

INF 4130

September 27, 2018

Stein Krogdahl

- Today's topics are from Chapter 14:
 - Matchings in (undirected) graphs
 - Flow in networks (network = directed graphs with capacities, flows, etc.)
- These topics are strongly connected to:
 - Convexity, polyhedrons with integer corners etc.
 - These are treated more thoroughly in other courses, e.g. in
 - INF3100 - Linear optimization
 - To some extent also in INF4120 - Mathematical optimization

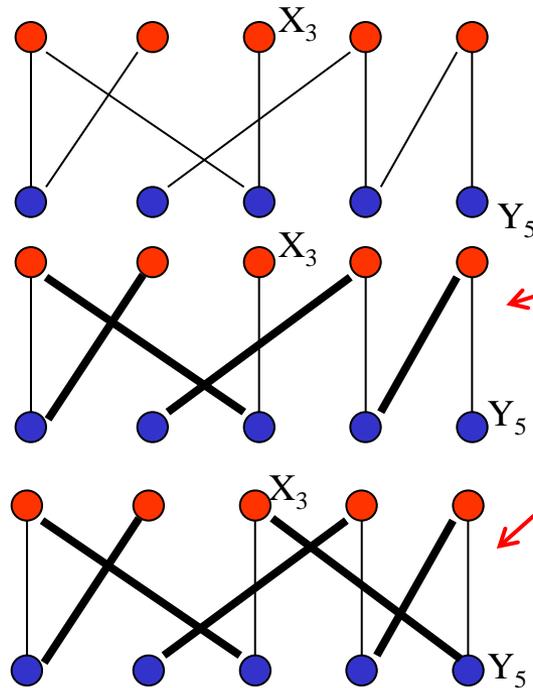
General information

- Deadline for Mandatory Assignment 1 is moved to **October 4**
- There is a lot of text on today's slides:
 - This is to make the slides more selfcontained for later study
 - I will to large extent explain things only relative to the figures
- Next week, last hour:
 - Guest lecture by International Grand Master in chess Rune Djurhuus (also master degree from IfI): «Chess playing programs»
- In two weeks, last hour:
 - Guest lecture by Torbjørn Rogstad, Professor at IfI in Bio-informatics: «Algorithms in Bio-informatics».

Matchings in undirected bipartite graphs, Ch.14.1

Bipartite graph = The set of nodes can be partitioned into two sets X and Y , so that each edge has one end in X and the other in Y

It is the same as a *two-colorable graph* or a graph without *odd loops*:



The node set X , e.g. workers in a workshop

The node set Y , e.g. the jobs of the day

Edges: Who has competence for doing the different jobs?

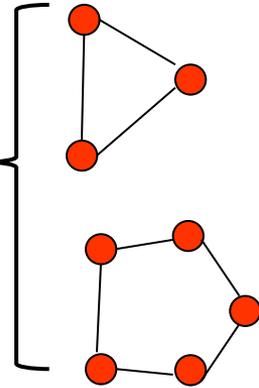
- Here, we are not able to find a "perfect matching", and thus all jobs cannot be done that day.
- However, if we add the edge $X_3 - Y_5$ we are suddenly able to find a "perfect matching", so that all jobs can be done.

Can be used in many different areas, e.g.:

Teaching assistants (X) each have a wishlist from the list of "groups" (Y). Can each teaching assistant get a group from his/her wishlist?

Some variations over the same theme:

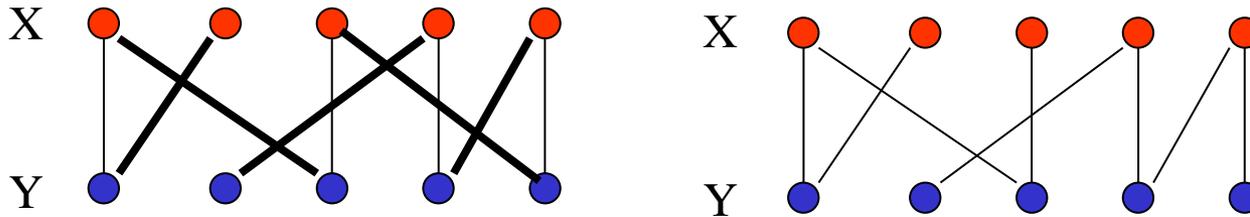
- We might have $|X| \neq |Y|$, and then there is obviously no perfect matching
- Even if there is no *perfect* matching, we are often interested in finding a match that is as large as possible.
- We can have «weights» on the edges, and ask for the matching with max. sum of weights



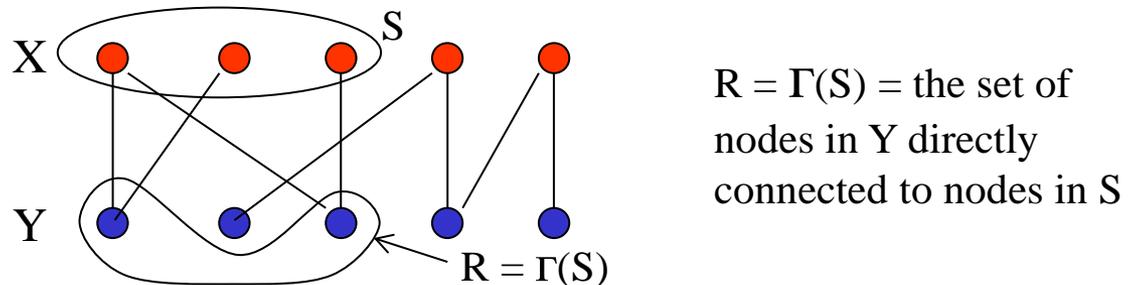
Hall's Theorem (1935):

When can we find a perfect matching?

Bipartite graphs with and without a perfect matching (same as on previous slide)



Below: A subset S of X is only connected to R in Y , and R has fewer nodes than S .



Then we can obviously not find a perfect matching. But this also works the other way around:

Hall's Theorem: There is a **perfect matching** if and only if there is **no subset S in X so that $\Gamma(S)$ has fewer nodes than S .**

Proof in the easy direction, as indicated above: If there is such an S so that $\Gamma(S)$ is smaller than S , there is obviously no perfect matching. We are not able to match each node in S with separate nodes in $\Gamma(S)$.

Proof in the difficult direction : The «Hungarien» algorithm will either give a perfect matching, or it will, when it stops without giving a perfect matching, point out an S where $|S| > |\Gamma(S)|$

Algorithms: The naive "greedy algorithm" doesn't work

Instance: Given a bipartite graph. **Question:** Find, if possible, a perfect matching.

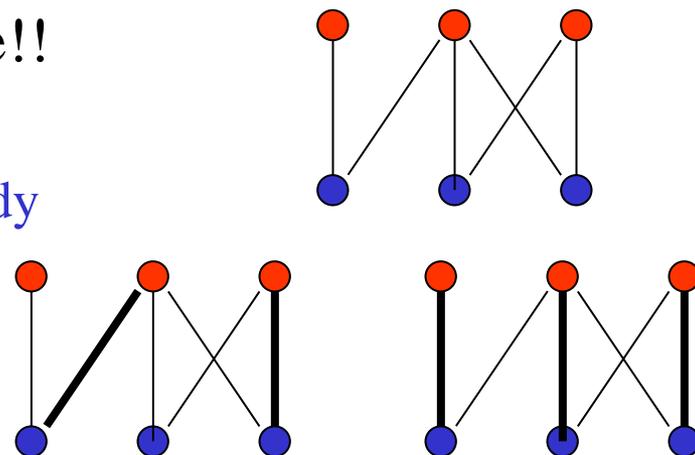
We could try a simple *greedy approach*, which could go as follows:

Look repeatedly at the edges of the graph, and include an edge in the matching if it has no node in common with an already included edge.

But the greedy strategy is not working here!!

Example:

Given the upper bipartite graph to the right. A greedy approach may, after two steps, give the matching to the lower left. However, there exists a matching with three edges (lower right), but we cannot use a simple greedy scheme to extend the left matching to one with three edges.



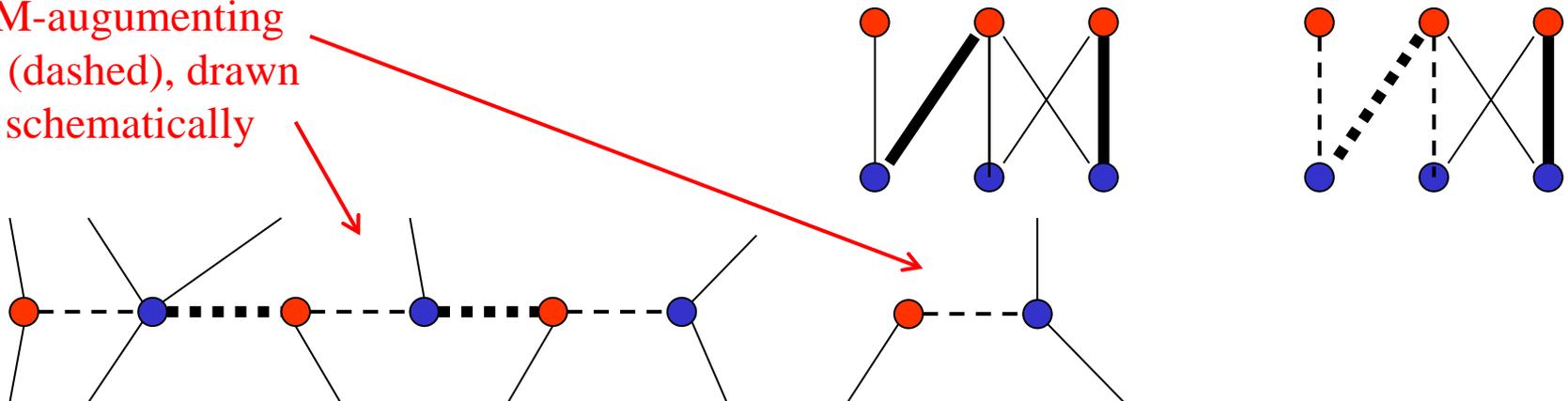
The Hungarien algorithm to find a perfect matching

We now assume that $|X| = |Y|$

- With the simple greedy strategy we only looked for «fully independant» edges, but even if we do not find one, there can be larger matchings
- However, it turns out that if we instead look for «**M-augumenting paths**», and (if we find one) use that to find a larger matching, the algorithm will work.
 - We will prove this below, and at the same time prove Halls Theorem
- An M-augumenting path:
 - Must first of all be an «M-alternating path», which is a (simple) path where alternating edges are in M and not in M.
 - In addition both end-nodes of the path must be «unmatched» (and then one end-node will be in X and the other in Y)

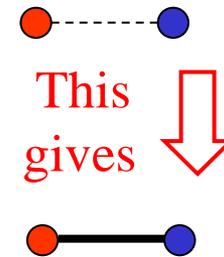
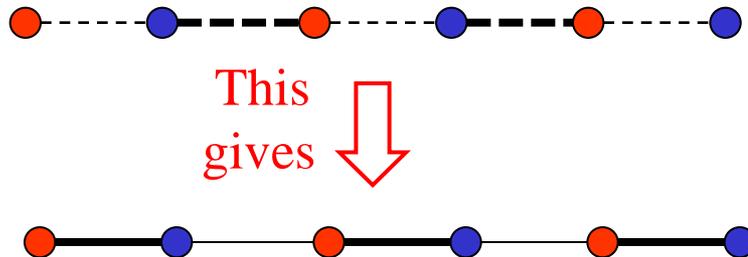
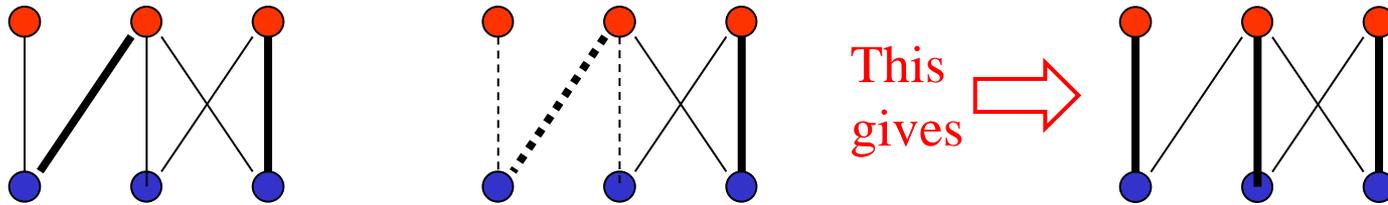
A bipartite graph with a matching M, and an M-augumenting path relative to M (dashed):

Two M-augumenting paths (dashed), drawn more schematically



We can «use» an M-augmenting path to obtain a larger matching

- If we have found an M-augmenting path, we can «obviously use this» to find a matching which is one edge larger (and is written $M \oplus P$):
- This new matching is, for the three dashed M-augmenting paths above, as follows:



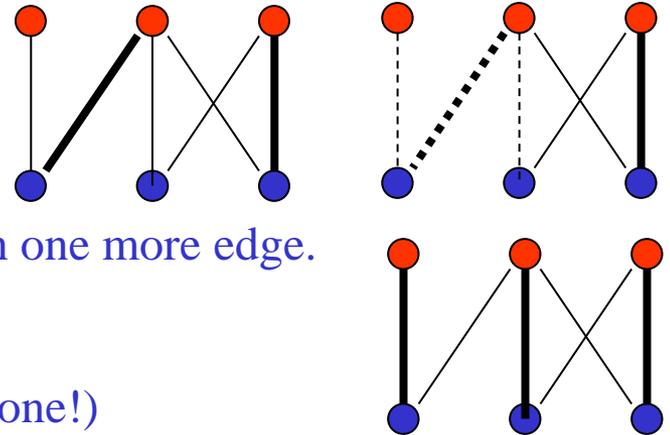
How can we find possible augmenting paths?

- The Hungarian algorithm goes as follows:

- Start with an empty matching, and repeat:

- a. Search for an augmenting path (see below)

- b. If you find one, use this to find a matching with one more edge.



Repeat a. and b. until

- Either: You have a perfect matching (and you are done!)

- Or: You cannot find an augmenting path relative to the current match M , by using the *Hungarian tree-building process* described below.

- In the last case, the situation will show us a subset S of X where the size of $\Gamma(S)$ (those in Y connected to S by an edge) is smaller than that of S .
 - Thus, if the algorithm stops in this way, we have a proof showing that there can't be any perfect matching in this graph.

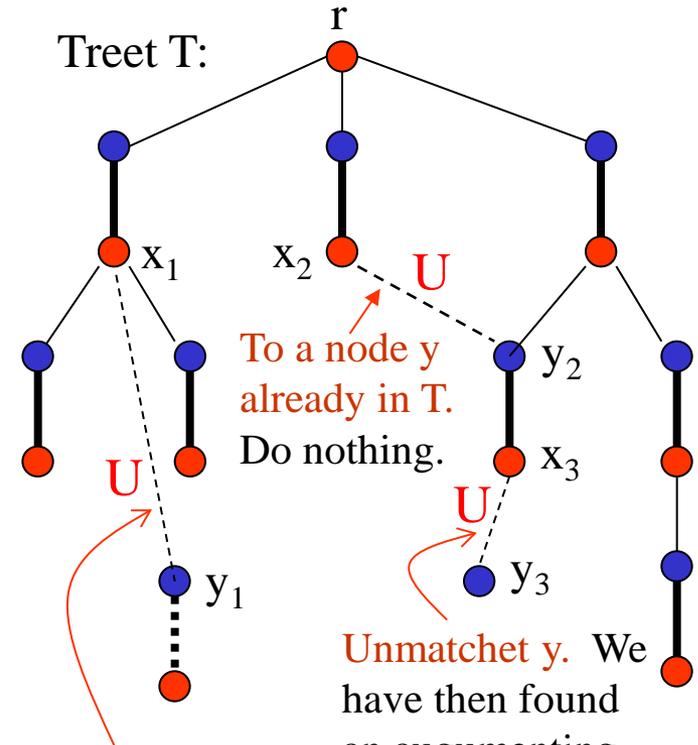
- The search for an augmenting path is done as follows (see figure on next slide):

- Choose an unmatched node 'r' in X . This node will be the root in a tree where all paths out from the root are alternating paths.

- We then grow the tree by adding two and two edges as explained in the next slide, until we we have found an augmenting path, or we cannot grow the tree any further by using legal steps.

To search for an augmenting path we “grow a tree”

- We assume that we have a matching M that is *not* perfect, and we will search for an augmenting path
- To try to find such a path we will build an *alternating tree* T . At the start the tree will consist only of a root node ‘ r ’ in X , which must be unmatched (and such a node can always be found when M is not perfect and $|X| = |Y|$)
- Building the alternating tree is done by repeating the following steps:
 - a) We search for an edge U which has its end-node in T , but is itself *not* in T . The other end-node y may be inside or outside T .
 - b) If we can find such an edge, there are three cases:
 1. The node y is already in T : Then we do nothing
 2. The node y is **unmatched**. We have then found an **augmenting path**, and we can use that to find a larger M (as seen earlier)
 3. The node y is a **matched** node in Y . We then include in T the chosen edge $U=(x,y)$, and the edge adjacent to y in the matching. The tree T will then be extended by two edges/nodes.



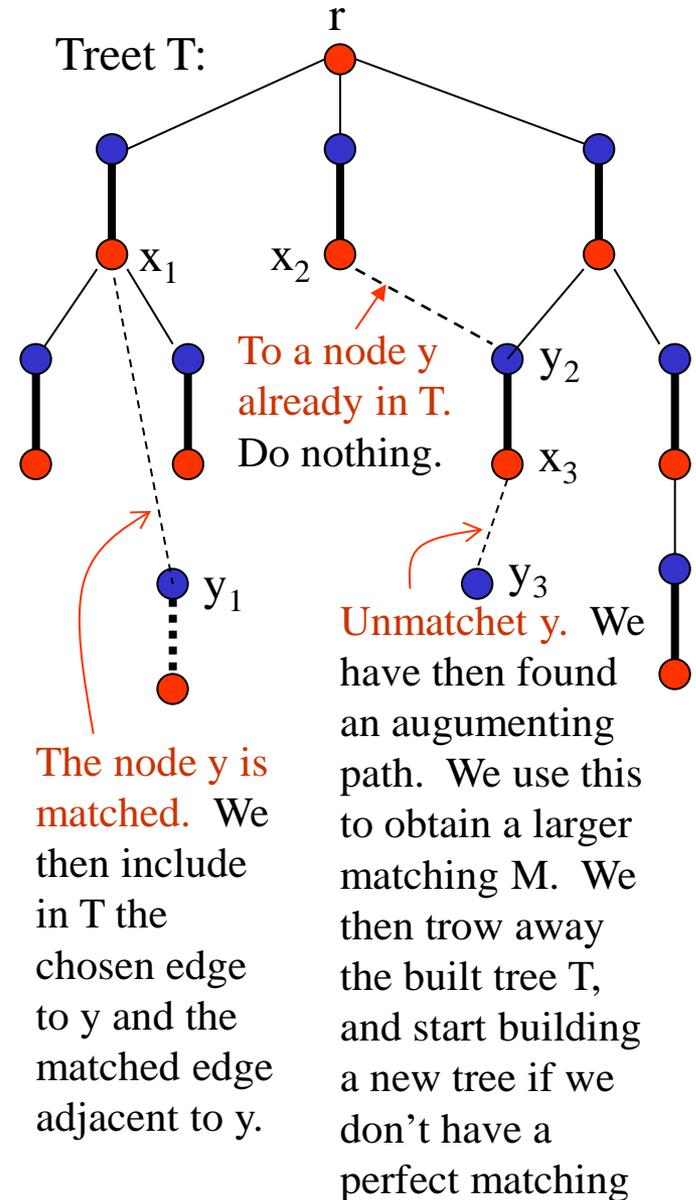
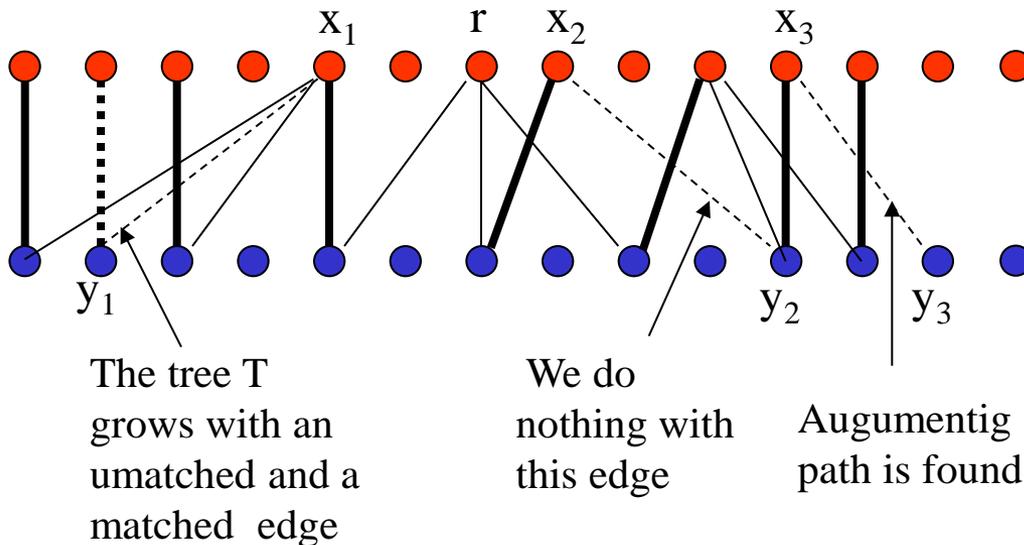
The node y is matched. We then include in T the chosen edge to y and the matched edge adjacent to y .

We have then found an augmenting path. We use this to obtain a larger matching M . We then throw away the built tree T , and start building a new tree if we don't have a perfect matching

Different drawings of a half-grown tree

A half-grown tree, drawn as to the right, looks nice and clean. Note that here only the the node and edges of the tree are drawn, and a few potential new ones. There may be a number of other nodes and edges.

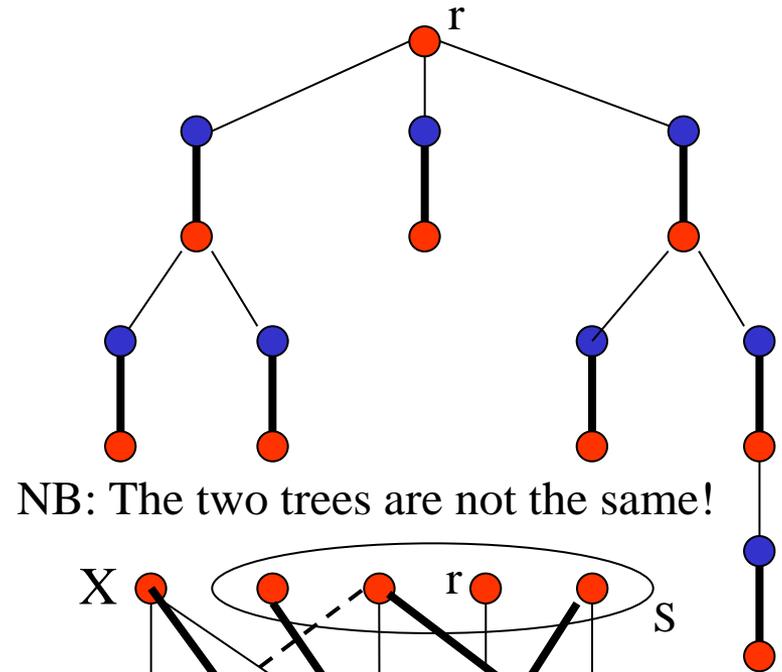
But the tree can obviously also be drawn inside the bipartite graph. Then it will look as shown below (where all nodes of the graph, but only the edges currently of interest, are drawn). However, it is easier to get an overview in the picture to the right



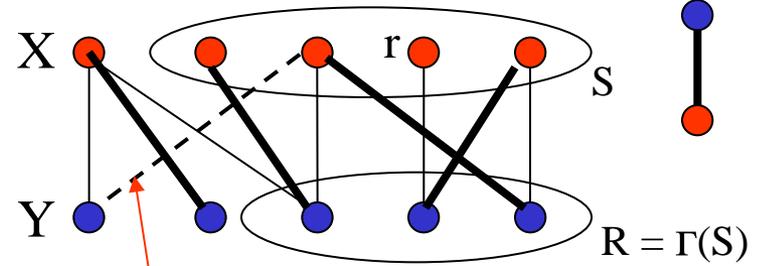
Termination of the Hungarien algorithm

The case when we don't find a perfect matching

- Assume that, when we are growing a tree, the algorithm stops because we can't find any edge between a red node in the tree and a blue node outside the tree. Then at least *this* search did not find any augmenting path. Our hope is then that this stopping situation will point out a «Hall-situation» which shows that no perfect matching can be found at all:
- We want: A subset S of X such that the set of nodes $R = \Gamma(S)$ in Y is smaller than S .
- For this we simply choose S to be the red nodes in the tree T . The number of nodes in S is then one larger than the number of blue nodes in the current tree (node r makes up an extra red node)
- We then claim that the only nodes in Y connected to a node in S are the blue nodes in T . Proof:
- If there were an edge from S to $Y-R$, then the algorithm would not have stopped.



NB: The two trees are not the same!



No such edge exists (or we would have used it!)

Thus, Hall's Theorem is proven:

The Hungarien algorithm can be run on any bipartite graph with $|X|=|Y|$, and it will **either** give a perfect matching **or** it will give a set S in X so that $|S| > |\Gamma(S)|$, proving that no perfect matching exists.

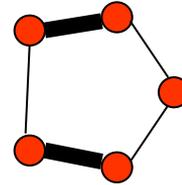
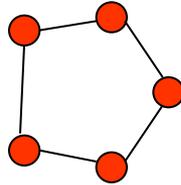
Variations of the matching problem

- Studied until now:
 - Find a perfect matching in a bipartite graph with $|X| = |Y|$ (or show that no one exists)
 - A sketch of a program for this algorithm is given at page 422/423
- Variants of the problem (which can also be solved in similar ways)
 - Find a matching with as many edges as possible (and then X and Y don't have to be of the same size)
 - We shall look at this as an exercise.
 - Given «weights» on the edges: Find a perfect matchingt with as high weight as possible.
 - Is described in the textbook (Ch 14.1.3), but is not part of the curriculum

Matchings in graphs that are *not* bipartite

Note: The slides below on this topic *is* included in the curriculum!

May have "odd" loops:

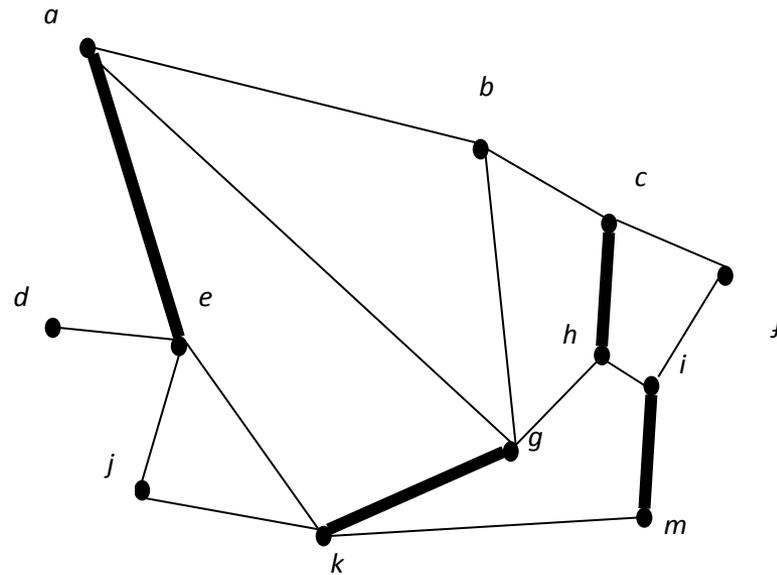


These are "difficult" for matchings.

The matching problem for general graphs:

- Pose the same questions as for bipartite graphs:
 - Find a perfect matching (or show that no one can be found)
 - Find a matching with as many edges as possible
- These can also be solved in polynomial time!
- We look at an algorithm for largest matching in general graphs:
 - This algorithm is only *slightly* more complicated to describe
 - But it is *much more complicated to prove that it really works*
 - It is part of this years curriculum to know the algorithm itself, but *not* how it can be proven correct (we won't even look at that)
 - The algorithm is a generalization of that for the bipartite case, with one more case in a few places.

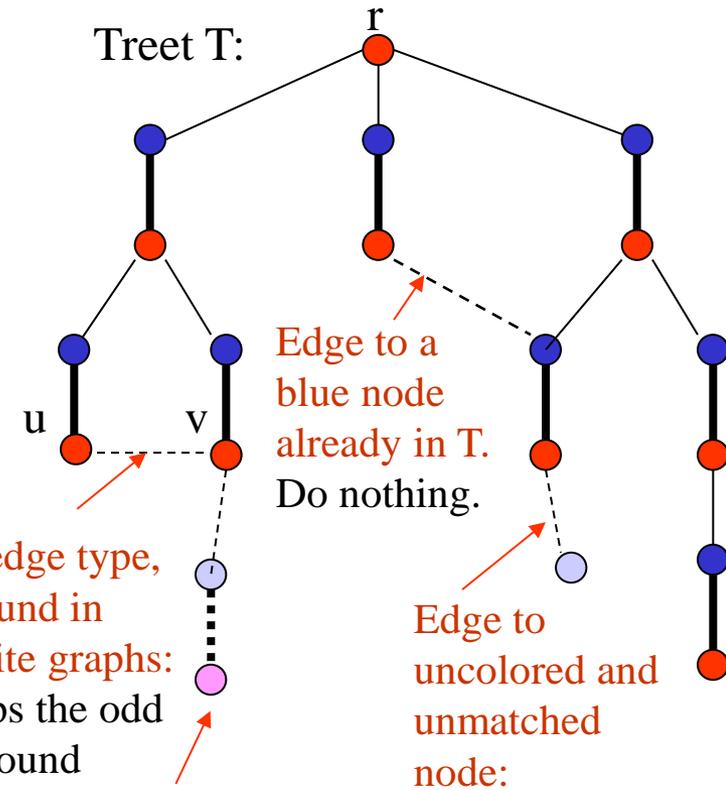
Example: A non-bipartite graph with a non-perfect matching



The main step in the «Extended Hungarien Algorithm»

New elements in the extended algorithm:

- There should be *no* node colors at the outset
- Each tree building starts with an unmatched node. We color it red, and it will be the root of the new tree
- As the graph is not bipartite, there may be edges from red to red nodes, like the edge (u,v) in the figure to the right. This will form an «odd loop» with the rest of the tree. (there may also be blue-to-blue edges, but we don't care about these!)
- This loop is treated by simply «collapsing» it (including its internal edges) to one red node.
- If a tree stops without finding an augmenting path, start with another unmatched node as root.



New edge type, not found in bipartite graphs: Collaps the odd loop found

Edge to a matched uncolored node: We color the node blue, and the corresponding matched node red, and include both nodes in the tree

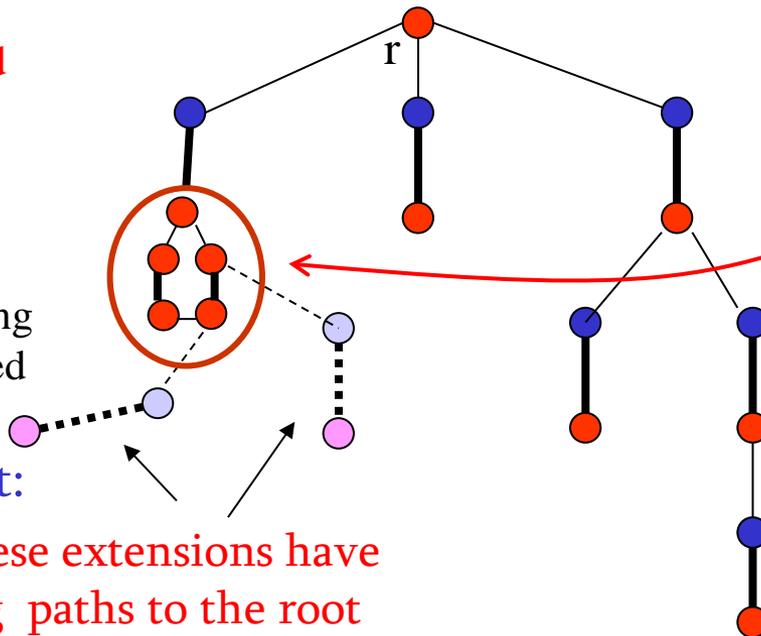
We have then found an augmenting path, and we can use it to get a larger matching.

Red collapsed nodes:

They all have an alternating path back to the root, starting with a matched node

Important:

Both of these extensions have alternating paths to the root



The end of a treebuilding in the Extended Hungarian algorithm

If we find an augmenting path to an unmatched node in the graph with some collapsed nodes:

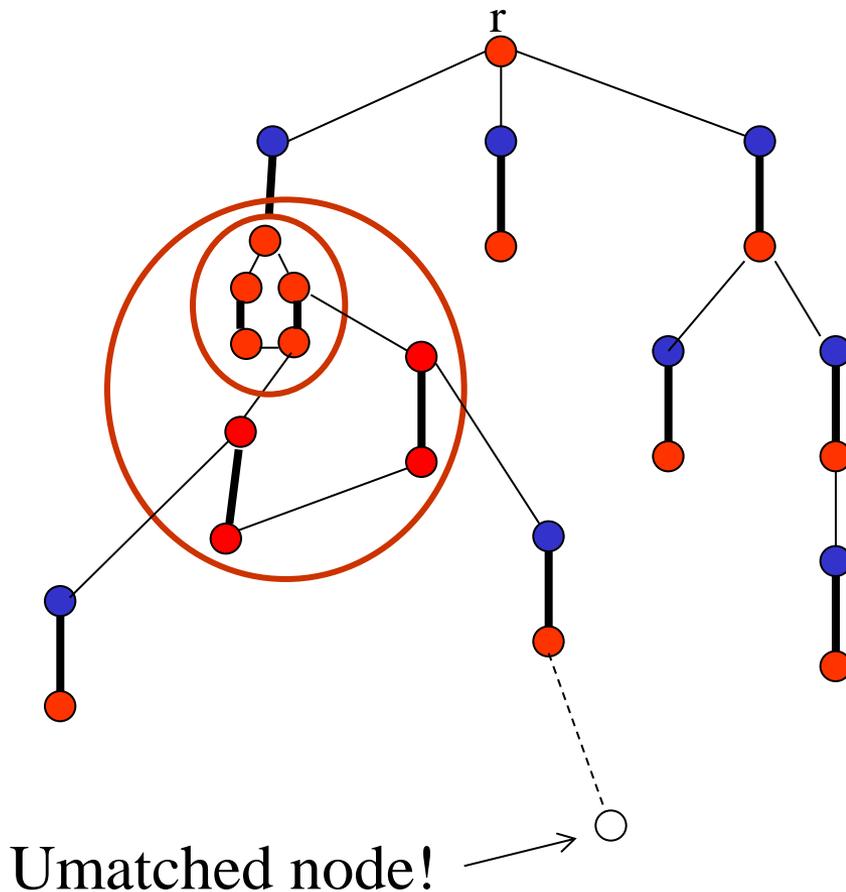
We go backwards along the alternating path, and along the way we unpack the collapsed (red) nodes, and find the alternating path through them.

We thereby get an alternating path in the original graph back to the root.

We can use this to find a matching that is one edge larger than the one we have.

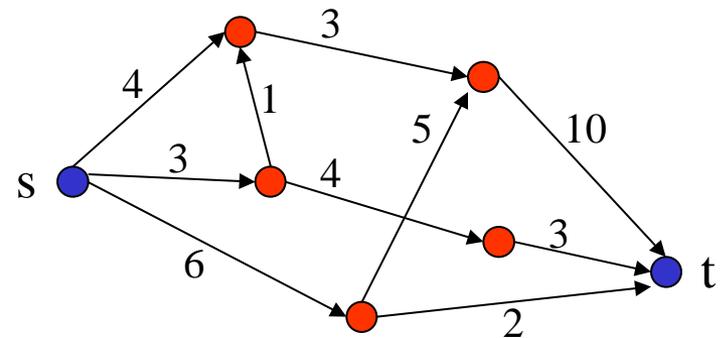
Otherwise the treebuilding stops because there are (1) no unexplored edge from a red node, and (2) no more unmatched nodes that can be the root of a new tree.

- *Then no larger matching will exist!!*
- But this is quite complex to prove, and such a proof is *not* part of the curriculum



Flow in Networks, Ch. 14.2

- This stuff is, to some extent, also covered in «the Weiss Book» (INF2220), so one may also read about it there.
- The use of the word «Network» is simply a tradition in this area. It is the same as directed graphs, usually with some weight, capacity etc. for edges and/or nodes.
- There are a lot of practical problems that can be seen as flow problems in networks.
 - Data nets, where there is a flow of data packages through the edges.
 - Different types of pipe-networks where fluid or gas can flow, and where each pipe has a *capacity*
 - Networks of roads with different capacities, where cars are «flowing» on the roads.
- The networks we shall study here have:
 - A capacity c on each of the edges
 - One *source* node s and one *sink* node t
 - And the goal is usually to find a largest possible flow from s to t

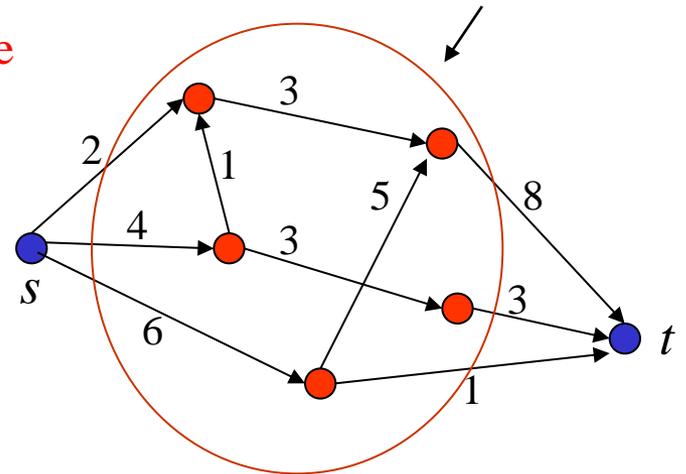


A network with capacities

Flow in networks: Ch. 14.2

- A flow f in a network is composed of one flow $f(e) \geq 0$ for each edge e , with the follow properties:
 - **Flow conservation:** For each node except s and t , the sum of flow into the node is equal to the sum of the flow out of the node (where *into* and *out of* is defined according to the directions of the edges).
 - **In networks with capacities:** Each edge has a capacity $c(e) \geq 0$, and the flow $f(e)$ must be in the interval 0 to $c(e)$
- We assume during the following discussion:
 - There are no edges leading into s or out of t .
 - $val(f)$ is by definition the sum of the flow out of s .
 - **Lemma:** The sum of the flow into t is the same as $val(f)$
 - Can be proved by summation of the flow in and out of all «internal» (red) nodes

A network without capacities. A legal flow is given:

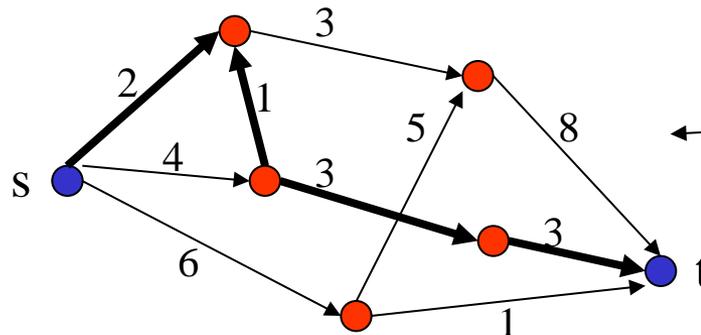


Concepts used in the book, that we don't use here.

These details are therefore not important at the exam!

- A *semipath* through the network is a path from s to t in the underlying undirected graph.
- The *characteristic flow* of a semipath S : This is a flow with value 1 where it follows the edge forward, and value -1 in the other edges (and it is therefore not a proper flow as $f(e)$ may be negative)
- **Lemma for networks without capacities:** Two legal flows may be summed at each edge, and the result is a new legal flow.
- **Lemma for networks without capacities:** If we multiply the edgeflow of each edge with a certain non-negative constant, we get a new legal flow

No capacities.
The numbers
are edge flows.

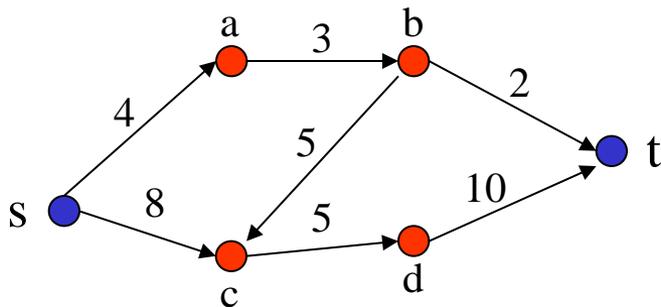


A (legal) *flow*
and a
semipath.

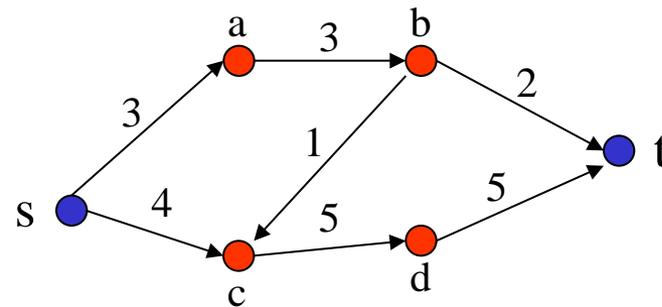
Flow in networks with capacities

- Each edge has a capacity $c(e)$, and the flow $f(e)$ must be between 0 and $c(e)$.
- Our goal:
 - Given a network with capacities
 - We want to find edgeflows $f(e)$ that
 - Satisfy the capacity requirement $0 \leq f(e) \leq c(e)$
 - Forms a maximum flow (there are no legal flow with larger $val(f)$)
- The example below: To the left is a network with given capacities.
 - We can easily find a legal flow of 7 (given right). Can we find any larger?

A network with capacities

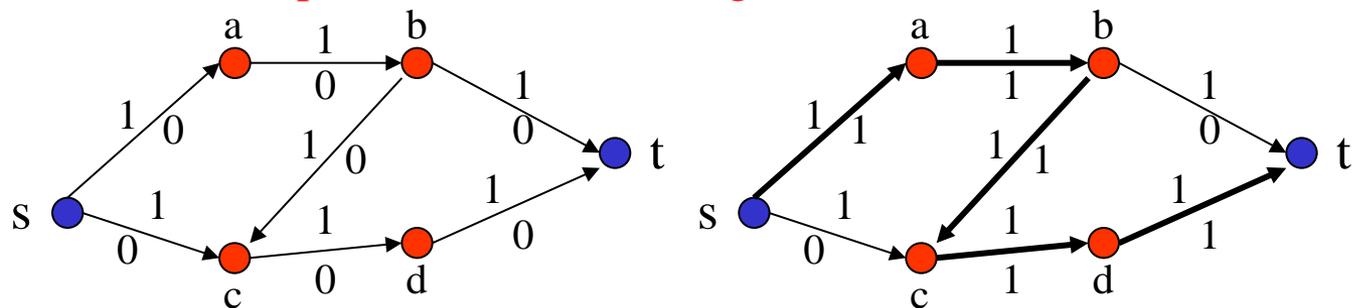


A flow f with $val(f) = 7$. Is this a maximum flow?



The naive greedy algorithm is again not working

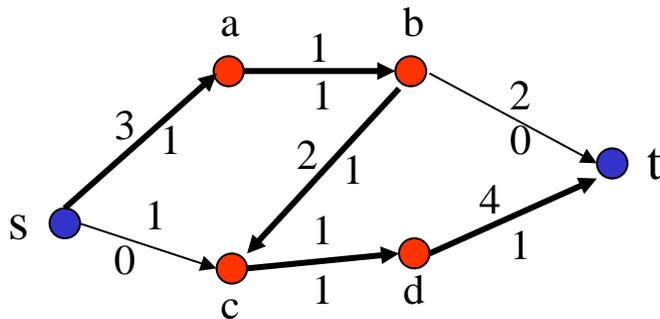
- The naive greedy algorithm (that in fact don't work!) could be as follows:
 - The step:
 - Find a directed, simple path from s to t where all the $f(e)$ are non-negative and *smaller* than $c(e)$
 - Increase the flow along this path as much as possible (dictated by the edge that has the smallest $c(e) - f(e)$ along the path)
 - Repeat this step until no such pathes can be found.
- In the figures below the capacity is given *above* the edges (all $c(e) = 1$) and the current flow is given below the edge (initially zero everywhere)
 - We first find a simple flow-increasing path, e.g. s - a - b - c - d - t . We can increase each edgeflow along this path with 1, and get the situation to the right.
 - Now $val(f) = 1$. But this is not a maximum flow, as we can easily find a flow with $val(f) = 2$
 - **BUT**, there is no flow-increasing path in this network that can bring us to a flow with value 2. Thus this simple scheme won't bring us to a maximum flow.



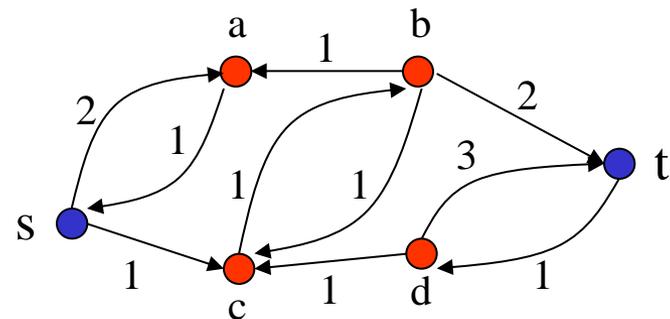
The f -derived network $N(f)$

- What we haven't taken into account on the previous slide, is that we, while searching for a larger matching, also can *decrease* the flow for edges with nonzero flow. And by utilizing this, we in fact get a working algorithm!
- To get an overview of the ways we can *change* the current flow on each edge we can construct the «*f-derived network*» often referred to as N_f , Nf , or $N(f)$. We shall here use $N(f)$.

(Note: the capacities below are different from those on the previous slide)



A network with capacities (above the edge) and a flow (under)

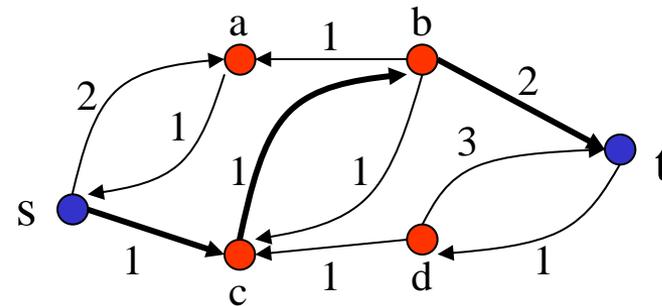
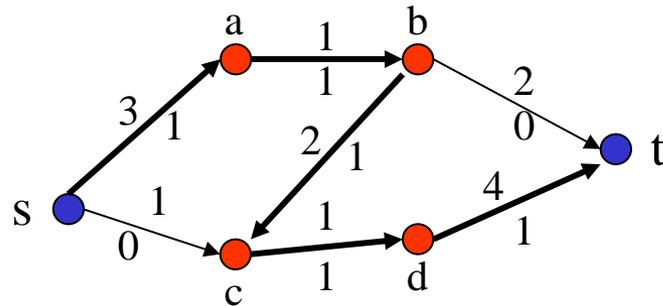


The f -derived network, of the situation to the left

Also, look at Figure 14.8 in the textbook (page 435)

Augmenting paths

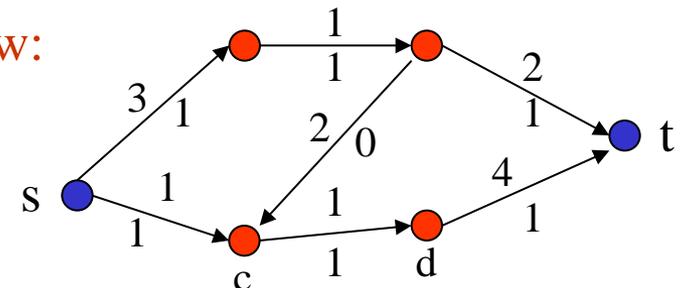
The network, capacities, flow and $N(f)$ is as in the previous slide:



- We search for paths from s to t in the f -derived network $N(f)$
 - Such paths are called f -augmenting paths
 - The search for such paths can be done e.g. with *breadth-first* or *depth-first* in $N(f)$.
 - We can e.g. choose the path $P = s-c-b-t$. The max. flow-increase along this path is here 1 (called h = minimum of possible increment over all the path edges)
- We then obtain the corresponding flow increase by, for all edges of P:
 - If the edge-direction in N is the same as in P : Increase the flow with h
 - If the edge direction in N is opposite to that of P : Reduce the flow with h

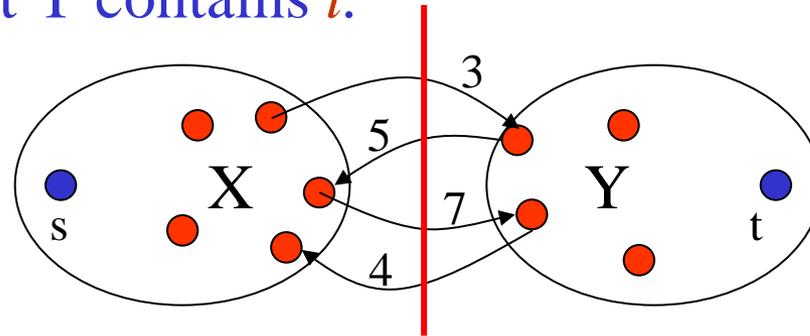
This gives the new flow:

- We then forget the old f -derived network, and build a new one relative to the new flow.



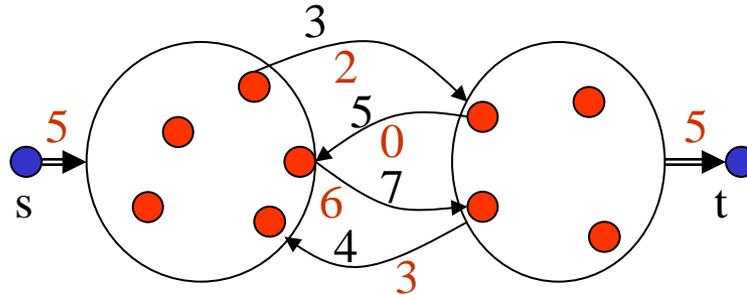
Cuts in networks

- A *cut* in a network is defined by a set X of nodes containing s but not t . The set of the rest of the nodes is then called Y , and we know that Y contains t .



- The *capacity* of a cut $K=(X,Y)$, written $cap(K)$, is the sum of the capacities of all edges leading from a node in X to a node in Y (disregarding edges from Y to X)
- In the figure above, the capacity of the cut is $3 + 7 = 10$
- Thus, the capacity of the edges leading from Y to X do not influence the capacity of the cut.

More about cuts in networks

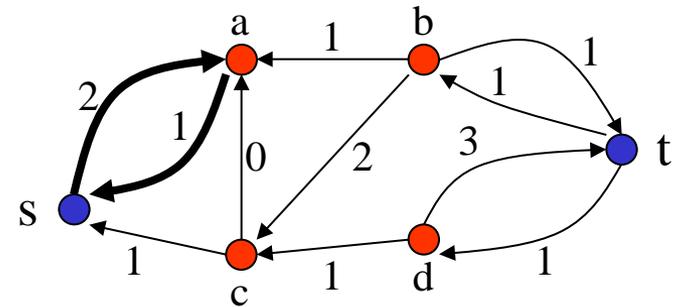
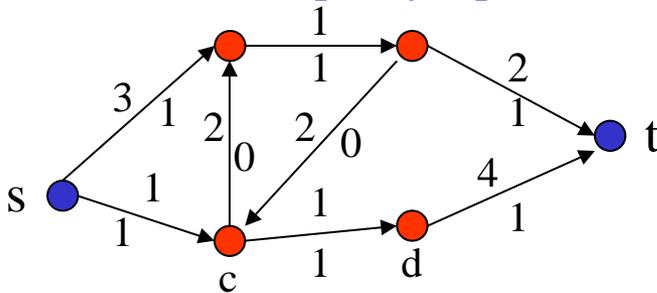


- Lemma:** Given a legal flow f and a cut $K = (X, Y)$. Then $val(f) \leq cap(K)$.
 This can be shown as follows:
 - By adding together the flow in/out of all nodes in $X' = X - s$, we find that
 (flow out of s) + (flow backwards over K) = (flow forwards over K)
 - As we by definition know that: (flow out of s) = $val(f)$ we know that
 $val(f) = (\text{flow forward over } K) - (\text{flow backwards over } K)$
 - The right hand side of the above equality is called the *flow over K* . As the last term is non-negative we know that the first term, which equals $cap(K)$ will obey: $val(f) \leq cap(K)$. Thus this is true for any cut K .
 - In the figure above: $val(f) = 5 = 2 + 6 - 0 - 3 \leq cap(K) = 3 + 7 = 10$
- This gives us a way to decide whether a given flow is optimal
 If we have a flow f and a cut K so that $val(f) = cap(K)$
 then we have a maximum flow, and there is no cut with smaller capacity!

The Ford-Fulkerson algorithm

The FordFulkerson-algorithm goes as follows:

- Start with zero flow (which is always a legal flow)
- The main step (and at the start of this we generally have any legal flow):
 - Find the f -derived network $N(f)$ (that shows all possible changes for the edgesflows)
 - Find, if possible, an f -augmenting path from s to t , and find the maximum increase it allows (before any of the edgeflows exceed the capacity or will go under zero).
 - Do the changes that this f -augmenting path indicate
- Repeat this step until we can no longer find an f -augmenting path in $N(f)$
 - The algorithm stops when there are no directed path from s to t in $N(f)$.
 - A proof showing that we now have a maximum flow, is that we can now point out a cut with capacity equal to the current flow. Thus, there can be no larger flow!

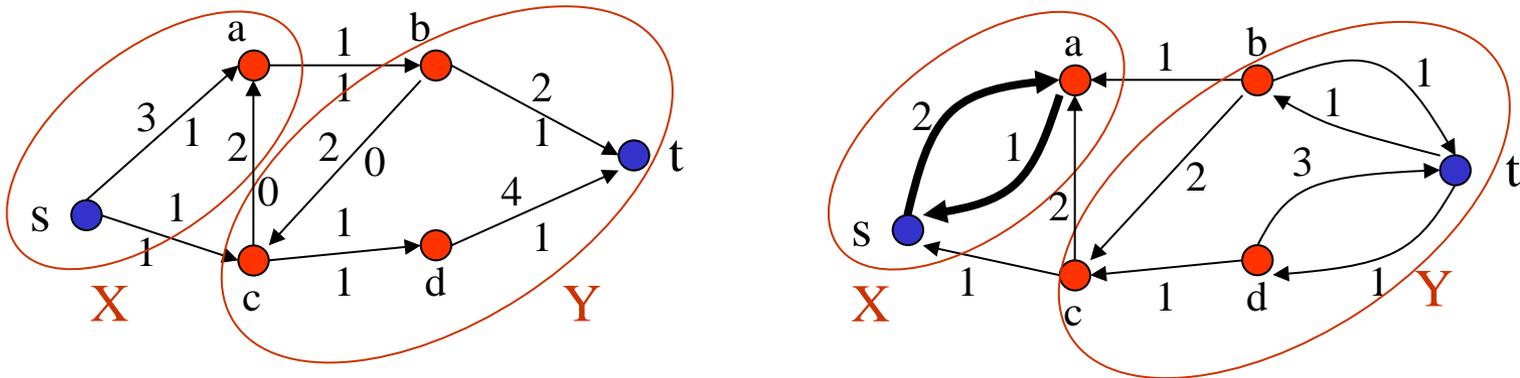


You should also look at the program at page 438

We will as an exercise look closer at this algorithm exemplified by figure 14.9

Termination of the Ford-Fulkerson algorithm

It stops when there is no connection from s to t in $N(f)$.

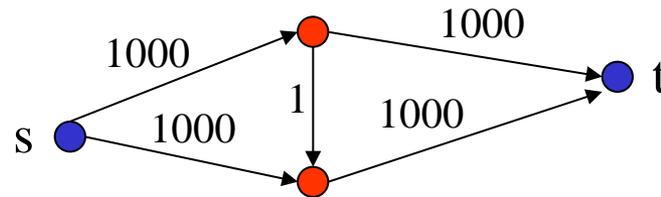


- **As indicated:** To show that we now have a maximum flow, we will show that we can construct a cut K with capacity equal to the current flow. That is: $cap(K) = val(f)$.
- It turns out that such a cut is easy to find: Let X be the set of nodes reachable from s in $N(f)$, and let Y be the rest of the nodes (including t).
- As no edges in $N(f)$ is leading from X to Y , we know by the def. of $N(f)$:
 - All edges in N (the original network) from X to Y are used to its full capacity.
 - All edges in N leading from Y to X have flow $f = 0$
- This means that $cap(K)$ equals the current flow over K , which again is $val(f)$.
- Thus, we know we have a maximum flow, and we have proven the following Theorem:

Theorem (Max-flow, min-cut): In a network with capacities we can find a flow f and a cut K so that $val(f) = cap(K)$. Then we know that we have a maximum flow, and that no cut has lower capacity.

Variations of the Ford-Fulkerson algorithm

- The **Ford-Fulkerson algorithm** says nothing about which f -augmenting path should be chosen in each step, if there are more than one.
- If we do not decide anything about the choice of f -augmenting paths, we know:
 - If all capacities are (positive) integers, then the number of steps can be as large as the size of the largest flow (but indeed, no larger as each incr. is an integer!):



n = number of nodes
 m = number of edges

- If the capacities are real numbers, the algorithm can in theory loop for ever.
- **Proposal 1:** All the time, choose the f -augmenting path that gives that largest possible increment in the flow. This one can be found by an algorithm similar to a shortest path algorithm)
 - This gives a worst-case-time: $O(m \log(n) \log(\max\text{-flow}))$
- **Proposal 2:** (Edmonds og Karp) All the time, choose the f -augmenting path that has the smallest number of edges (can be found by a breadth-first search)
 - This gives a worst-case-time: $O(n m^2)$
(and this independent of the max. flow, and this is very convenient)

Variations of the problem of max. flow

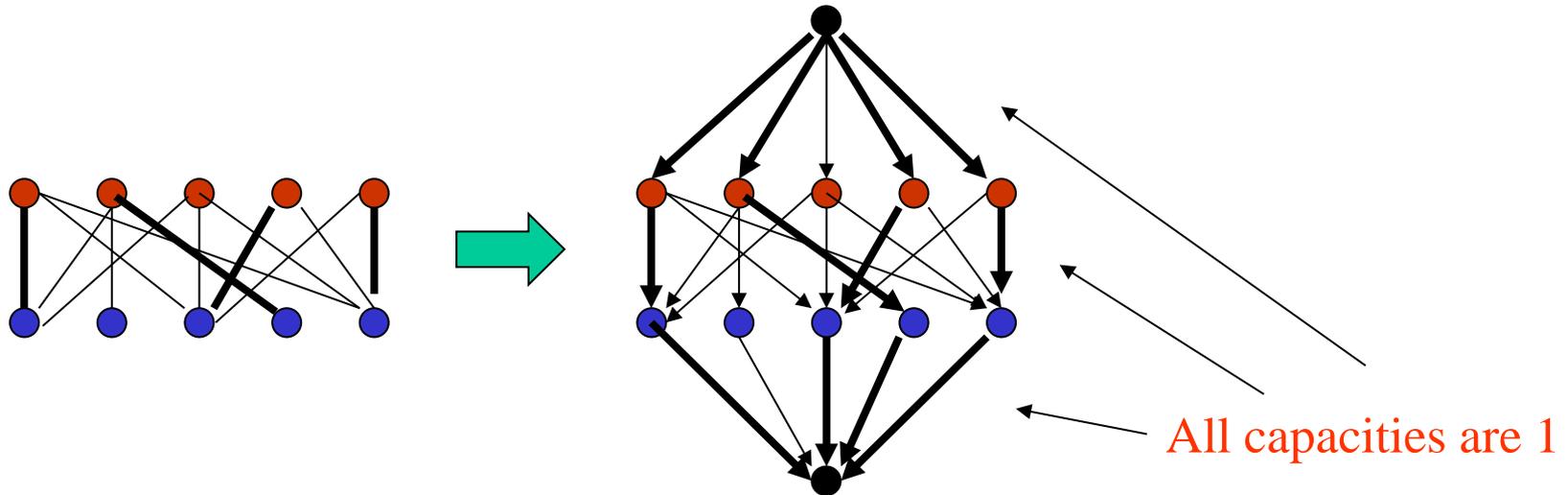
- First of all, there are alternatives to the Ford-Fulkerson algorithm
 - *Dinic* has designed an algorithm with time $O(n^2 m)$
 - *Goldberg and Tarjan* («preflow push algorithm», time $O(n^2 \sqrt{m})$)
- We may also have a minimal flow for each edge
 - Then it is an interesting problem just to find a *possible* flow
 - But after that you can proceed as in Ford-Fulkerson
- We may also have a price on each edge, saying how much a flow of one will cost over this edge. We want to minimize total cost for getting a certain total flow through the network.
 - On old algorithm here is the «Out-of-kilter algorithm» (not polynomial), but many later algorithms runs in polynomial time.
- We can also have multiple sources and/or multiple sinks, with different requirements to the flow in and out of these
- We may also have different "commodities" that should flow in the network (cars, busses, trucks, ... in a street network) , and the edges may have a different capacity and cost for each commodity.
 - This is a field of active research, in connection with e.g. traffic planning, routing in communication networks, etc.

Kap. 14.2.7: A connection between flow in networks and matching in bipartite graphs

A simple but important lemma, which is obvious from the Ford-Fulkerson algorithm and the max-flow-min-cut Theorem:

1. If we have interger capacities, then we can always find an interger max. flow.
2. And thus the Lemma: When all the capacities are 1, we can find a max. flow where all edgeflows are either 0 or 1.

Such a flow can be seen as pointing out a subset of the edges (those with flow 1)



Concerning the above picture, we will as an exercise look at:

- That searching for an M -augmenting path in the bipartite graph to the left, corresponds to searching for an f -augmenting path to the right.