(\frac{\sqrt{5}-1}{2}\right)^{i-2} \cdot \left(\frac{3-\sqrt{5}}{2}\right) = \left(\frac{\sqrt{5}-1}{2}\right)^{i-2} \cdot \left(\frac{\sqrt{5}-1}{2}\right)^2$$

$$= \left(\frac{\sqrt{5}-1}{2}\right)^i = \sigma^i$$

- This completes the proof

Consequence

7.5.1.2 Lemma:

It holds that

$$\forall n \geq 0 : \lim_{n\to\infty} a_0 + \sum_{i=2}^{n} (a_i + a_{i+1}) = a_0 + (a_2 + a_3) + (a_3 + a_4) + \ldots = \frac{1}{1-\sigma} = S$$

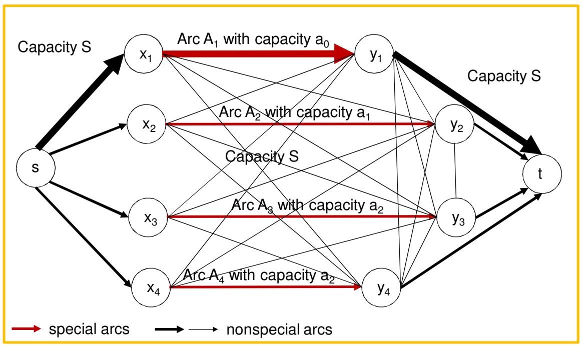
Proof:

We conclude that:

$$\forall n \geq 0 : \lim_{n\to\infty} a_0 + \sum_{i=2}^{n} \left(a_i + \underbrace{a_{i+1}}_{=a_{i-1}-a_i} \right) = \lim_{n\to\infty} a_0 + \sum_{i=2}^{n} a_{i-1} = \lim_{n\to\infty} a_0 + \sum_{i=1}^{n-1} a_i$$

$$= \lim_{n\to\infty} \sum_{i=0}^{n-1} a_i = \underbrace{\sum_{i=0}^{\infty} \sigma^i}_{\text{Geometric series with } 0<\sigma<1} = \frac{1}{1-\sigma} = S$$

Step 0 – augmentation path (s,x_1,y_1,t)



Step 0 - consequences

- Augmentation value is a_0
- This is true since

$$\sigma = \frac{\sqrt{5}-1}{2} < 1 \text{ and } a_0 = \sigma^0 = 1 < S = \frac{1}{1-\sigma}$$

- Hence, the residual capacities in the special arcs amount to

$$(a_0 - a_0, a_1, a_2, a_2) = (0, a_1, a_2, a_2)$$

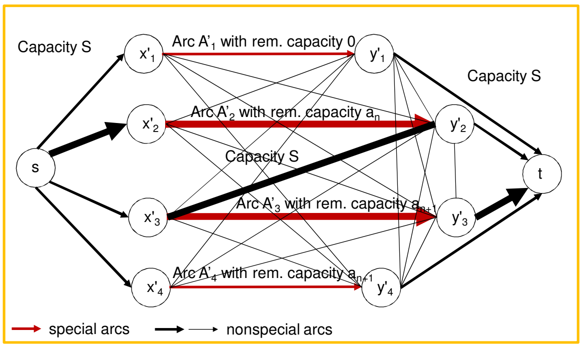
Step n≥1 – assumptions

- Due to the preceding steps, we have the following remaining capacities on the special arcs

$0, a_n, a_{n+1}$, and a_{n+1}
Note that we order now the special arcs such that, after this step, we have the arcs A_1', A_2', A_3', and A_4' with the remaining capacities $(0, a_n, a_{n+1}, a_{n+1})$. Order the connected nodes x_1', x_2', x_3', and x_4' as well as y_1', y_2', y_3', and y_4', accordingly.

- Note that step 0 has provided such a situation

Step n≥1 – augmentation path $(s,x'_2,y'_2,x'_3,y'_3,t)$



Step n≥1 – consequences

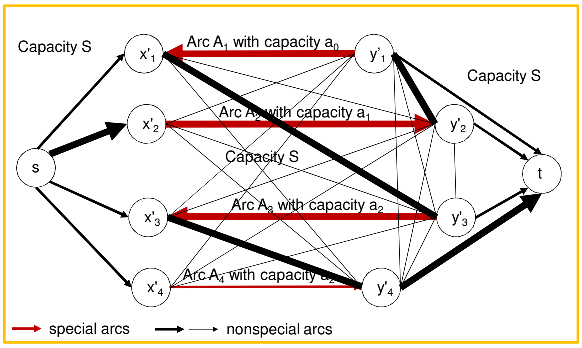
The chosen augmentation path increased the total flow by a_{n+1} units since we used the special arcs A'_2 and A'_3 in forward direction. Since $a_{n+1}=\sigma^{n+1}<a_n=\sigma^n$, due to $\sigma<1$, a_{n+1} is the bottleneck on the chosen path

Note that the inner nonspecial arcs are somehow symmetric, i.e., we have always arcs with capacity S in both directions from x to y and vice versa.

After using this augmentation path, we obtain the following residual capacities on the special arcs:

$$\left(0,\underbrace{a_n-a_{n+1}}_{a_{n+2}},a_{n+1}-a_{n+1},a_{n+1}\right)=\left(0,a_{n+2},0,a_{n+1}\right)$$

Second augmentation path $(s,x'_2,y'_2,y'_1,x'_1,y'_3,x'_3,y'_4,t)$



Second augmentation – consequences

The chosen augmentation path increased the total flow by a_{n+2} units since we used the special arc A_2' in forward direction and the special arc A_1' and A_3' in backward direction . Since $a_{n+2} = \sigma^{n+2} < a_{n+1} = \sigma^{n+1}$, due to σ<1, a_{n+2} is the bottleneck on the chosen path

Note again that the inner nonspecial arcs are somehow symmetric, i.e., we have always arcs with capacity S in both directions from x to y and vice versa.

After using this augmentation path, we obtain the following residual capacities on the special arcs: $\left(0+a_{n+2}, a_{n+2}-a_{n+2}, 0+a_{n+2}, a_{n+1}\right)$ $=\left(a_{n+2}, 0, a_{n+2}, a_{n+1}\right)$

Consequences of step n≥1

- Step n ends with residual capacities appropriate for conducting the succeeding step $n+1$
- Hence, each step augments the total flow by $a_{n+1}+a_{n+2}$

  It holds that: $a_{n+2} = a_n - a_{n+1} \Leftrightarrow a_{n+2} + a_{n+1} = a_n$
- Therefore, the flow is augmented by a_n

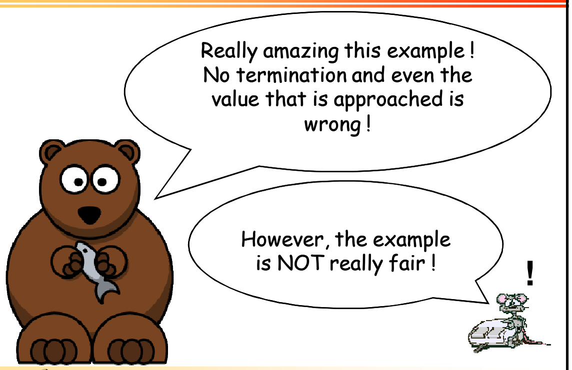
All in all, after n steps, we therefore obtain the total flow $\sum_{i=0}^{n} a_i$

Consequently, there is always an improvement possible and the algorithm does not terminate and the total flow approaches $\sum_{i=0}^{\infty} a_i = \frac{1}{1-\sigma} = S$

No termination and …

- However, the max flow in our pathological example is obviously $4 \cdot S$
- So the Ford-Fulkerson algorithm approaches one-fourth the optimal flow value
- Therefore, the algorithm is not correct

Worth to mention



In the sense of fairness

- The raised question of finiteness of the Ford Fulkerson algorithm is in a sense a mathematical but not a practical one, since computers always work with rational numbers
- Hence, it is reasonable to assume that data can be represented by a finite number of bits
- A practical question, which is however related to that of finiteness, will ask how many steps may be required by a computation as a function of the total number of bits in the data

7.5.2 Complexity analysis

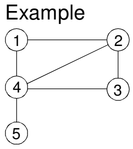
- In what follows, we analyze the complexity of the Ford-Fulkerson algorithm for integral capacity values
- Unfortunately, it turns out that – depending on the given capacity values of the considered instance – this labeling procedure may require in the worst case an exponential amount of time
- Fortunately, there exists an efficient algorithm for the max flow problem, which is, in fact, a rather simple modification of the labeling algorithm
- In order to analyze the labeling procedure and to prepare a modified version of it, we first examine a fundamental graph algorithm called $search(v)$
- Such a procedure is required in both algorithms

Graph representations

- A graph $G = (V, E)$ can be represented in many alternative ways
  - Adjacency matrix:
    - A matrix $A_G = [a_{i,j}]_{1 \le i \le |V|, 1 \le j \le |V|}$, with binary entries such that
    - $a_{i,j} = 1$ if arc $(i,j) \in E$ and $a_{i,j} = 0$ otherwise
    - However, in case of graphs that are sparse in that the number of their arcs is far less than $O\left(\binom{|V|}{2}\right) = O(|V|^2)$, this representation is the most economical one. E.g., if we have 100 nodes and 500 edges, an representation with 10,000 (!) binary entries has to be stored

Graph representations

- Adjacency lists: For each node $v \in V$ $A(v)$ gives an ordered list of successors, i.e., we have $A(v) = [v_1, v_2, \dots, v_{l(A(v))}]$, with $(v, v_i) \in E, \forall i \in \{1, \dots, l(A(v))\}$
- Example



$A(1) = [2,4], A(2) = [1,3,4],$
$A(3) = [2,4], A(4) = [1,2,3,5], A(5) = [4]$

- In what follows, we assume that the graph $G = (V, E)$ is connected, i.e., there are no isolated nodes

Algorithm $search(v)$

Input: A graph G, defined by adjacency lists and a node v
Output: The graph with the nodes reachable by path from the node v marked

$Q = \{v\}$
while $Q \neq \emptyset$ **do**
 let u be any element of Q
 remove u from Q
 mark u
 for all $u' \in A(u)$ do
  if u' is not maked **then** insert u' into Q
end while

Complexity

7.5.2.1 Theorem:

The algorithm $search(v)$ marks all nodes of G connected to v in $O(|E|)$ time.

Proof:

Correctness: We assume that a node u is connected to node v by a path p. Clearly, it can be shown by induction on the path length that u will be marked. If, otherwise, node u is not connected to node v u will not be marked since this would lead to the contradictory conclusion that there is a path from node v to node u

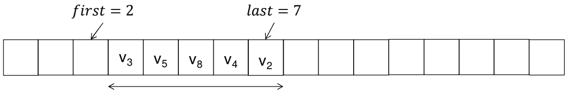
Proof of Theorem 7.5.2.1

Time bound:

- In order to estimate the running time of $search(v)$, we have to consider three components:
  1. Initialization: this takes constant time
  2. Maintaining the set Q: We store the set Q as a queue with a $first$ and $last$ pointer (variables) in order to enable insertion and deletion in constant time (see the next slide for a brief illustration). The pointers (variables) $first$ and $last$ are initialized to zero while Q is stored as a simple array with $|V|$ entries. Array Q is empty if and only if it holds $first = last$. We remove from top and add at the tail of the queue (FIFO principle).

Applied data types

- Add v to Q:
  - $last$=$last$+1
  - $Q[last] = v$
- Remove:
  - $first = first + 1$
  - $v = Q[first]$



The contents of Q, in order of arrival

Proof of Theorem 7.5.2.1 – Time bound

3. Searching the adjacency lists: we have constant time for each element of the lists. Since the total number of these elements is $2 \cdot |E|$, the time required is $O(|E|)$

Therefore, we have a total asymptotic running time of $O(|E|)$. This completes the proof

LIFO queue (i.e., a stack)

- Add v to Q:
  - $last = last + 1$
  - $Q[last] = v$
- Remove:
  - $v = Q[first]$
  - $last = last + 1$

$last = 5$

v_3	v_5	v_8	v_4	v_2								

The contents of Q, in order of arrival. Q is empty if and only if $last = 0$

Selecting rules applied to Q

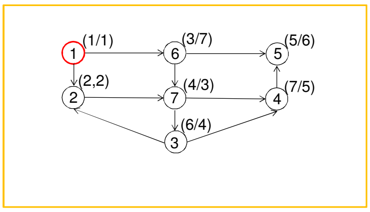
- The procedure $search(v)$ was not completely specified
- We have not defined yet exactly how the next element v is chosen from Q in the while loop
- There are many possibilities
- Two best known are …
  - *FIFO*: The node that waited longest is chosen (breadth first search (BFS))
  - *LIFO*: The node that was lastly inserted is chosen (depth first search (DFS))

Directed graphs

- The procedure $search(v)$ can be applied to directed graphs (i.e., so-called digraphs) without any changes

Example

- We apply BFS and DFS to the digraph below
- The resulting numbers (BFS/DFS) give the indices of the step at that the respective node is labeled
- Starting node is node 1



Algorithm $findpath(v)$

Input: A digraph $G=(V,E)$, defined by adjacency lists and two subsets S,T of V
Output: A path in G from a node in S to a node in T if this path exists

```
for all v ∈ S do label[v] = 0
if v ∈ T then return(v); break;
Q = S
while Q ≠ ∅ do
 let u be any element of Q
 remove u from Q
 for all u′ ∈ A(u) do
  if u′ is not labeled then begin
    label[u′] = u
    if u′ ∈ T then return path(u′); break; else insert u′ into Q
  end (begin)
 end (do)
end while
return "no S-T path available in G"
```

Algorithm $path(v)$

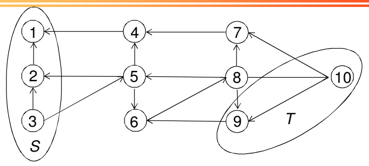
Input: For all nodes $u \in V$: $label[u]$ generated by procedure *findpath*
Output: Path from a node in S to $v \in T$

if $label[v] = 0$
  then return(v); **break**;
  else return$\left(path(label[v])\right) \| (v)$; **break**;
end if

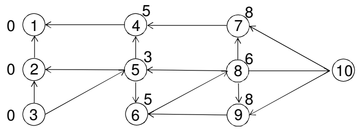
$\|$ stands for concatenation of paths

Note that the procedure is recursive!

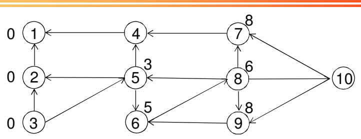
Example



- We apply the procedure $findpath(S,T)$ with FIFO queue (bfs) and obtain the labels (resulting in a path with a minimum number of arcs)



Example – Path reconstruction



- We apply $path(9)$ and obtain

$$
\begin{aligned}
& path(9) \\
&= path(8) \| (9) = path(6) \| (8,9) = path(5) \| (6,8,9) \\
&= path(3) \| (5,6,8,9) = (3,5,6,8,9)
\end{aligned}
$$

Complexity of the Ford Fulkerson procedure

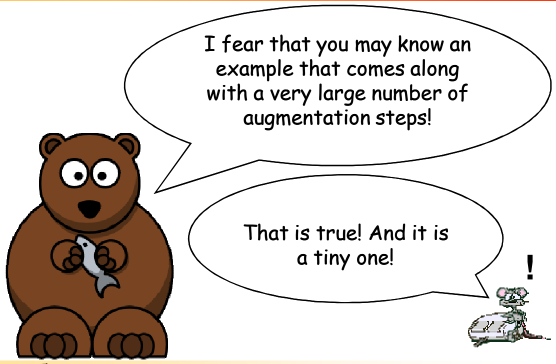
- We now analyze the complexity of the Ford-Fulkerson algorithm more in detail
- We apply the algorithm to a network $N = (s, t, V, E, c)$ and observe the following
  - The initialization step of the procedure takes time $O(|E|)$
  - Each iteration step involves the scanning and labeling of vertices. It can be stated that each edge (u, v) is considered at most twice – once for scanning node u and once for v. Moreover, we have to follow back the found path that has a length of at most $O(|V|)$ steps
  - Thus, each iteration takes time $O(|V| + |E|)$

Complexity of the Ford Fulkerson procedure

- All in all, in case of integral capacities, if v is the value of the max flow and S is the number of conducted augmentation steps of the applied Ford-Fulkerson algorithm, we have $S \leq v$ and a total asymptotic running time complexity of $O\big((|V| + |E|) \cdot S\big) = O(|E| \cdot S)$
- In order to define the running time by the input data of a given instance, we obtain the asymptotic running time

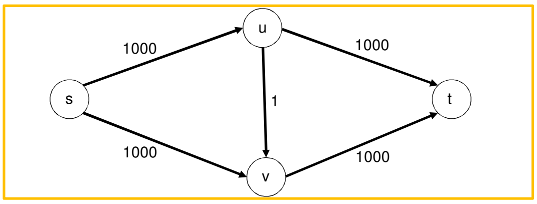
$$O\left(|E| \cdot \left(\sum_{(x,y)\in E} c(x,y)\right)\right)$$

Worth to mention



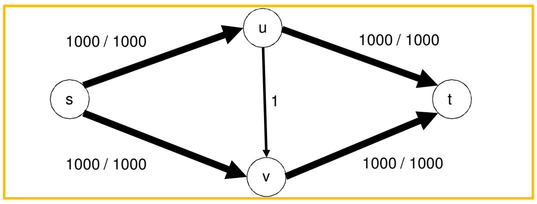
Worst case example

- Consider the following network with total capacity of 4,001
- We will see that the Ford Fulkerson algorithm requires 2,000 iterations to generate an optimal solution



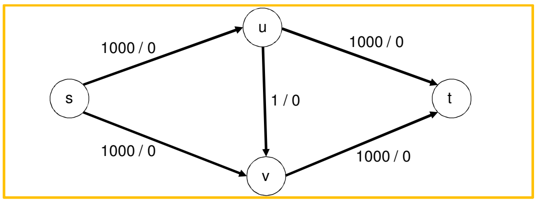
Worst case example – Optimal solution

- The maximum flow obviously amounts to 2000
- Illustration of the optimal solution



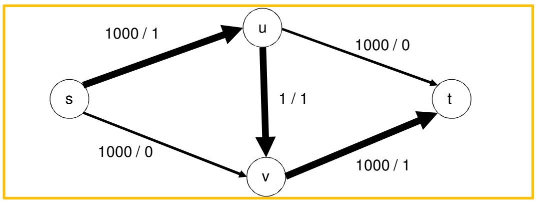
Worst case example

- In what follows, we apply the labeling algorithm starting from the initial zero flow
- We commence with the zero flow on each edge



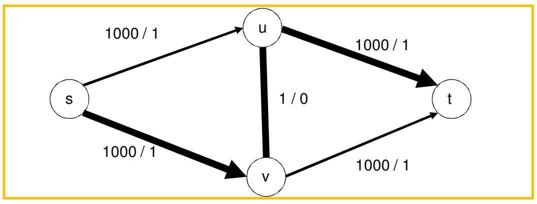
Solving the worst case example 1

- We start with the initial flow (s, u, v, t) with flow 1
- We obtain the following updated network



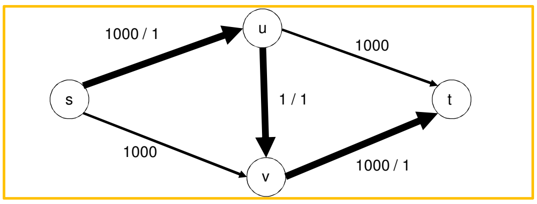
Solving the worst case example 2

- We start with the initial flow (s, v, u, t) with flow 1
- We obtain the following updated network



Solving the worst case example 3

- We start with the initial flow (s, u, v, t) with flow 1
- We obtain the following updated network



After two augmentation steps, we have

- A total flow of 2
- Hence, there exists a sequence of $1{,}000$ iterations, each comprising two augmentation steps with the paths (s, u, v, t) and (s, v, u, t), that generates the optimal solution with total flow $2{,}000$
- Therefore, the asymptotic runtime bound

$$\boxed{O\left(|E| \cdot \left(\sum_{(x,y)\in E} c(x,y)\right)\right)}$$

- is actually tight since we can replace the $1{,}000$ values by an arbitrarily large M-value

Exponential running time

- If $M = c^{|V|}$ holds (with $c \geq 2$), the Ford-Fulkerson algorithm executes

$$\boxed{O\left(|E| \cdot c^{|V|}\right)}$$

- steps
- Hence, we have an exponential running time

Towards a new max flow algorithm

- Suppose that we wish to apply the labeling routine to a network $N = (s, t, V, E, c)$ with initial zero flow $f = 0$
- We need not examining capacities and flows in this ease; it is a priori certain that all arcs in *A* are forward, and that there are no backward arcs Consequently, our task of labeling the network in order to discover an augmenting path is done by applying procedure $findpath$ to $N = (s, t, V, E, c)$ with $S = \{s\}$ and $T = \{t\}$
- Subsequently, we augment the current flow by applying $findpath$ to a modified network $N(f) = (s, t, V, E(f), ac)$ that results from the current flow f
- This modified network is defined next

A flow-oriented network definition

7.5.2.2 Definition

Given a network $N = (s, t, V, E, c)$ and a feasible flow f of N. Then, we define the network $N(f) = (s, t, V, E(f), ac)$ with $E(f)$ comprising the arcs

1. *If $(u, v) \in E$ and $f(u, v) < c(u, v)$, then $(u, v) \in E(f)$ and $ac(u, v) = c(u, v) - f(u, v)$*
2. *If $(u, v) \in E$ and $f(u, v) > 0$, then $(v, u) \in E(f)$ and $ac(v, u) = f(v, u)$*

The value $ac(u, v)$ is denoted as the augmenting capacity of arc $(u, v) \in E(f)$

Avoiding multiple copies of arcs in $E(f)$

- If E contains both arcs $(u, v) \in E$ *and* $(v, u) \in E$, then $E(f)$ may have multiple copies of these arcs. However, in this case we may replace one arc $(u, v) \in E$ by a new node w and two additional arcs $(u, w), (w, v) \in E$ with identical capacity, i.e., it holds that $c(u, w) = c(w, v) = c(u, v)$
- Therefore, we can assume that $E(f)$ has no multiple arcs

Interesting attributes of $N(f)$

- Take any s-t cut $(W, \overline{W})$ of $N(f)$
- The value of this cut is the sum of the augmenting capacities of all arcs of $N(f)$ going from W to $\overline{W}$
- Such an arc $(u, v) \in E(f)$ may be either a forward arc (case 1 in Definition 7.5.2.2, i.e., $ac(u, v) = c(u, v) - f(u, v)$) or a backward arc (case 2 in Definition 7.5.2.2, i.e., $ac(v, u) = f(v, u)$)
- Thus, all in all, if we directly compare the value of $(W, \overline{W})$ in $N(f)$ with the value of $(W, \overline{W})$ of N, we see that the first one is equal to the second one minus the forward flow of f across the cut plus the backward flow of f against the cut

Interesting attributes of $N(f)$

- But for every cut $(W, \bar{W})$ and flow f we know that the flow of f over forward arcs minus the flow of f (i.e., $|f|$) over backward arcs coincides with the total flow of f that leaves source s
- We define $|f| = \sum_{(s,v) \in E} f(s,v)$
- Consequently, we conclude that the value of $(W, \bar{W})$ in $N(f)$ coincides with the value of $(W, \bar{W})$ of N minus the total flow $|f|$ of flow f
- Hence, this proves the following Lemma 7.5.2.3 since in both networks the value of the minimum cut equals the value of the maximum flow

Consequence

7.5.2.3 Lemma

If $|\tilde{f}|$ is the value of the maximum flow in network N, then the value of the maximum flow in $N(f)$ is $|\tilde{f}| - |f|$

Layered network

7.5.2.4 Definition

A layered network $L = (s, t, U, A, b)$ with $d + 1$ layers is a network with vertex set $U = U_0 \cup \cdots \cup U_d$, while $\forall j \in \{1, \ldots, d\}: U_{j-1} \cap U_j = \emptyset$, $U_0 = \{s\}$, and $U_d = \{t\}$. The set of arcs A is defined by

$$\boxed{A \subseteq \bigcup_{j=1}^{d} \left(U_{j-1} \times U_j\right)}$$

Maximal flows

7.5.2.5 Definition

Let $N = (s, t, U, A, b)$ be a layered network. An augmenting path in N with respect to some flow g is denoted as forward if it uses no backward arc. A flow g of N is called maximal (not maximum) if there is no forward augmenting path in N with respect to g

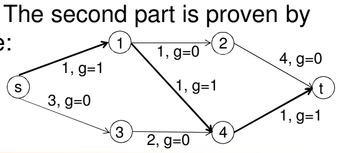
Maximum, maximal flow

7.5.2.6 Conclusion

All maximum flows are maximal. However, not all maximal flows are maximum flows.

Proof:

If f is a maximum flow it cannot be augmented. Hence, it is maximal. The second part is proven by the following example:



Maximum flow amounts to 2
However, g is maximal but $|g| = 1$

Auxiliary network $AN(f)$

- We introduce the auxiliary network $AN(f)$ as a layered network to a network $N(f)$ with a flow f
- We create $AN(f)$ by carrying out a breadth-first search on $N(f)$ while copying only the arcs in $AN(f)$ that lead us to new nodes and only the nodes that are at lower levels than node t
- If a node is added all incoming arcs from previously added nodes are integrated. However, there is no backward arc
- Hence, $AN(f)$ is generated out of $N(f)$ in time $O(|E(f)|) = O(|E|)$
- Using the auxiliary network, we can easily find the shortest augmenting path (with a minimal number of edges) with respect to the current flow.

7.6 An efficient max flow algorithm

- In what follows, we introduce a polynomial max flow approach
- It has an asymptotic running time of $O(|V|^3)$

Basic structure of the max flow procedure

- It operates in stages
  - At each stage – depending on the current flow f – it constructs the network $N(f)$ and, according to it, it generates the auxiliary network $AN(f)$
  - Then, we find a maximum flow g in the auxiliary network $AN(f)$ and add this flow g to flow f

Basic structure of the max flow procedure

- Adding g to f entails adding $g(u,v)$ to $f(u,v)$ if arc (u,v) is a forward arc in $AN(f)$ and subtracting $g(u,v)$ from $f(u,v)$ if arc (u,v) is a backward arc in $AN(f)$
- The procedure terminates when s and t are disconnected in $N(f)$
- This proves that f is optimal

7.6.1 Pseudo code of the procedure

Input: A network $N = (s,t,V,E,c)$
Output: The maximum flow f of N

$f = 0$; $done = false$;
while (NOT $done$) **do**
  $g = 0$;
  construct the auxiliary network $AN(f) = (s,t,U,F,ac)$;
  if t is NOT reachable from s in $AN(f)$ **then** $done = true$;
  else while there is a node with $throughput[v] = 0$ **do**
    if $v = s$ OR $v = t$ **then go to** incr
    else delete v and all incident arcs from $AN(f)$
    let v be the node in $AN(f)$ with minimal nonzero $throughput[v]$;
    $push(v, throughput[v])$;
    $pull(v, throughput[v])$;
  end while; end if
  incr: $f = f + g$ **Comment**: End of the current stage
end while

Pseudo code of $push(y,h)$

Comment: Increases the flow g by h units pushed from y to t
$Q = \{y\}$ **Comment**: Q is organized as a queue
for all $u \in U - \{y\}$ **do** $req[u] = 0$;
$req[y] = h$ **Comment**: $req[u]$ defines how many units have to be pushed out of u
while $Q \neq \emptyset$ **do**
    let v be an element of Q
    remove v from Q
    for all u such that $(v,u) \in F$ and **until** $req[v] = 0$ **do**
        $m = \min\{ac(v,u), req[v]\}$;
        $ac(v,u) = ac(v,u) - m$;
        if $ac(v,u) = 0$ **then** remove arc (v,u) from F
        $req[v] = req[v] - m$;
        $req[u] = req[u] + m$;
        add u to Q
        $g(v,u) = g(v,u) + m$;
    end until
end while

Pseudo code of $pull(y,h)$

Comment: Increases the flow g by h units pull from y to s
$Q = \{y\}$ **Comment**: Q is organized as a queue
for all $u \in U - \{y\}$ **do** $req[u] = 0$;
$req[y] = h$ **Comment**: $req[u]$ defines how many units have to be pulled out of u
while $Q \neq \emptyset$ **do**
    let v be an element of Q
    remove v from Q
    for all u such that $(u,v) \in F$ and **until** $req[v] = 0$ **do**
        $m = \min\{ac(u,v), req[v]\}$;
        $ac(u,v) = ac(u,v) - m$;
        if $ac(u,v) = 0$ **then** remove arc (u,v) from F
        $req[v] = req[v] - m$;
        $req[u] = req[u] + m$;
        add u to Q
        $g(u,v) = g(u,v) + m$;
    end until
end while

7.6.2 Analysis of the algorithm

7.6.2.1 Lemma

An arc a of $AN(f)$ is removed from F at some stage only if there is no forward augmenting path with respect to flow g in $AN(f)$ that passes through a.

Proof:

Arc a is deleted at a stage for two reasons

1. It may either be that $g(a) = c(a)$ or
2. *$a = (v,u)$ with $throughput(v) = 0$ or $throughput(u) = 0$*

Proof of Lemma 7.6.2.1

- Suppose that $g(a) = c(a)$
- This means that arc a is now saturated and may appear in an augmenting path in $AN(f)$ with respect to g only as a backward arc. Hence, the proposition follows
- Let us now consider the case when v *or* u has throughput zero
- Then, no input or output by another arc exists at the arc a and, therefore, $a = (v, u)$ cannot be used in any forward path
- This completes the proof

Result of each stage

7.6.2.2 Lemma

At the end of each stage, g is a maximal flow in $AN(f)$.

Proof:

- By Lemma 7.6.2.1, an arc is deleted only if it cannot belong to a forward augmenting path
- This never changes again since capacities are only reduced and arcs and nodes are deleted
- However, a stage ends only when node s or node t is deleted due to a zero throughput

Proof of Lemma 7.6.2.2

- Therefore, due to Lemma 7.6.2.1 and zero throughput in s or t, after completing a stage, there are no forward augmenting paths at all, and hence g is maximal
- This completes the proof

Improvement

7.6.2.3 Lemma

The s-t distance in $AN(f+g)$ at some stage is strictly greater than the s-t distance in $AN(f)$ at the previous stage.

Proof:

- The auxiliary network $AN(f+g)$ coincides with the auxiliary network of $AN(f)$ with respect to flow g
- Since g is maximal (Lemma 7.6.2.2), there is no forward augmenting path in $AN(f+g)$

Proof of Lemma 7.6.2.3

- Hence, all augmenting paths have length greater than the s-t distance in $AN(f)$ (that is the length of g)
- We conclude that the s-t distance in $AN(f+g)$ is strictly greater than the s-t distance in $AN(f)$
- This completes the proof

Correctness and complexity

7.6.2.4 Theorem

The max flow algorithm (with pseudo code given under 7.6.1) correctly solves the max-flow problem for a network $N = (s, t, V, E, c)$ in asymptotic time $O(|V|^3)$.

Proof:

Correctness:

After performing the last stage, we have s and t being disconnected. Hence, the total augmentation flow in network $N(f)$ is zero.

Proof of Theorem 7.6.2.4

- By Lemma 7.5.2.3, we know that the total size $|g|$ of the maximum flow g in network $N(f)$ amounts to $|g| = |\hat{f}| - |f|$, while $|\hat{f}|$ is the total size of the maximum flow in the original network N
- Thus, we obtain $|g| = |\hat{f}| - |f| = 0$ and, therefore, $|\hat{f}| = |f|$
- This proves the optimality of the current flow f

Time bound

- Due to Lemma 7.6.2.3, we have at most $|V|$ stages, since the s-t distance increases monotonously

Proof of Theorem 7.6.2.4

- At each stage at most each node is chosen to transfer its minimal throughput
- Moreover, at most each arc is used completely only one time (afterwards, it is deleted)
- However, an arc may be also used partially and this can happen many times
- But, push and pull operations are initiated by each node at most once (afterwards, the node is deleted since its throughput is now zero)

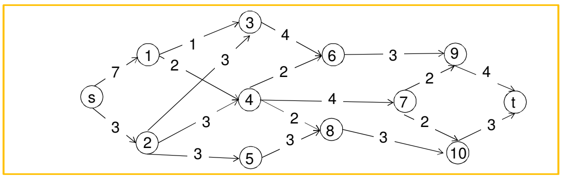
  Each push and pull operation contains at most $|V|$ steps by enumerating the nodes systematically

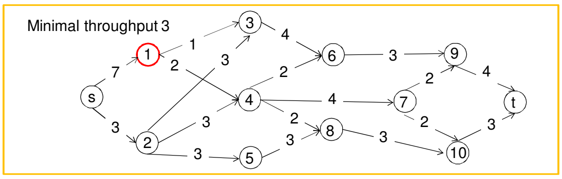
Proof of Theorem 7.6.2.4

- All in all, we have
  - At most $|V|$ stages
  - At each stage
    - At most $|V|^2$ steps that use an arc partially
    - At most $|E|$ steps that use an arc completely
  - Thus, the total asymptotic running time amounts to

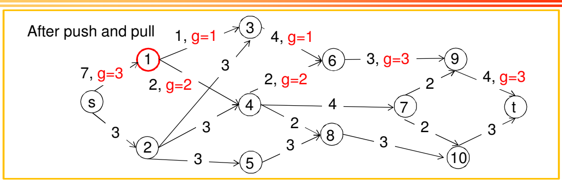
$$\boxed{O\left(|V|\cdot\left(|V|^2+|E|\right)\right)=O\left(|V|\cdot\left(|V|^2\right)\right)=O\left(|V|^3\right)}$$

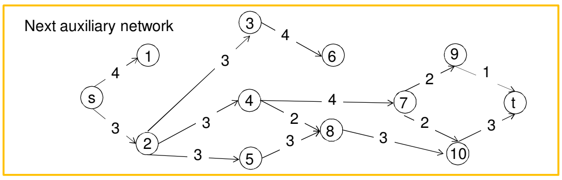
Example



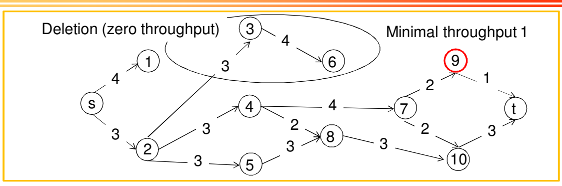


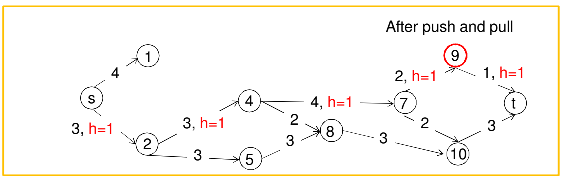
Example – stage 1: first node





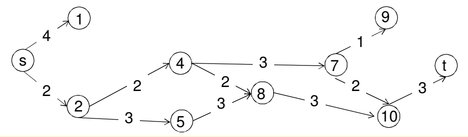
Example – stage 1: second node



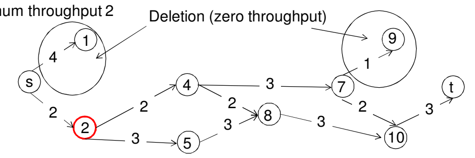


Example – stage 1: third node

Next auxiliary network

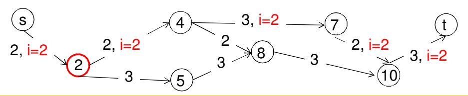


Minimum throughput 2



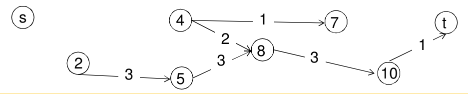
Example

After push and pull



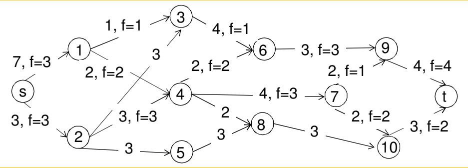
Next auxiliary network

t is not reachable from s anymore



Example – termination

- Since t is not reachable from s in $AN(g + h + i)$, the procedure terminates
- The maximal flow is given through $g + h + i$ and has a total size of 6



Additional literature to Section 7

- Edmonds, J.; Karp, R.M. (1972): Theoretical Improvement m Algorithmic Efficiency for Network Flow Problems. Journal of the ACM, vol. 19, no. 2 (April 1972), pp. 248-264.
- Ford, L.R. JR., and Fulkerson, D.R. (1962): Flows in Networks, Princeton University Press, Princeton, N.J., 1962.

The efficient max flow algorithm was originally proposed in

- Karzanov, A.V. (1974): Determining the Maximal Flow in a Network with the Method of Preflows. *Soviet mathematics Doklady,* 15 (1974), pp. 434-437.

Additional literature to Section 7

The efficient max flow algorithm was considerably simplified in:

- Malhotra, V.M.; Kumar, M.P., and Maueshwari, S.N. (1978): An $O(|V|^3)$ Algorithm for Finding Maximum Flows in Networks," *Inf. Proc. Letters,* 7 (no. 6) (October 1978), pp. 277-278.
- Tarjan, R.E. (1983): Data structures and network algorithms. In SIAM CBMS-NSF Regional Conference Series in Applied Mathematics 44, Philadelphia, 1983. SIAM.