INF 4130
22 October 2013

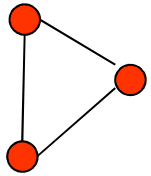
Stein Krogdahl

- Today's topics from Chapter 14:
  - Matchings in (undirected) graphs
  - Flow in networks (network = directed graphs with capacities etc.)

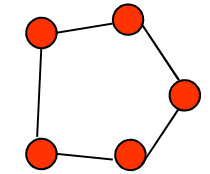
- These topics are stongly connected to:
  - Convexity, polyhedrons with integer corners etc.
  - This is treated more generally in other courses, e.g. in INF-MAT 4110: Mathematical Optimization.

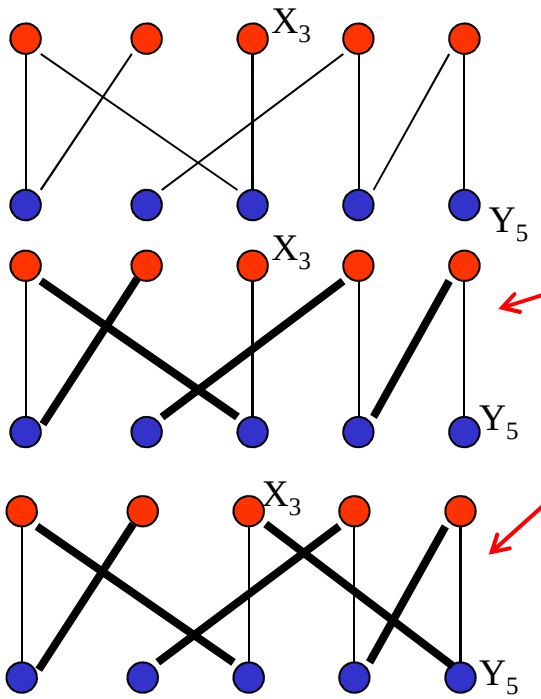
Matchings in undirected bipartite graphs, Ch.14.1

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 differnt jobs?

- We are here 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 assistents (X) each have a wishlist from the list of ”groups” (Y). Can each teaching assistant get a group from his/her wishlist?

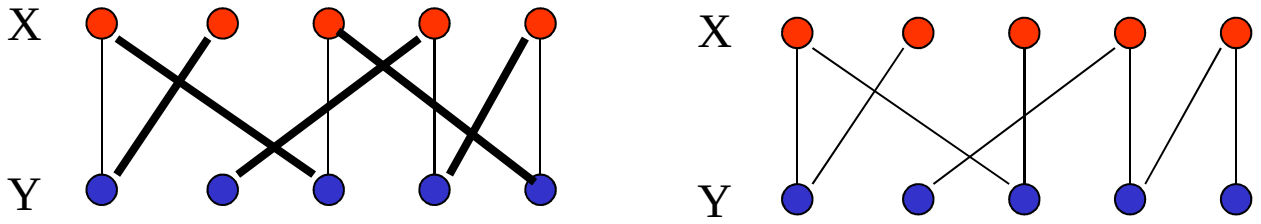
Some variations over the same theme:

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

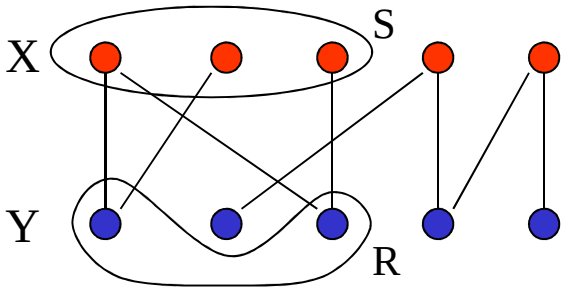
Hall's Theorem:

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.



R = Γ(S) = the set of nodes in Y directly connected with nodes in 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 so that R = Γ(S) has fewer nodes than S.

Proof in the easy direction, as indicated above: If there is such an S so that R = Γ(S) is smaller than S, there is obviously no perfect matching. We are not able to match each node in S with its own node in R.

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 with $|S| > |\Gamma(S)|$

The naive ”greedy algorithm” don’t work

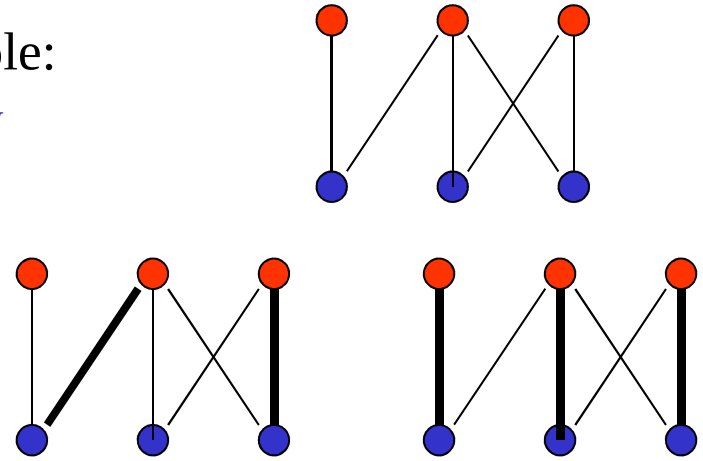
Instace: Given a bipartite graph. Question: Find, if possible, a perfect matching.

We could try a simple *greedy approach*, which can 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.

The greedy strategy is not working here. Example:

Given the bipartite graph to the upper 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 the one with three edges.



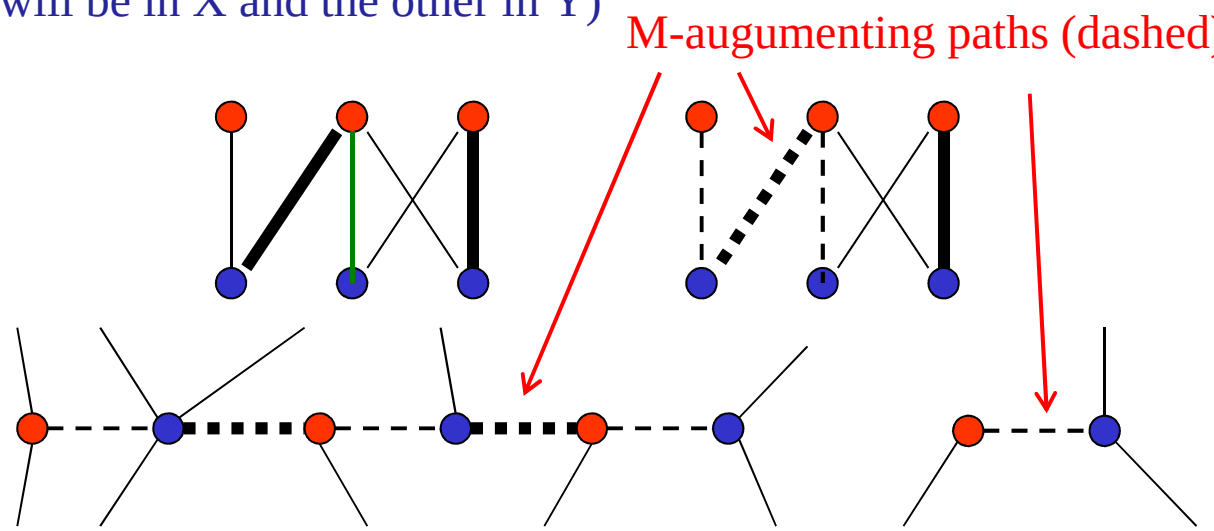
Reminder from INF2220: In connection with greedy algoritms the elements often also has a weight, A place where such a greedy algorithm really works is if we want the heaviest span tree in an undirected graph, where the edges have weights. Algorithm:

> ”Look at the edges in order of decreasing weight, and include those that does not make a loop with those already included (Kruskal’s algorithm).

The Hungarien algorithm to find a perfect matching

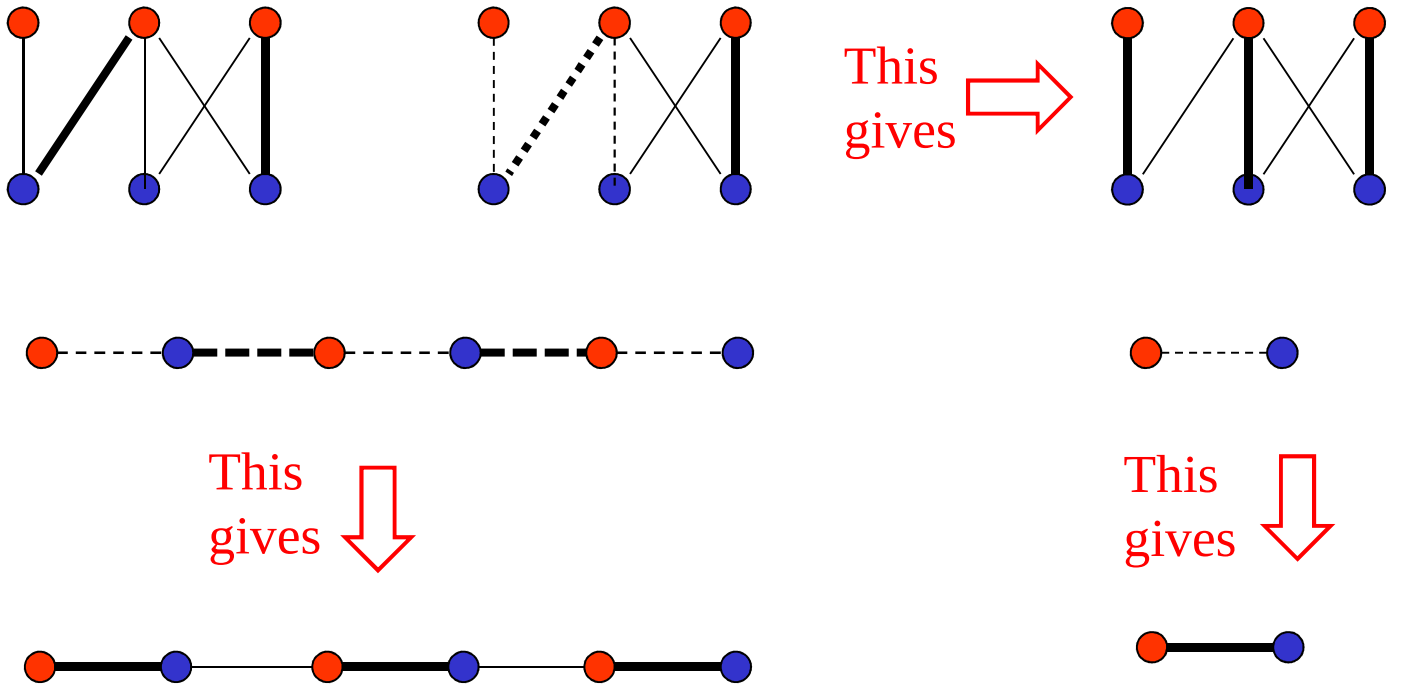
We assume here that |X| = |Y|

- With the simple greedy strategy we only looked for «fully independant» edges, when we wanted to increase the size of the current matching M.
- This was obviously too simple, but it turns out that if we instead look for «M-*augumenting* paths», and each time pick the best, the algorithm will work.
  - This can be shown directly, but we'll do it so that we also show 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)



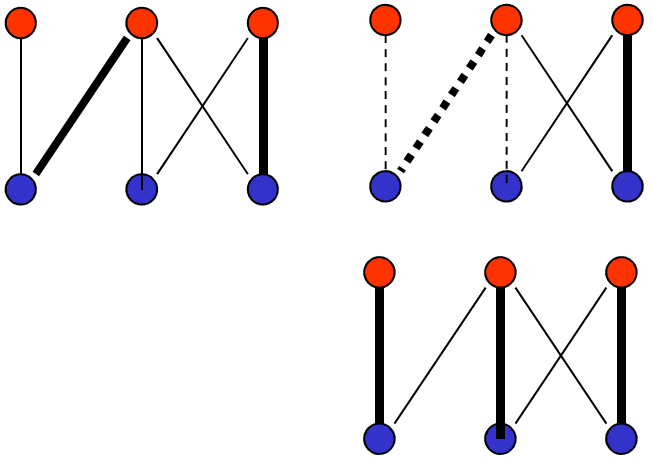
We can «use» an augumenting path to obtain a larger matching

- If we have found an augumenting path, we can obviously «use this» to find a matching which is one larger (written $M \oplus P$):
- The results are shown for the dashed M-augumenting paths below:



How can we find possible augumenting paths?

- The Hungarien algorithm goes as follows:
  - Start with an empty matching
  - Search for an augumenting path
  - Use this to find a matching with one more edge



Repeat this until either:

– You have a perfect matching

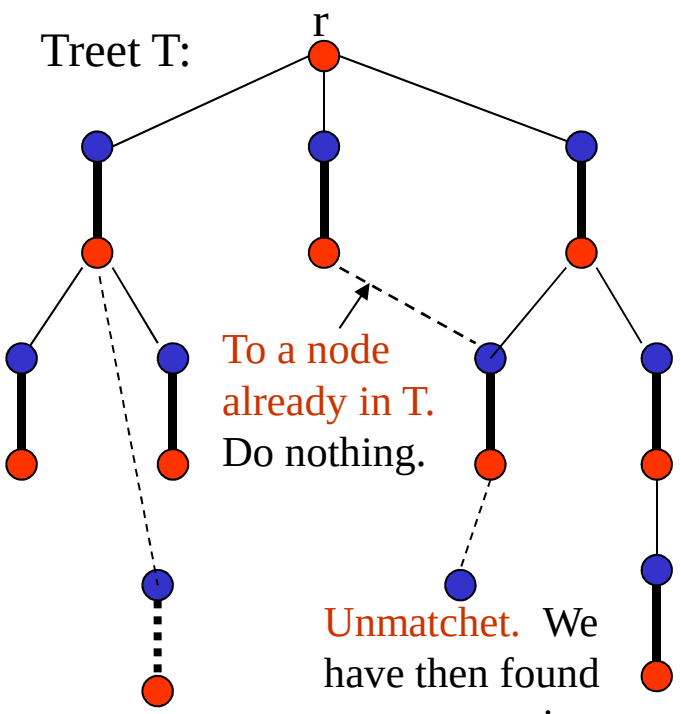
– Or you cannot find an augumenting path relative to the current M

- In the last case, the sitiation will hopefully show us a subset S in X that is connected only to the subset R in Y, where R is smaller than X.
- Thus, when the algorithm stops, we can show that there is no perfect matching.

- The search for an augumenting path is done as follows (See figure on next slide):
  – Choose an unmatched node ‘r’ in X. This node should be the root in a tree where all paths out from the root are alternating paths (which is, in fact, always the case in a bipartite graph)
  – We then grow the tree by two and two edges as explained in the next slide, until we we have found an augumenting path, or we cannot grow the tree any further by using legal steps.

Growing a tree to find an augumenting path

- We assume that we have a matching M that is not perfect, and we will search for an augumenting path
- This step should build an *alternating tree* T, and at the start the tree will consist only of a root node ‘r’ in X which must be an unmatched node in X (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 step:
- We search for an edge, not in T, between a *red* node in T and another node (which must be blue)
- If we find such an edge, there are three cases:
  1. The blue node is already in T: Do nothing
  2. The edge leads to an unmatched node in Y. We have then found an augumenting path, and we can use that to find a larger M.
  3. The edge leads to a matched node y in Y. We then include in T the chosen edge from T, and the edge adjacent to y in the matching. The tree T will then be extended by two edges/nodes.



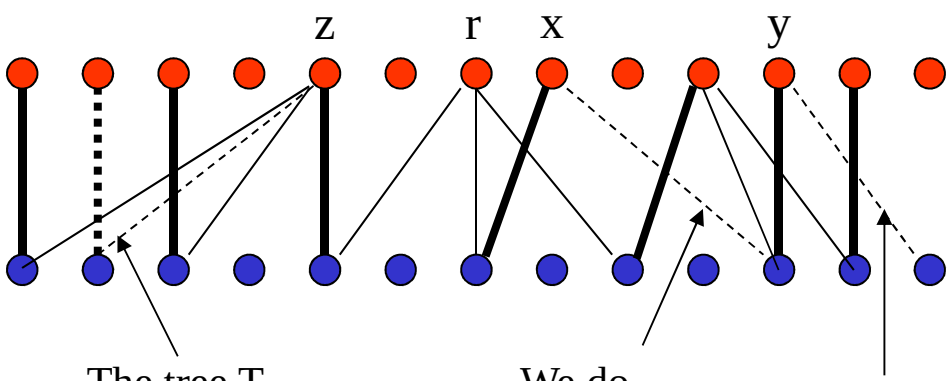
The node y is matched. We then include in T the chosen edge to y and the matched edge adjacent to y.

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

Growing a tree to find an augumenting path

The tree to the right looks nice and clean. Note that only the the edges of the tree, and a few potential new ones, are drawn. There may be a number of other nodes.

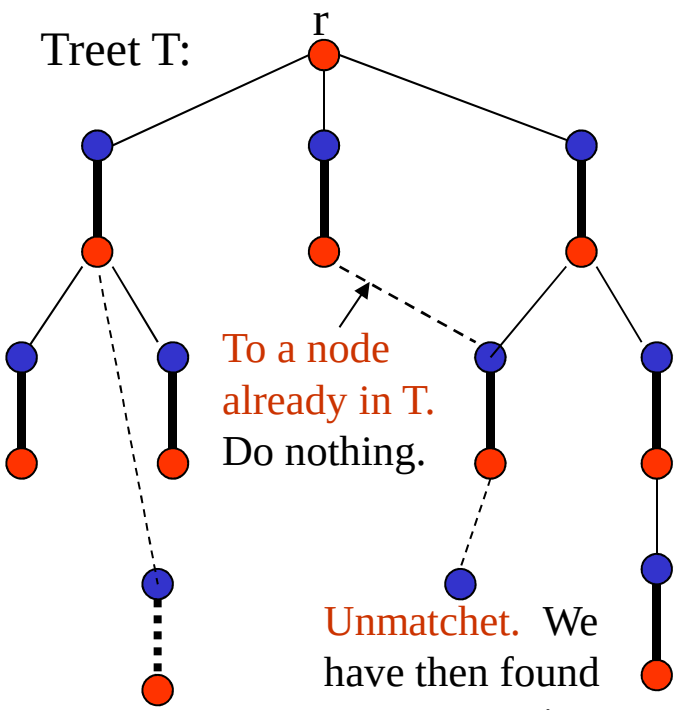
But the tree can obviously also be drawn in the bipartite graph. Then it looks as shown below (where all nodes, but not all edges, is drawn)



The tree T grows with an umatched and a matched edge

We do nothing with this edge

Augumentig path is found



To a node already in T. Do nothing.

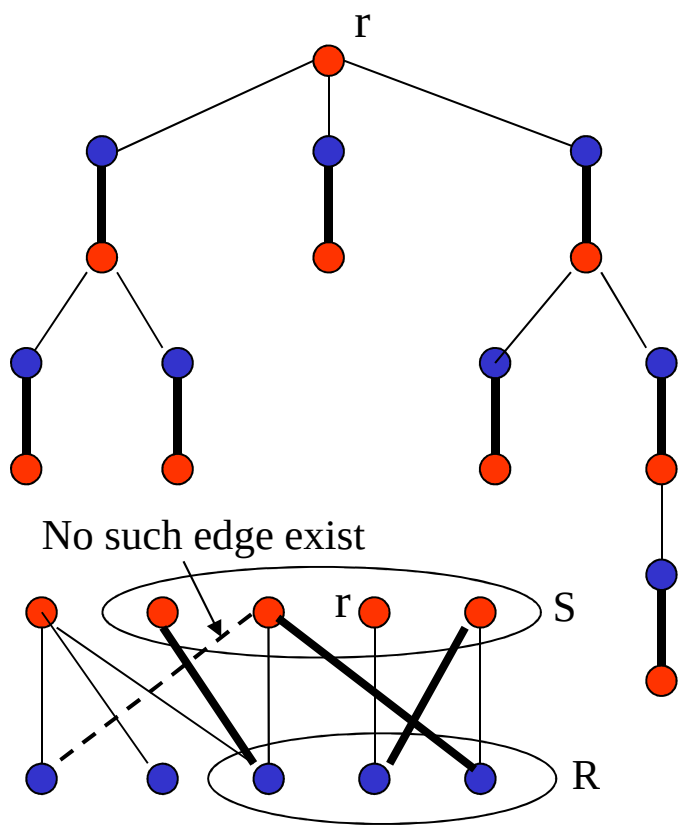
The node y is matched. We then include in T the chosen edge to y and the matched edge adjacent to y.

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

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 don't find any unused edge between a red node in the tree to a blue node outside the tree. Then at least this search did not find any augumenting path. Our hope is then that the situation will show us a «Hall-situation» that shows that no perfect matching can be found at all:
- We want: A subset of S of X such that the the set of nodes R = Γ(S) in Y is smaller than S.
- We then simply choose S as the red nodes in the tree T. The nuber of nodes in S is one larger than the number of edges in the current matching.
- 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 above are different

Thus, Hall 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 us a pair of sets S and R showing that no perfect matching can exist.

Variations over the matching problem

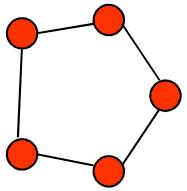
- Studied until now:
  - Find a perfect matching in a bipartite graph (or show that no one exists)
  - A sketch of a program for this algorithm is given at page 422/423

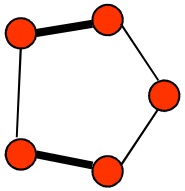
- Variants of the problem (which can also be solved in similar ways)
  - Find a matching with as many edges as possible (and then X and Y don't have to be of the same size)
    - We shall look at some of these as exercises next Thusrday
  - Given «weights» on the edges: Find a perfect matchingt with as high weight as possible.
    - Is described in the textbook (Ch 14.1.3), but is not part of the curriculum
  - Flow in «networks», where the matching for bipartite graphs occurs as a special case.
    - Also treated as an exercise next Thursday
    - Flow in networks will be studied in the next hour today.
  - Generalizing to graphs that are not bipartite
    - Will be discussed at next slides

Matchings in graphs that are *not* bipartite

The slides on this theme is included in the curriculum

May have ”odd” loops:





These are ”difficult” for matchings.

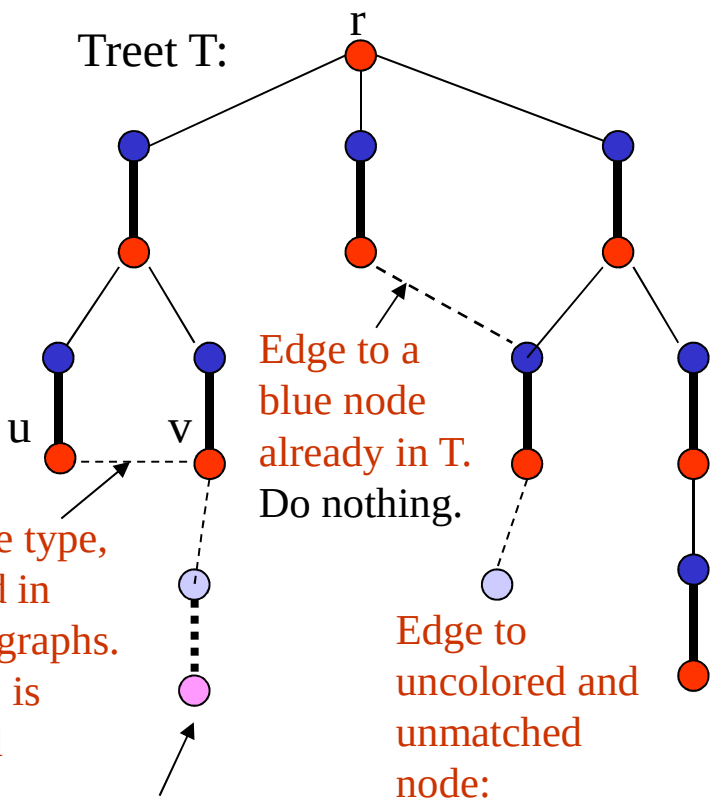
Generalization of matchings beyond bipartite graphs:

- Pose the same questions for general graphs:
  - Find a perfect matching (or show that no one can be found)
  - Find a meching as many edges as possible
  - With weights on the edges: Find a (perfect?) matching with the largest possible weight
- All thes can be solved in polynomical time
- Algorithm for matching in general graphs
  - This algorithm are sligthly more complicated to describe*, but it is considerably more complex to prove correct*.
  - As part og the curriclium you should know this algorithm, but not how it it can be proven correct.
  - The algorithm is a generalization of that for the bipartite case, with one more case a few places.

The step in the «extended Hungarien algorithm»”

New elements in the algorithm:

- There should be no node colors at the outset
- Each tree building starts with unmatched node. We color it red, and it will be the root of the new tree
- When the graph is not bipartite, there can be edges in the tree from 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.
- This loop is treated by simply collapsing it (including its internal edges) to one red node.
- If it stops without finding an augumenting path, start with another unmatched node as root.

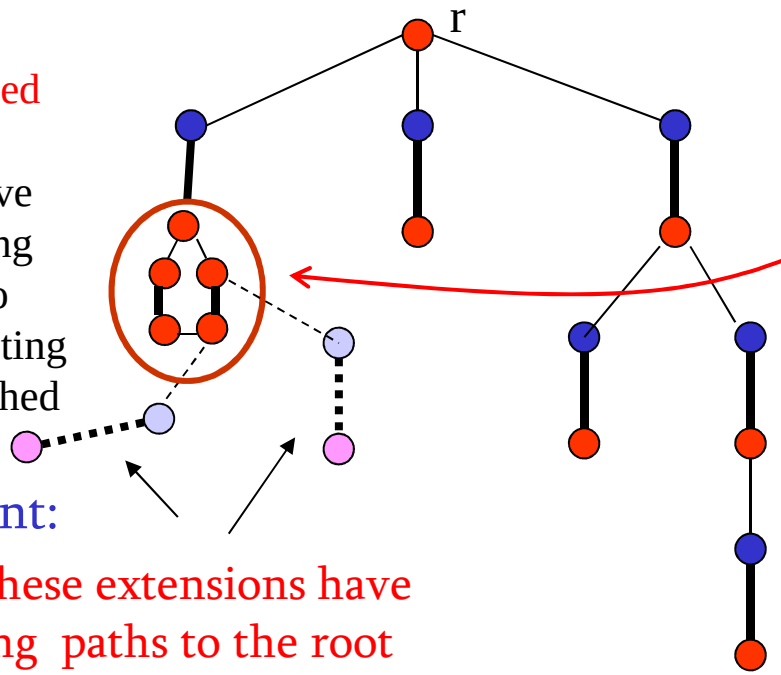


Edge to a blue node already in T.
Do nothing.

New edge type, not found in bipartite graphs. Odd loop is collapsed

Edge to uncolored and unmatched node:
We have then found an augumenting path, and we can use it to get a larger matching.

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

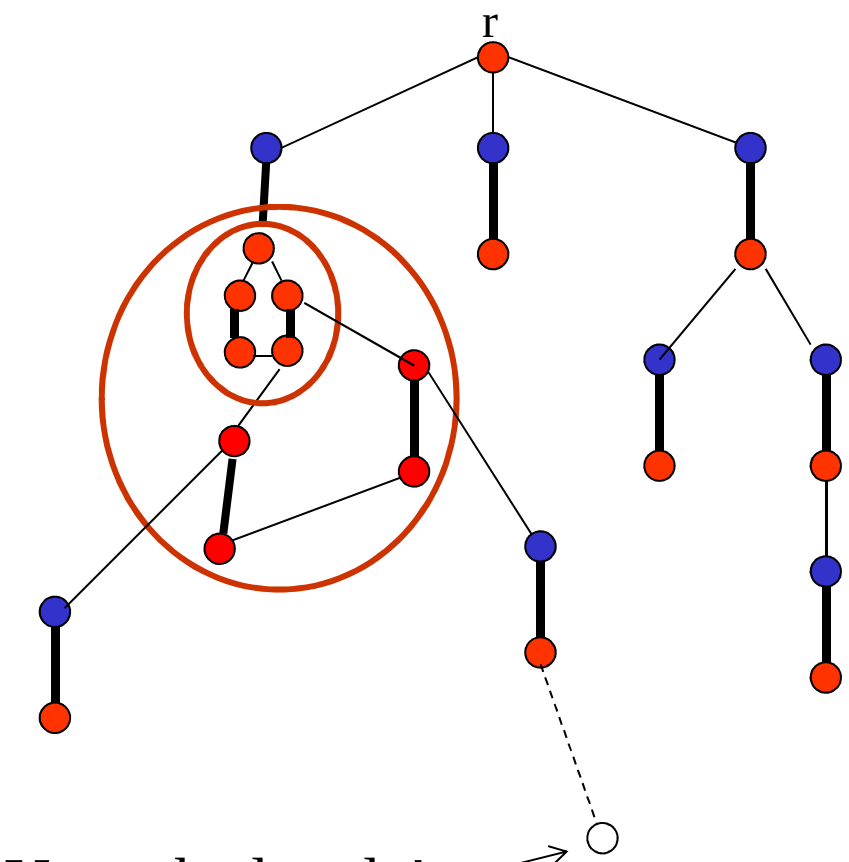


Red collapsed nodes:
They all have an alternating path back to the root, stating with a matched node

Important:

Both of these extensions have alternating paths to the root

The end of the treebuilding step in the extended Hungarian algorthms



If you find an augementing path:

We then go backwards along the alternating path, and along the way we unpack the collapsed nodes, and find the alternating path througt them.

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

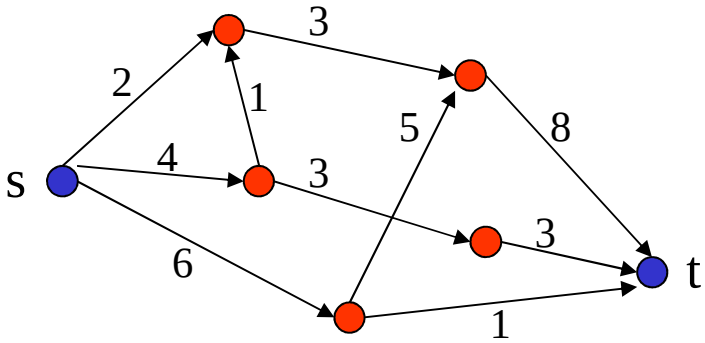
We can use this to find a matching that is one edge larger than the one we have.

Otherwise the treebuilding stops because there are no more unmatched nodes, and no edge from a red node to a node that is uncolored and unmatched.

- Then no larger matching exist
- But this is quite complex to prove, and it is not part of the curriculum

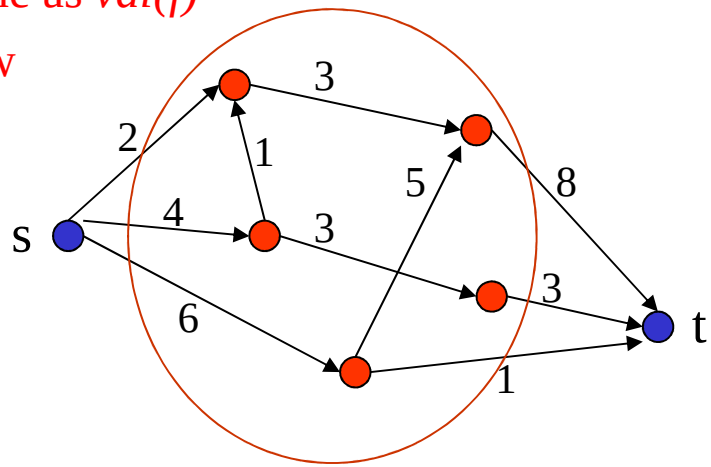
Flow in Networks, Ch. 14.2

- This stuff is, to some extent, also covered in the Weiss Book, 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 each edge.
- 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 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 on each of the edges
  - One *source* node *s* og one *sink* node *t*
  - And the goal is usually to find a largest possible flow from s to t



Flow in networks, Ch. 14.2

- A flow *f* in such a network is composed of a flow $f(e) \geq 0$ for each edge *e*, with the follow properties:
  - **Flow conservation:** For each node, except for *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 between 0 and $c(e)$.
- 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 into and out of all nodes.

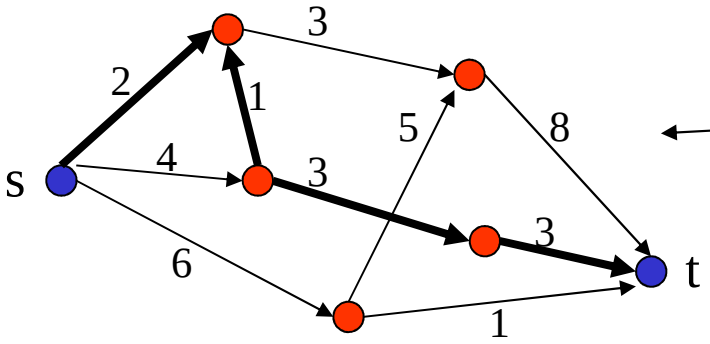


Concepts used in the book, that we don't use here.

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 constant, we get a new legal flow

No capacities. The numbers are edge flows.

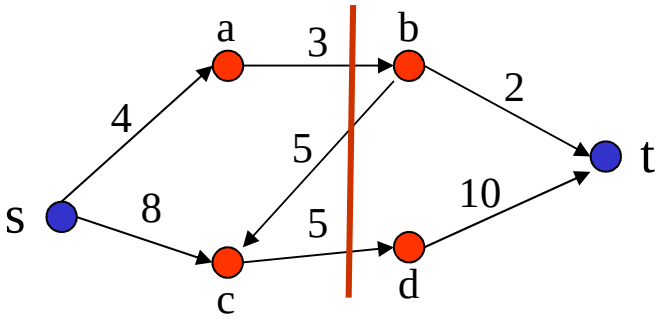


A (legal) *flow* and a *semipath*.

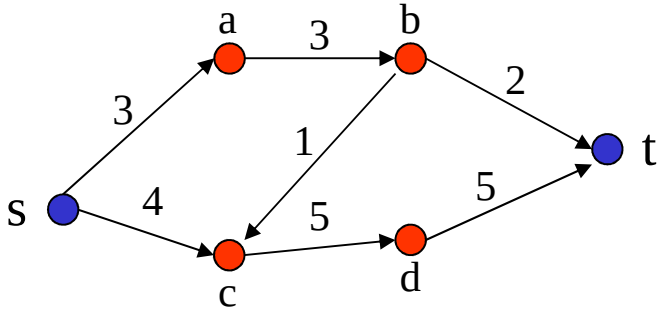
Flow in networks, with capacities

- 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
    - Satesfy 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 see: Maksimum flow is 7, and such a flow is given to the right.

A cut with capacity 8.
More about that later.

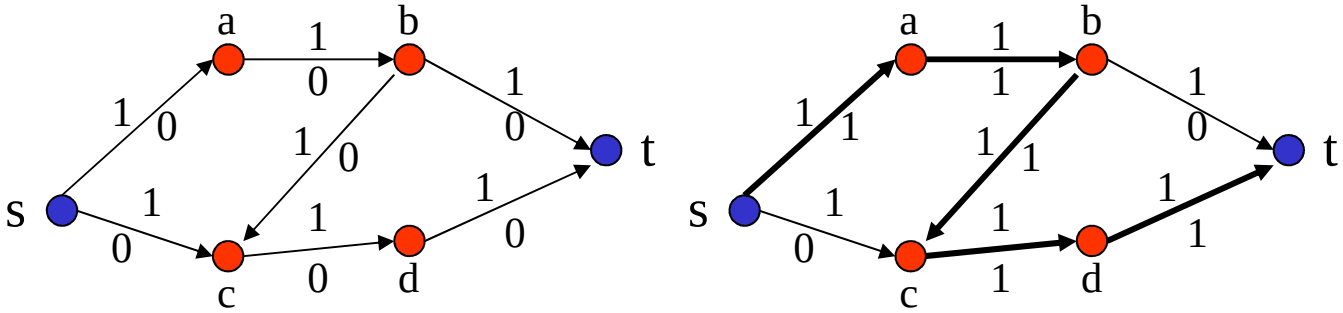


A maximum flow, of 7



The naive greedy algorithm is again not working

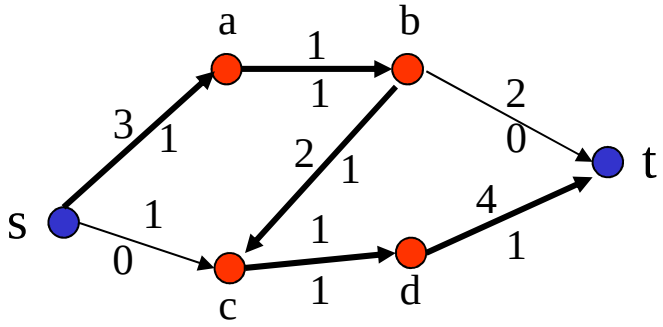
- The naive greedy algorithm (that in fact don't work!) would be as follows:
  - The step:
    - Find a directed, simple path from s to t where all the current *f(e)* are positive and *smaller* than *c(e)*
    - Increase the flow along this path as much as possible (dictated by the edge that has the 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 flowincreasing 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 flowincreasing path in the right network that can bring us to a flow with value 2. Thus this simple scheme won't bring us to a maximum flow.



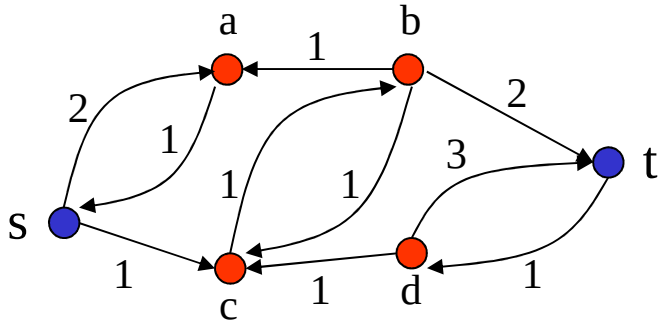
The *f*-derived network *N(f)*

- 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 set up the *f*-derived network referred to as N_f, *Nf*, or *N(f)*. We will here use *N(f)*.

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



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

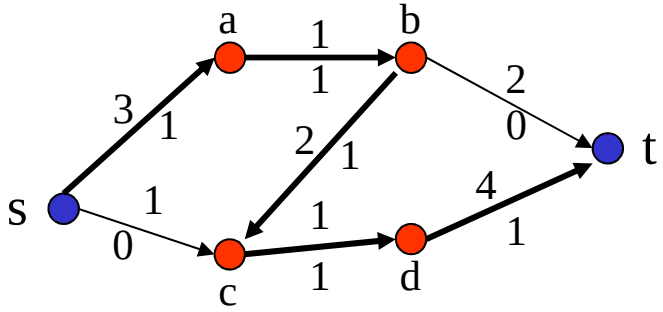


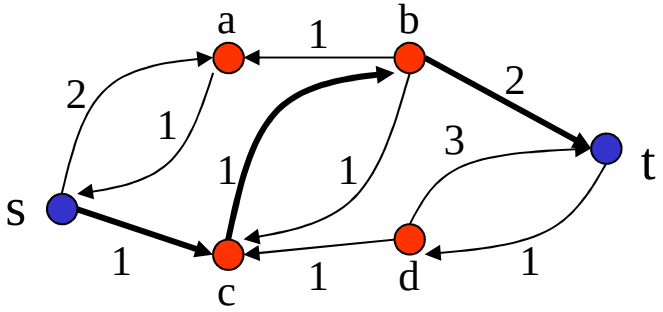
The *f*-derived network, of the situation to the left

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

f-forbedringsveier

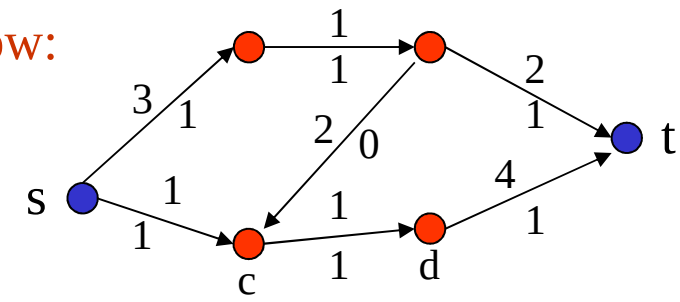
(Same network, capacities, and *N(f)* as on the previous slide:)





- We the search for paths from s to t in the *f*-derived network *N(f)*
  - Such paths are called *f*-augumenting paths
  - The search can be done e.g. bredth-first or depth-first in *N(f)* from s.
  - We can e.g choose the path *F = s-c-b-t*. The maximal flow-increase along this path is 1 (assume, in general h).
- We then perform the corresponding flow change, by:
  - Increasing the flow with h for the edges where their direction is the same as in *F*
  - Reduce the flow with h where edge direction is opposite to that of F

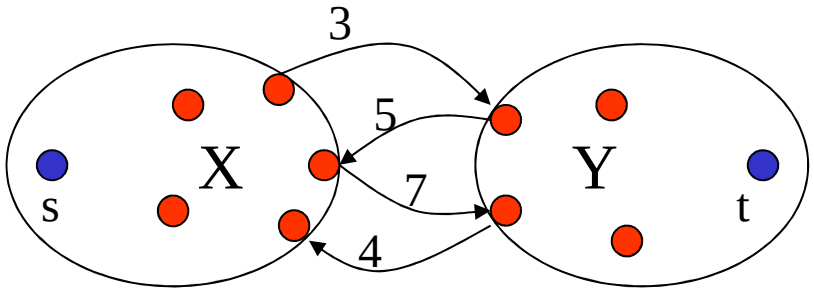
This gives the new flow:



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

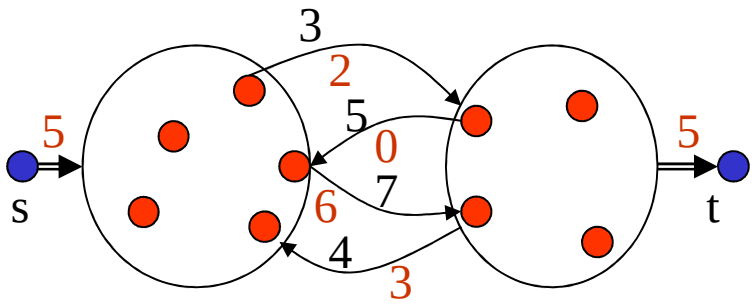
Cuts in networks

- A *cut* in a network is simply a devision of the set of nodes into two sets X and Y, where *s* is in X and *t* is in Y.



- 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.
- In the figure above, the capacity of the cut is 3 + 7 = 10
- Thus, the capacity of the edges 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)* ≤ *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)

  – This means (as (flow out of s) = *val(f)*):

  val(f) = (flow forward over K) – (flow backwards over K)

  – The right hand side of the above equality is called the *flow over K*, and (as all flows are positve) we no that it will not exceed *cap(K)*

  – Thus, we know: *val(f)* ≤ *cap(K)*.

  – In the figure above: *5 = 2 + 6 – 0 – 3 ≤ 3 + 7*

- 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)*

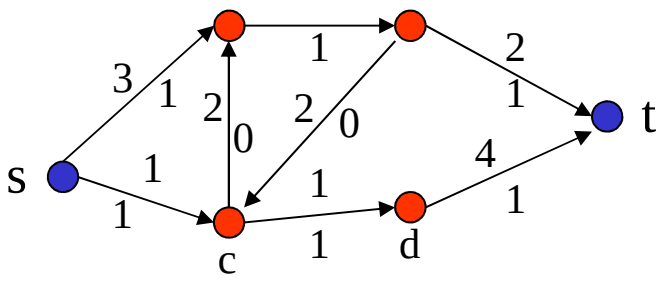
  then we have a maximum flow, and there is no *cut with smaller capacity*!

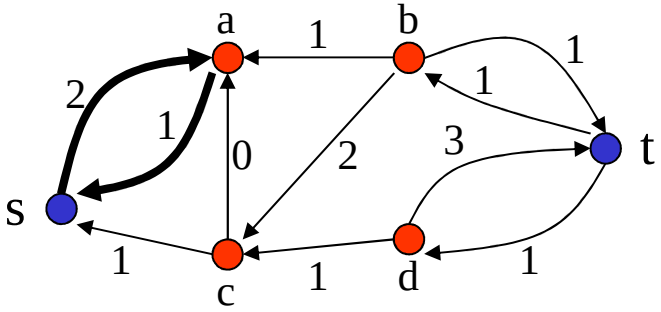
The Ford-Fulkerson algorithm

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*-augumenting 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*-augumenting path indicate

- Repeat this step until we can no longer find an *f*-augumenting 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 show a cut with capasity equal to the current flow. Thus, there can be no larger flow!



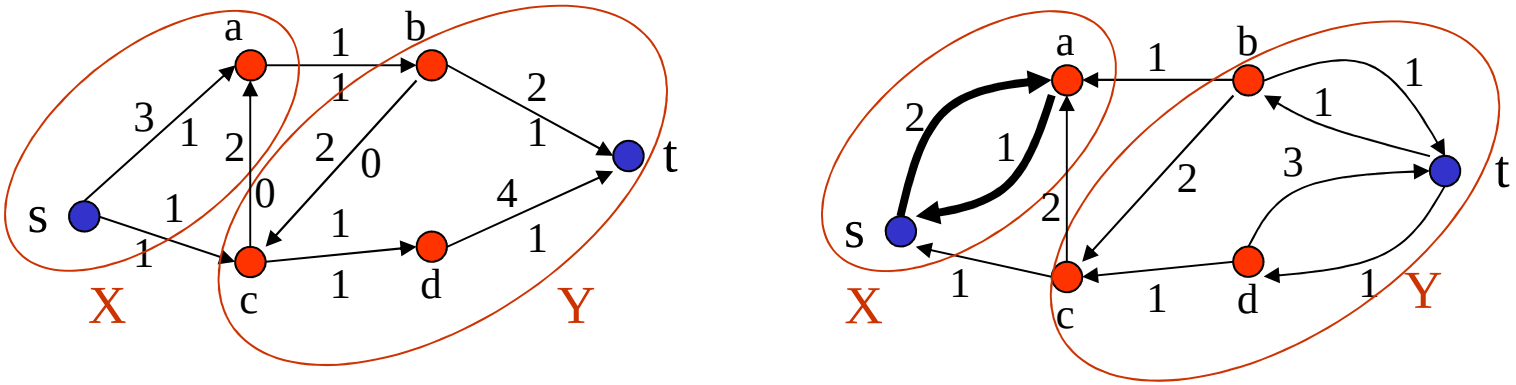


You should also look at the program at page 438

We will next Thursday look closer at this algorithm exemplified by figure 14.9

Termiation of the Ford-Fulkerson algorithm

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



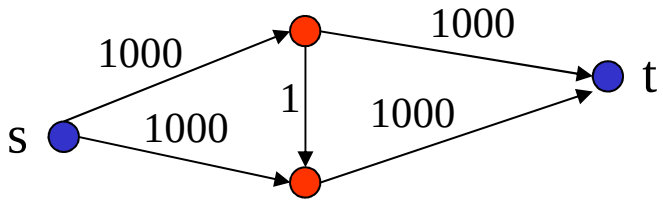
- As indicated: To show that we now have a maximum flow, we will show that we can construct a cut *K* with capacity equal to the current flow. That is: *cap(K)=val(f).*
- It turns out that such a cut is easy to find: Let X be the set of nodes reachable from *s* in *N(f)*, and let Y be the rest of the 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
- From the definition of *cap(K)* we see that the current flow over K 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 lewer capacity.

Variations of the Ford-Fulkerson algorithm

- The **Ford-Fulkerson algorithm** says nothing about which *f*-augumenting path should be chosen in each step, if there are more than one.

- If we do not decide anything about the choice of *f*-augumenting paths, we know:
  - If all capacities are (positive) intergers, then the number of steps can be at least at large as the size of the largest capacity. Example:



n = number of nodes
m = number of edges

  - If the capacities are real numbers, the algorithm can in theory loop forever.

- **Proposal 1:** All the time, choose the *f*-augumenting path that gives that largest possible incremet in the flow. This one can be found by an algorithm similar to a shortest path algorithm)
  - This gives a worst-case-time: $O(m \log(n) \log(\text{max-flow}))$

- **Proposal 2:** (Edmonds og Karp) All the time, choose the *f*-augumenting path that has the smallest number of edges (can be found by a bredth-first search)
  - This gives a worst-case-time: $O(n m^2)$

  (and this independant of the max. flow, which is very convenient)

Varianter av problemet med maksimal flyt

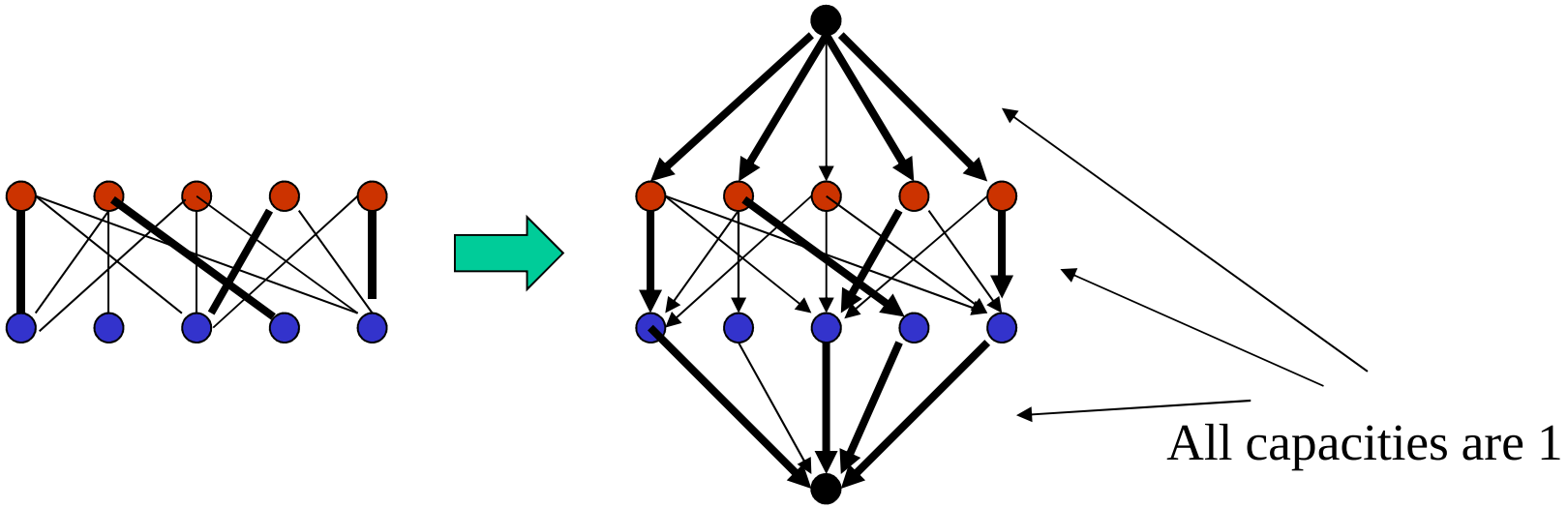
- First of all, there are alternatives to the Ford-Fulkerson algorithm
  - *Dinac* has designed an algorithm
  - *Goldberg and Tarjan* (preflow push algorithm)
- We may also have a minimal flow for each edge
  - Then it is an interesting problem just to find a *possible* flow
  - But after that you can proceed as Ford-Fulkerson
- We may also have a price on each edge, saying how much a flow of 1 costs over this edge.
  - For this problem there is a well known algorithm: The Out-of-kilter algorithm.
- 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 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 algorihtm:

1. If we have interger capacities, Ford-Fulkerson will always find an interger max. flow
2. 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 next Thursday look at:

– That searching for an M-augumenting path in the bipartite graph to the left, corresponds to searchong for an *f*-augumenting path to the right.