Network Flow

Mark Floryan and Tom Horton
CS4102 – Spring 2021

Topics

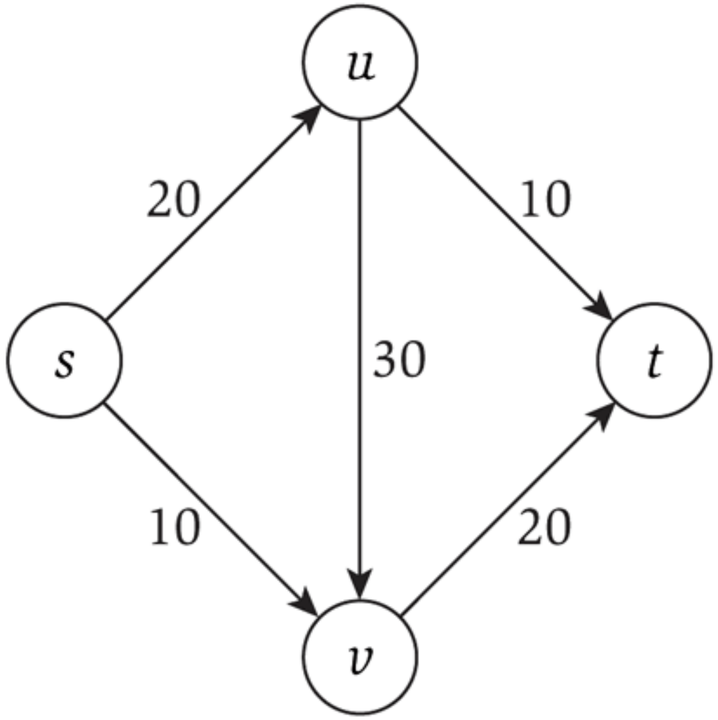
Topics in this slide-deck:

- Flow-Networks
  - Max-flow problem
  - Ford-Fulkerson Algorithm
- Related Graph Problems
  - Bi-partite Matching
  - Minimum Cut
- A Short introduction to reductions
  - …but many more reductions coming at end of course

Maximal Flow

Flow networks

- Consider a flow network, which is a specialized directed graph with:
  - A single source node s
  - A single terminus node t
  - Capacities on each edge
    - That must be integer!
- What is the maximum flow you can send from s to t?



Applications

- Transportation networks
  - How many people can be routed?
- Computer networks
- Electrical distribution
- Water distribution

- Note that all these applications have multiple sources and multiple sinks!
  - Whereas the flow networks we study do not, yet

Some rules

- Source node has NO incoming flow
- Terminal (sink) node has NO outgoing flow
- Internal nodes has net zero flow
  - all units of flow going in must be going out as well
- No edge is over capacity
- GOAL: Find the maximum flow that can be "pushed" through the network

Algorithm notation

- Graph G has vertices V and edges E
  - $s \in V$ is the source
  - $t \in V$ is the sink (terminus)
- $f(u,v)$: the flow on the edge from u to v
  - $f(v,u)$: the backflow on the edge from v to u
- $c(u,v)$: the capacity on the edge from u to v
- $c_f(u,v)$: the *residual* capacity on the edge from u to v
- G_f is the graph where the edges weights are the residual capacities
  - THIS is usually the graph we actually use when running the algorithm we are about to see.

Ford-Fulkerson: Algorithm overview

- Consider the *residual* capacities
  - Meaning how much capacity is left after taking into account how much flow is going through that edge
- Find a path from *s* to *t* such that the minimum residual capacity is greater than zero
  - Since everything is integer, it must be 1 or more
- Update the residual capacities after taking into account this new flow
- Repeat until no more such paths are found

Backflow

- Each edge has forward flow and backflow
  - The two must always be inverses of each other!
- This allows for modeling of flow “returning” along a given edge

Ford-Fulkerson Algorithm

1. $f(u,v) = 0$ for all edges (u,v)
2. While there is an "augmenting" path p from s to t in G_f such that $c_f(u,v) > 0$ for all edges $(u,v) \in p$
   a. Find $c_f(p) = \min\{c_f(u,v) \mid (u,v) \in p\}$
   b. For each edge $(u,v) \in p$
      i. $f(u,v) = f(u,v) + c_f(p)$ send flow along the path
      ii. $f(v,u) = f(v,u) - c_f(p)$ send backflow the other way

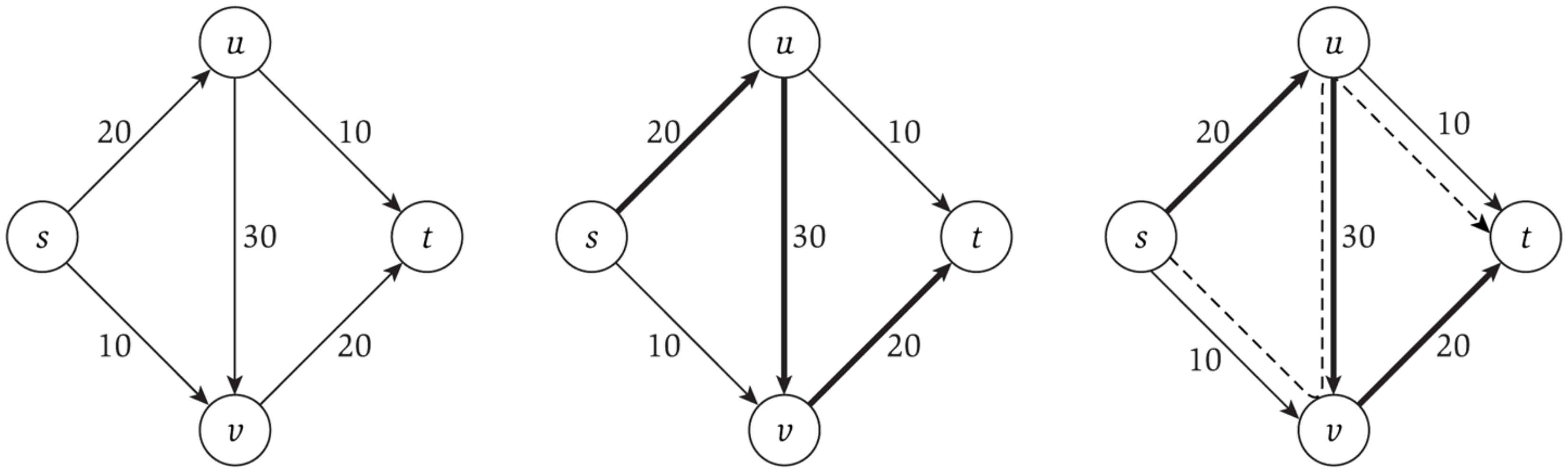
Running time

- Is O(E*f)
  - E is the number of edges
    - Maximum time to find an augmenting path via depth-first search
      - Can also use breadth-first search!
  - f is the maximum flow of the final graph
    - Minimum flow on an augmenting path is 1, so the maximum number of steps is thus f

What type of search?

- "While there is a path p from s to t in G_f"
  - A depth-first search is the Ford-Fulkerson algorithm
    - Each augmenting path can be found in O(m) time
    - And there can be f paths
    - So the running time is O(mf)
    - Will not terminate with irrational edge values
  - A breadth-first search is the Edmonds-Karp algorithm
    - Runs in $O(nm^2)$
      - Total number of augmentations is O(nm)
      - And finding each augmentation takes O(m)
    - Guaranteed termination with irrational edge values
    - Run-time is independent of the maximum flow of the graph

Our example



(a) (b) (c)

Minimum Cut

Definition: Cut

- Given a flow network, we want to *cut* edges…

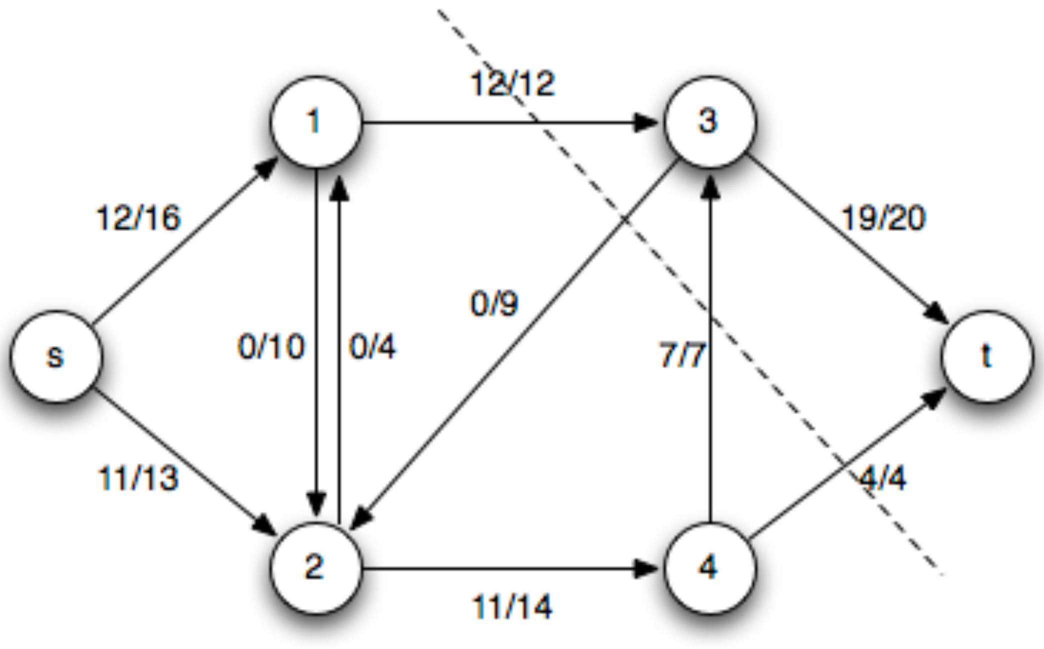
- A cut C = (A, B) where:
  - A is a set of vertices (A is a subset of V)
  - B is a set of vertices (B also a subset of V)
  - A intersect B = null set (no shared vertices)
  - A union B = V (all vertices in either A or B)

- What do we care about?
  - Well, we care about the edges that go across this cut.
  - Either from node in A to a node in B or vice versa.

Definition: Net Flow across Cut

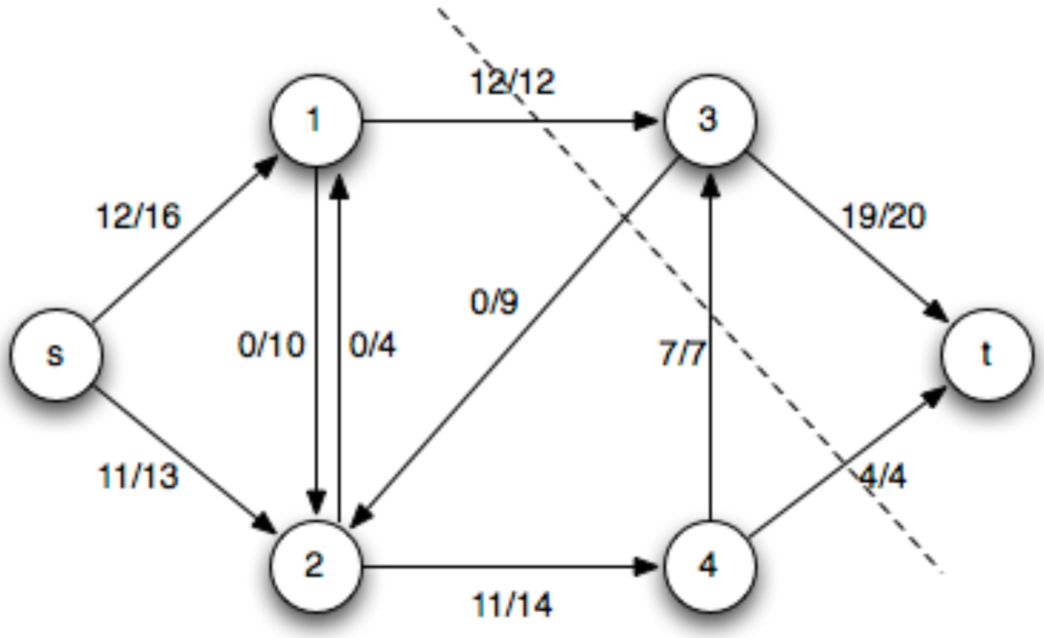
- Given a cut C = (A, B)

- The **net flow across the cut** C = (A, B) is the sum of the flows on its edges from A to B minus the sum of the flow on its edges from B to A



Definition: Flow-value lemma

- Let ***f*** be any flow and C = (A, B) be any cut
  - The net flow across (A, B) equals the value of the flow ***f***



Proof: Flow-value lemma

- Let ***f*** be any flow and C = (A, B) be any cut
  - The net flow across (A, B) equals the value of the flow ***f***

- Proof by induction on the size of B
  - B.C. B = {t} (B is only the sink)
    - Clearly this is true as the flow across the cut is everyone sinking into t, which is the definition of the flow f
  - I.H. Assume true for some cut C = (A, B)
  - I.S. Move one node from A to B
    - Choose an a' to move that has at least one edge to a node in B
    - We know the flow f never changes
    - How does the value of the new cut C' = (A', B') change?
    - It doesn't! Why? See next slide…

Proof: Flow-value lemma cont.

- Let ***f*** be any flow and C = (A, B) be any cut
  - The net flow across (A, B) equals the value of the flow ***f***

- Why does C' = (A', B') have the same net flow?
  - Local equilibrium: net flow coming into a' from nodes in A only must equal the flow going out across the cut to nodes in B
  - After a' is moved:
    - everything going across the cut now goes to something in B from B, everything going to or from a node in A now goes across the cut.
  - Thus, by local equilibrium the value of the cut C' is equivalent to the value of the cut C
  - Induction done!

Max-flow min-cut theorem

- The max-flow min-cut theorem states that the maximum value of an s-t flow is equal to the minimum capacity of an s-t cut

- In other words, if you look at all the possible cuts in the graph, and find the smallest capacity of those cuts, then that value is the value of the maximum flow for that network.

Another definition: Weak Duality

- Let ***f*** be any flow and C = (A, B) be any cut
- Then:
  - Value of f <= capacity of C
- Note: We are talking about the CAPACITY of C, not the value of the flow across C

Another version of MaxFlow-MinCut

- The following three statements are equivalent:

- For some flow f
  - 1) There exists a cut whose capacity equals the value of f
  - 2) f is a maximum flow
  - 3) There is no augmenting path with respect to f

- Let's prove this!
  - 1 $\rightarrow$ 2
  - 2 $\rightarrow$ 3
  - 3 $\rightarrow$ 1

How to determine the min cut?

- Use the Ford-Fulkerson algorithm to determine max flow
  - Time is O(mf)
- Worst case is each edge needs a cut
  - So we can determine the min cut in O(m) additional time
  - O(mf) + O(m) = O(mf)

Reductions

Algorithm for min-cut

- Imagine that I presented you with a new algorithm to determine min-cut
  - Everybody uses max-flow to determine min-cut, but imagine it anyway

- What could you tell me about that algorithm?
  - About it's running time?

Max-flow vs. min-cut

- These two problems are “equivalent”
  - Specifically, if you can solve one, you can solve the other
- Alternatively, we can say that one problem *reduces* to the other
  - The problem of finding min-cut reduces to the problem of finding max-flow (plus a polynomial time conversion)

Reduction

- A reduction is a transformation of one problem into another problem
  - Min-cut is reducible to max-flow because we can use max-flow to solve min-cut
  - Formally, problem A is reducible to problem B if we can use a solution to B to solve A
- We note that the reduction happens in polynomial time
- And signify it with a $\leq_p$

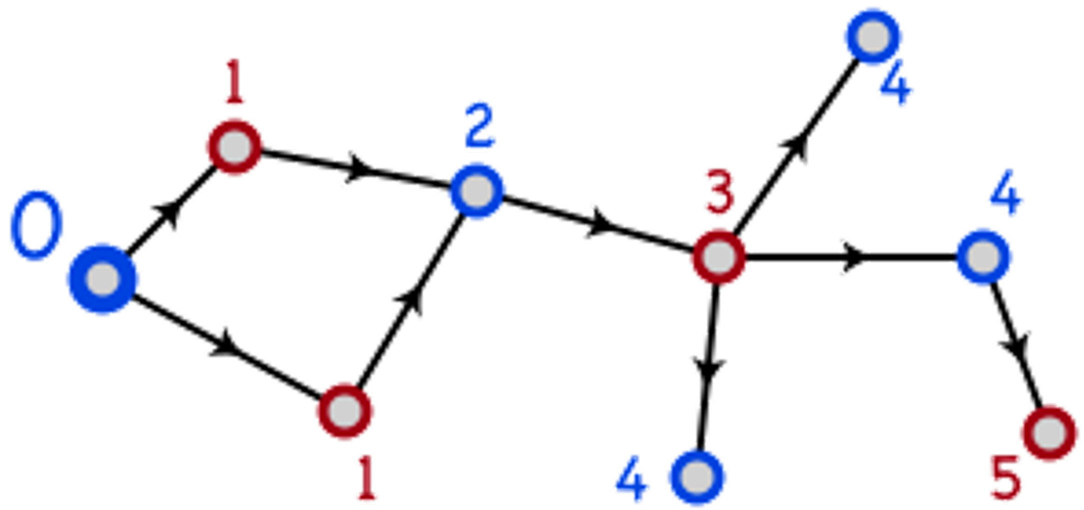
Reducing both ways

- We know that:
  - Min-cut $\leq_P$ max-flow
  - Max-flow $\leq_P$ min-cut
- Because they reduce both ways, they are *polynomial-time equivalent*
- Often times you can’t directly compare algorithms
  - So you show that they reduce both ways

Bipartite Graphs

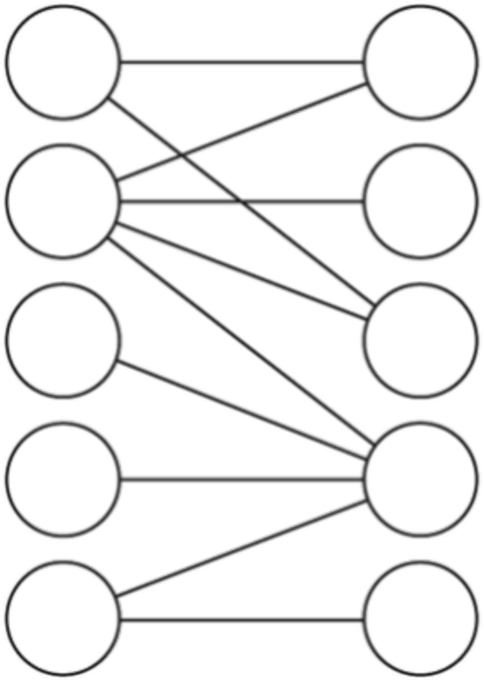
Bipartite Graphs

- A graph is *bipartite* if node set V can be split into sets X and Y such that every edge has one end in X and one end in Y
  - X and Y are typically colored red and blue
    - Or Boolean true/false



Notes and assumptions

- We assume the graph is connected
  - Otherwise we will only look at each connected component individually
- A triangle cannot be bipartite
  - In fact, any graph with an odd length cycle cannot be bipartite



Bipartite Determination Algorithm

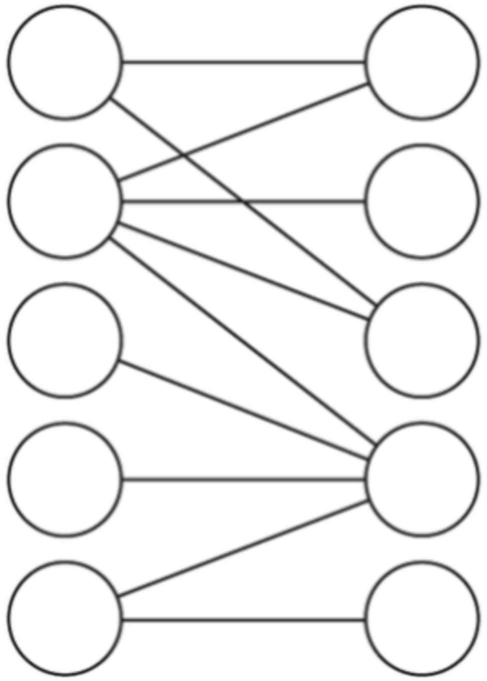
- Pick a starting vertex, color it red
- Color all adjacent nodes blue
  - And all nodes adjacent to that red
  - Etc.
- If you ever try coloring a red node blue, or a blue node red, then the graph is not bipartite

- Does this algorithm sound familiar?

Bipartite Matching

Bipartite Matching

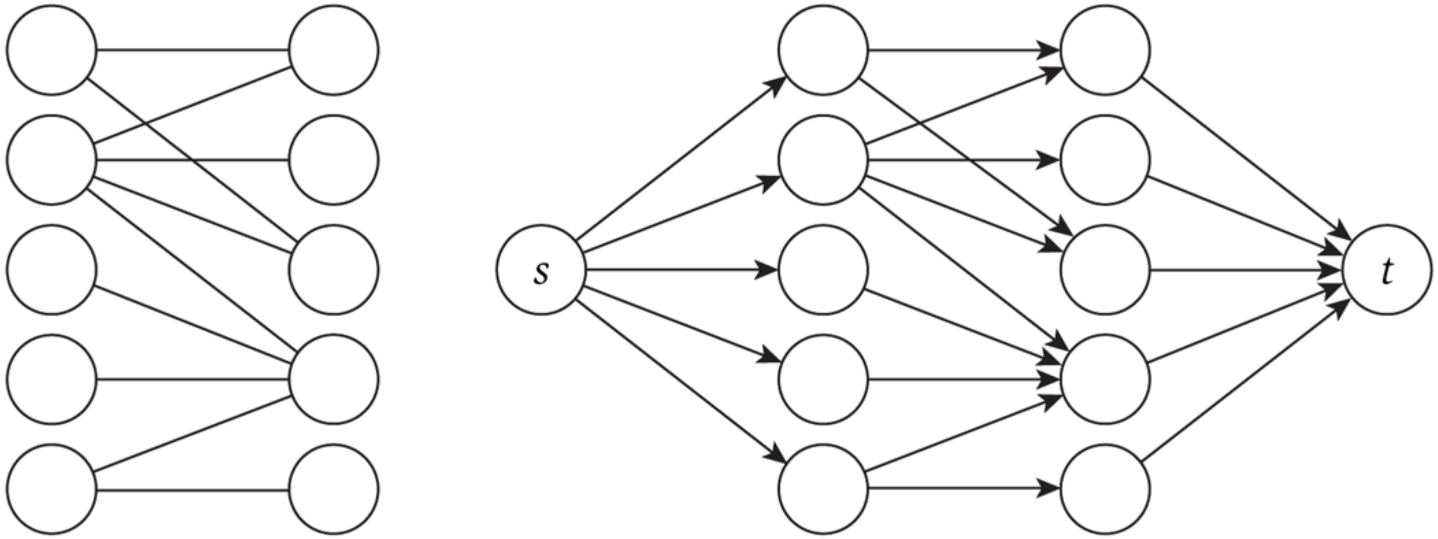
- Given a bipartite graph G, can we find a matching M in G such that $M \subseteq E$ and each node appears on exactly one of M
  - In other words, find a subset of edges that connect every node on the left to one (and only one!) node on the right
  - Since the graph is bipartite, all edges connect one on the left with one on the right



Reduction!

- To solve this, we reduce it to a maximal flow problem by creating a graph G':
  - Direct all edges from the left to the right
  - Add a source node, with edges to every node on the left side
  - Add a terminus node, with edges to every node on the right side
- Compute maximal flow!
  - The maximal flow in G' is the maximum matching in G

Reduction, diagrammatically



Why does this work?

- Each node on the left can be in at most one matching
    - This is enforced by the edge of capacity one leading into it
- Likewise for each node on the right
- The bottleneck will be how it flows across the bipartite "barrier"

Reduction details

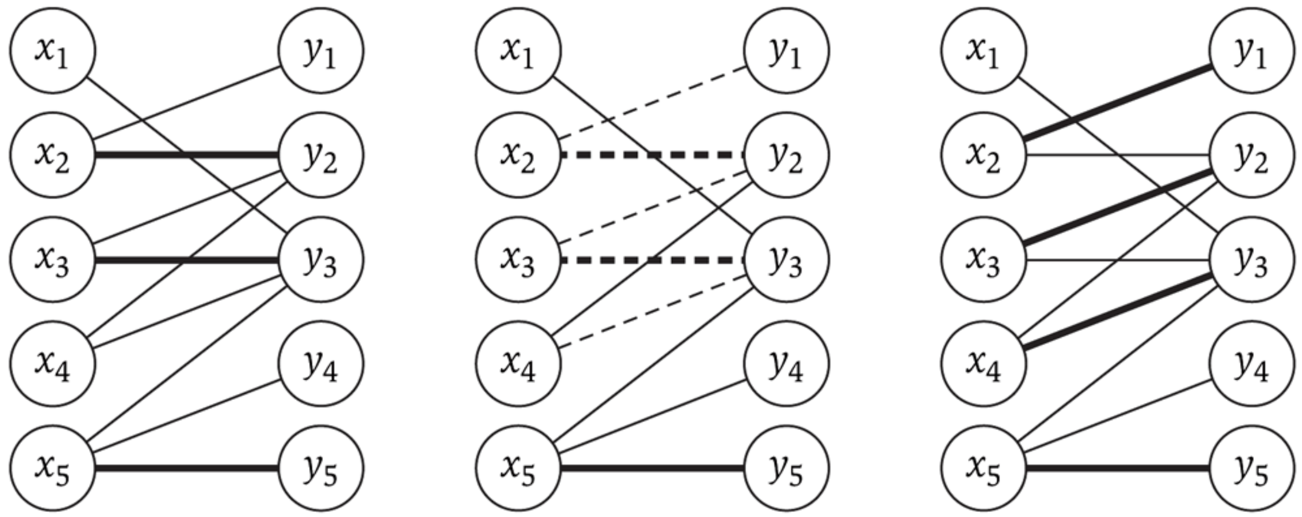
- We have transformed (in polynomial time) a bipartite matching problem into a maximal flow problem
- Specifically, bipartite-matching $\leq_p$ max-flow
  - Because we can transform bipartite matching to max-flow in polynomial time
- But is it the case that max-flow $\leq_p$ bipartite-matching?
  - Not so much: a solution to bipartite matching does not help us with a non-bipartite graph

Running time

- Max flow runs in O(E*f)
  - But the max flow is (at most) n/2
    - If every node in the graph has flow through it, then there are n/2 units of flow moving through the graph
  - So the running time is equivalent to O(E*n)

Imperfect bipartite matchings

- These exist, and the algorithm may produce them, depending on the graph
  - The following shows an augmenting path (in the middle) used to achieve the maximal flow on the right



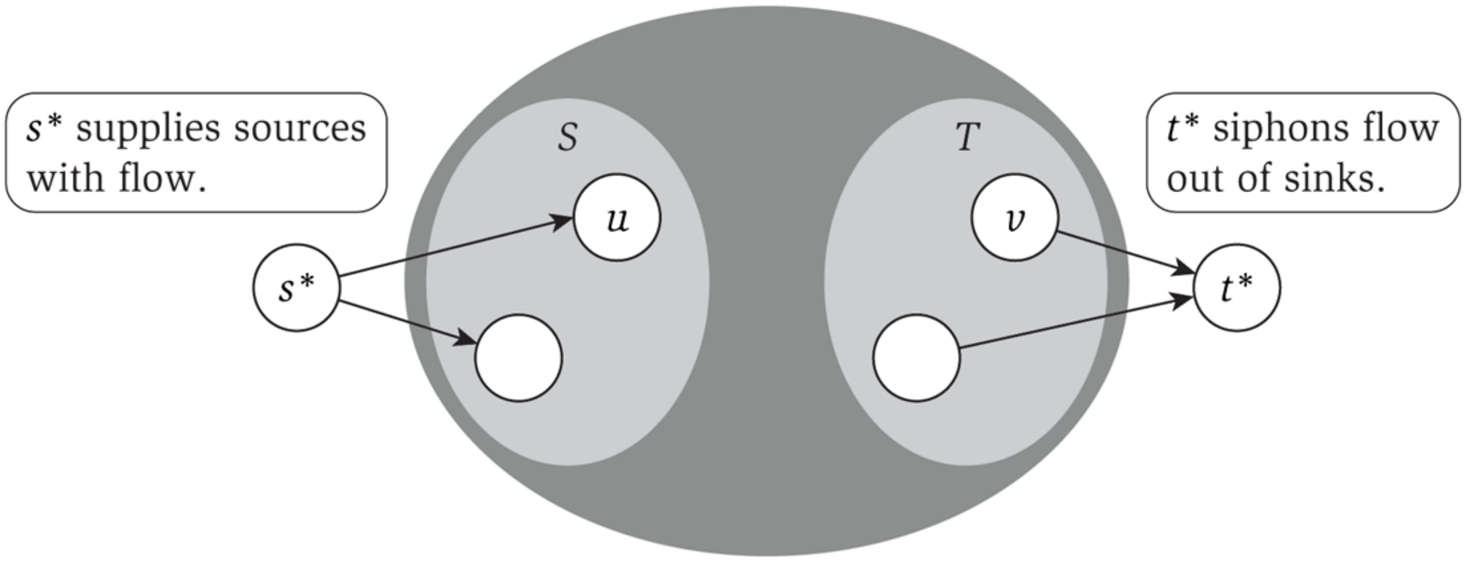
Max-flow variations

Finding a Circulation

- Real world applications don't have just one source and sink
  - Instead there are multiple ones: power production / consumption, etc.
- We designate a set S to be all the nodes that are sources
  - We can also view them has having negative demand
- Likewise, we designate a set T to be all the nodes that are sinks
  - They have positive demand
- Networks with multiple sources and sinks (modeled using demand) are called *circulation networks*

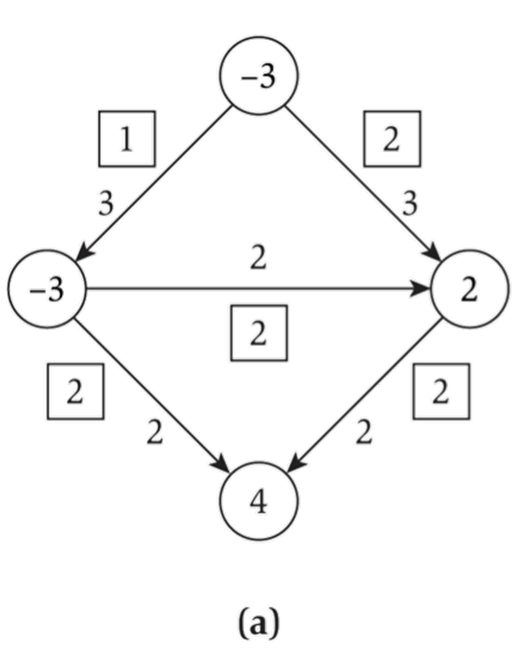
Reduction to max-flow

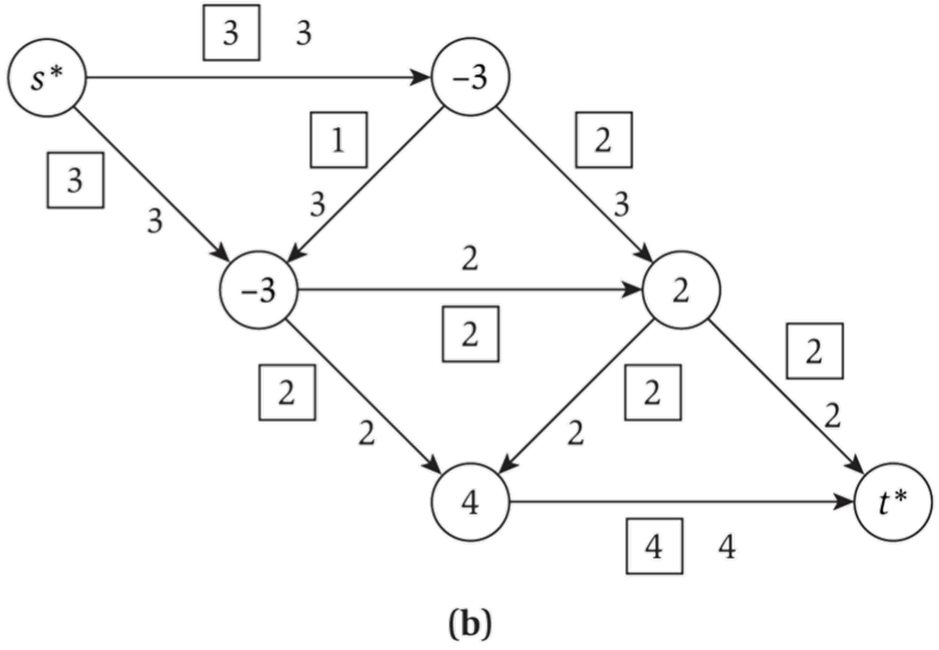
- With a few modifications, we can make this a max-flow problem:
  - Create a ‘super source’ s* with edges to each node in S
    - The capacity of that edge is the size of the source of the node in S
  - Likewise with the set T


s^* supplies sources with flow.
S
u
s^*
T
v
t^*
t^* siphons flow out of sinks.

Conversion example

- Converting a graph with multiple sources and sinks to a single-source-single-sink max-flow problem:





Circulation notes

- A circulation problem is aiming for *feasibility*, not max flow
  - But we use max flow to solve it
- We set each edge from the super-source to each individual source to be the absolute value as the individual source's demand
- Max-flow is then run
- If the total amount leaving the single-source is the SAME as the capacity of each outgoing edge, then the circulation is feasible

Edge lower bounds

- So far, we have considered only the capacity of an edge: the upper bound on the flow
- We also want to consider a lower bound on the flow on an edge
  - i.e. forcing a certain amount of flow through an edge
- We will reduce this to a circulation problem
  - Which can then be reduced to a max-flow problem

Handling lower bounds

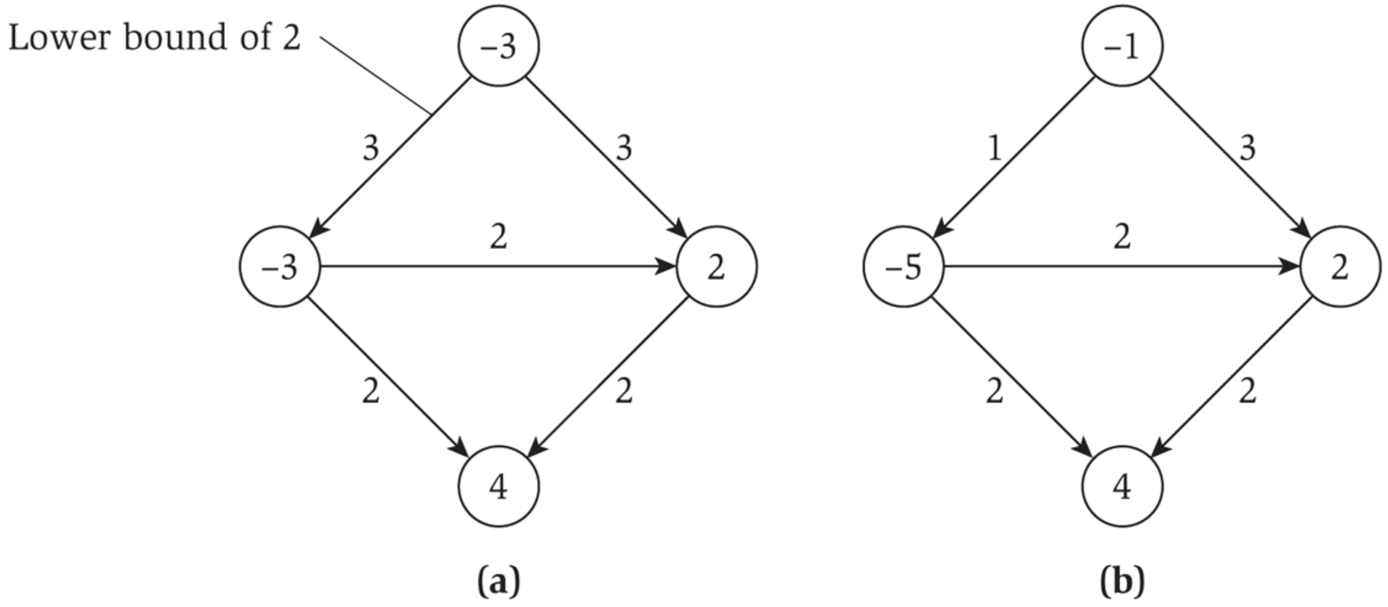
- A lower bound forces flow across an edge
  - Which increases demand at the start of the edge (to compensate for the flow across the edge)
  - And decreases demand at the terminus of the edge (as some flow is fulfilling the demand)

Solving a flow with lower bounds

- Given a circulation network G, construct a new graph G' such that for each edge e from u to v with a lower bound l_e:
  - We decrease the capacity on that edge by l_e
    - As that is the flow that is moving through the edge
  - We increase the demand at u by l_e
  - We decrease the demand at v by l_e
- Then solve G' as a circulation problem
  - i.e. add a super-sink and super-terminus, and solve as a max-flow problem

Eliminating a lower bound

- Diagrammatically…



Summary

What did we learn?

- Max-flow / min-cut
  - The problems, relationship between them, etc.
  - Ford-Fulkerson algorithm and related proofs that this approach is optimal
- Bi-partite matching
  - First example of a reduction. Use the algorithm from one problem to solve another problem.
- More reductions
  - Solving variations of max-flow by converting the problem into an instance of "normal" max-flow.