CSC373: Algorithm Design, Analysis and Complexity
Fall 2017

Allan Borodin

October 4, 2017

Week 4 : Annoucements

Assignment 1 is due Thursday October 5 at 2PM.
Link to Markus: https://markus.teach.cs.toronto.edu/csc373-2017-09
Questions on assignment will be taken after announcements.
Term test 1 will be held in the tutorial rooms October 12, 5-6. This is a common exam with the daytime section.
The term test will be closely based on some questions occuring in Assignment 1. More specifically, there will be three questions: one divide and conquer, one greedy and one dynamic programming
Aids allowed: one 8 by 11 sheet of paper, two sides, hand written.

Note about Piazza: I think it is very good when students answer questions posed on Piazza. Almost always these are good responses. **However**, while it is good to clarify questions and point out possible ambiguities, I do not want solutions posted on Piazza. Each student is responsible for their own work and if a solution is provided on Piazza, we may not be able to grade that question.

Annoucement on next slide about UNICEF fund raiser

Unicef Fund Raiser

Dear Professor,

I am writing to you on behalf of the University of Toronto UNICEF group to request your assistance for our upcoming yearly event "Prof-in-a-Prop," which is a part of our National UNICEF Day campaign.

This year, "Prof-in-a-Prop" will run from October 1st to October 31st. All of the proceeds from our Prof-in-a-Prop campaign will go directly to UNICEF Canada and thus directly to children in need in the developing world.

The main purpose of the event is to raise funds for UNICEF and to create awareness for UNICEF Canada's campaigns worldwide. We are requesting the participation of numerous professors on campus to help us make this event a success.

The "Prof-in-a-Prop" event sends U of T UNICEF volunteers to stop by lecture halls to collect donations from students for UNICEF's campaigns. If the class donates enough money, the professor will wear a Halloween costume – such as bunny ears, a banana or a hotdog suit – for an entire lecture. This year, we have set our target goal at $1 per student in each participating class.

In the past, we have found that students are very receptive to this idea, and the event has consistently been our major fundraiser for the year. It also allows us to engage large numbers of students on campus, giving us the opportunity to educate them about various global issues.

As National UNICEF Day is October 31st, we always appreciate the opportunity to spread the word about UNICEF's work in over 190 countries worldwide and how students can contribute to the cause.

Therefore, we at U of T UNICEF ask if you would be interested in participating in our "Prof-in-a-Prop" campaign. If you would be willing to take part in our event, please contact me at the email address or phone number listed below.

If you have any questions or concerns, I would be more than happy to speak with you at length.

Thank you for your time and consideration.

Sincerely,

Tea Cimini
Co-President
University of Toronto UNICEF
tea.cimini@mail.utoronto.ca

Todays agenda

- I am happy to answer any questions about the assignment. Tomorrow in tutorials, you can ask details about solutions. **Note:** When we pose a question such as "Does algorithm A correctly solve the given problem", your answer should either be an example showing that the algorithm is not correct, or a reasonably convincing proof that the algorithm is correct.
- Finish the slides from the last lecture on edit distance and some concluding remarks on dynamic programming.
- Begin Flow networks
- Ford-Fulkerson algorithmic scheme and augmenting paths
- max-flow-min-cut theorem
- Efficient implementations of the Ford-Fulkerson scheme

The sequence alignment (edit distance) problem

The edit distance problem

- Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ over some finite alphabet S.
- **Goal:** find the best way to "match" these two strings.

- Variants of this problem occur often in bio-informatics as well as in spell checking.
- Sometimes this is cast as a maximization problem.
- We will view it as a minimization problem by defining different distance measures and matching symbols so as to minimize this distance.

A simple distance measure

- Suppose we can delete symbols and match symbols.
- We can have a cost $d(a)$ to delete a symbol a in S, and a cost $m(a, b)$ to match symbol a with symbol b (where we would normally assume $m(a, a) = 0$).
- As in any DP we consider an optimal solution and let's consider whether or not we will match the rightmost symbols of X and Y or delete a symbol.

The DP arrays

- The semantic array:

$$E[i,j] = \text{the cost of an optimal match of } x_1 \dots x_i \text{ and } y_1 \dots y_j.$$

- The computational array:

$$E'[i,j] = \begin{cases} 0 & \text{if } i = j = 0 \\ d(y_j) + E'[i, j-1] & \text{if } i = 0 \text{ and } j > 0 \\ d(x_i) + E'[i-1, j] & \text{if } i > 0 \text{ and } j = 0 \\ \min\{A, B, C\} & \text{otherwise} \end{cases}$$

where $A = m(x_i, y_j) + E'[i-1, j-1]$, $B = d(x_i) + E'[i-1, j]$, and $C = d(y_j) + E'[i, j-1]$.

- As a simple variation of edit distance we consider the maximization problem where each "match" of "compatible" a and b has profit 1 (resp. $v(a, b)$) and all deletions and mismatches have 0 profit.
- This is a special case of unweighted (resp. weighted) bipartite graph matching where edges cannot cross.

The traveling salesman problem (TSP)

We recall from last week that computing the maximum cost (i.e. profit) of a simple path is a generalization of the Hamiltonian path problem which is a variant of the traveling salesman problem.

Traveling salesman problem (TSP)

Given a graph $G = (V, E)$ with a cost function $c : E \to \mathbb{R}_{\geq 0}$ determine if the cost of a simple cycle containing all the nodes (i.e. cycle length is $n = |V|$) assuming the graph has such a Hamiltonian cycle.

Without loss of generality, we can assume a complete graph (using $c(e) = \infty$ for any missing edges).

It is is roughly equivalent to consider the least cost Hamiltonian path problem. Namely, finding a least cost *simple* path of length *exactly* (and NOT at most) length $n - 1$ from some given starting node u. For the same reason as in the maximum cost path discussion, the least cost Hamiltonian path problem cannot be obtained by modifying the least cost path DP. Namely, we cannot dismiss the possibility of cycles in the path.

Not all exponentials are equal; using DP to obtain a better exponential time algorithm

A naive way to compute the least cost Hamiltonian path problem (with some given initial node u) is to consider all $(n-1)!$ simple paths of length $n-1$. As we noted last week, $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$. We also mentioned that by using an appropriate DP, we can reduce that time complexity to $O(n^2 2^n)$ which is still of course exponential but grows much slower than the factorial function $(n!)$.

Here is the idea as expressed in the following semantic array: For each subset $S \subseteq V$ with $u \in S, v \notin S$
$C[S, v]$ is the least cost simple path from u to v containing each node in S exactly once.

Computing the entries of $C[S, v]$

I am now going to just use C and not C' since by now I hope the distinction between "what we want to compute" and "how we are going to compute it" is hopefully clear. We compute $C[S, v]$ "inductively as follows:

If $|S| = 1$, then $C[S, v] = c(u, v)$ % S must be $\{u\}$
Else if $|S| > 1$, then $C[S, v] = \min_{x \notin S} C[S, x] + c(x, v)$

We note that the least cost Hamiltonian path problem is "NP-hard to approximate" to any constant whereas there are efficient approximations if the cost function satisfies the triangle inequality.

DP concluding remarks

- In DP algorithms one usually has to first generalize the problem (as we did more or less to some extent for all problems considered). Sometimes this generalization is not at all obvious.

- What is the difference between divide and conquer and DP?

  - In divide and conquer the recursion tree never encounters a subproblem more than once.
  - In DP, we need memoization (or an iterative implementation) as a given subproblem can be encountered many times leading to exponential time complexity if done without memoization.
  - See also the comment on page 169 of DPV as to why in some cases memoization pays off since we do not necessaily have to compute every possible subproblem. (Recall also the comment by Dai Tri Man Le.)

Flow networks

- I will be following CLRS, second edition for the basic definitions and results concerning the computation of max flows.
- We will depart from the usual convention and allow negative flows. While intuitively this may not seem so natural, it does simplify the development.
- The DPV, KT and CLRS (third edition) texts use the more standard convention of just having non-negative flows.

Definition

A flow network (more suggestive to say a capacity network) is a tuple $F = (G, s, t, c)$ where

- $G = (V, E)$ is a "bidirectional graph"
- the source s and the terminal t are nodes in V
- the capacity $c : E \to \mathbb{R}^{\geq 0}$

What is a flow?

A flow is a function $f : E \to \mathbb{R}$ satisfying the following properties:

1. Capacity constraint: for all $(u, v) \in E$,

$$f(u, v) \leq c(u, v)$$

2. Skew symmetry: for all $(u, v) \in E$,

$$f(u, v) = -f(v, u)$$

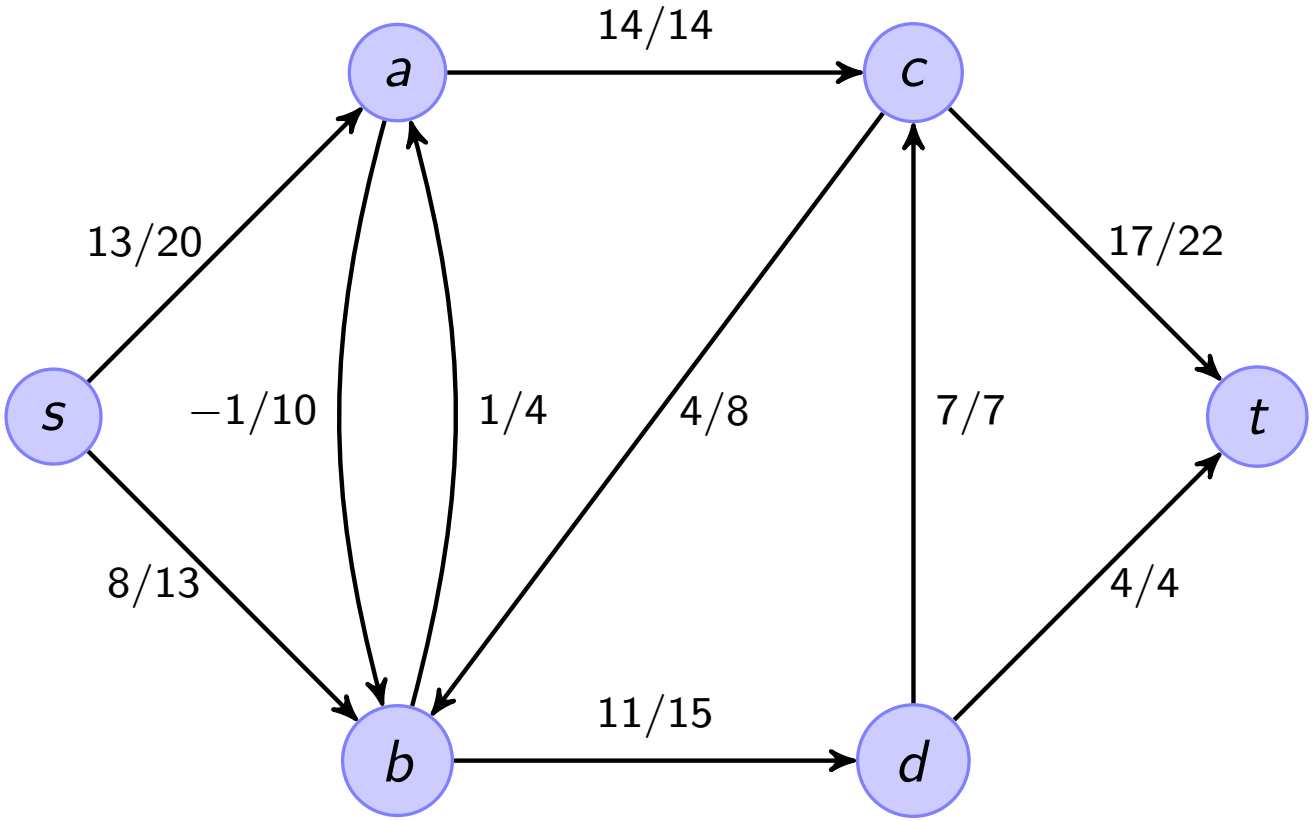
3. Flow conservation: for all nodes u (except for s and t),

$$\sum_{v \in N(u)} f(u, v) = 0$$

Note

Condition 3 is the "flow in = flow out" constraint if we were using the convention of only having non-negative flows in one direction. .

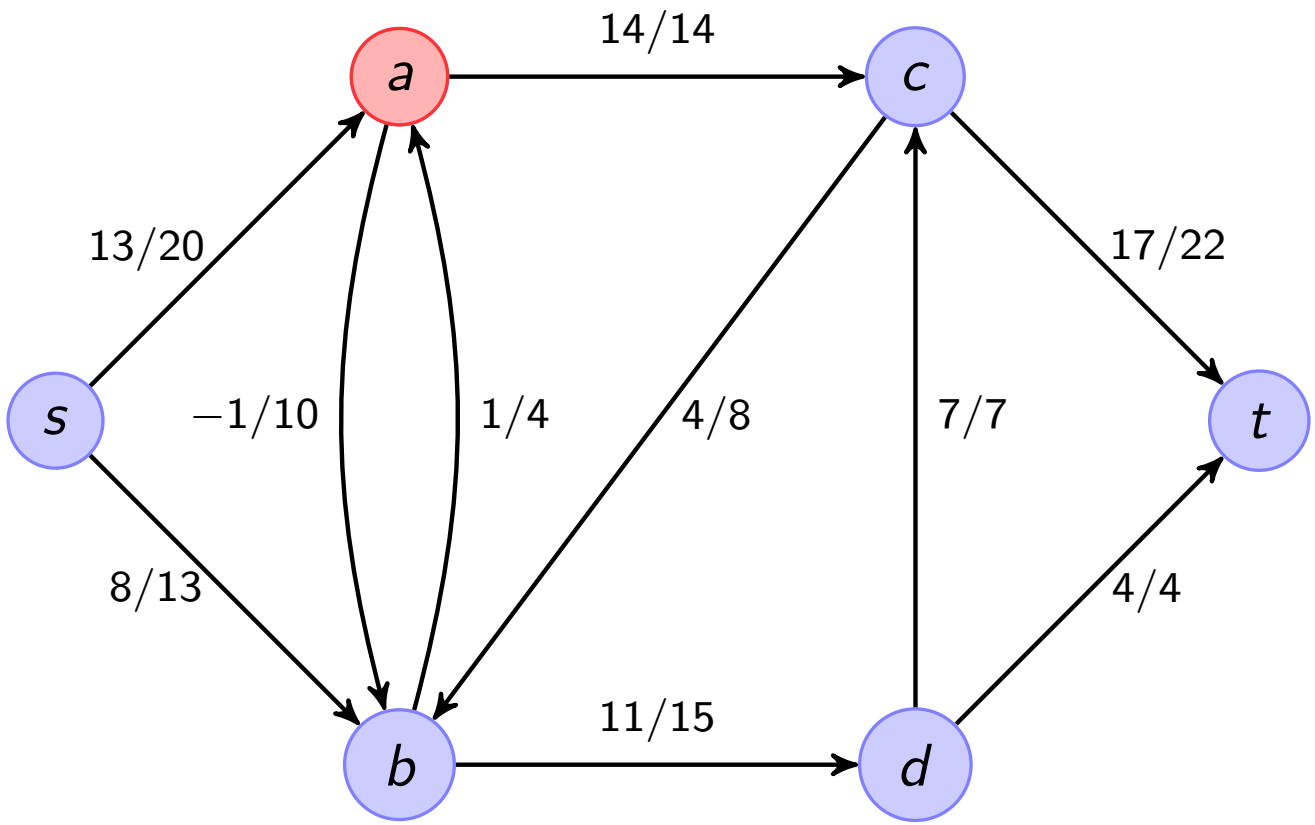
An example



The notation x/y on an edge (u, v) means

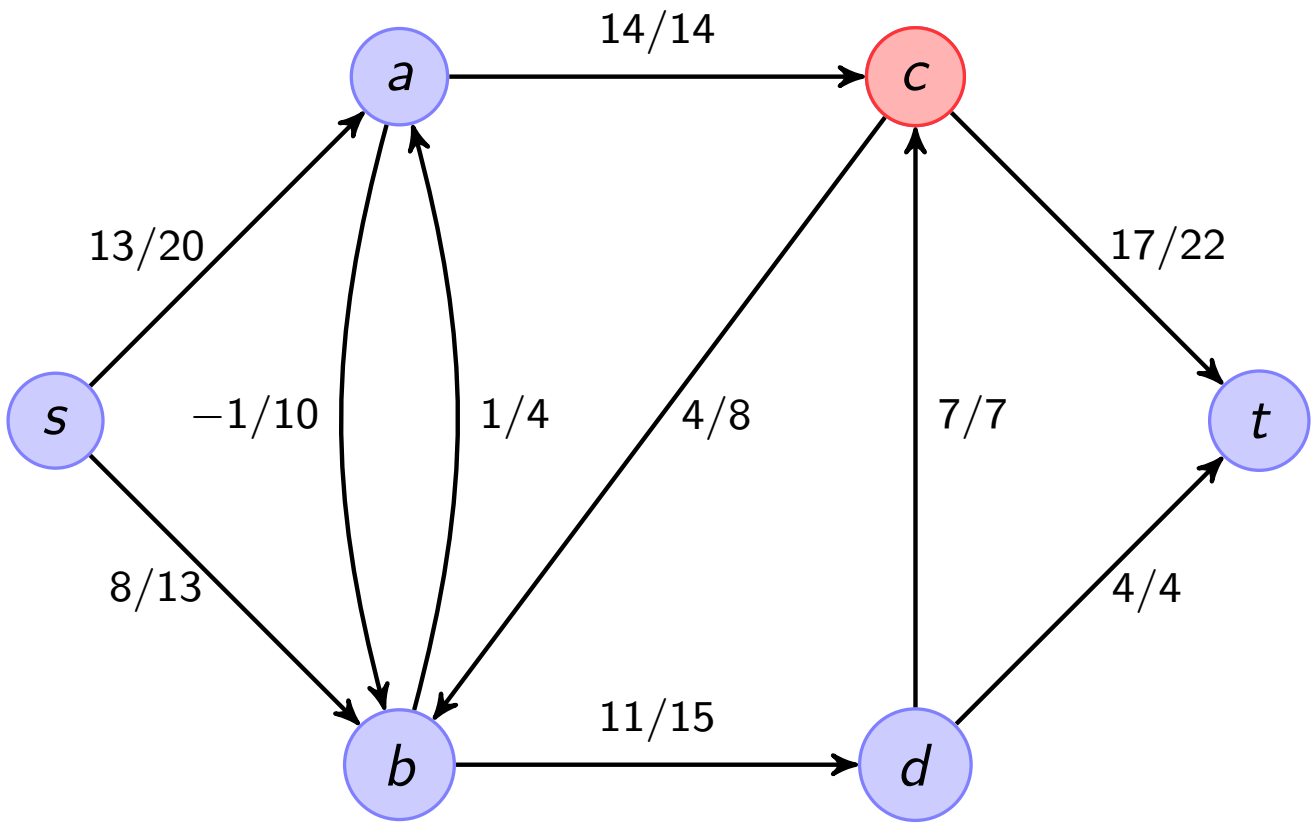
- x is the flow, i.e. $x = f(u, v)$
- y is the capacity, i.e. $y = c(u, v)$

An example of flow conservation



- For node a: $f(a,s) + f(a,b) + f(a,c) = -13 + (-1) + 14 = 0$

An example of flow conservation



- For node a: $f(a,s) + f(a,b) + f(a,c) = -13 + (-1) + 14 = 0$
- For node c:

$$f(c,a) + f(c,b) + f(c,d) + f(c,t) = -14 + 4 + (-7) + 17 = 0$$

The max flow problem

The max flow problem

Given a flow network, the goal is to find a valid flow that maximizes the flow out of the source node s.

- As we will see this is also equivalent to maximizing the flow into the terminal node t. (This should not be surprising as flow conservation dictates that no flow is being stored in the other nodes.)
- We let $val(f)$ denote the flow out of the source s for a given flow f.
- We will study the Ford-Fulkerson augmenting path scheme for computing an optimal flow.
- I am calling it a "scheme" as there are many ways to instantiate this scheme although I dont view it as a general "paradigm" in the way I view (say) greedy and DP algorithms.

So why study Ford-Fulkerson?

- Why do we study the Ford-Fulkerson scheme if it is not a very generic algorithmic approach?
- As in DPV text, max flow problem can also be represented as a linear program (LP) and all LPs can be solved in polynomial time.
- I view Ford-Fulkerson and augmenting paths as an important example of a local search algorithm although unlike most local search algorithms we obtain an optimal solution.
- The topic of max flow (and various generalizations) is important because of its immediate application and many applications of max flow type problems to other problems (e.g. max bipartite matching).
  - That is many problems can be polynomial time transformed/reduced to max flow (or one of its generalizations).
  - One might refer to all these applications as "flow based methods".

A flow f and its residual graph

- Given any flow f for a flow network $F = (G, s, t, c)$, we define the residual graph $G_f = (V, E_f)$, where
  - V is the set of vertices of the original flow network F
  - E_f is the set of all edges e having positive residual capacity

$$c_f(e) = c(e) - f(e) > 0.$$

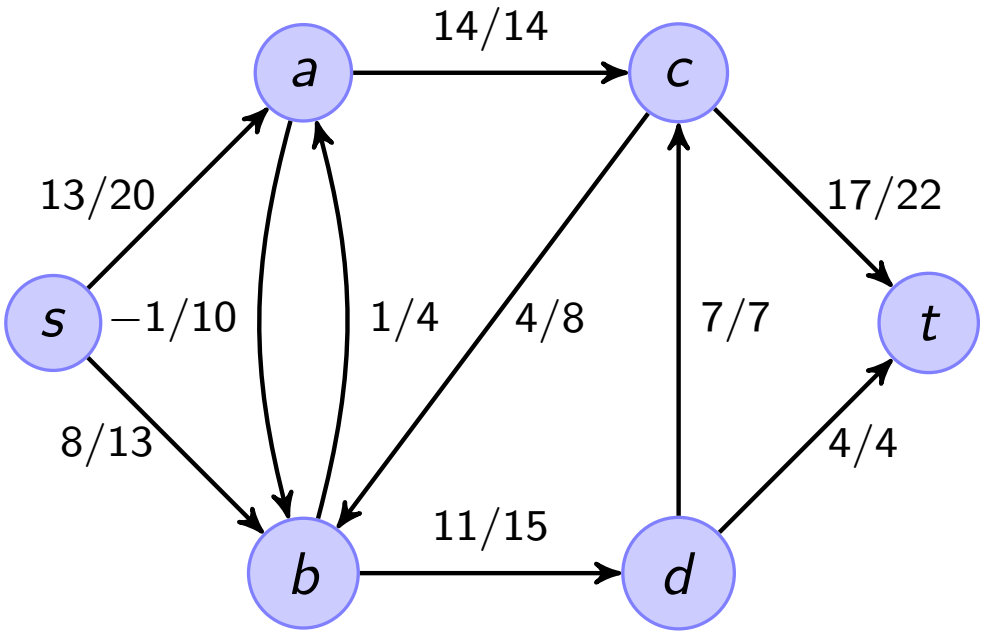
- Note that $c(e) - f(e) \geq 0$ for all edges by the capacity constraint.

Note

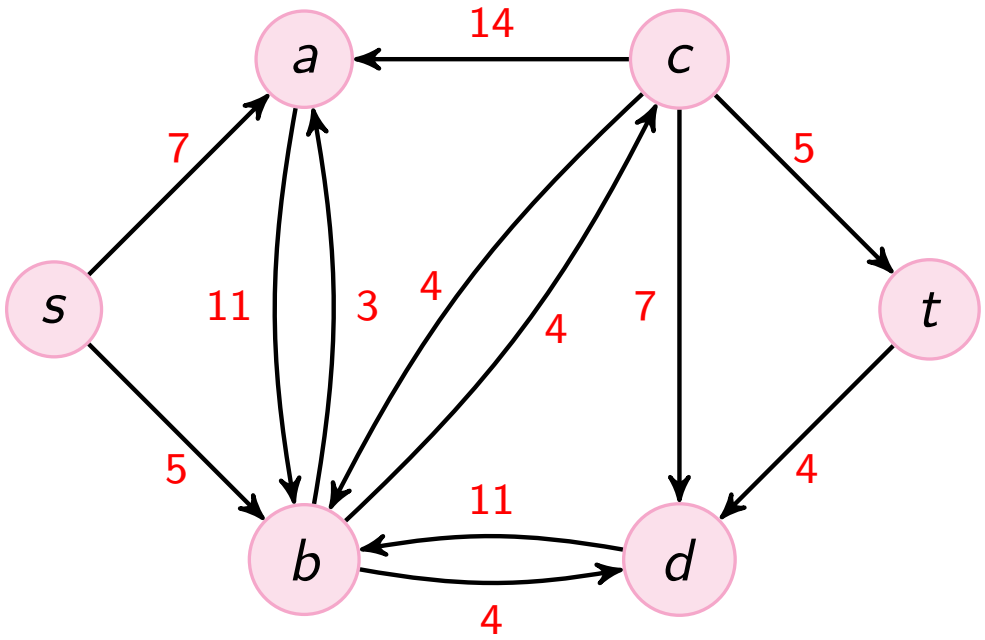
With our convention of negative flows, even a zero capacity edge (in G) can have residual capacity.

- The basic concept underlying the Ford-Fulkerson algorithm is an **augmenting path** which is an s-t path in G_f.
  - Such a path can be used to augment the current flow f to derive a better flow f'.

An example of a residual graph



The previous network flow



The residual graph

The residual capacity of an augmenting path

- Given an augmenting path π in G_f, we define its residual capacity $c_f(\pi)$ to be the

$$\min_e \{c_f(e) \mid e \in \pi\}$$

- **Note:** the residual capacity of an augmenting path is itself is greater than 0 since every edge in the path has positive residual capacity.

- **Question:** How would we compute an augmenting path of maximum residual capacity?

Using an augmenting path to improve the flow

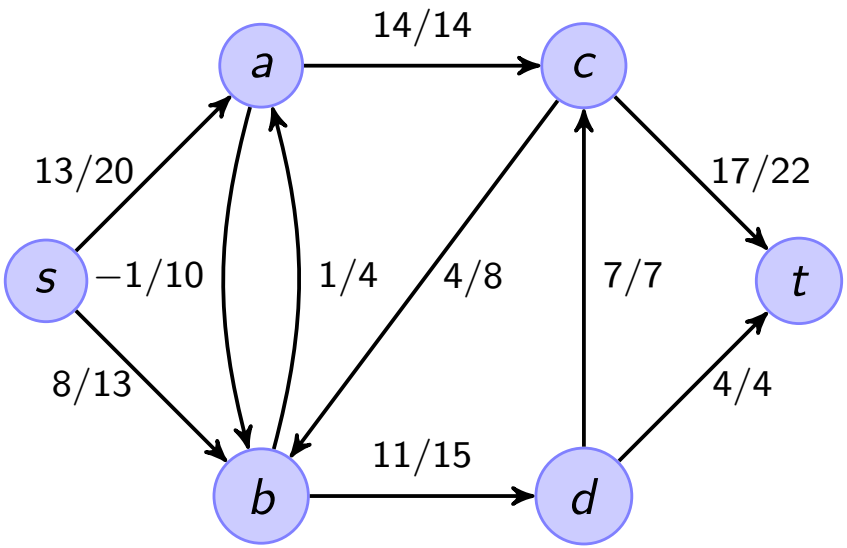
- We can think of an augmenting path as defining a flow f_π (in the "residual network"):

$$f_\pi(u,v) = \begin{cases} c_f(\pi) & \text{if } (u,v) \in \pi \\ -c_f(\pi) & \text{if } (v,u) \in \pi \\ 0 & \text{otherwise} \end{cases}$$

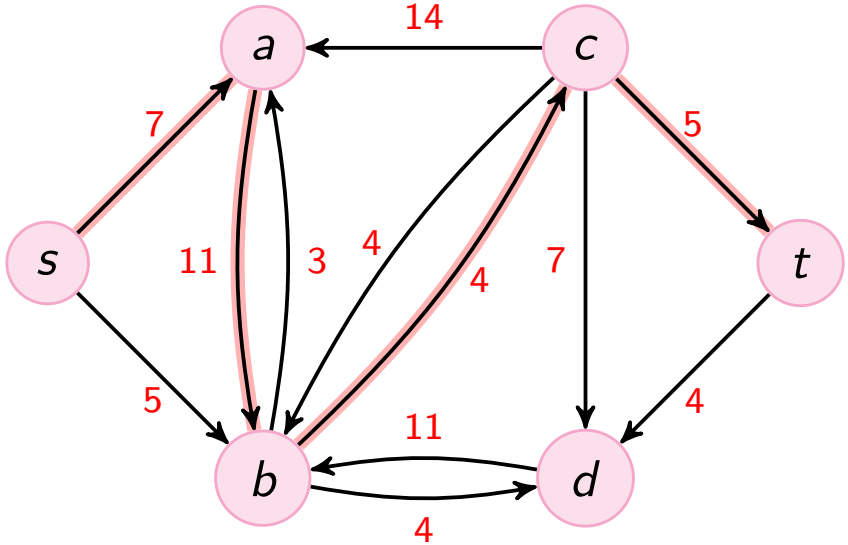
Claim

$f' = f + f_\pi$ is a flow in F and $val(f') > val(f)$

Deriving a better flow using an augmenting path

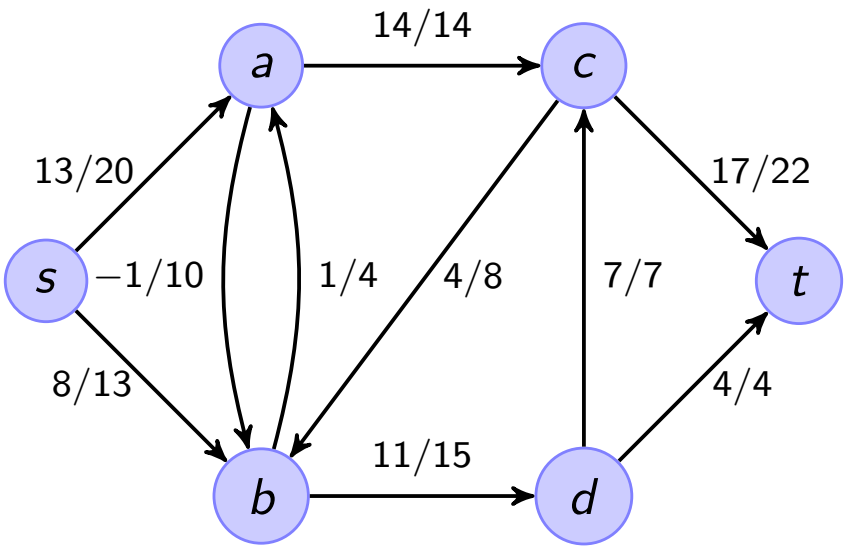


The original network flow

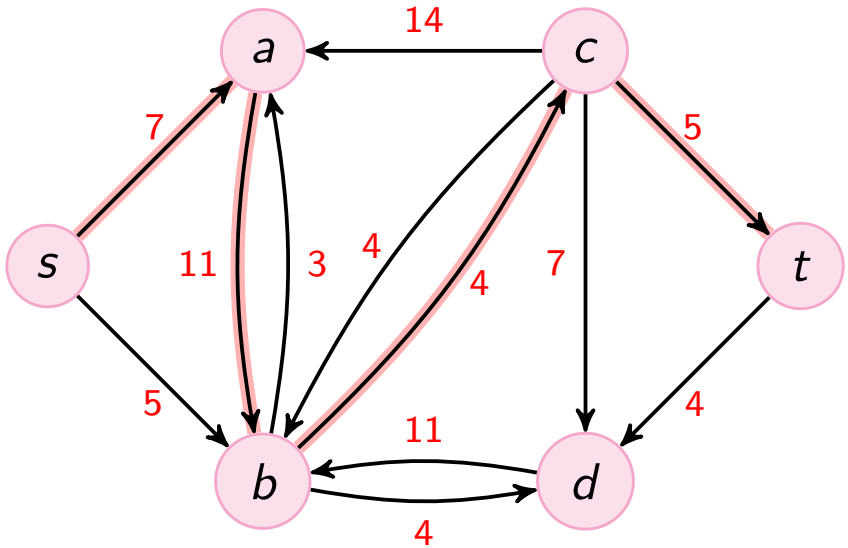


An augmenting path π with $c_f(\pi) = 4$

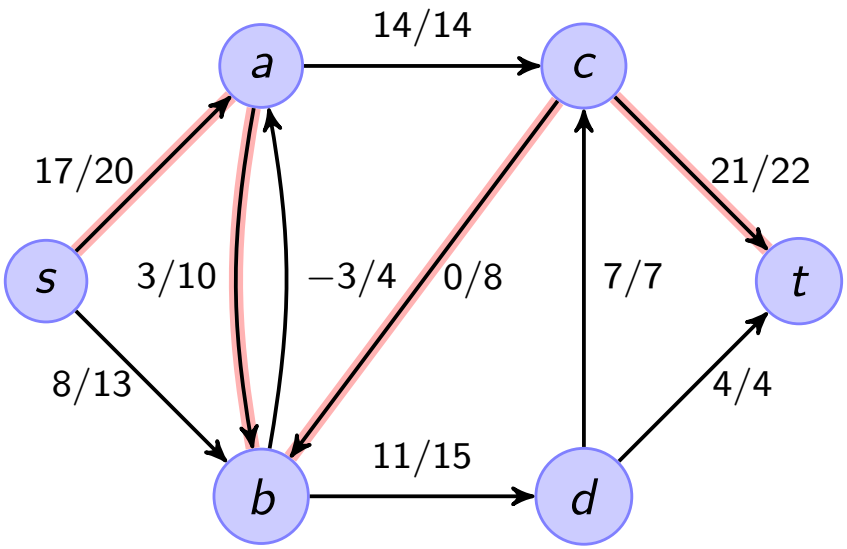
Deriving a better flow using an augmenting path



The original network flow

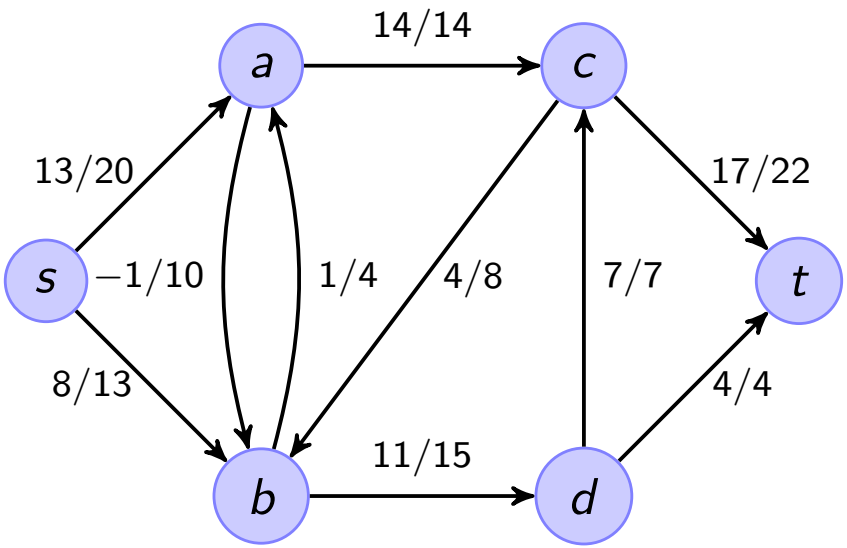


An augmenting path π with $c_f(\pi) = 4$

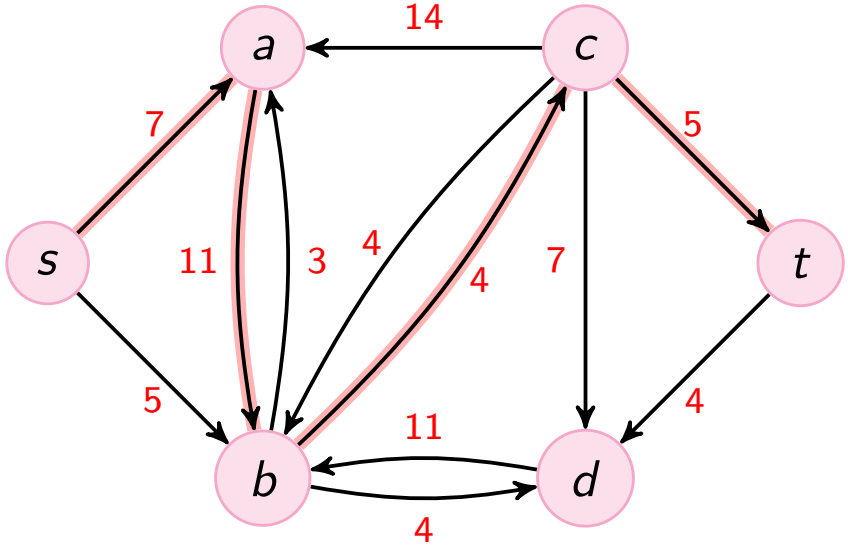


The updated flow whose value = 25

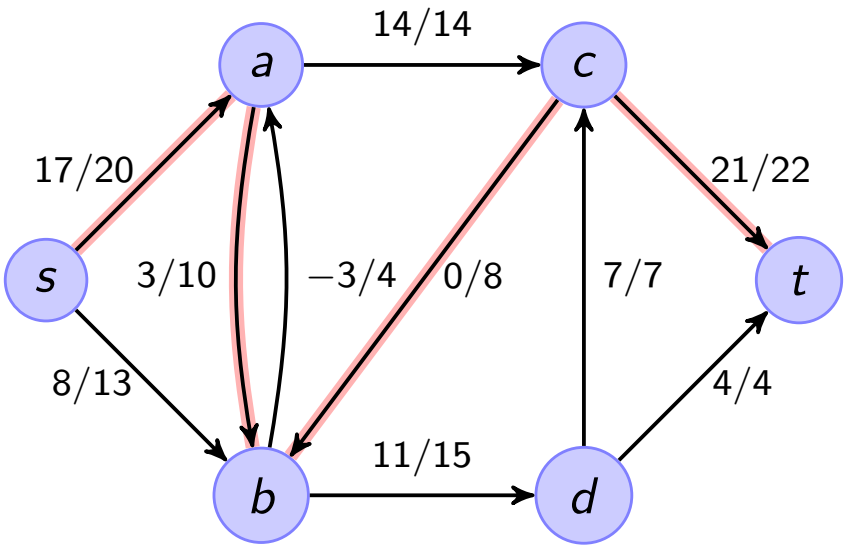
Deriving a better flow using an augmenting path



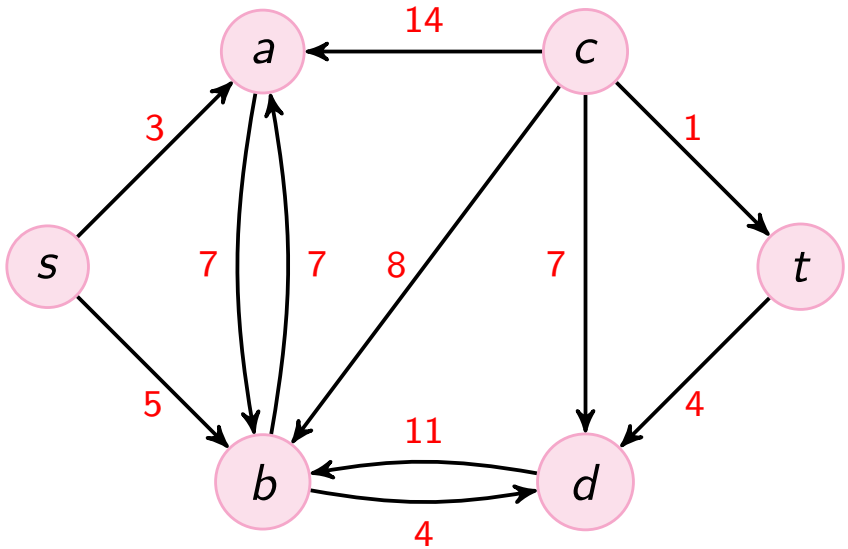
The original network flow



An augmenting path π with $c_f(\pi) = 4$



The updated flow whose value = 25



Updated res. graph with no aug. path

The Ford-Fulkerson scheme

The Ford-Fulkerson scheme

/* *Initialize* */
$f := 0$; $G_f := G$
WHILE there is an augmenting path π in G_f
  $f := f + f_\pi$
    /* *Note this also changes* G_f */
ENDWHILE

Note

I call this a "scheme" rather than an algorithm since we haven't said how one chooses an augmenting path (as there can be many such paths)

Ford Fulkerson as a local search

- Local search is one of the most popular approaches for solving search and optimization problems.
- Local search is often considerd to be a "heuristic" since local search algorithms are often not analyzed but seem to often produce good results.
- For both search (i.e finding any feasible solution) and optimization, local search algorithms define some local neighborhood of a (partial) solution S, which we will denote as $Nbhd(S)$

The local search meta-algorithm

The local search meta-algorithm

Initialize S
WHILE there is a "better" solution $S' \in Nbhd(S)$
    $S := S'$
ENDWHILE

- Here "better" can mean different things.
  - For a search problem, it can mean "closer" to being feasible.
  - For an optimization problem it usually means being an improved solution.
- There are many variations of local search and we will hopefully study local search later but for now we just wish to observe the sense in which Ford-Fulkerson can be seen as a local search algorithm.
  - We start with a trivial initial solution.
  - We define the local neighbourhood of a flow f to be all flows f defined by adding the flow of an augmenting path f_π to f.

Many issues concerning local search

- How do we define the local neighbourhood and how do we choose an $S' \in Nbhd(S)$?
- Can we guarantee that a local search algorithm will terminate? And if so, how fast will the algorithm terminate?
- Upon termination how good is the local optimum that results from a local search optimization?
- How can we escape from a local optimum (assuming it is not optimal)?

Local search issues for the Ford-Fulkerson scheme

- Does it matter how we choose an augmenting path for termination and speed of termination?

- That is, does it matter how we are choosing the $S' \in Nbhd(S)$?
  - **Answer:** YES, it matters but there are good ways to choose augmenting paths so that the algorithm is poly time.
  - Note that the $Nbhd(S)$ here can be of exponential size but that is not a problem as long as we can efficiently search for solutions in the local neighbourhood.

- Upon termination how good is the flow?
  - **Answer:** The flow is an optimal flow. This will be proved by the max flow - min cut theorem.
  - Note that this is unusual in that for most local search applications a local optimum is usually not a global optimum.

The max-flow min-cut theorem

- We will accept some basic facts and look at the proof of the max-flow min-cut theorem as presented in our old CSC 364 notes.
- Amongst the consequences of this theorem, we obtain that

If any implementation of the Ford Fulkerson scheme terminates, then the resulting flow is an optimal flow.

- A cut (really an s-t cut) in a flow network is a partition (S, T) of the nodes such that $s \in S$ and $t \in T$.
- We define the capacity $c(S, T)$ of a cut as

$$\sum_{u \in S \text{ and } v \in T} c(u, v)$$

- We define the flow $f(S, T)$ across a cut as

$$\sum_{u \in S \text{ and } v \in T} f(u, v)$$

Max-flow min-cut continued

Some easy facts

- One basic fact that intuitively should be clear is that

$$f(S, T) \leq c(S, T)$$

  for all cuts (S, T) (by the capacity constraint for each edge).

- And it should also be intuitively clear that $f(S, T) = val(f)$ for any cut (S, T) (by flow conservation at each node).

- Hence for any flow f, $val(f) \leq c(S, T)$ for every cut (S, T).

The max-flow min-cut theorem

The following three statements are equivalent:

1. f is a max-flow
2. There are no augmenting paths w.r.t. flow f (i.e. no s-t path in G_f)
3. There exists some cut (S, T) satisfying $val(f) = c(S, T)$
   - Hence this cut (S, T) must be a min (capacity) cut since $val(f) \leq c(S, T)$ for all cuts.

Note

The name follows from the fact that the value of a max-flow $=$ the capacity of a min-cut

The proof outline

1. $\Rightarrow$ 2. If there is an augmenting path (w.r.t. f), then f can be increased and hence not optimal.

The proof outline

1. $\Rightarrow$ 2. If there is an augmenting path (w.r.t. f), then f can be increased and hence not optimal.

2. $\Rightarrow$ 3. Consider the set S of all the nodes reachable from s in the residual graph G_f.
   - Note that t cannot be in S and hence $(S, T) = (S, V - S)$ is a cut.
   - We also have $c(S, T) = val(f)$ since $f(u, v) = c(u, v)$ for all edges (u, v) with $u \in S$ and $v \in T$.

The proof outline

1. $\Rightarrow$ 2. If there is an augmenting path (w.r.t. f), then f can be increased and hence not optimal.

2. $\Rightarrow$ 3. Consider the set S of all the nodes reachable from s in the residual graph G_f.
   - Note that t cannot be in S and hence $(S, T) = (S, V - S)$ is a cut.
   - We also have $c(S, T) = val(f)$ since $f(u, v) = c(u, v)$ for all edges (u, v) with $u \in S$ and $v \in T$.

3. $\Rightarrow$ 1. Let f' be an arbitrary flow. We know $val(f') \leq c(S, T)$ for any cut (S, T) and hence $val(f') \leq val(f)$ for the cut constructed in 2.

A consequence of the max-flow min-cut theorem

Corollary

If all capacities are integral (or rational), then any implementation of the Ford-Fulkerson algorithm will terminate with an optimal integral max flow.

Rational capacities

Why does the claim about integral capacities imply the same for rational capacities?

The runtime of Ford-Fulkerson

Observation

Each augmenting path has residual capacity at least one.

- The max-flow min-cut theorem along with the above observation ensures that with integral capacities, Ford-Fulkerson must always terminate and the number of iterations is at most:

$$C = \text{the sum of edge capacities leaving } s.$$

Notes

- There are bad ways to choose augmenting paths such that with irrational capacities, the Ford-Fulkerson algorithm will not terminate.
- However, even with integral capacities, there are bad ways to choose augmenting paths so that the Ford-Fulkerson algorithm does not terminate in polynomial time.

Bad example for naive Ford-Fulkerson

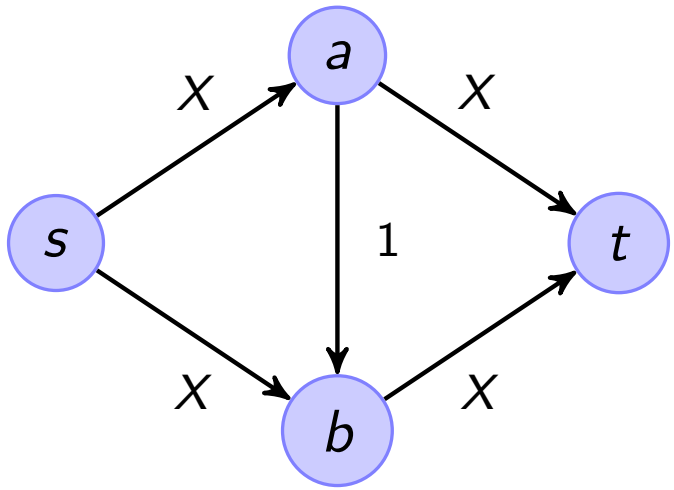


Figure: The numbers denote the capacities of the edges.

- The max-flow is clearly $2X$.
- A naive Ford-Fulkerson algorithm could alternate between
  - pushing a 1 unit flow along the augmenting path $s \to a \to b \to t$
  - pushing a 1 unit flow along the augmenting path $s \to b \to a \to t$
- This would resul in $2X$ iterations, which is exponential if say X is given in binary.

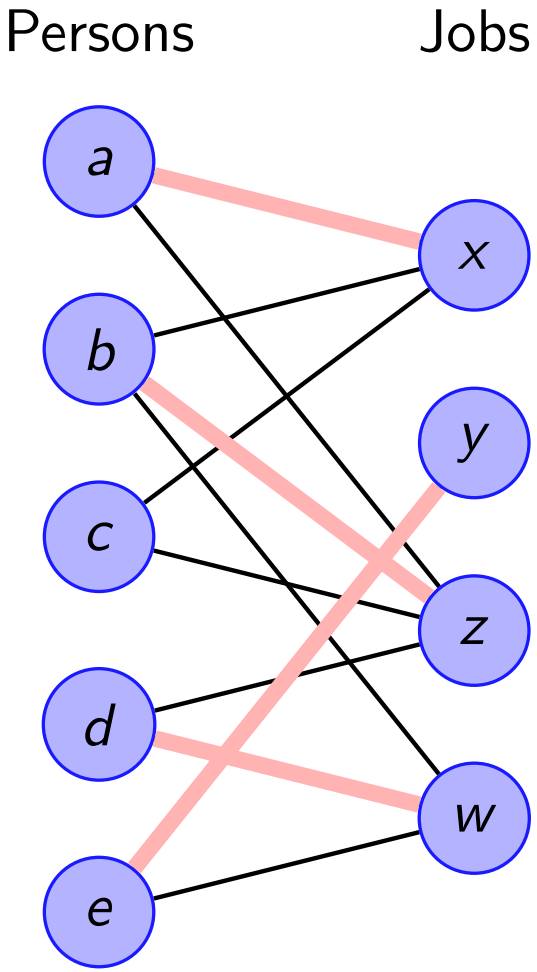
Some ways to achieve polynomial time

- Choose an augmenting path having shortest distance: This is the Edmonds-Karp method and can be found in CLRS. It has running time $O(nm^2)$, where $n = |V|$ and $m = |E|$.
- There is a "weakly polynomial time" algorithm in KT
  - Here the number of arithmetic operations depends on the length of the integral capacities.
  - It follows that always choosing the largest capacity augmenting path is at least weakly polynomial time.
- There is a history of max flow algorithms leading to a recent $O(mn)$ time algorithm (see http://tinyurl.com/bczkdfz).
- Although not the fastest, next lecture I will present Dinitz's algorithm which has runtime $O(n^2m)$.
  - A shortest augmenting-path method based on the concept of a blocking flow in the leveled graph.
  - Has an additional advantage (i.e. an improved bipartite matching bound) beyond the somewhat better running time of Edmonds-Karp.

An application of max-flow: the maximum bipartite matching problem

The maximum bipartite matching problem

- Given a bipartite graph $G = (V, E)$ where
  - $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \varnothing$
  - $E \subseteq V_1 \times V_2$
- **Goal:** Find a maximum size matching.



- We do not know any efficient DP or greedy optimal algorithm for solving this problem.
- But we can efficiently reduce this problem to the max-flow problem.

The reduction

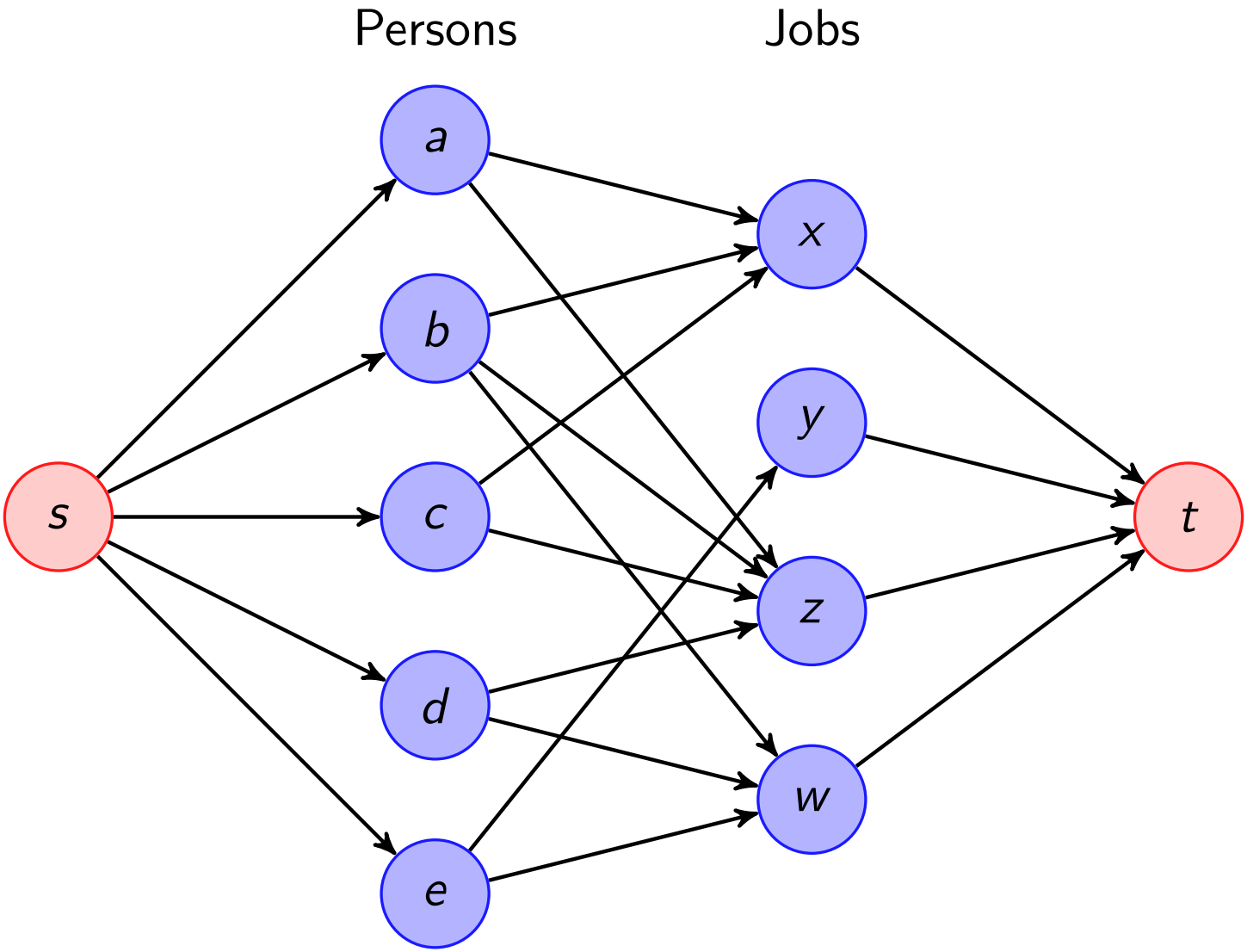


Figure: Assign every edge of the network flow a capacity 1.

The reduction preserves solutions

Claims

1. Every matching M in G gives rise to an integral flow f_M in the newly constructed flow network F_G with $val(f_M) = |M|$
2. Conversely every integral flow f in F_G gives rise to a matching M_f in G with $|M_f| = val(f)$.

Let $m = |E|, n = |V|$

- Time complexity: $O(mn)$ using any Ford Fulkerson algorithm since the max flow is at most n and $C = n$ since all edge capacities are integral and set to 1.
- Dinitz's algorithm can be used to obtain a runtime $O(m\sqrt{n})$.

A few more comments on this reduction

- When we get to our next big topic (NP completeness), we will be focusing on decision problems and as a decision problem we have $|M| \geq k$ iff $val(f_M) \geq k$.
- The reduction we are using is very efficient (linear time in the representation of the graph) and it is a special type of polynomial time reduction which we will call a polynomial time transformation.

Alternating and augmenting paths in graphs

There is a graph theoretic concept of an augmenting path relative to a matching (in an arbitrary graph).

- An alternating path π relative to a matching M is one whose edges alternate between edges in M and edges not in M.
- An augmenting path is an alternating path that starts and ends with an edge not in M.

- The reduction provides a 1-1 correspondence between augmenting paths in the bipartite G wrt. M_f and augmenting paths in G_{f_M}.