Overview

- Network Flow
  - Flows, constraints, augmenting paths, residual graphs
  - Ford-Fulkerson
  - S-t cuts, cut capacities, max flow min-cut theorem
  - Reductions
- NP-Completeness
  - Reductions
  - Problems to Know
  - Proofs

Network Flow

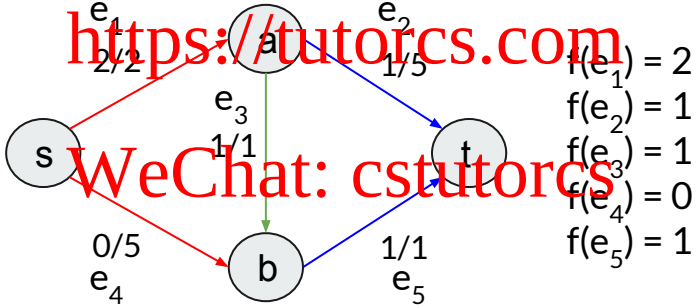
Flow

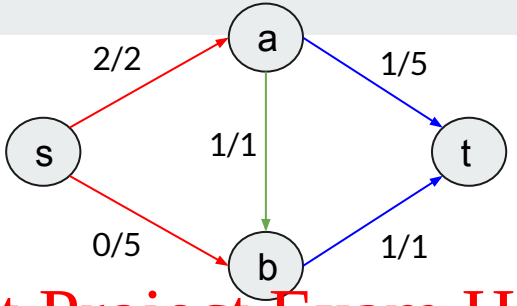
- Flow network: G = (V, E) with capacity c_e for each edge e
  - has a single source node s and single sink node t

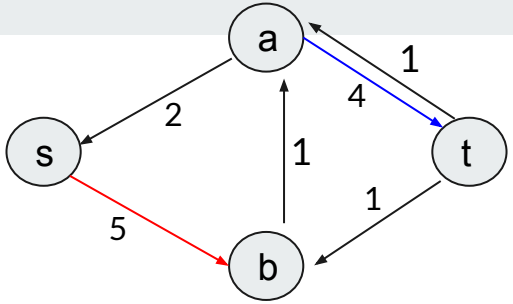
- Flow: a s-t flow is a mapping from the set of edges to non-negative real numbers
  - Constraints:
    - Capacity: $0 \leq f(e) \leq c_e$ for each edge e
    - Conservation: for all internal nodes, the flow going out must equal the flow coming in

Flow

- Value: value v(f) of flow f is the
  - flow coming **out** of **s (source)** = also the flow **entering** the **t (sink)**

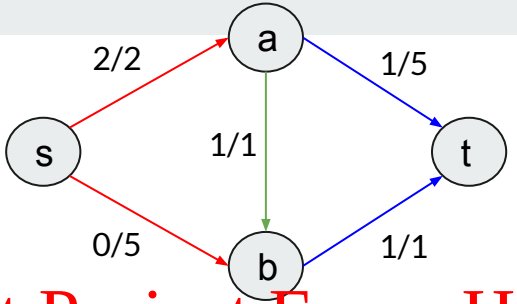


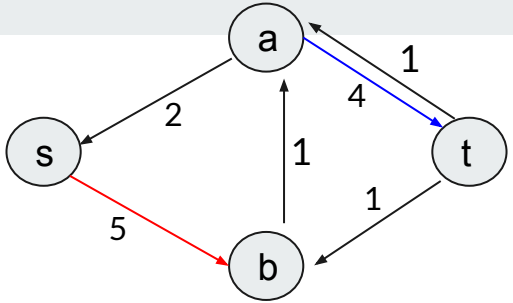




Residual Graphs

- Residual graph: G_f for flow network G and flow f where each directed edge in G_f has a value which denotes the amount of leftover flow that could be pushed in that direction
  - Forward Edges: Have value c_e - f(e)
  - Backwards Edges: Have value f(e)
- Augmenting path: path from s to t in the residual graph



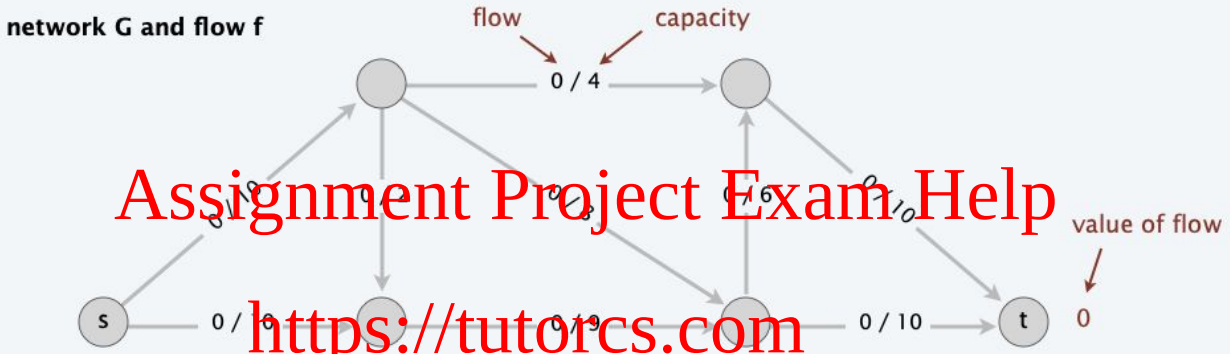


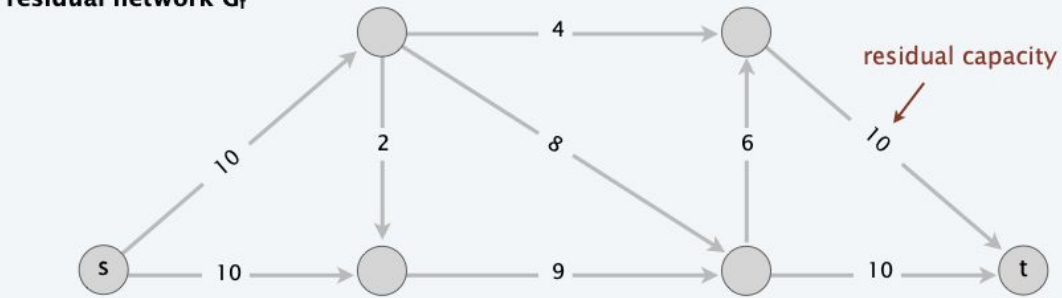
Residual Graphs

Ford-Fulkerson Algorithm

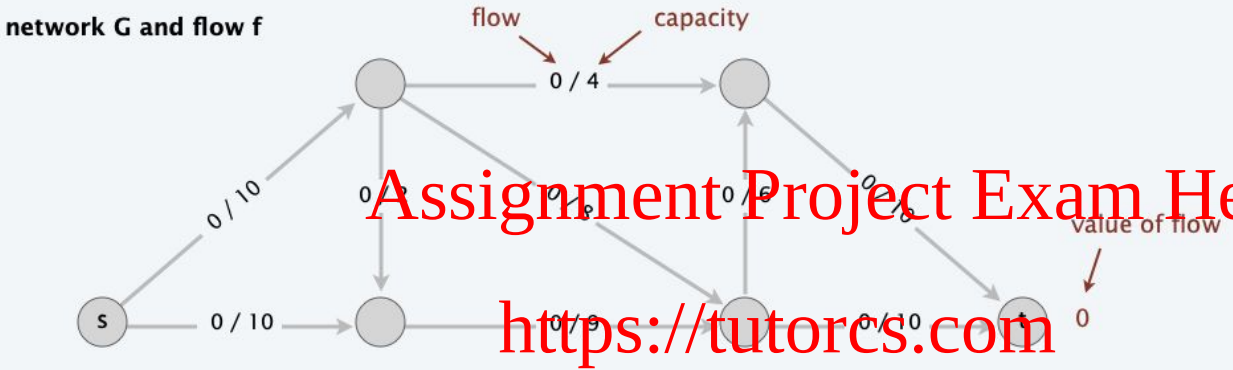
- Flow is 0 initially for all edges
- While there is an augmenting path:
  - Use the path to update $f(e)$ for each edge e in the path
  - Update the residual graph
- Return final flow (max flow)
- Runtime 🤦‍♂️?
  - Runtime: with integer capacities, O(mC) - m is # of edges in G, C is max-flow
  - “Pseudopolynomial” - A polynomial bound on the runtime based on the magnitude of the inputs

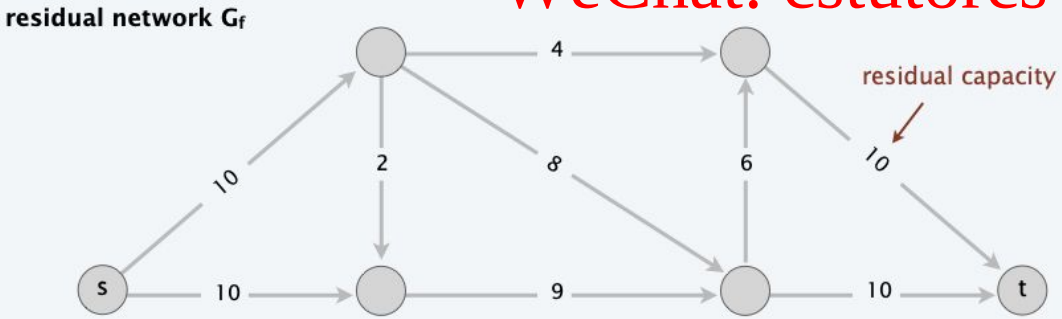
Ford–Fulkerson algorithm demo





Ford–Fulkerson algorithm demo

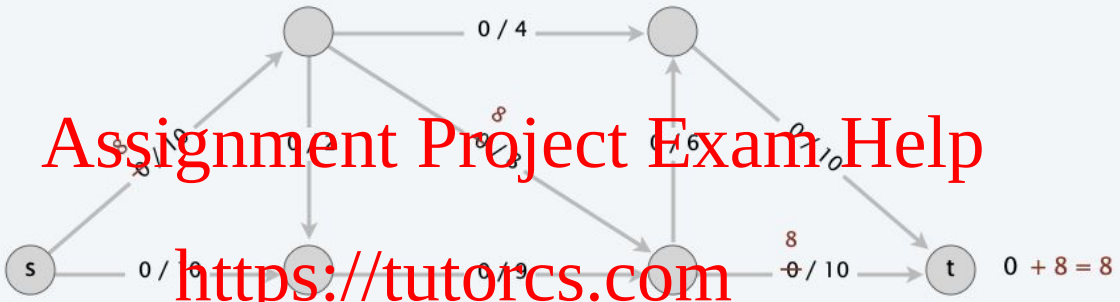




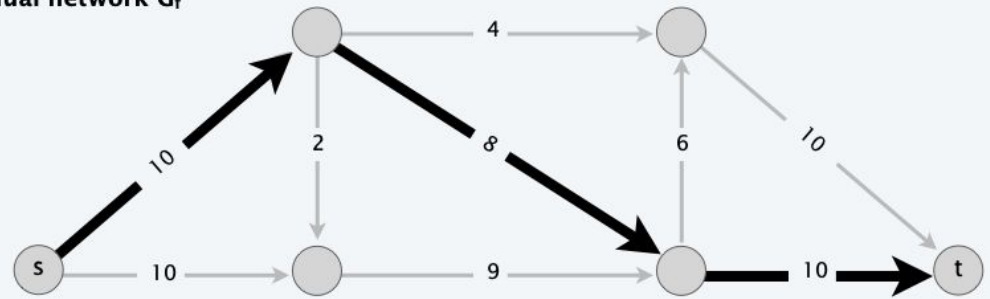
- How do I pick a path to send flow?
- How much flow do I push along a path?

Ford–Fulkerson algorithm demo

network G and flow f

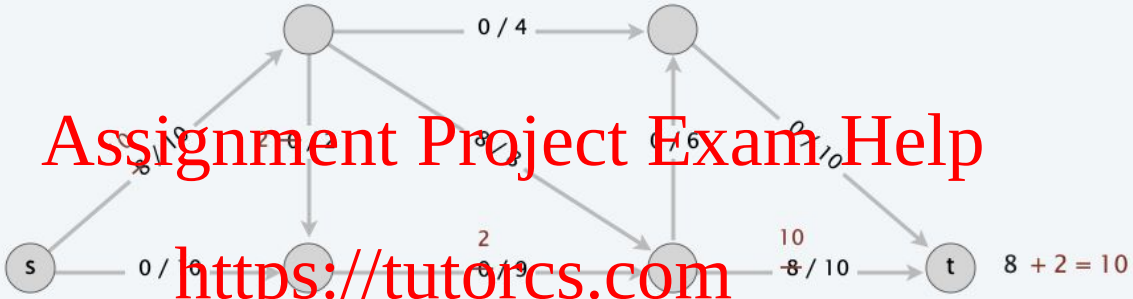


residual network G_f

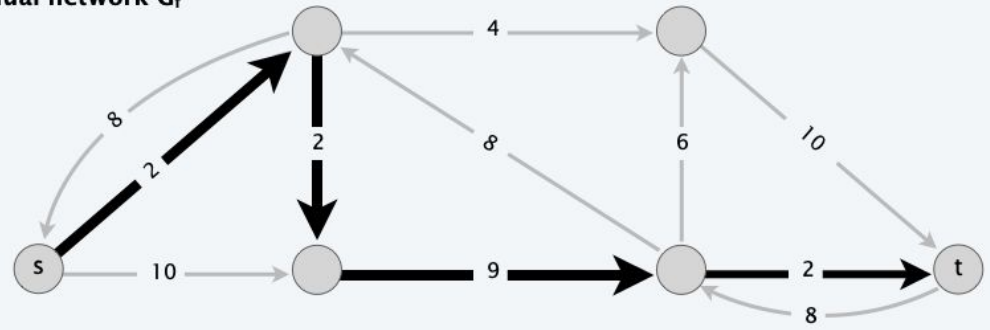


Ford–Fulkerson algorithm demo

network G and flow f

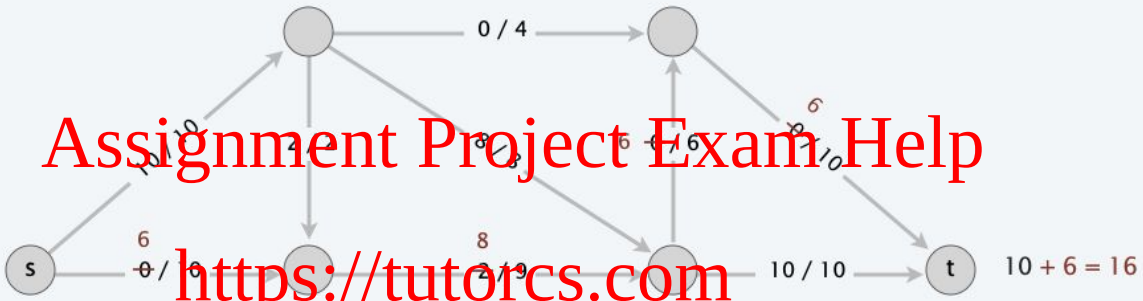


residual network G_f

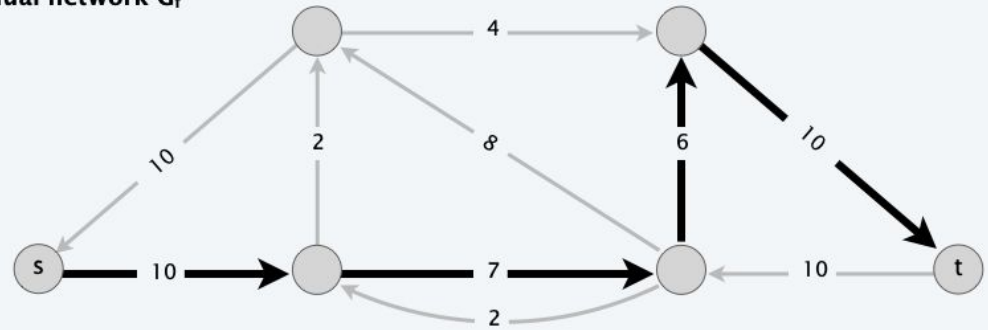


Ford–Fulkerson algorithm demo

network G and flow f

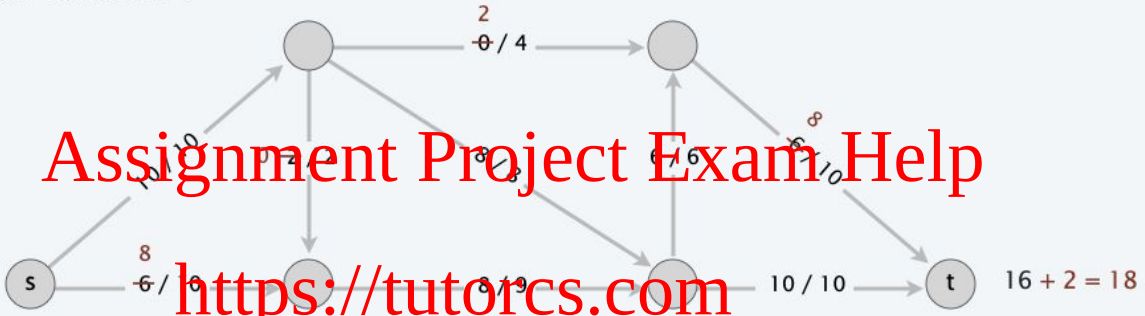


residual network G_f

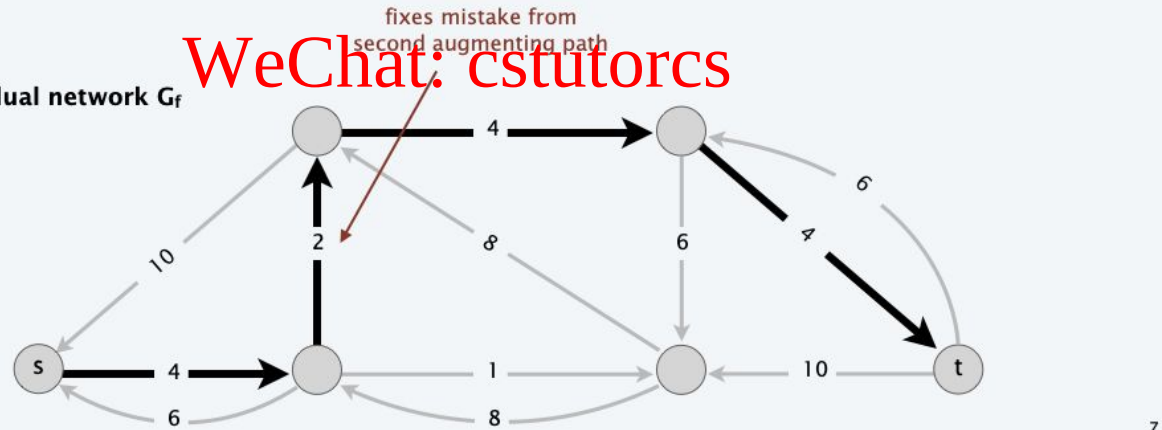


Ford–Fulkerson algorithm demo

network G and flow f

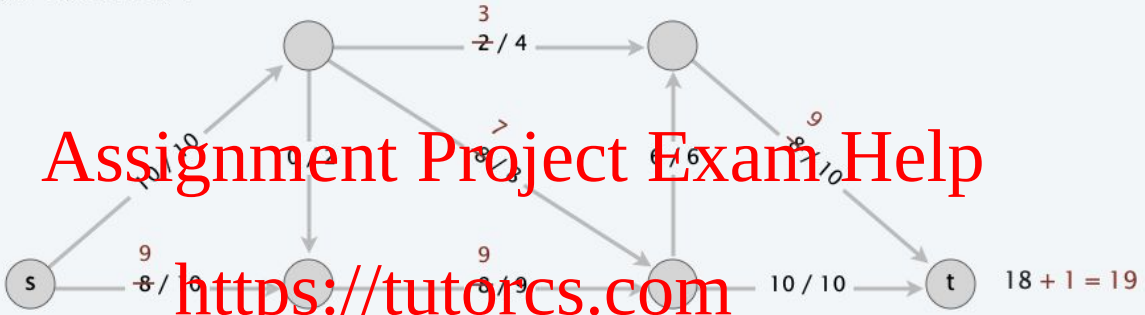


residual network G_f

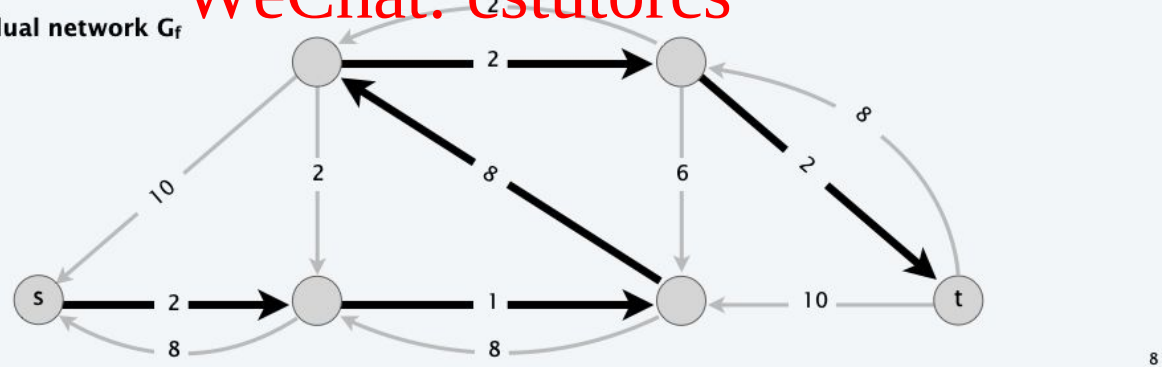


Ford–Fulkerson algorithm demo

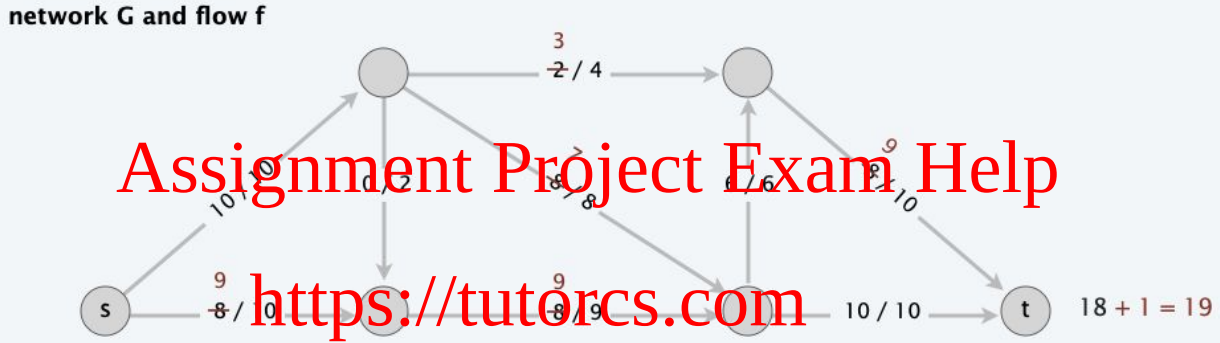
network G and flow f



residual network G_f



Ford–Fulkerson algorithm demo

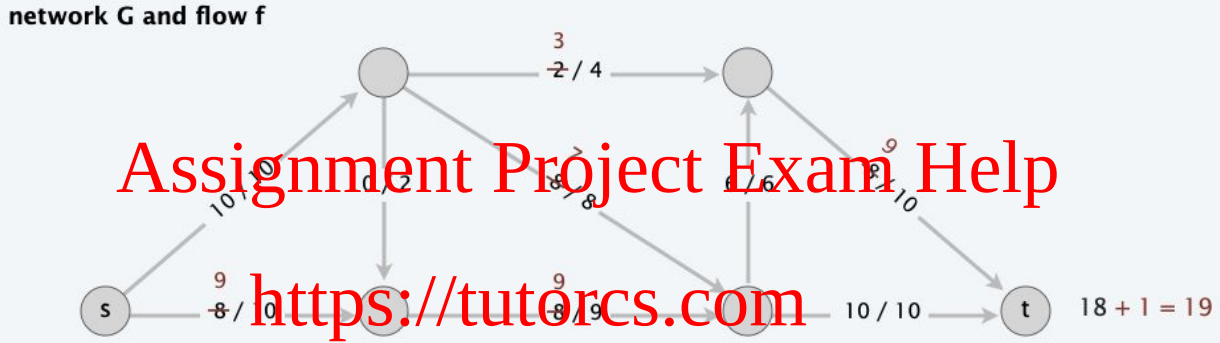


1

Where is the min-cut?

- Can I tell from this graph?
- Is it easy?

Ford–Fulkerson algorithm demo

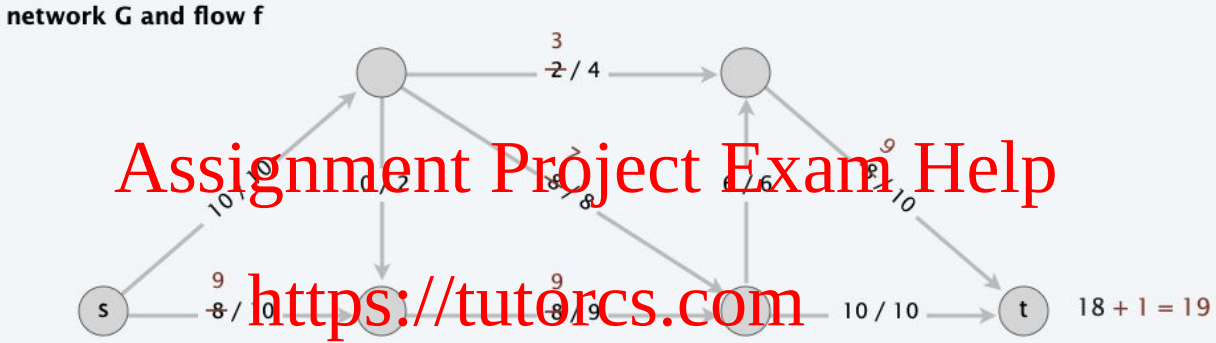


1

Where is the min-cut?

- Can I tell from this graph?
- Is it easy? ❌

Ford–Fulkerson algorithm demo

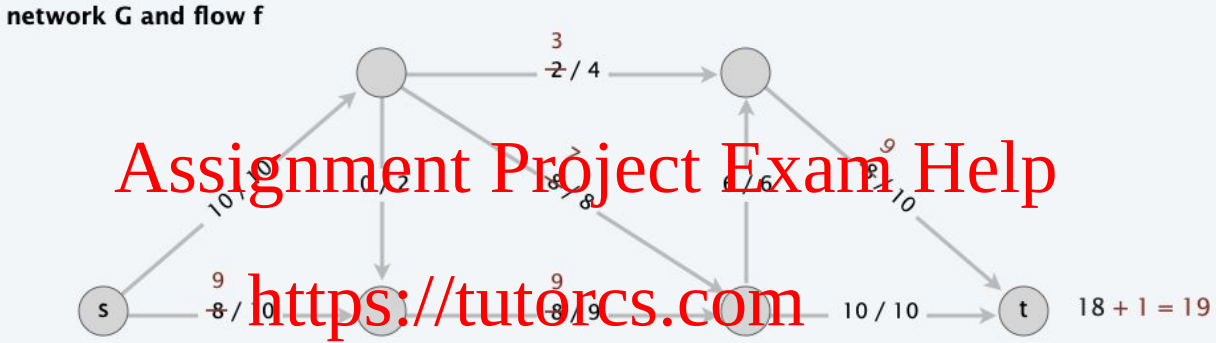


2

Before constructing the residual graph, what is the **capacity** of this min-cut?

- Do I have enough information?

Ford–Fulkerson algorithm demo



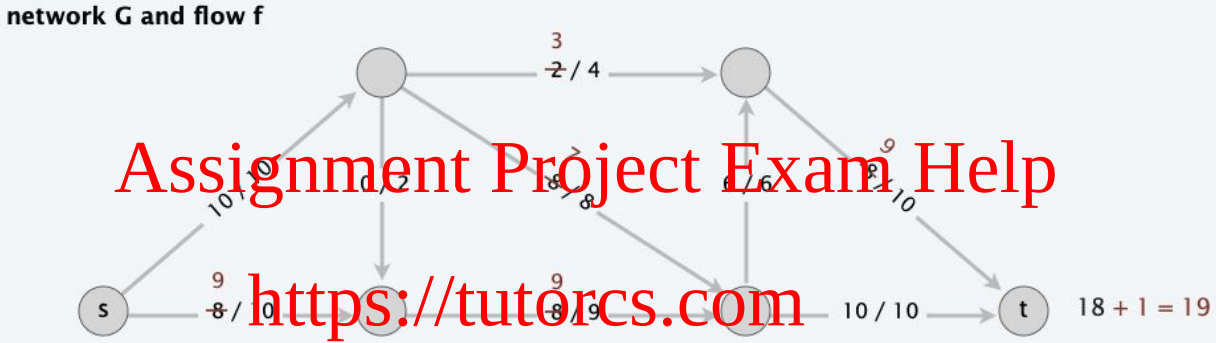
3

Can there ever not be a min-cut?

- By well-ordering principle that **has** to be a minimum

Can the min-cut not be equal to the max-flow? Explain without just citing the theorem)?

Ford–Fulkerson algorithm demo



4

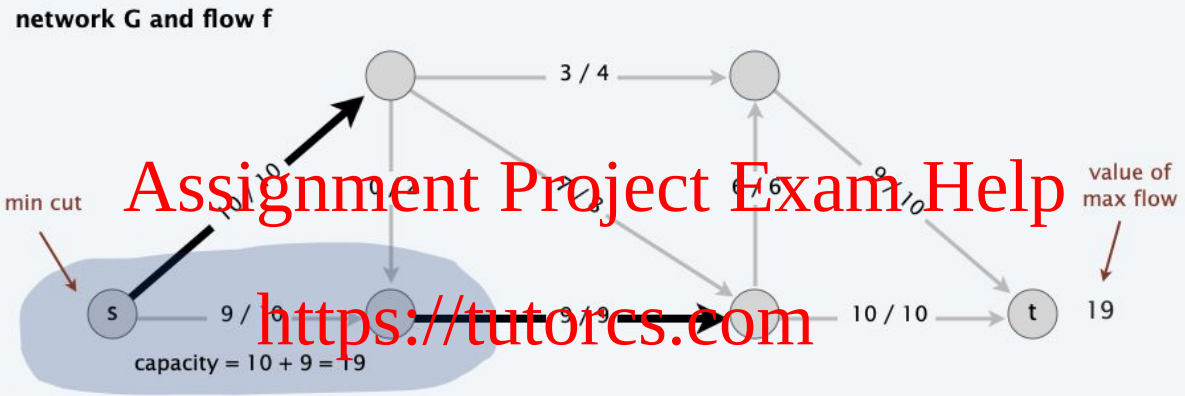
Are capacities of cuts (usually) > or < than values of flows?

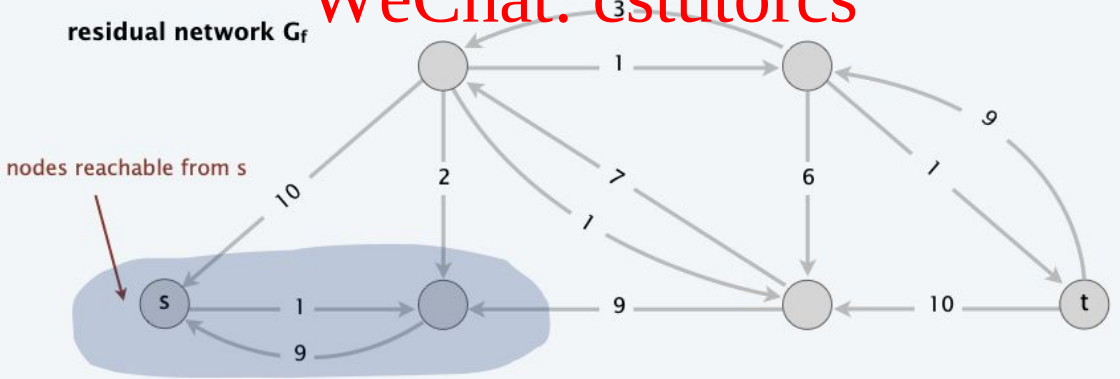
a. >
b. <

Can a cut have capacity less than the max-flow?

- Flow goes from s to t
- Cut is amount of flow allowed from s to t

Ford–Fulkerson algorithm demo





Cuts

- s-t cut: partition (A,B) of V where s has to be in A and t has to be in B
- Capacity of s-t cut: sum of capacities leaving A

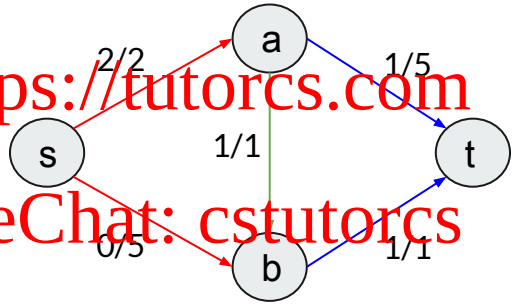
Max-flow min-cut Theorem:

- Max flow (no augmenting path) = an s-t cut (A', B') that has the lowest capacity and v(f) = c(A', B')
- To find the min-cut, use BFS/DFS starting on s on the residual graph with the max flow
- **Min cut does not always have to be unique!**
- For e.g., in the homework, we found a min cut by doing a BFS from t on the reversed residual graph.

Prelim Tip

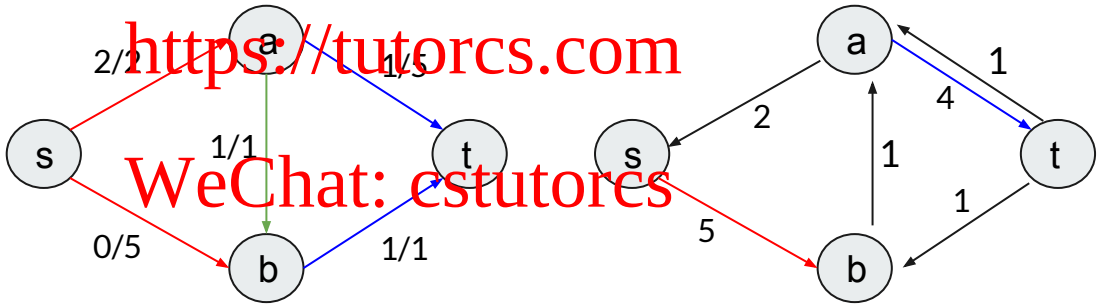
- Practice calculating residual graphs, finding augmenting paths, augmenting a residual graph, finding min-cuts, etc.
- If you're able to solve these quickly, you can get through the first half of the prelim fast!

Example Problem



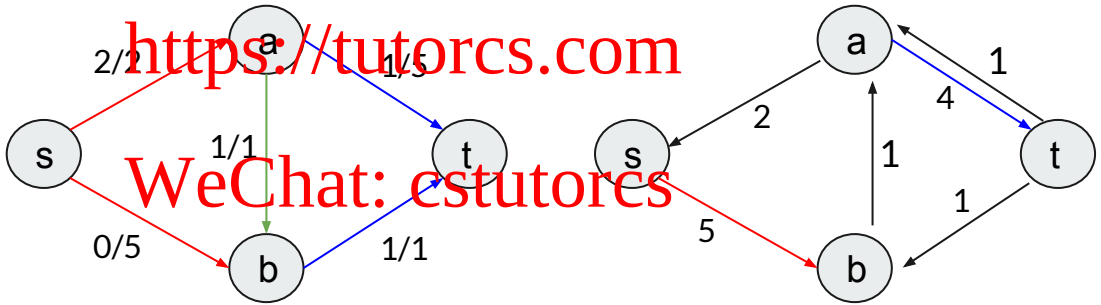
- What is the value of the flow? Is this a max (s,t) flow?
- Draw the residual graph and determine if there's an augmenting path

Example Problem



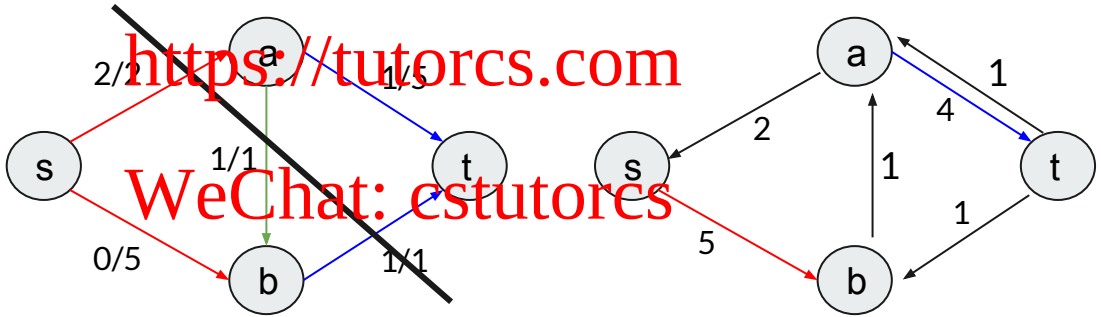
- What is the value of the flow? (2)
- Is this a max (s,t) flow? (no)
- Draw the residual graph and determine if there's an augmenting path (s → b → a → t)

Example Problem



- Find min s-t cut

Example Problem



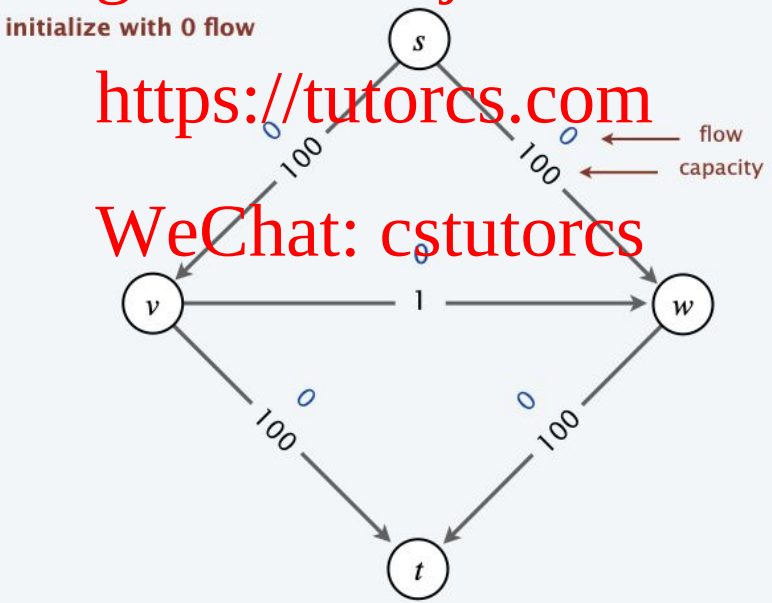
- Find min s-t cut

Why do we enforce, integer value flows 🙋‍♂️?

Example of Pseudo polynomial Runtime!!!!!

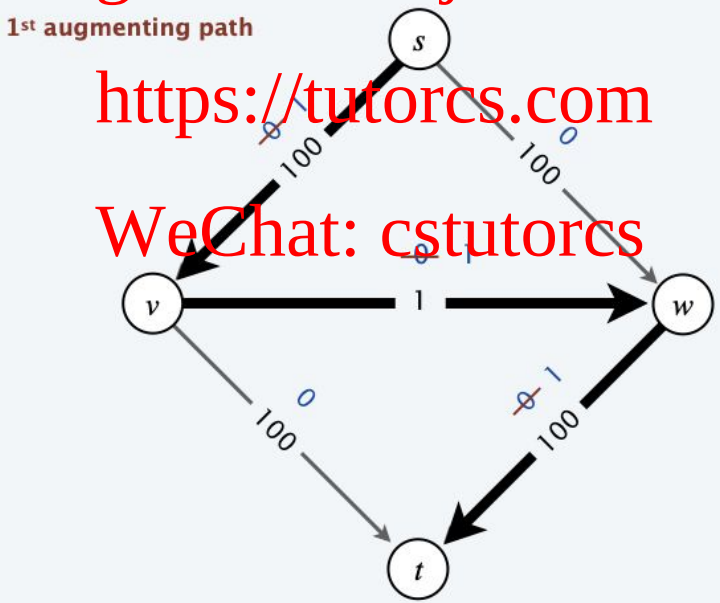
Ford–Fulkerson algorithm: exponential-time example

Bad news. Number of augmenting paths can be exponential in input size.



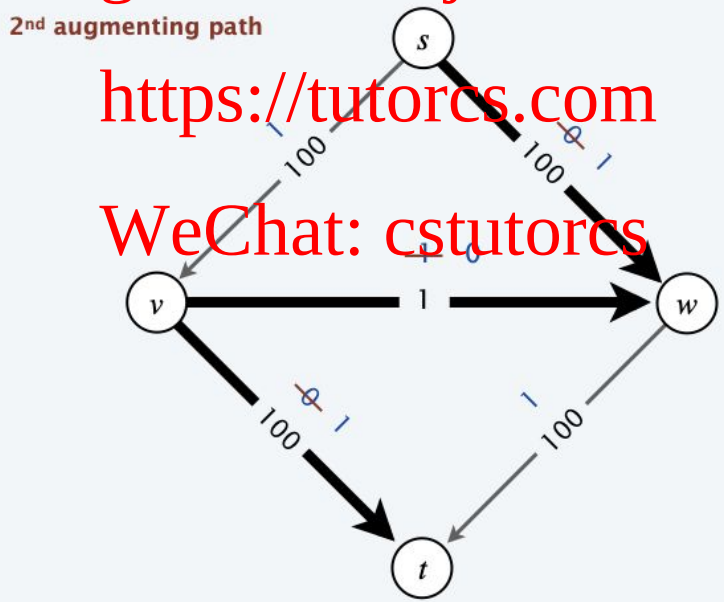
Ford–Fulkerson algorithm: exponential-time example

Bad news. Number of augmenting paths can be exponential in input size.



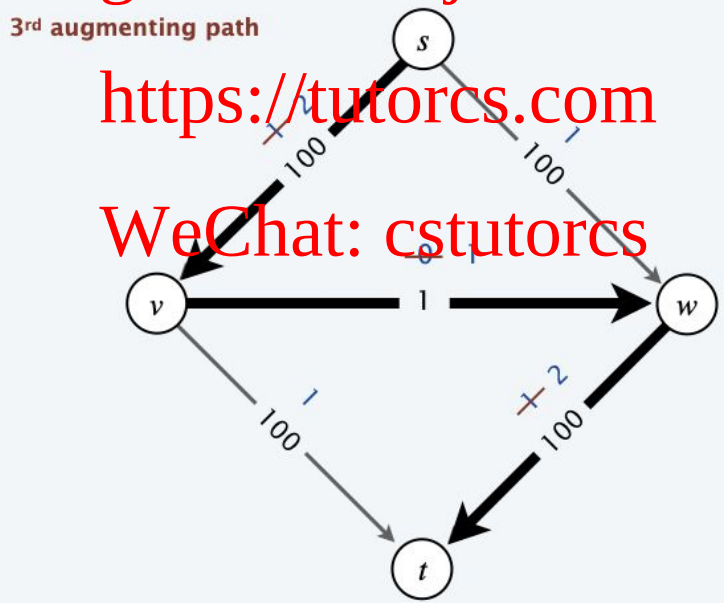
Ford–Fulkerson algorithm: exponential-time example

Bad news. Number of augmenting paths can be exponential in input size.



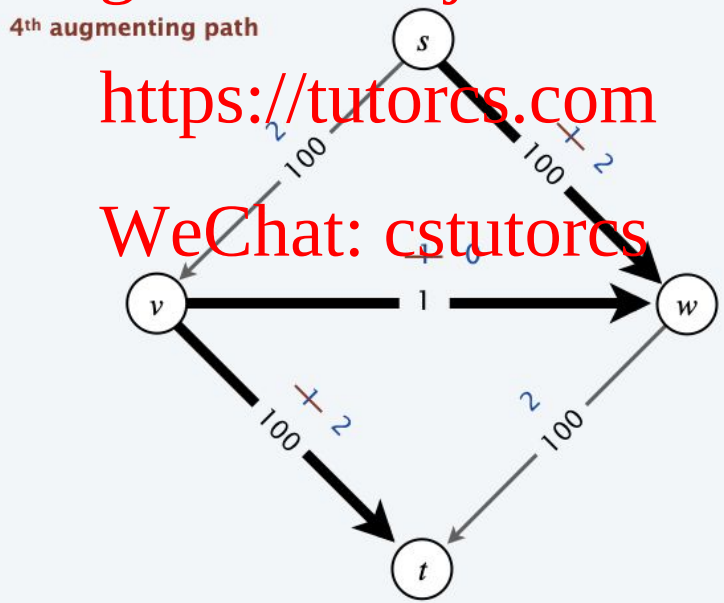
Ford–Fulkerson algorithm: exponential-time example

Bad news. Number of augmenting paths can be exponential in input size.



Ford–Fulkerson algorithm: exponential-time example

Bad news. Number of augmenting paths can be exponential in input size.

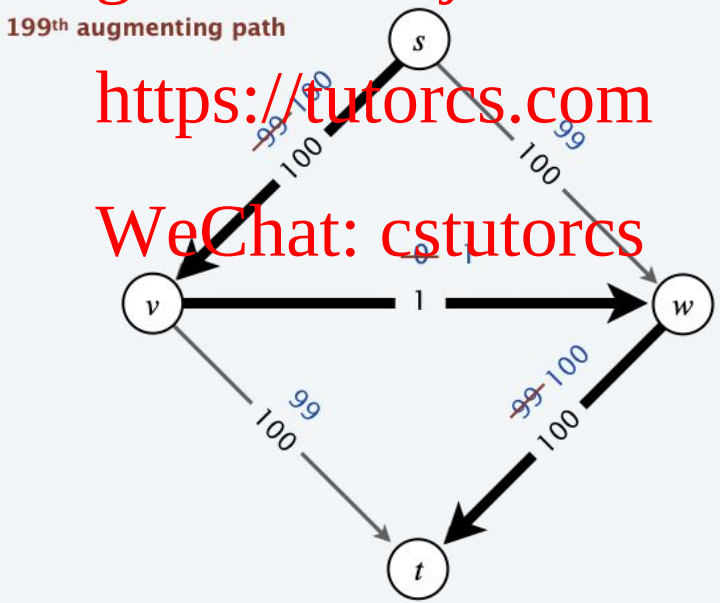


Ford–Fulkerson algorithm: exponential-time example

Bad news. Number of augmenting paths can be exponential in input size.

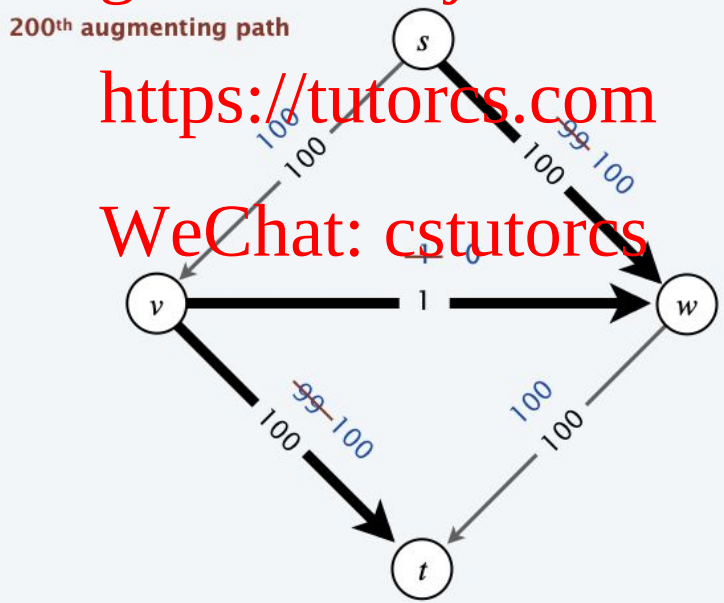
Ford–Fulkerson algorithm: exponential-time example

Bad news. Number of augmenting paths can be exponential in input size.



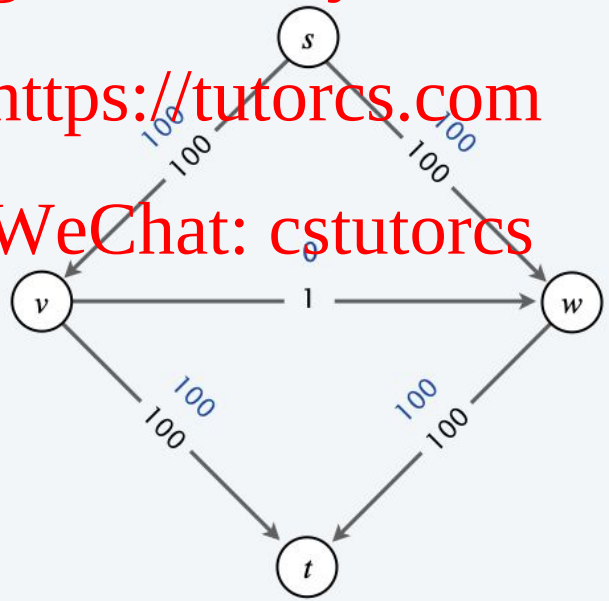
Ford–Fulkerson algorithm: exponential-time example

Bad news. Number of augmenting paths can be exponential in input size.



Ford–Fulkerson algorithm: exponential-time example

Bad news. Number of augmenting paths can be exponential in input size.

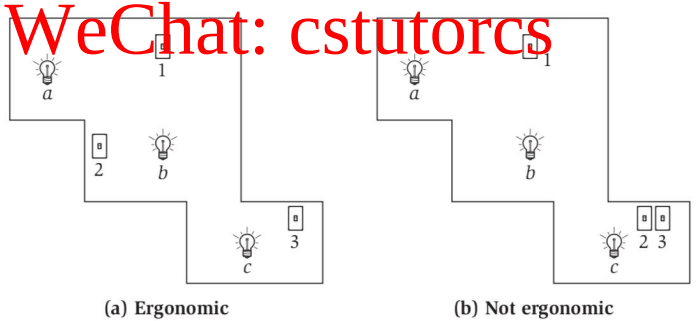


Reductions

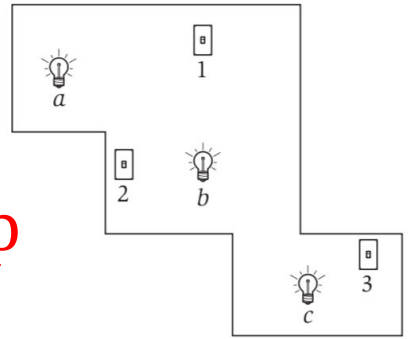
- Feasibility Reductions
  - Bipartite Matching
  - Interview scheduling
  - Hospital to patients
- Min cut (optimizing a certain quality) Reductions
  - Computer applications and costs
  - Project selection

Matching Problem #1 (ch. 7, ex. 6)

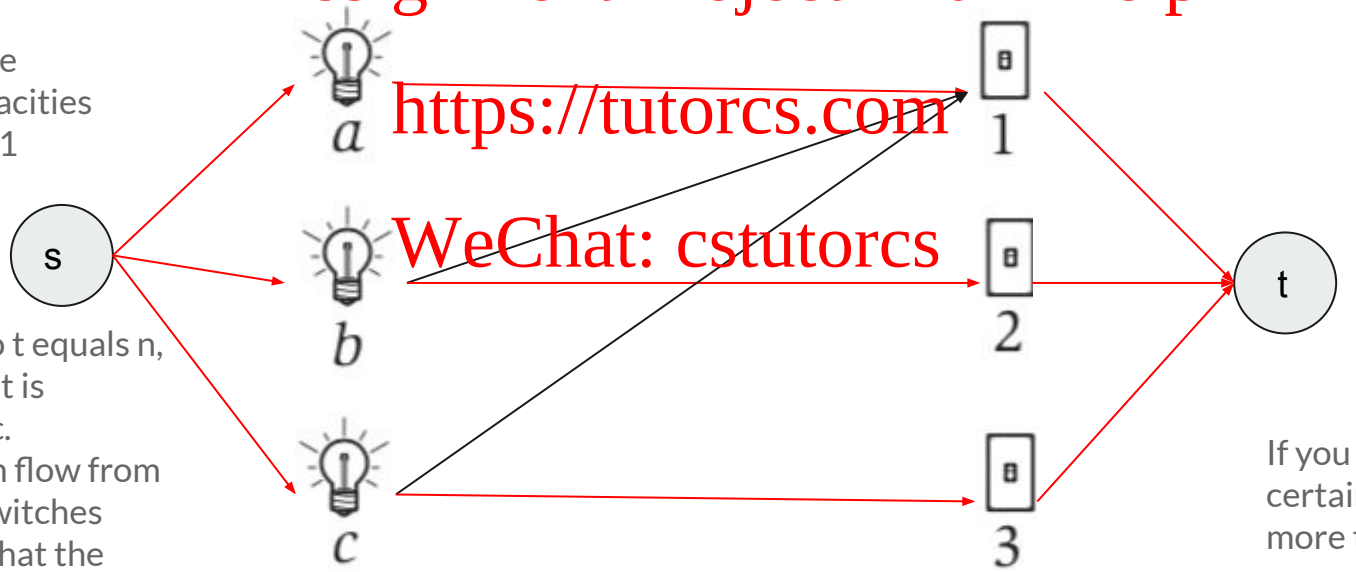
- Given locations n light switches and n light bulbs
- Each light switch can be paired to one light bulb
- Determine if there exists an “ergonomic” set of pairings switches and bulbs (every bulb can be seen from its switch)



Matching Problem #1 (ch. 7, ex. 6)



Edge capacities are 1



- If flow into t equals n, then layout is ergonomic.
- Edges with flow from bulbs to switches indicate what the ergonomic set of pairings is

If you were allowed to use certain bulbs or switches more than once, increase corresponding edge capacities

Matching Problem #2 (Ch. 7, s.e. 2)

Given

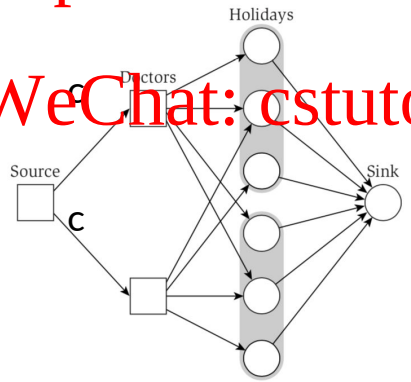
- k *vacation periods*, where each period is a set of *vacation days* corresponding to a particular holiday
- n doctors, where each doctor has a set of vacation days that they are available to work

Goal

- Determine if there is an assignment of doctors to work on vacation days, with the following constraints
  1. Each doctors can be assigned to work at most c vacation days
  2. No doctor can work more than one vacation day in the same vacation period

Matching Problem #2 (Ch. 7, s.e. 2)

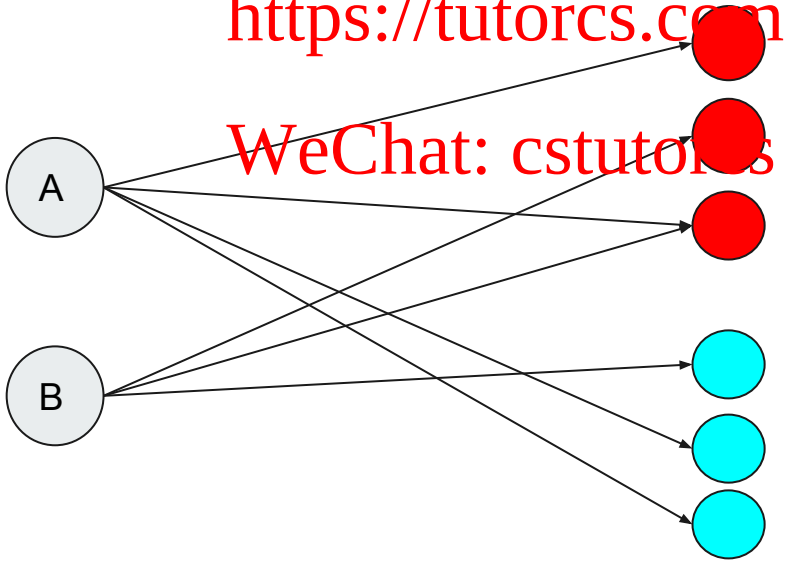
- First Approach: Set up bipartite graph, use edge capacities to satisfy constraint #1 (at most c days per doctor)



- Problem: How do we enforce at most 1 vacation day per period for each doctor?

Matching Problem #2 (Ch. 7, s.e. 2)

- **In general**: We want to assign individual flow limits from left nodes to specific sets of right nodes

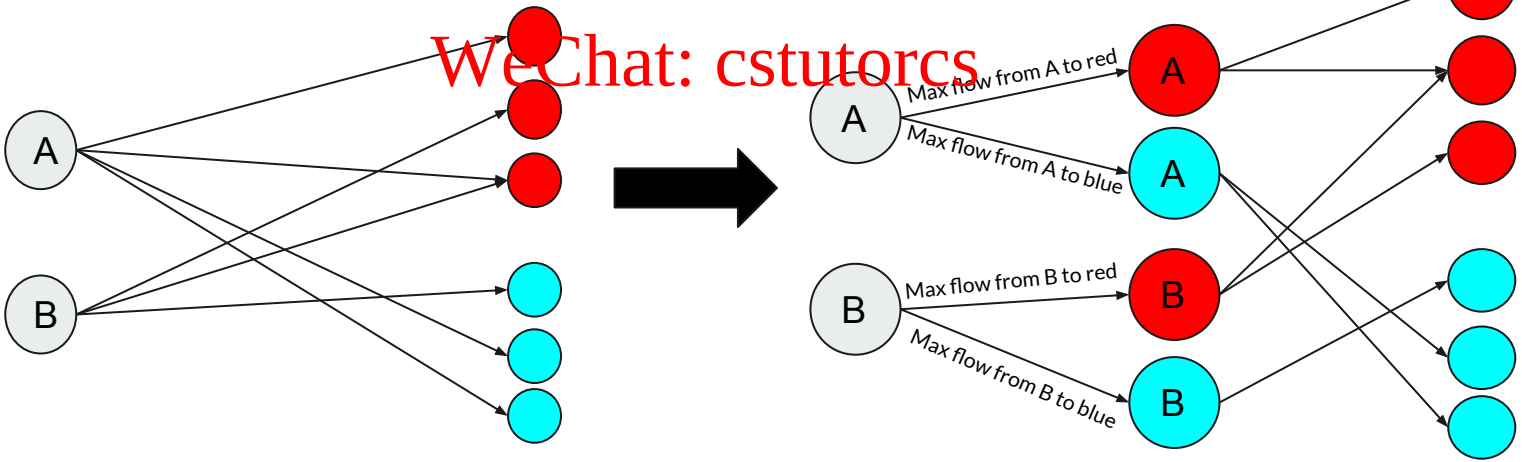


Ex. We want to assign individual limits to
- A → Red
- A → Blue
- B → Red
- B → Blue

(s, t, edge capacities not shown)

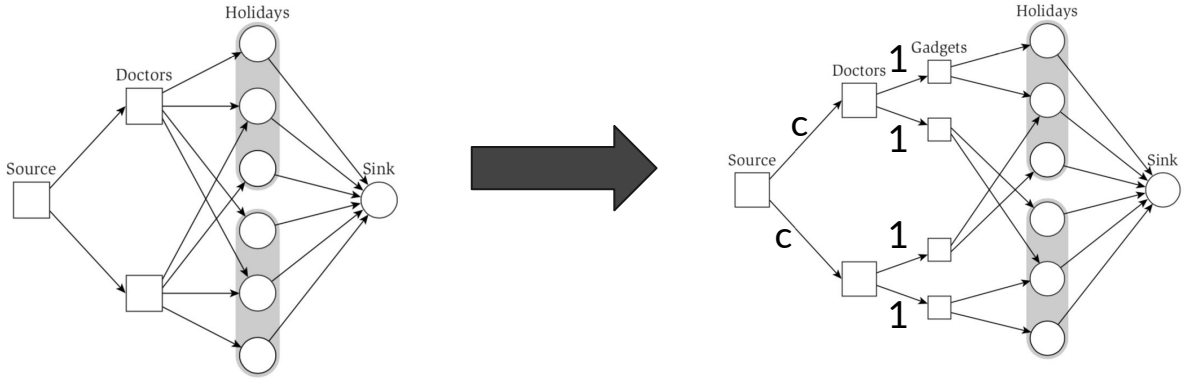
Matching Problem #2 (Ch. 7, s.e. 2)

- **In general**: We want to assign individual flow limits from left nodes to specific sets of right nodes
  - Add a "gadget" for each (left node, subset of right nodes) pair
  - Edge capacity going into gadget is flow limit from the corresponding left node to set of right nodes



Matching Problem #2 (Ch. 7, s.e. 2)

- Set up bipartite graph
- Use [source → doctor] edge capacities to limit each doctor to c days in total (restraint 1)
- Add layer of "gadgets", use [doctor → gadget] capacities limits from each doctor to 1 day per vacation period (constraint #2)
- Valid matching exists if Max Flow = # of vacation days



Conversion to Matching:

Flow from gadget (i,j) to vacation day k means Doctor i should work on Vacation Day k

Matching Problem #2 (Ch. 7, s.e. 2)

Runtime

- Reduction takes O(n * # of vacation days) time
- Ford fulkerson is O(mC), because each edge has integer capacity
  - In this case, is O(# doctors * # holidays), C is O(number of holidays)
  - You need to be explicit about what m and C are in big O to apply the limit on Ford Fulkerson runtime

Proof Of Correctness

- Valid Matching Existence → Max flow = # of vacation days
  - Explain how if you have a valid matching, you can set flow through each edge such that flow into t = # of vacation days
  - This is max flow because (V\t, t) is (s,t) cut with min cut of # of vacation days
- Max flow = # of vacation days→ Valid Matching Existence
  - If max flow is number of vacation days, each vacation day has flow to it from some gadget
  - Flow from a gadget for doctor i to vacation day k means Doctor i should work on Vacation Day k
  - Show that this matching obeys both constraints

Min Cut Reduction

- Reductions to Network Flow allows us to use Min Cut-Max Flow theorem to solve problems with the following properties
  - Goal is to find an "optimal" partition set of "objects" into sets A and B
    - Optimal: maximize total reward minus total penalty
    - Ex. Select some subset of objects (don't select the others)
  - Reward/Penalty comes from:
    a) An object being in set A
    b) An object being in set B
    c) A pair of objects being split across the two sets
  - May contain "dependencies" → basically just (c) above in disguise

- Ex. Project Selection (Section 7.11 in textbook)
  - Need to split projects into "selected" and "not selected"
  - Projects have positive or negative reward for being selected
  - Has dependencies (certain projects cannot be selected without others being selected)

Min Cut Reduction

To use network flows, we want to convert goal into "minimize penalty of partition"

- Set up flow network such that the capacity of any s-t cut (A, B) is ***exactly equal*** to the penalty for partitioning the set of objects into A and B
- Why this is useful: Max-flow = Min cut = Min total penalty
  - Determine max flow then find out its residual graph

Converting "Maximize Reward" objective into "Minimize Penalty" objective

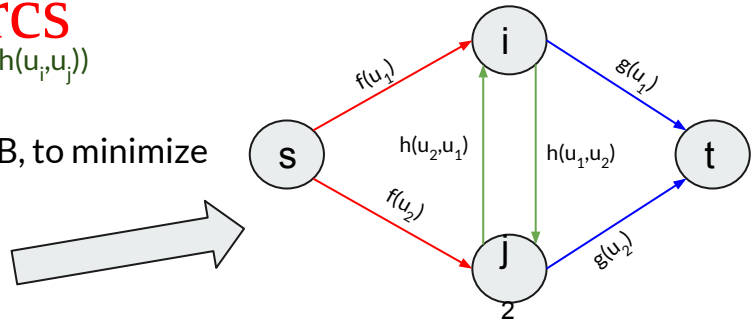
- Reward for selecting object to be in A → Penalty for not selecting it (It being in B)

Min Cut Reduction

- Setting up Flow Graph:

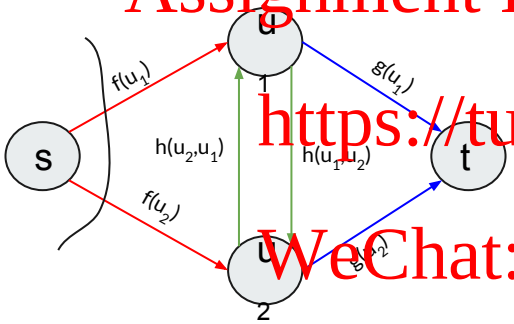
- For each object, create a node u in the flow networks
  - $s \rightarrow u$ capacity: Penalty for u being in B (f(u))
  - $u \rightarrow t$ capacity: Penalty for u being in A (g(u))
  - $u_i \rightarrow u_j$ capacity: Penalty for u_i being in A but u_j being in B ($h(u_i,u_j)$)

- Example: Goal is to partition $\{u_1, u_2\}$ into subsets A and B, to minimize total "penalty"

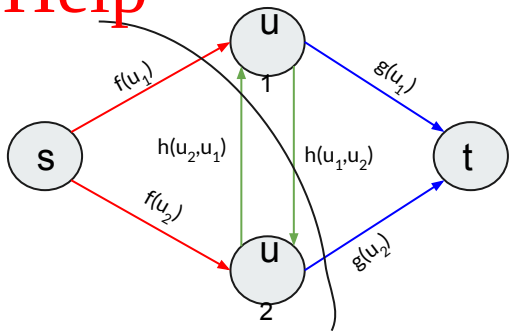


Min Cut Reduction (Possible Partitions from last slide)

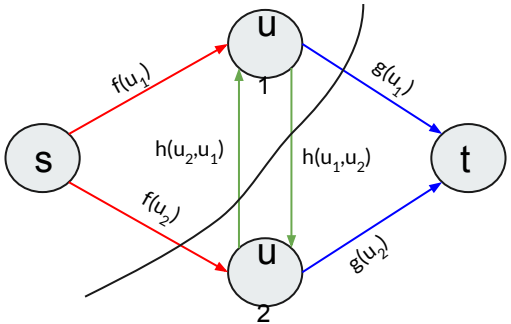
A = {}
B = $\{u_1, u_2\}$



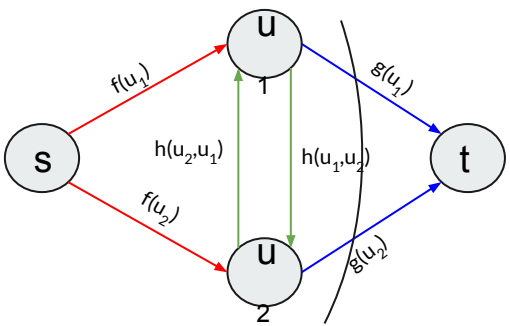
A = $\{u_2\}$
B = $\{u_1\}$



A = $\{u_1\}$
B = $\{u_2\}$

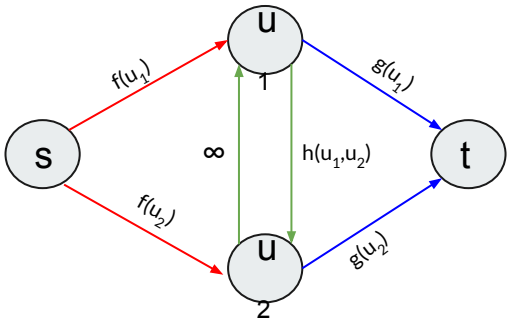


A = $\{u_1, u_2\}$
B = {}



Min Cut Reduction

- **Dependencies**
  - ex. u_j can only be in A if u_i is in A too
- How to handle this
  - Use infinite capacity for $u_j \rightarrow u_i$ edge
  - Why this works:
    - If an s,t cut (A,B) were to place u_j in A but u_i in B, the cut would have infinite capacity because of $u_j \rightarrow u_i$ edge



$A = \{u_2\}, B = \{u_1\}$ would not happen because of infinite capacity from $u_2 \rightarrow u_1$

Application Porting (Chapter 7, Exercise 29)

- The problem
  - You have a collection of n software applications running on an old system
  - Need to choose some subset of applications from {2, …, n} to be ported to a new system (#1 must stay behind)
  - Goal is to choose subset to maximize monetary benefit
- Rewards/Penalties
  - Receive $+b_i$ benefit for porting application i to the new system
  - Expense x_{ij} for porting one of i or j to the new system but not both
  - Projects have positive or negative reward for being selected
  - Has dependencies (certain projects cannot be selected without others being selected)

How to set up flow network to solve with min cut reduction?

Nodes? Edges? Edge Capacities?

Application Porting (Chapter 7, Exercise 29)

- Nodes?
  - Applications
    - Excluding application 1 (since it's always with s)

- Let's try to construct a graph, where for a given cut A/B, a node being in B represents the application being ported, and a node being in A represents the application not being ported.
- How do we set up edge capacities between the applications and s/t, so that minimizing the capacity of the cut also maximizes our profit?

Application Porting (Chapter 7, Exercise 29)

- Formally, given a cut A/B, our profit equals:

$$\sum_{u \in B} b_u - \sum_{u \in B, v \in A} x_{u,v}$$

- Maximizing this equals minimizing:

$$\sum_{u \in B, v \in A} x_{u,v} - \sum_{u \in B} b_u$$

Let B be the sum of rewards for every application $2..n$, i.e. $B = \sum_{u \in B} b_u + \sum_{v \in A} b_v$

We can add a constant to the equation we are minimizing and have it be equivalent, so maximizing profit is equivalent to minimizing

$$\sum_{u \in B, v \in A} x_{u,v} - \sum_{u \in B} b_u + B = \boxed{\sum_{u \in B, v \in A} x_{u,v} + \sum_{v \in A} b_v}$$

Application Porting (Chapter 7, Exercise 29)

- This equation is much easier to work with! We want the capacity of an A/B cut to equal that value.
-
- For every node *v* in A (representing an application not ported), we would like a *b_v* capacity edge going across the cut.
    - Hence, add a *b_i* capacity edge from every application node *i* to the sink, *t*.
- For every pair of nodes *u* in B and *v* in A (representing u being ported, but not v), we would like a $x_{u, v}$ capacity edge going across the cut.
    - Hence, add an $x_{u, v}$ capacity edge between every pair of application nodes.
    - Note that $x_{u, v} = x_{v, u}$, so we are essentially adding an edge from u to v and an edge from v to u, both with equal capacity .

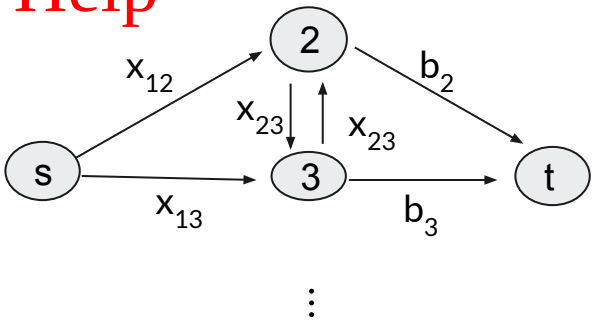
Application Porting (Chapter 7, Exercise 29)

A more intuitive explanation - you can think of b_i as the "penalty" for not porting application i, since that is the missed reward/opportunity cost you are paying for not porting i. Hence, we would like the i -> t edge to represent that penalty.

- Edges + Capacities
  - s ⟶ application i
    - Penalty for i being ported
      - x_{1i}
  - application i ⟶ t
    - Penalty for i not being ported
      - b_i
  - application i ⟶ application j
    - Penalty for j being ported but not i
      - x_{ij}

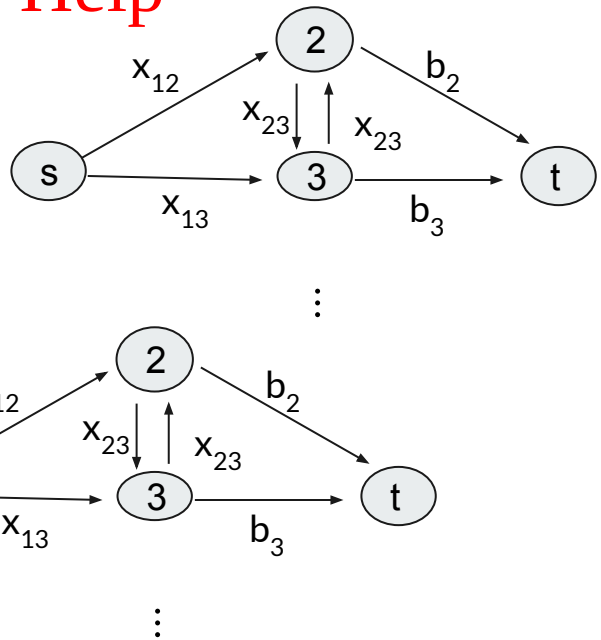
Application Porting (Chapter 7, Exercise 29)

- Edges + Capacities
  - s → application i
    - Penalty for i being ported
      - x_{1i}
  - application i → t
    - Penalty for i not being ported
      - b_i
  - application i → application j
    - Penalty for j being ported but not i
      - x_{ij}



Application Porting (Chapter 7, Exercise 29)

- Edges + Capacities
  - s → application i
    - Penalty for i being ported
      - x_{1i}
  - application i → t
    - Penalty for i not being ported
      - b_i
  - application i → application j
    - Penalty for j being ported but not i
      - x_{ij}



Prelim Tip

- For setting up max-flow reductions, try to think of the setup as “layers” of nodes. Then, you can reason about edges between layers of nodes more easily.

- Setting the edge capacities in min-cut reductions can be confusing: as a first step, decide what you want your sets A and B in the cut to represent

    - For e.g. in the application porting problem, we decided that a node being in B meant that it was ported, and it being in A meant it wasn’t ported.
    - This will help you intuitively reason about what the capacity of any given cut should be, to ensure that minimizing the capacity equals maximizing the profit.

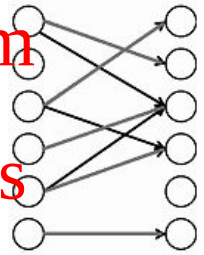
NP-Completeness

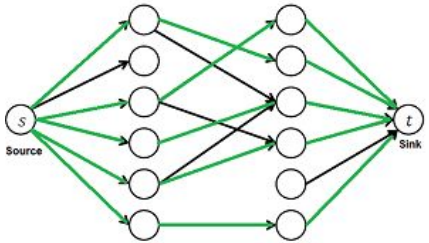
Comparing “Hardness” of Problems

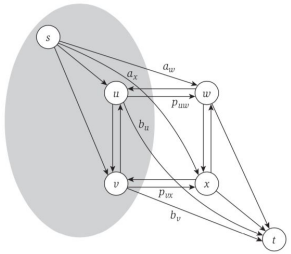
- Another application of reductions
- X is at least as “hard” as Y if
  - If we’re able to solve X efficiently, then we can also solve Y efficiently
- $Y \leq_P X$
  - X is at least as “hard” as Y
    - Y can be reduced to X using a *polynomial time reduction*
- Bipartite Matching $\leq_P$ Finding a max flow in a flow network

Reductions

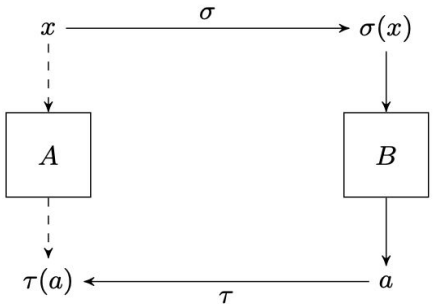
- Solving one problem by *converting* it into an instance of another problem
- Create efficient algorithms by bootstrapping off algorithms you already know
  - Bipartite Matching → Max Flow
  - Finding Min Cut → Max Flow







Polynomial Time Reductions



Taken from these notes

- For polynomial time reduction from Y to X
  a. Polynomial time preprocessing to convert to instances of X
  b. Polynomial number of calls to a black-box subroutine that solves X
  c. Polynomial time post-processing on the result to get a result for Y.
- σ: *polynomial* time function σ mapping instances of Y to instances of X,
- τ: *polynomial* time function τ mapping results of X back to results of Y.
- For decision problems
  - τ is the identity function
  - To show equivalence, we need to show that

y is a yes-instance of Y *iff* σ(y) is a yes-instance of X

Important Considerations

- $Y \leq_P X$ implies that if we can solve X in polynomial time, we could also solve Y in polynomial time.
- The contrapositive of this also holds: if we cannot solve Y in polynomial time, then we cannot solve X in polynomial time.
- **The direction of the reduction is a common tripping point** - remember that to show that X is at least as hard as Y, we want to convert every instance of Y to some instance of X, not the other way around.
- Also note that $\leq_P$ is transitive, since compositions of polynomial-time reductions are still polynomial time.

What is NP-Completeness?

NP - The set of decision problems where there exists a certifier and certificate: verify “yes” solution in polynomial time

NP-Complete - The set of decision problems *X* that are in NP such that any decision problem *Y* in NP may be reduced to *X* (NP-Hard)

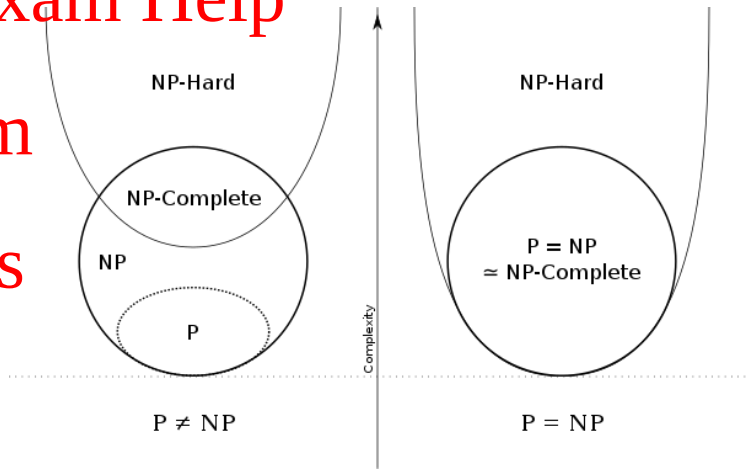
P - The set of decision problems that can be solved in polynomial time

Complexity Classes: NP

- Formally, a problem X is in NP if for every instance *x* of X, there exists a polynomial time verifier $\psi(w)$ that takes in a polynomial-length witness *w*, where $\psi(w)$ is true iff *x* is a yes-instance of the problem.
- This shows that there is a polynomial time algorithm (an *efficient certifier*) to verify that x is a yes-instance of X.
- You can also show a problem A is in NP by showing that $A \leq_P B$, where B is some problem in NP. (The former option is often easier, though).

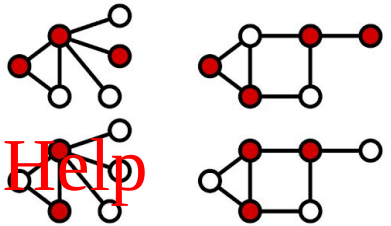
Complexity Classes: NP-hard problems

- We define NP-hard problems to be problems that are at least as hard as every problem in NP. (Implying that they are at least as hard as the hardest problems in NP)
- This means that there is a polynomial time reduction from every problem in NP to an NP-hard problem.

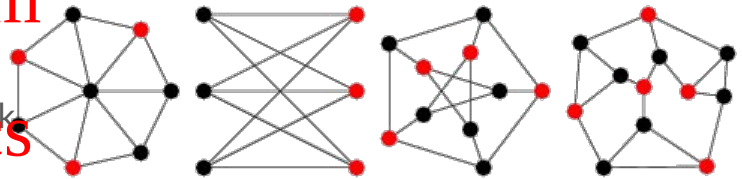


Example NP Hard Problems

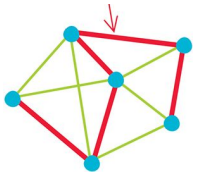
SAT/3SAT: Given a CNF form formula involving *n* variables, is there a variable assignment that makes the formula true?



Vertex Cover: Does there exist of subset of vertices of size ≤ k which covers every edge?

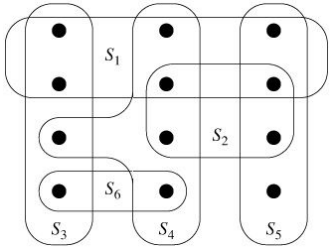


Independent Set: Does there exist of subset of vertices of size ≥ k with no shared edges?



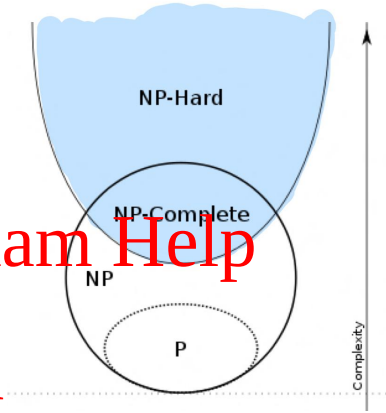
Hamiltonian Path: Does there exist a path which contains every vertex exactly once?

Set Cover: Given subsets of objects, does there contain a set of size ≤ k of these subsets whose union covers all objects?

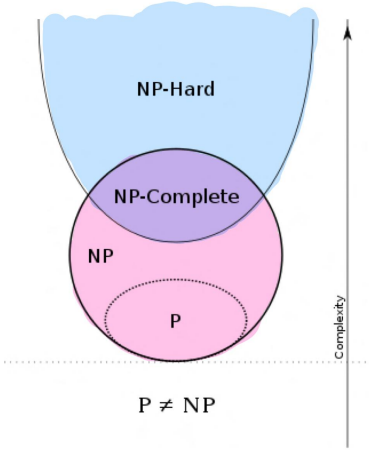


NP-completeness

- NP Complete: In NP and NP Hard
- Step 1: To show that a problem X is NP-hard, we can just reduce from any NP-complete problem to X.
- Step 2: Show X is in NP (Polynomial time verifier exists)



Step 1: Reducing to X from an NP-complete problem gives us a lower bound for the hardness of X



Step 2: Showing that X can be deterministically verified in polynomial time gives us an upper bound for the hardness of X.

Showing NP-completeness

Here is the general structure for proving that X is NP-complete.

1. Show that X is NP-hard. This involves showing that $Y \leq_p X$, where Y is some NP-complete problem.
    - Define a mapping σ from an arbitrary instance y of Y to some instance x of X.
    - Show that y is a yes-instance of Y $\Rightarrow$ $\sigma(y)$ is a yes-instance of X.
    - Show that $\sigma(y)$ is a yes-instance of X $\Rightarrow$ y is a yes-instance of Y.
    - Show that the mapping σ takes polynomial time.

Showing NP-completeness (step 1 continued)

Technically we can reduce using any NP-complete problem, but a key skill is being able to identify similar patterns between X and an NP-complete problem you know of, to make the reduction easier. As you learn about more NP-complete problems, the range of problems you can use for reductions will increase!

If X involves...

- selecting at most k items, try reducing from Vertex Cover/Set Cover
- selecting at least k things, try Independent Set/Set Packing/Clique/Near-Clique
- giving an ordering over n items, try reducing from Hamiltonian Cycle/Hamiltonian Path/Travelling Salesman.

If all else fails/these patterns don't apply, SAT/3SAT is a good starting point! Section 8.10 in the book gives a useful overview of common NP-complete problems you can use.

Showing NP-completeness

2. Show that X is in NP. This involves showing that for any instance of X, there exists a polynomial length string *w* that can be verified in polynomial time to check if the instance is a yes-instance. (sounds scary but is usually the easy part)

- Define a polynomial length *w* for every instance x that can be computed in polynomial time.
- Define the predicate ψ, and show that $\psi(w)$ is true iff x is a yes-instance.
- Show that $\psi(w)$ takes polynomial time to compute.

Proving Y is NP-Complete (Summary)

1. ***Y* is in NP** - motivate the certificate and certifier
   a. Certificate: A string of polynomial length with respect to the instance of Y, describing a "solution"
      i. e.g. A set of vertices for Vertex Cover
   b. Certifier: A function to process the certificate and verify solution in polynomial time
2. **Reduction** - show *Y* is at least as hard as a known NP-Hard problem *X*, which is $X \leq_p Y$
   a. Assume a black box to solve *Y*, and translate an arbitrary instance of *X* to a specific instance of *Y*.
   b. Transform an input to *X* to an equivalent input to *Y*.
3. **Reduction takes polynomial time** - show *X* is polynomial time reducible to *Y*
4. **Yes Solution to *X* implies Yes Solution to *Y***
5. **Yes Solution to *Y* implies Yes Solution to *X***

Which NP-Hard Problem to reduce from?

Problem to reduce from	Types of Problems this is useful for proving NP-Hard:
3-SAT	constraint satisfaction problems (3-SAT can be naturally reduced to many NP-Complete problems, this is a good problem to reduce from when a problem does not have a natural connection to another seen NP-Hard Problem)
Vertex Cover or Set Cover	covering problems or problems that involve selecting a subset that covers the whole (note that Set Cover is a more expressive version of Vertex Cover, so reductions from Set Cover will also have a natural reduction from Vertex Cover)
Independent Set	packing problems or problems that invlove selecting a subset in which pairs of options conflict
Hamiltonian Cycle, Path or Traveling Salesman	problems that involve connecting or sequencing a subset of things, particularly if there are constraints or cost invloved in sequencing pairs of things

Harder as k **decreases**

Harder as k **increases**

Important: Look at Section 8.10 for a comprehensive list of problems.

Example: Diverse Subset (ch 8. Ex. 2)

A store trying to analyze the behavior of its customers will often maintain a two-dimensional array A, where the rows correspond to its customers and the columns correspond to the products it sells. The entry $A[i, j]$ specifies the quantity of product j that has been purchased by customer i.

Here's a tiny example of such an array A.

	liquid detergent	beer	diapers	cat litter
Raj	0	6	0	3
Alanis	2	3	0	0
Chelsea	0	0	0	7

Example: Diverse Subset

One thing that a store might want to do with this data is the following. Let us say that a subset S of the customers is *diverse* if no two of the of the customers in S have ever bought the same product (i.e., for each product, at most one of the customers in S has ever bought it). A diverse set of customers can be useful, for example, as a target pool for market research.

We can now define the Diverse Subset Problem as follows: Given an $m \times n$ array A as defined above, and a number $k \leq m$, is there a subset of at least k of customers that is *diverse*?

Show that Diverse Subset is NP-complete.

Proof: Diverse Subset is in NP

Notice that a verifier can take a subset of customers S, check if the size of S is at least k, and then for each customer check if no two customers have bought the same product in O(nk) time. (why?) So, Diverse Subset is in NP.

The Reduction

We create a reduction from Independendent Set (IS) to Diverse Subset (DS). Recall the IS problem:

> *Given a graph G and a number k, does G contain an independent set of size at least k?*

We will show that

$$\text{Independent Set} \leq_p \text{Diverse Subset}$$

Constructing The Reduction

Given a graph G = (V, E), and a number k as an instance of Independent Set, construct an instance of Diverse Subset as follows: Construct a customer for each node of G, and construct a product for each edge of G. In other words, we have |V| customers, and |E| edges. Now, construct an array where customer v bought 1 unit of product e, if the edge e is incident on the node v. We then ask if this array has a diverse subset of size k.

Proof of Correctness

We claim that G has an independent set of size at least k if and only if this instance has a diverse subset of size at least k:

(<=) If there is a diverse subset of size k, then the nodes corresponding to the customers in the subset all have the property that none of them are incident on the same edge (by construction). So, they form a diverse subset.

(=>) If there is an independent set, then no two of the nodes in the set are incident on the same edges, and thus the set of customers corresponding to the set of nodes in this IS have the property that no two of them have bought the same product, so they form a diverse set.

Prelim Tip

We've been working on NP completeness for a while, so it might be hard for you to decide whether a given problem is in P or if it is NP-hard.

If you are asked to decide whether a problem is in P or is NP-hard, as a first step, it is easier to try to come up with a polynomial time algorithm for the problem. If you find yourself failing, then try to think of a reduction from an NP complete problem.

You might see similarities to a known polynomial time problem you have seen before, in which case try reducing **to** that polynomial time problem.

You might see similarities to a known NP-complete problem you have seen before, in which case try reducing **from** that NP-complete problem.

Prelim Tip

Ways to show a problem X is in P:

- Give a poly time algorithm for it! From scratch, no reductions.
- Alternatively: If you know a problem A is solvable in polynomial time, show that $X <=_p A$, i.e. construct a poly time reduction **from X to A.** For example, we are able to reduce **from** bipartite matching **to** max flow.

Showing X is NP-hard:

- Pick a problem A that is NP-complete. Show that $A <=_p X$, i.e. construct a poly time reduction **from A to X**. For e.g., we reduced Independent Set **to** Diverse Subset Remember the direction of the reduction!

Note: This example reduces from the 3D matching problem, which wasn't covered in lecture, so we did not cover these slides in recitation. If you are interested in the solution, feel free to read through this! **You are not expected to know about the 3D matching problem.**

Example 2: Madison's Refrigerator Magnets

Your friends' preschool-age daughter Madison has recently learned to spell some simple words. To help encourage this, her parents got her a colorful set of refrigerator magnets featuring the letters of the alphabet (some number of copies of the letter *A*, some number of copies of the letter *B*, and so on), and the last time you saw her the two of you spent a while arranging the magnets to spell out words that she knows.

Somehow with you and Madison, things always end up getting more elaborate than originally planned, and soon the two of you were trying to spell out words so as to use up all the magnets in the full set—that is, picking words that she knows how to spell, so that once they were all spelled out, each magnet was participating in the spelling of exactly one

Madison's Refrigerator Magnets

of the words. (Multiple copies of words are okay here; so for example, if the set of refrigerator magnets includes two copies each of C, A, and T, it would be okay to spell out *CAT* twice.)

This turned out to be pretty difficult, and it was only later that you realized a plausible reason for this. Suppose we consider a general version of the problem of *Using Up All the Refrigerator Magnets*, where we replace the English alphabet by an arbitrary collection of symbols, and we model Madison's vocabulary as an arbitrary set of strings over this collection of symbols. The goal is the same as in the previous paragraph.

Prove that the problem of Using Up All the Refrigerator Magnets is NP-complete.

Madison's Refrigerator Magnets

Simplify this. The problem has more detail than required, but can be basically boiled down to the problem of whether or not given a multiset of symbols M and a set S of strings over those symbols, can you create a multiset consisting of strings (or words) from S such that the words use up all the letters from M.

What problem does this resemble in spirit? Hint: Refer to section 8.10 in the book for an idea of which problem you might want to reduce from)

Madison's Refrigerator Magnets

Obviously, Using Up All The Refrigerator Magnets (UUATRM) is in NP. A verifier can take the multiset of of words Madison formed, and for each word, count the number of times each letter is used in the word, and add all these totals up. Once it's gone through the list of words, it can compare the count of letters used to the count of letters available, and accept/reject the certificate accordingly. This takes polynomial time, so UUATRM ∈ NP. (Exercise: verify this)

The Reduction

We reduce from the 3-D Matching (3DM) to UUATRM. Recall the 3DM Problem:

> *Given disjoint sets X, Y, and Z, each of size n, and given a set $T \subseteq X \times Y \times Z$ of ordered triples, does there exist a set of n triples in T so that each element of $X \cup Y \cup Z$ is contained in exactly one of these triples?*

We want to show that

$$\text{3DM} \leq_p \text{UUATRM}$$

Creating the reduction

- Given an instance of 3DM with 3 sets X, Y, Z, where |X| = |Y| = |Z| = n, and a list of triples T, we create the instance of UNATRM as follows:
- Each element in each of the sets maps to a unique letter. So, the alphabet contains 3n letters, all appearing exactly once (in other words, 3n fridge magnets, where each represents a different letter).
- The list of tuples becomes the list of words that Madison knows

Proving correctness

We want to show that our given instance of 3DM has a matching if and only if all the magnets can be used up in the instance of UUATRM:

(=>) Suppose that we have a valid 3D Matching M in our instance. Then, every tuple in M corresponds to a word, and by our construction, that means that the n words have used up all 3n letters.

(<=) Now, suppose the magnets are all used up. Then, we know we got exactly n words, since there are exactly 3n letters and each word uses exactly 3 letters. So, the list of triples corresponding to each word is a valid 3D Matching. (why?)