Lecture 12

- Network flow problems in graphs
- Matching problems in bipartite graphs

(Reading: Weiss Ch. 9.4)

Network flow problems

- Consider using a directed weighted graph to model a kind of transport network
  - Some examples:
    - Edges are highway segments, nodes are interchanges between highways, weights are how many cars each highway segment can carry
    - Edges are network links, nodes are network switches, weights are network link bandwidths
    - Edges are lines in a power transmission network, nodes are interconnect stations, weights are power carrying capacities of the lines
- Such models have these features:
  - Edge weights are *capacities*: the maximum flow of "traffic" the edge can carry
  - Some nodes in the graph may be *sources* of flow (flow can originate there, e.g. a power station in the power network)
  - Some nodes may be *sinks* of flow (they can absorb flow, e.g. a neighborhood at the end of a power line)
  - Some nodes only transmit flow (flow coming in must equal flow going out, e.g. a power grid interconnect station)
- We will look at solving the *maximum flow* problem in such a graph

Graphs and flows

- To model a network flow problem, we can construct a "capacity graph" G:
  - a weighted directed graph G = (V,E)
  - weights are nonnegative integers, and will be interpreted as flow capacities. (Take nonexistent edges and edges with capacity of 0 to be equivalent.)
  - one distinguished source vertex *s* in V, with no edges entering it
  - one distinguished sink vertex *t* in V, with no edges leaving it

- A **flow** in a capacity graph is an assignment of a number to each edge in the graph such that:
  - The flow number on an edge is nonnegative, and does not exceed the capacity of the edge. (This the *capacity constraint* on flow)
  - Except for the distinguished source and sink *s* and *t*, the total of flow numbers on edges entering a vertex exactly equals the total of flow numbers on edges leaving it. (This is the *conservation constraint* on flow)

- We will be interested in flow assigments that maximize the total flow coming out of the source vertex *s* (by conservation, this also maximizes the flow coming in to the sink *t*)

Flow value and maximum flow

- More formally:
  - We have a weighted directed graph G=(V,E).
  - Elements of E are triples $e = (u,v,c)$, with $u, v \in V$, and $c \geq 0$. For convenience, we can write the capacity of edge e as c_e
  - There are distinguished vertices $s, t \in V$
- A **flow** in G is a function f from edges in G to real numbers, with these constraints:
  - *capacity constraint*: for each edge $e = (u,v,c)$ in E, flow does not exceed capacity:

$$0 \leq f(e) \leq c_e$$

  - *conservation constraint*: For each vertex v in V, except for s and t, inflow = outflow:

$$\sum_{e \text{ entering } v} f(e) = \sum_{e \text{ leaving } v} f(e)$$

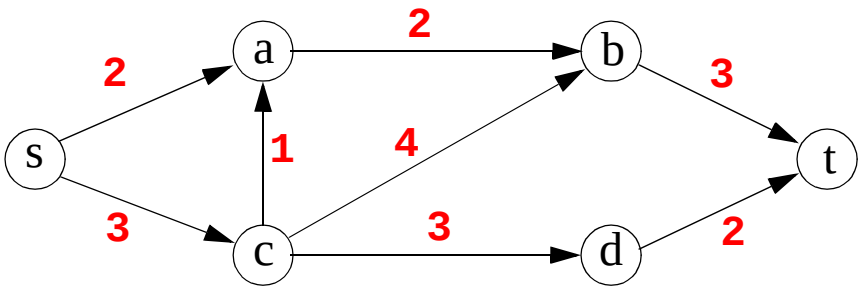
- (Source vertex s has no entering edges, but can generate flow. Sink vertex t has no exiting edges, but can absorb flow. All other vertices conserve flow.)
- Define the **value** of a flow f as the total flow on edges leaving the source s:

$$value(f) = \sum_{e \text{ leaving } s} f(e)$$

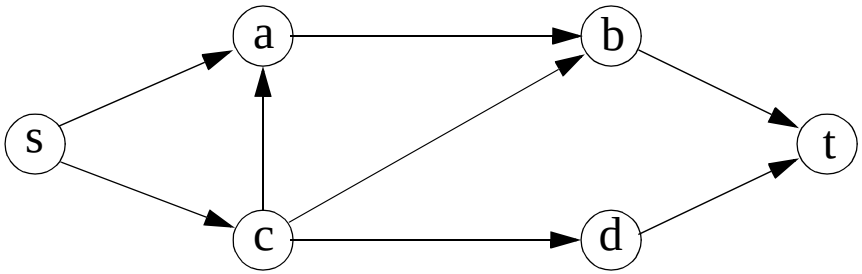
- For a given graph G, how to find a flow f that has maximum value?

Flow in a graph: example

- Consider this graph, with source and sink vertices and capacities as shown



- Find a flow in that graph. Show the result here:



- What is the value of that flow? ___
  Is there a better (higher value) one? ___
  What is an algorithm that is guaranteed to always find the maximum value flow?

Flow graphs and residual graphs

- A *flow graph* G_f for a capacity graph G:
  - has all and only the same vertices and edges as G
  - has edge weights that obey the capacity and conservation constraints of a flow in G

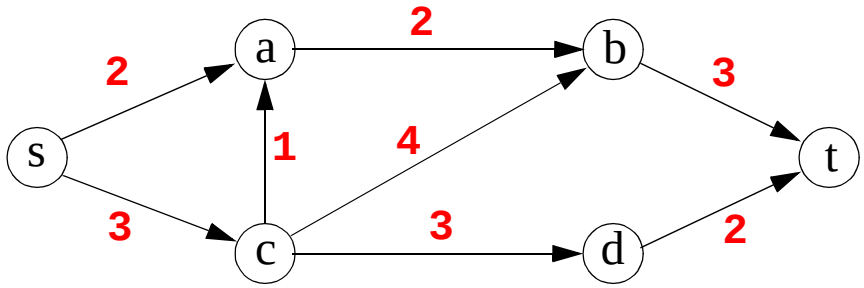
- A *residual graph* G_r for a capacity graph G and a flow graph G_f :
  - has all and only the same vertices as G
  - for every edge (u,v,c) in G, and corresponding edge (u,v,f) in G_f, there is an edge $(u,v,c-f)$ in G_r
  - these weights $c-f$ in G_r are called "residual capacities"
  - (if the residual capacity of an edge is 0, you can consider the edge to not exist at all)
  - (in a minute, we will see that G_r needs more edges than this...)

- So the residual graph shows how much unused capacity in the original graph G there is on each edge in a particular flow graph G_f

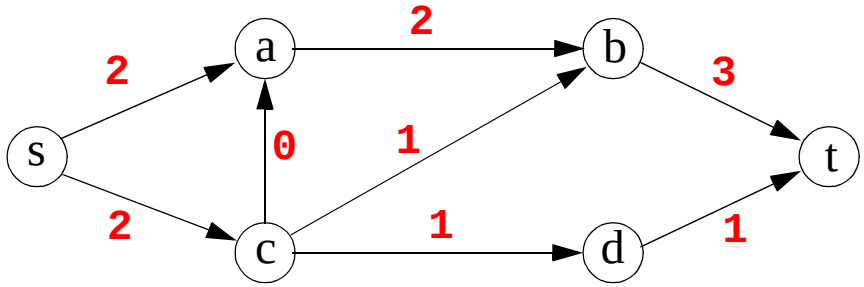
- We can use the idea of a residual graph to start developing an algorithm to solve the maximum flow problem

Example capacity, flow, and residual graphs

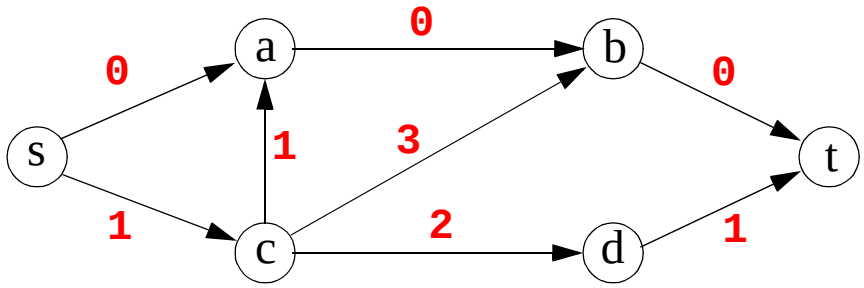
- Input capacity graph G:



- A flow graph G_f:



- The residual graph G_r for that G and G_f:



Using the residual graph

- Keep in mind that weights on edges in the residual graph show unused flow capacity, and weights on edges in the current flow graph satisfy the capacity and conservation constraints
- Suppose we can find a simple path P from source *s* to sink *t* in the residual graph
- Let the smallest edge weight on path P be *b*
  - (note we must have $b > 0$, since residual graph edges with 0 weight are nonexistent and can't be part of a path, and unused flow capacity is never negative)
- Now we can add that flow *b* to all edges in the path P, back in the flow graph, and get a flow graph with a higher value flow! (P is thus called an *augmenting path*.)
  - Doing this will not violate the capacity constraint, since *b* is not more than the unused flow capacity on any edge on the path P
  - Doing this will not violate the conservation constraint, since every vertex on P except *s* and *t* has one incoming edge and one outgoing edge, and we are adding the same amount, *b*, to each
  - Doing this will increase the flow value, since we are adding a positive number, *b*, to an edge coming out of *s*
- There might be more than one augmenting path in the residual graph. Which one should we pick? Try a greedy approach: pick the one with the largest *b*...

Toward a maximum-flow algorithm: greedy approach

- Let's try the following approach. Initially set all weights in the flow graph to 0. Then:
- 1. Compute the edge weights in the residual graph G_r from the input capacity graph G and the current flow graph G_f. G_r shows us where there is unused flow capacity that we can possibly use. Consider 0-weight edges in G_r as nonexistant.

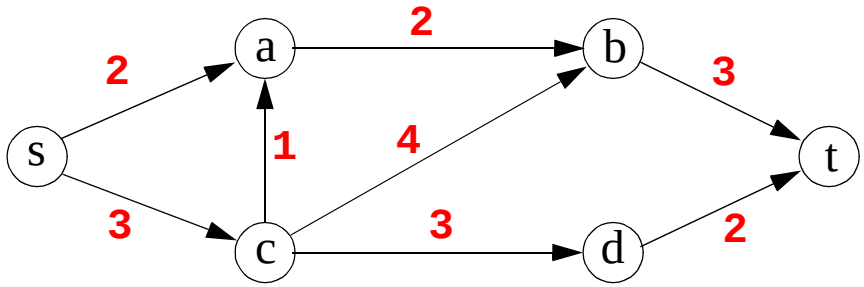
- 2. Consider simple paths from source *s* to sink *t* in G_r. These *s*-*t* paths are possible paths for pushing additional flow from *s* to *t*.
  - If there are no *s*-*t* paths in G_r, Done: G_f shows the maximum flow.
  - Each *s*-*t* path has a "bottleneck" edge. (The bottleneck edge on a path is the one that has the smallest edge weight -- i.e. capacity -- of any edge on that path.)
  - Find the *s*-*t* path P with the largest bottleneck edge weight. Call this edge weight *b*. P is an *augmenting path* P that will permit increasing flow from *s* to *t* by amount *b*.

- 3. Update the flow graph G_f with the additional flow along the augmenting path P:
  - Consider the edges in the augmenting path P, in G_f. Add P's "bottleneck" flow *b* to the current weight of each of those edges in the flow graph G_f.

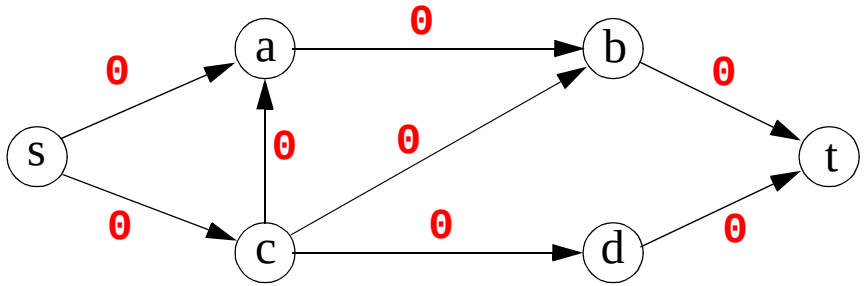
- 4. Go to 1.

First algorithm capacity, flow, and residual graphs: iteration 0

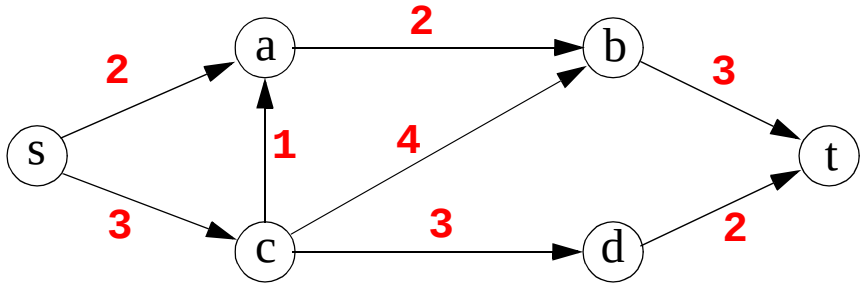
- Input capacity graph G:



- A flow graph G_f , initially all weights 0:

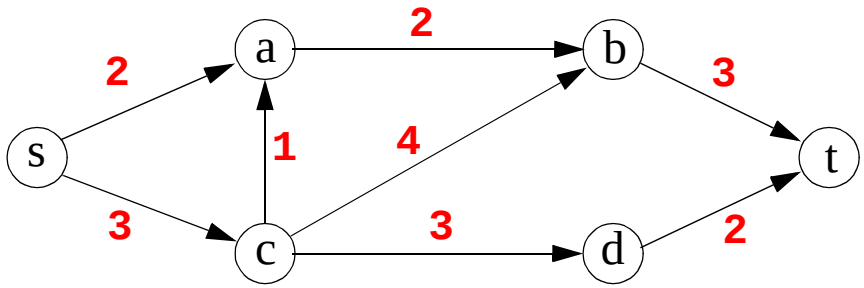


- Residual graph G_r:, first max-bottleneck *s*-*t* path is (s,c),(c,b),(b,t), bottleneck 3...

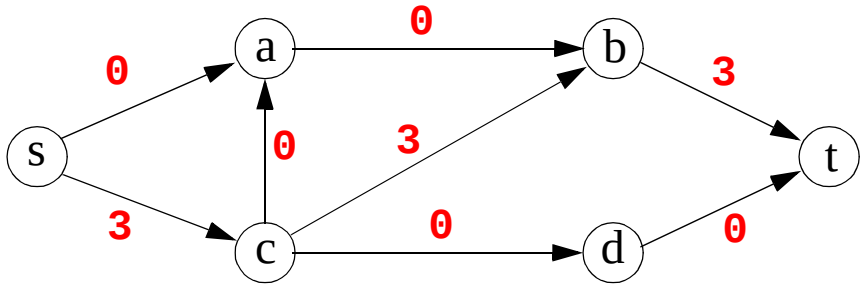


First algorithm capacity, flow, and residual graphs: iteration 1

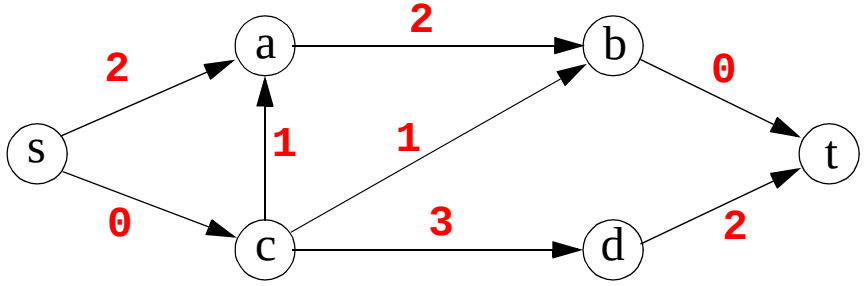
- Input capacity graph G:



- Update flow graph G_f, adding bottleneck weight along augmenting path



- Update residual graph G_r ... but now there is no augmenting path... done???



Fixing the greedy maximum-flow algorithm

- That simple greedy approach did not work.
- Evidently, we need some way of backtracking: being able to undo some flow on some edges, if we overcommitted and used 'too much' flow there
- Basic idea for how to do this, leaving everything else in the algorithm unchanged:
  - When computing the residual graph given the current flow graph, add some additional edges...
  - These additional edges will have the *same* weight as edges in the flow graph, but that point in the *opposite* direction! Call these additional edges *backward edges*.
  - (The forward edges we talked about before had a weight that is the difference between the capacity graph edge weight and the flow graph edge weight, and point in the same direction. They will still be there.)
  - Backward edges, as well as the regular forward edges, can be included in an augmenting path in the residual graph
  - However, when using the augmenting path bottleneck weight to update the flow graph weights, if an augmenting path edge is a backward edge, its weight gets *subtracted* from the current flow graph edge (undoing flow!), instead of added.
- It turns out that this idea works, and leads to the Ford-Fulkerson algorithm [1962]

Maximum-flow: Ford-Fulkerson algorithm

- Initially set all weights in the flow graph to 0. Then:

- 1. Compute the edge weights in the residual graph G_r from the current flow graph G_f:
  - For each edge (u,v,c) in G, and corresponding edge (u,v,f) in G_f, do the following:
    - Update forward edge (u,v,x) in G_r to be $(u,v, c-f)$
    - If $f>0$, create or update backward edge (v,u,x) in G_r to be (v,u, f)
  - Consider 0-weight edges in the resulting G_r as nonexistant.

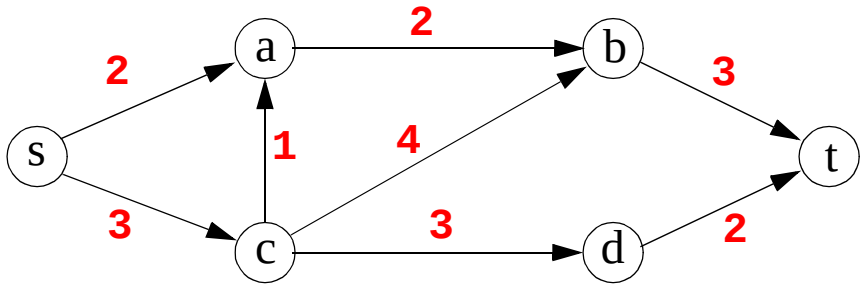
- 2. Find the s-t augmenting path P in G_r with largest bottleneck edge weight b. If there is no s-t path in G_r, Done: G_f shows the maximum flow.

- 3. Update the flow graph G_f with the flow along the augmenting path P:
  - For each edge (u,v,x) in P, update edge weights in the flow graph G_f:
    - If (u,v) is a forward edge in G_r, update edge (u,v,f) in G_f to be $(u,v, f+b)$.
    - If (u,v) is a backward edge in G_r, update edge (v,u,f) in G_f to be $(v,u, f-b)$.

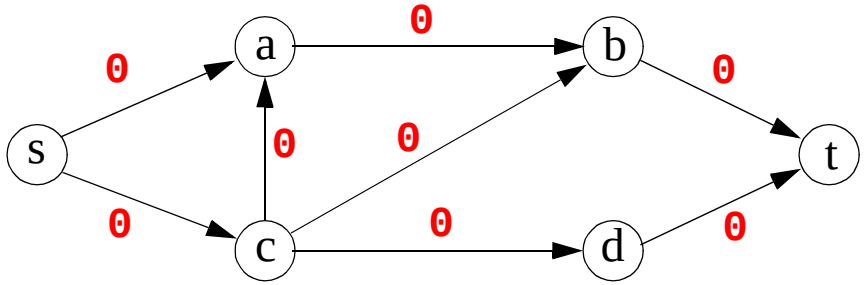
- 4. Go to 1.

Ford-Fulkerson algorithm: iteration 0

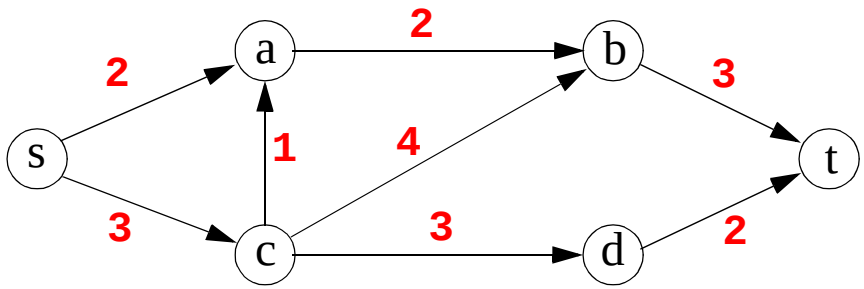
- Input capacity graph G:



- A flow graph G_f, initially all weights 0:

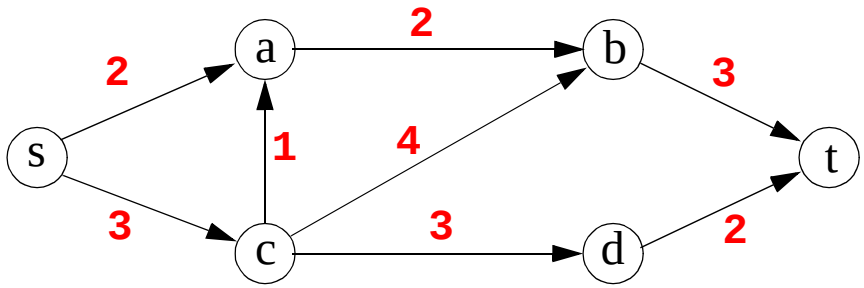


- Residual graph G_r:, first max-bottleneck s-t path is (s,c),(c,b),(b,t), bottleneck 3...

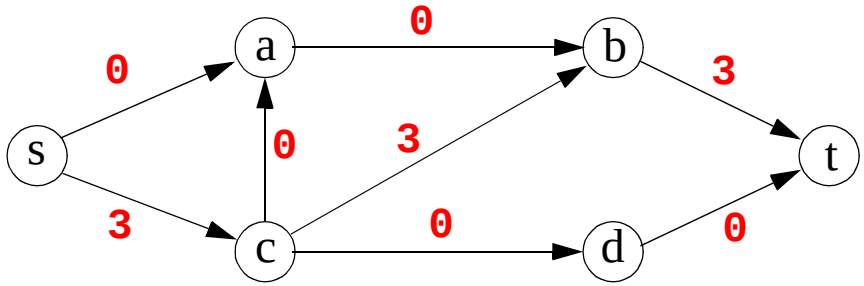


Ford-Fulkerson algorithm: iteration 1

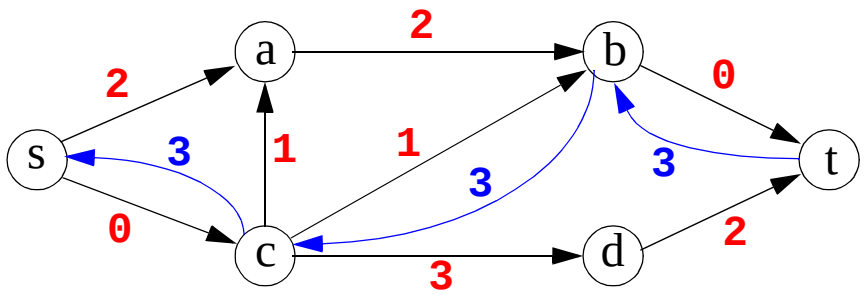
- Input capacity graph G:



- Update flow graph G_f, adding bottleneck weight along augmenting path

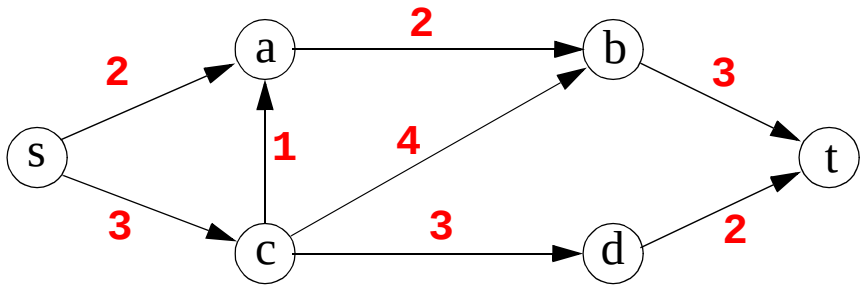


- Update residual graph G_r with fwd+bkwd edges, and find best augmenting path

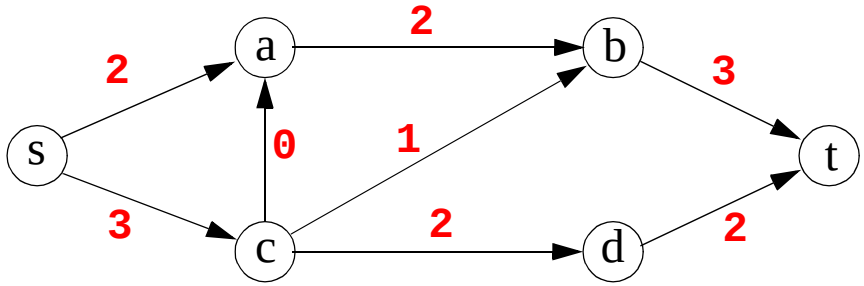


Ford-Fulkerson algorithm: iteration 2

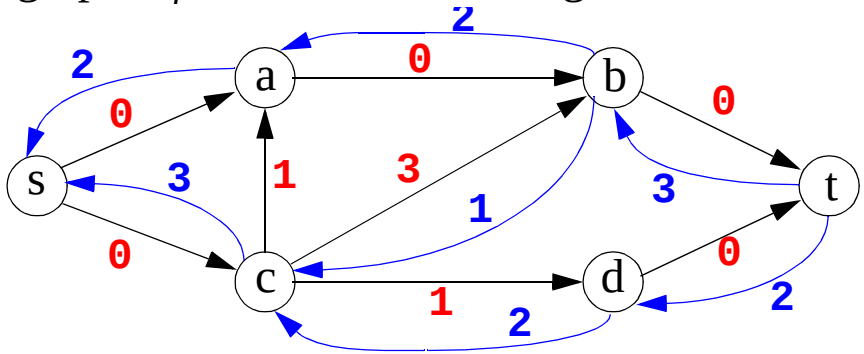
- Input capacity graph G:



- Update flow graph G_f, adding/subtracting bottleneck weight along augmenting path



- Update residual graph G_r with fwd+bkwd edges, and... no augmenting path, done



Implementing the Ford-Fulkerson algorithm

- As described, the Ford-Fulkerson algorithm requires 3 graphs:
  - the input capacity graph G
  - the flow graph G_f
  - the residual graph G_r
- These graphs can be implemented in various ways, as long as the implementation supports the needed operations (adding and deleting edges and updating edge weights, etc.)
- How to find the largest-bottleneck augmenting *s*-*t* path in the residual graph G_r?
  - A simple modification of Dijkstra's algorithm for finding shortest paths can find the largest-bottleneck path
- What is the worst-case time cost of the algorithm? Let the value of the maximum flow be F. If capacities are integers, it can be shown that the algorithm will find the maximum flow in O(log F) iterations. Each iteration involves running the modified Dijkstra's algorithm on the residual graph, so the total worst-case time cost is O(|E| log |E| log F).

Choosing an augmenting path in the residual graph

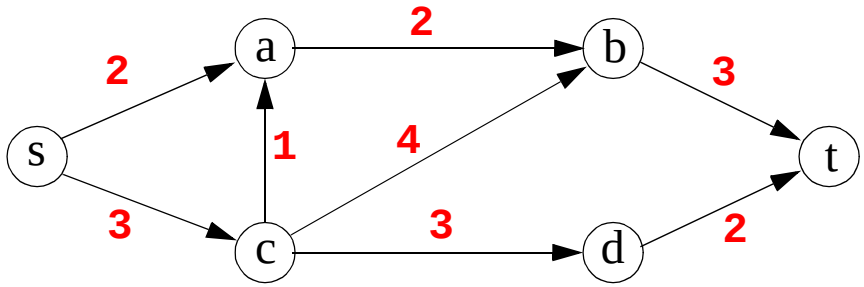
- The Ford-Fulkerson algorithm we just described is just one version of the basic maximum-flow algorithm (a very popular one)
- In step 2, it finds the *s*-*t* augmenting path P in G_r with largest bottleneck edge weight *b*
- But in fact, once we have the backward-edge idea working, *any* choice of augmenting *s*-*t* path in G_r will work in step 2, if the input graph G has integer weights:
  - If any augmenting path exists, its bottleneck weight will be greater than 0
  - Backward edges to the source vertex *s* never appear in an augmenting path (the path starts at *s*, and must be a simple path)
  - Therefore any augmenting path must increase the value of the flow by at least 1
  - And since the value of the flow can never be more than the sum of the weights on edges leaving *s* , the maximum flow will always eventually be reached
- So, if in the input graph G all capacities are integers, and the sum of the capacities on edges leaving the source vertex *s* is *C*, the algorithm will iterate at most *C* times, no matter which augmenting path is chosen in each iteration!
- However, the result can be a slow algorithm, if *C* is large. Choosing the largest-bottleneck augmenting path in step 2 is better.
- What about other ways of choosing the augmenting path?...

Another way to choose an augmenting path: Edmonds-Karp

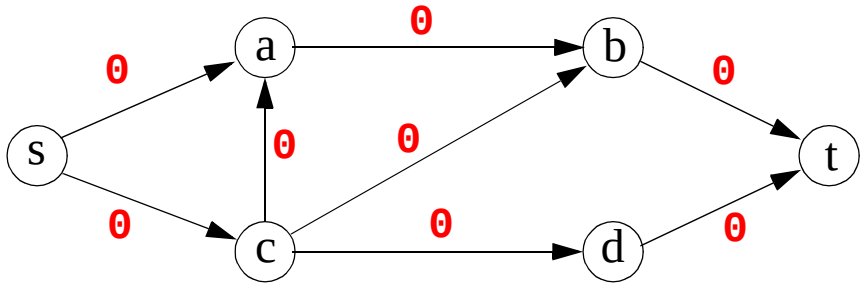
- A variant on the Ford-Fulkerson algorithm was published by Edmonds and Karp [1972]
- The variation is in step 2. Insead of finding the *s*-*t* path with the largest bottleneck, just find the *s*-*t* path with the fewest edges:
  - 2. Find the *s*-*t* augmenting path P in G_r with the fewest edges. Let its bottleneck edge weight be *b*. If there is no *s*-*t* path in G_r , Done: G_f shows the maximum flow.
- Since the path with the fewest edges can easily be found with breadth-first search, this variation is simple to implement
- This just looks like a simple heuristic: short paths are maybe less likely to have restrictive bottlenecks, just because they have fewer edges
- However, Edmonds and Karp were able to prove that their variation finds the maximum flow in worst-case time $O(|V| \, |E|^2)$, independent of the graph edge weights

Edmonds-Karp algorithm: iteration 0

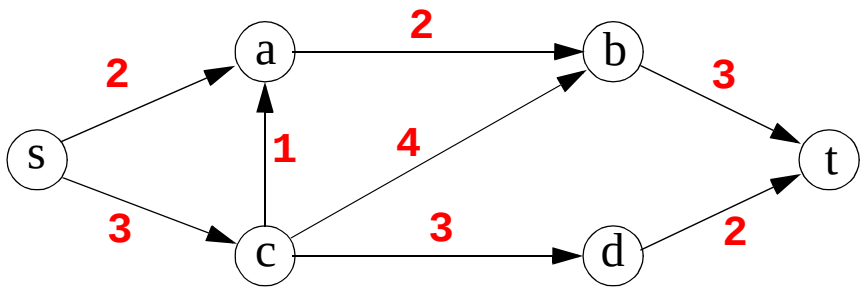
- Input capacity graph G:



- A flow graph G_f, initially all weights 0:

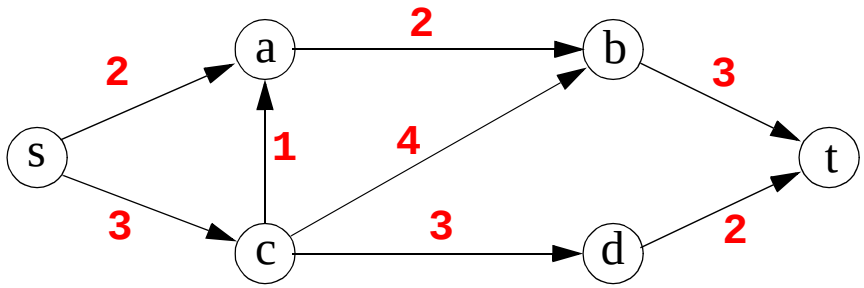


- Residual graph G_r:, a shortest s-t path is (s,a),(a,b),(b,t), bottleneck 2...

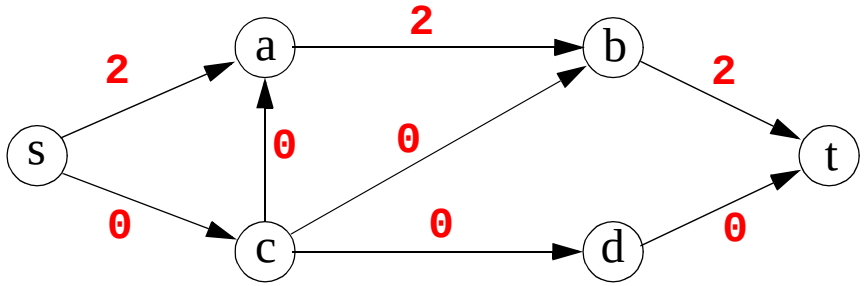


Edmonds-Karp algorithm: iteration 1

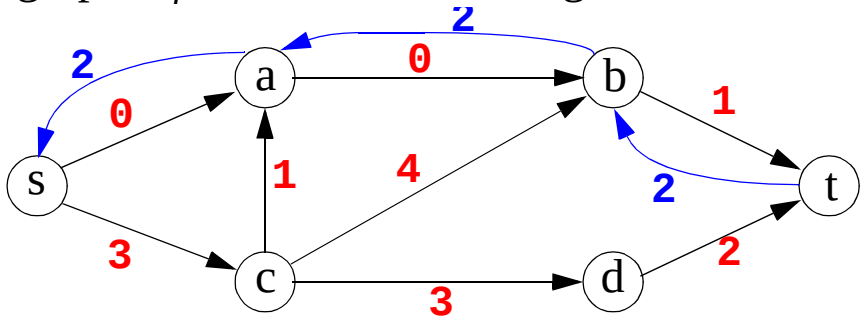
- Input capacity graph G:



- Update flow graph G_f, adding bottleneck weight along augmenting path

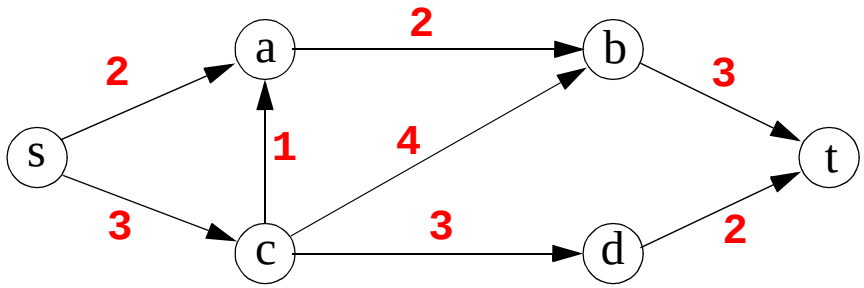


- Update residual graph G_r with fwd+bkwd edges, and find a shortest s-t path

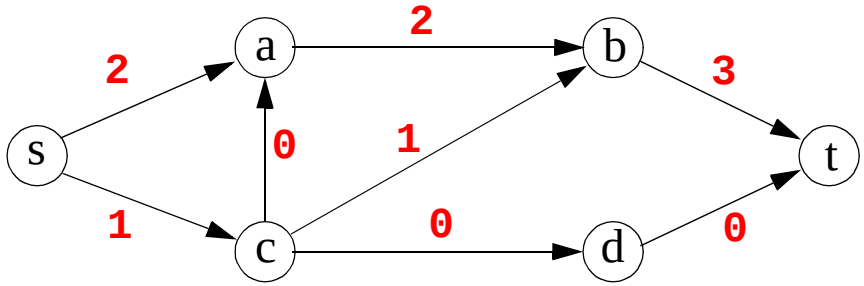


Edmonds-Karp algorithm: iteration 2

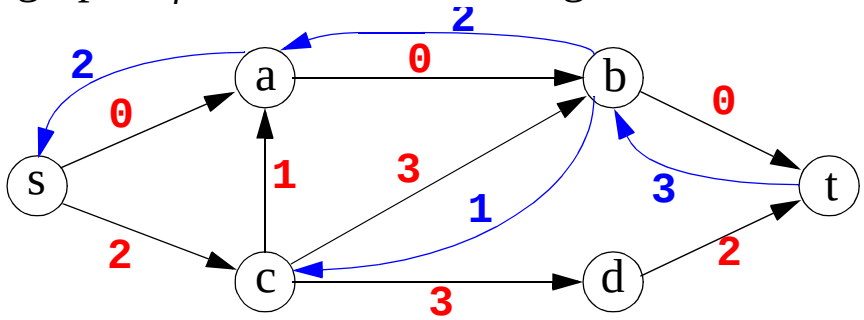
- Input capacity graph G:



- Update flow graph G_f, adding bottleneck weight along augmenting path

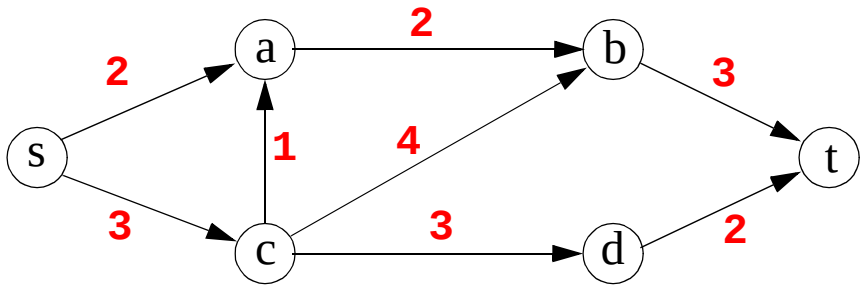


- Update residual graph G_r with fwd+bkwd edges, and find shortest s-t path

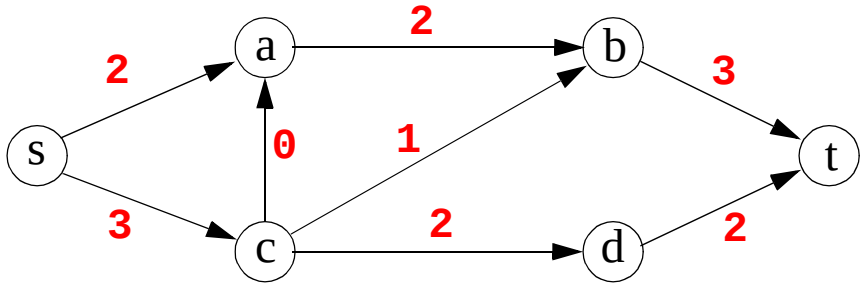


Edmonds-Karp algorithm: iteration 3

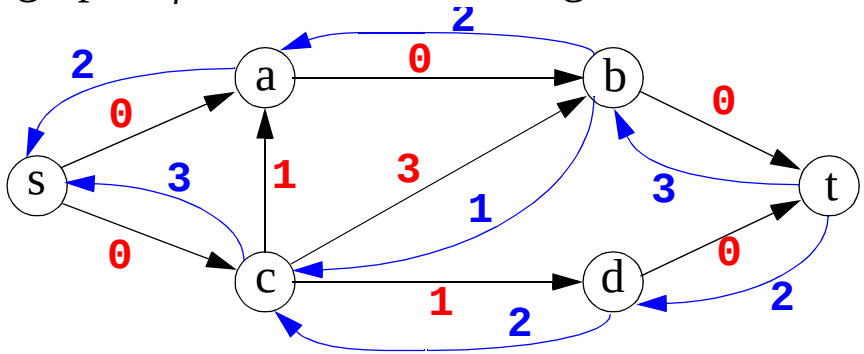
- Input capacity graph G:



- Update flow graph G_f, adding/subtracting bottleneck weight along augmenting path

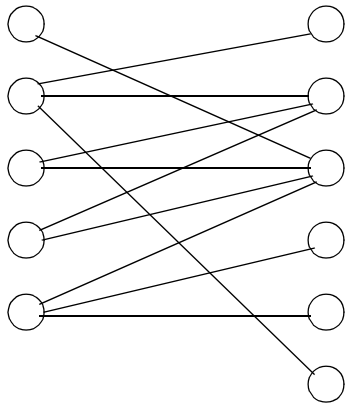


- Update residual graph G_r with fwd+bkwd edges, and... no augmenting path, done



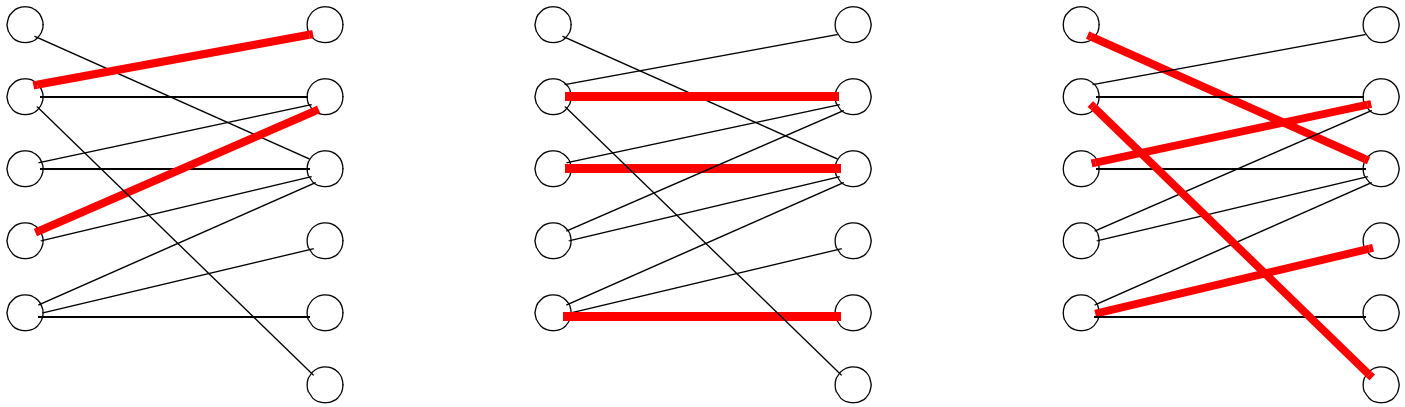
Bipartite graphs

- A *bipartite graph* is a graph G = (V,E) in which the vertices V can be partitioned into two subsets, such that no edge in E touches vertices in the same subset.
- More formally, G = (V,E) is bipartite if
  - There is a partition of V = $V_1 \cup V_2$, $V_1 \cap V_2 = \varnothing$, such that:
  - For every edge (u,v) in E: $u \in V_1$ and $v \in V_2$, or $u \in V_2$ and $v \in V_1$
- An example bipartite graph:



Matchings in bipartite graphs

- A *matching* in a bipartite graph is a set of edges $M \subseteq E$ such that each vertex in V appears in at most one edge in M
- A *perfect matching* in a bipartite graph is a matching M such that each vertex in V appears in exactly one edge in M
- Here are some matchings in the example bipartite graph. Edges in each matching shown in red. This graph does not have a perfect matching (do you see why?).



The matching problem in bipartite graphs

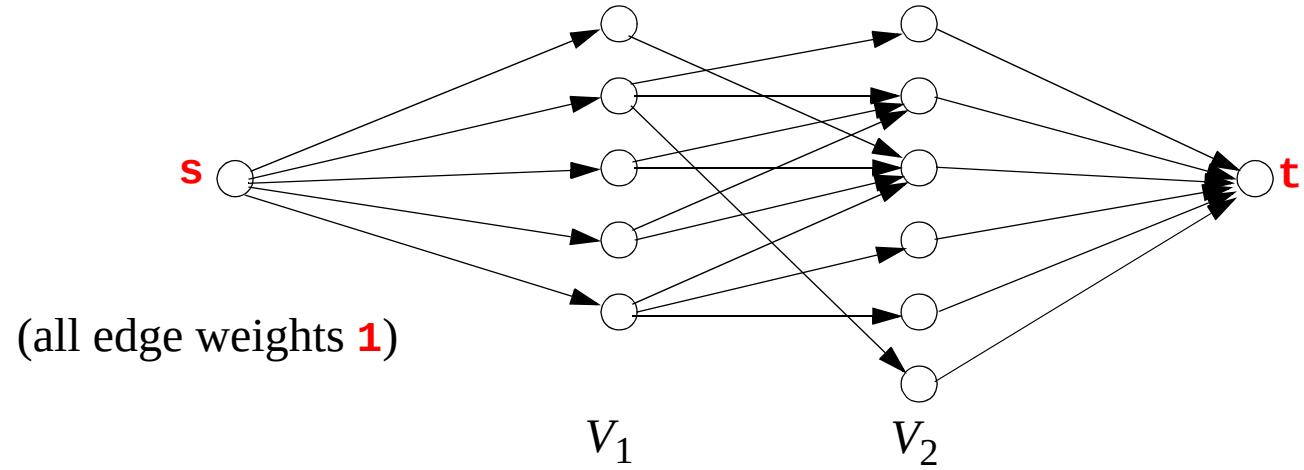
- The bipartite graph matching problem is:
  - Given a bipartite graph G = (V,E), with vertex set partition V = $V_1 \cup V_2$:
  - Find the *largest* subset M of E such that each vertex in V appears in at most one edge in M.

- The bipartite matching problem models many real world problems, e.g.:
  - V_1 is a set of job openings; V_2 is a set of applicants; edges indicate which applicants are qualified for which jobs. Find a pattern of hiring which pairs the maximum number of applicants with a job
  - V_1 is a set of computer processes; V_2 is a set of CPU cores; edges indicate which processes can run on which cores. Find an assignment of processes to hardware that maximizes hardware utilization
  - Etc.!

- How to solve the bipartite matching problem? We can modify the bipartite graph, and use a maximum-flow algorithm...

Using network flow to solve matching

- Start with a bipartite graph G = (V,E), with vertex set partition V = $V_1 \cup V_2$
- Make edges in E directed, from a vertex in V_1 to a vertex in V_2 , with weight 1
- Add an additional vertex *s*. Add edges from *s* to every vertex in V_1 , each with weight 1
- Add an additional vertex *t*. Add edges from every vertex in V_2 to *t* , each with weight 1



- Solve the maximum flow problem in the resulting graph!
- Since each vertex in V_1 has incoming capacity 1, its outgoing flow in the solution will be 0 or 1 (recall with integer capacities, flows are always integers)...
- So each V_1 vertex will be matched with at most one V_2 vertex; but as many of them as possible will be matched, to maximize the flow out of *s*

Next time

- ✔ Connectedness in graphs
- ✔ Spanning trees in graphs
- ✔ Finding a minimal spanning tree
- ✔ Time costs of graph problems and NP-completeness
- ✔ Finding a minimal spanning tree: Prim's and Kruskal's algorithms
- ✔ Intro to disjoint subsets and union/find

Reading: Weiss, Ch. 9, Ch 8