Week 14 - Friday

COMP 4500

Last time

- What did we talk about last time?
- Review of second third of course
  - Recurrence relations
  - Divide and conquer
    - Counting inversions
    - Closest pair of points
    - Integer multiplication
  - Master theorem
  - Dynamic programming
    - Weighted interval scheduling
    - Subset sum
    - Knapsack
    - Sequence alignment

Questions?

Assignment 7

Logical warmup

- Each of the following equations is made of matches and written using Roman numerals
- Unfortunately, each equation is wrong
- Move a single match stick in each equation to correct the error
  - VI = IV – III (12 matches)
  - XIV – V = XX (14 matches)
  - X = VIII – II (12 matches)
  - VII = I (7 matches but bizarre)

Review

Final exam

- Final exam:
  - Wednesday, April 24, 2024
  - 8:00 – 10:00 a.m.
- It will mostly be short answer
- There will be diagrams
- There might be a matching problem
- There will likely be a (simple) proof
- It will be 50% longer than previous exams, but you will have 100% more time

Maximum Flow

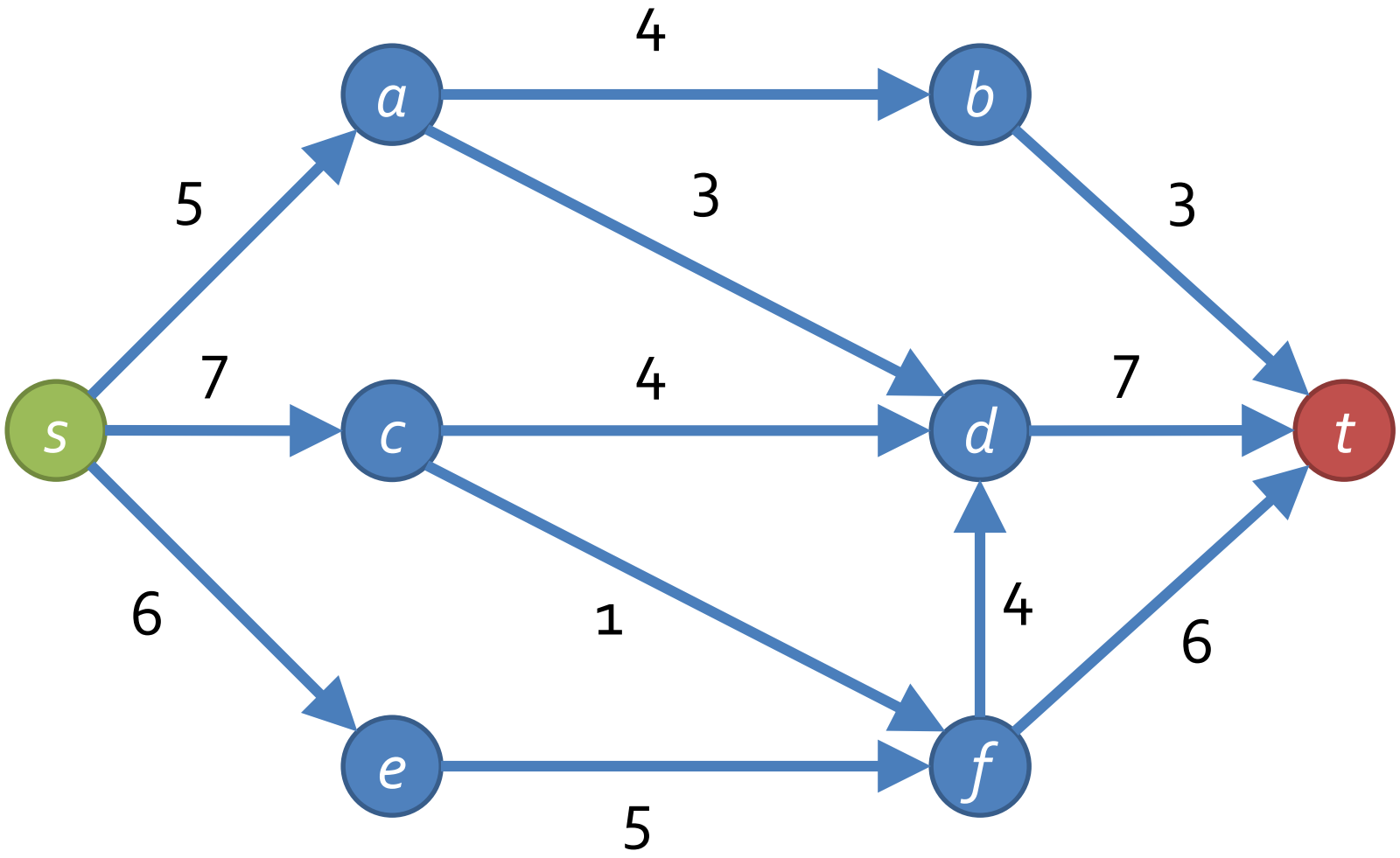
Flow networks

- A **flow network** is a weighted, directed graph with positive edge weights
  - Think of the weights as **capacities**, representing the maximum units that can flow across an edge
  - It has a **source *s*** (where everything comes from)
  - And a **sink *t*** (where everything goes to)
- Some books refer to this kind of flow network specifically as an ***st***-flow network

Maximum flow

- A common flow problem is to find the **maximum flow**
- A maximum flow is a flow such that the amount leaving s and the amount going into t is as large as possible
- In other words:
  - The maximum amount of flow gets from ***s*** to ***t***
  - No edge has more flow than its capacity
  - The flow going into every node (except ***s*** and ***t***) is equal to the flow going out

Flow network



Ford-Fulkerson algorithm

- Ford-Fulkerson is a family of algorithms for finding the maximum flow

1. Start with zero flow on all edges
2. Find an augmenting path (increasing flow on forward edges and decreasing flow on backwards edges)
3. If you can still find an augmenting path in the residual graph, go back to Step 2

Bipartite Matching

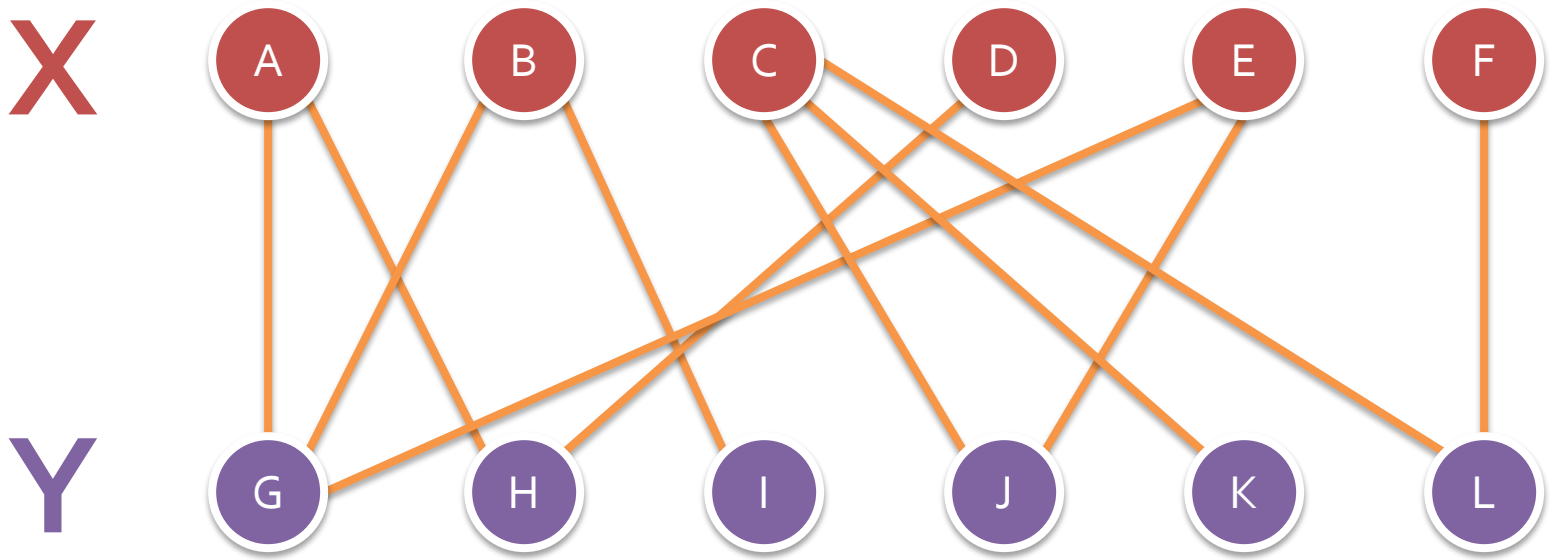
Bipartite graphs

- Recall that a bipartite graph is one whose nodes can be divided into two disjoint sets X and Y
- Every edge has one end in set X and the other in set Y
  - There are no edges from a node inside set X to another node in set X
  - There are no edges from a node inside set Y to another in set Y
- Equivalently, a graph is bipartite if and only if it contains no odd cycles

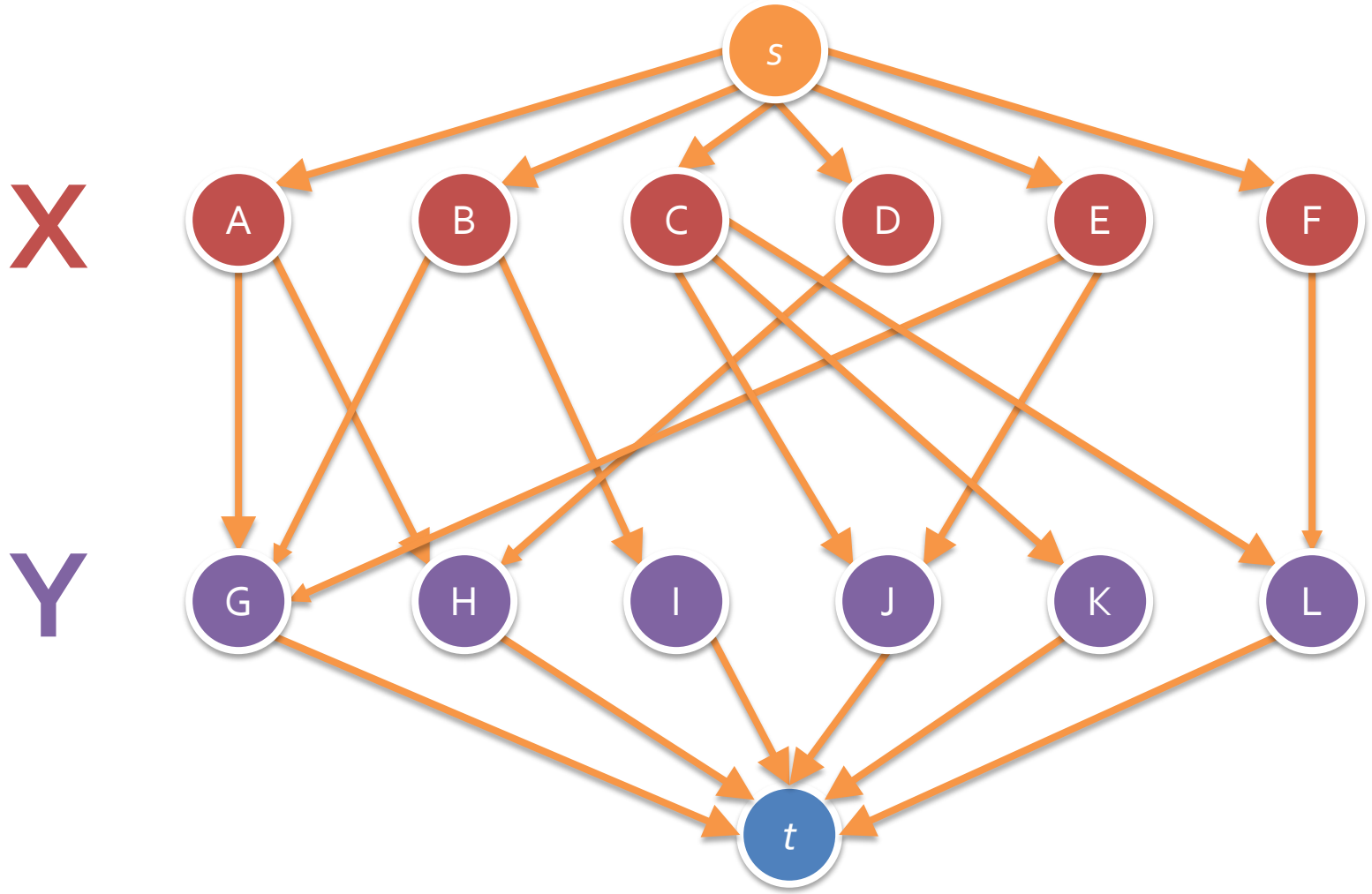
Maximum matching

- **Matching** means pairing up nodes in set X with nodes in set Y
- A node can only be in one pair
- A **perfect matching** is when every node in set X and every node in set Y is matched
- It is not always possible to have a perfect matching
- We can still try to find a **maximum matching** in which as many nodes are matched up as possible

Bipartite matching problem



Maximum flow problem



An easy change

- Take a bipartite graph *G* and turn it into a directed graph *G'*
- Create a source node ***s*** and a sink node ***t***
- Connect directed edges from the source to all the nodes in set X
- Connect directed edges from all the nodes in set Y to the sink
- Change all the undirected edges from X to Y to directed edges from X to Y
- Set the capacities of all edges to 1

Algorithmic changes

- We run the Ford-Fulkerson algorithm to find the maximum flow on our new graph
- Since all edges from X to Y have capacity 1, they will either have a flow of 1 or of 0
- If they have a flow of 1, they are in the matching
- If they have a flow of 0, they aren't
- The maximum flow value tells us how many nodes are matched

Maximal matching

- To make the algorithm go faster, we can start with a **maximal matching**
- A maximal matching is not necessarily maximum, but you can't add edges to it directly without removing other edges
- In essence, arbitrarily match unmatched nodes until you can't anymore
- Then start the process of looking for augmenting paths

Matching algorithm

1. Come up with a legal, maximal matching
2. Take an **augmenting path** that starts at an unmatched node in X and ends at an unmatched node in Y
3. If there is such a path, switch all the edges along the path from being in the matching to being out and vice versa
4. If there is another augmenting path, go back to Step 2

NP-Completeness

Characterizing hardness

- How can we compare the hardness of problems?
- How are we able to say that NP-complete problems are all the same level of hardness?
- We want a formal way to describe that problem ***X*** is at least as hard as problem ***Y***
- The tool we use to argue that ***X*** is at least as hard as ***Y*** is called a **reduction**

Reductions

- We imagine that we have a black box that can solve problem ***X*** instantly
- Can any instance of problem ***Y*** be solved by doing polynomial work to format the input for ***Y*** into input for ***X*** followed by a polynomial number of calls to the black box that solves ***X***?
- If the answer is **yes**, we write $Y \leq_P X$ and say that ***Y*** is polynomial-time reducible to ***X***

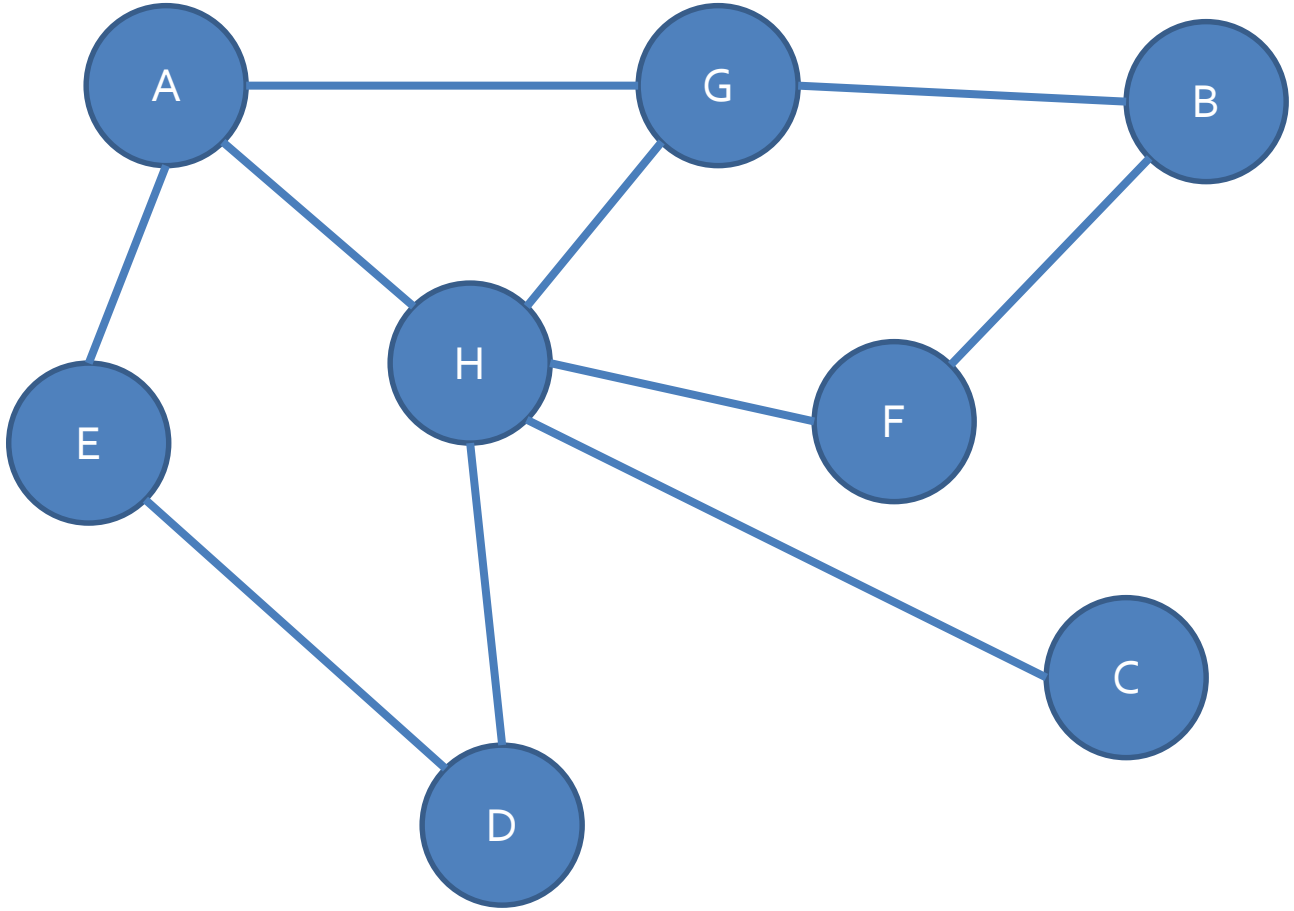
What about the other direction?

- We didn't really study logic in this class
- If you have an implication $p \rightarrow q$ that is true, its **contrapositive** $\sim q \rightarrow \sim p$ is also true
- Implication:
  - Suppose $Y \leq_P X$. If X can be solved in polynomial time, then Y can be solved in polynomial time.
- Contrapositive:
  - Suppose $Y \leq_P X$. If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.

Independent set

- Recall the independent set graph problem
- Given an undirected graph, find the largest collection of nodes that are not connected to each other
- Practical application:
  - Nodes represent friends of yours
  - An edge between those two nodes means they hate each other
  - What's the largest group of friends you could invite to a party if you don't want any to hate each other?

Independent set example



Hardness of independent set

- Independent set is an NP-complete problem
- We don't know a polynomial-time algorithm for it, but we don't know how to prove that there isn't one
- We just stated the **optimization** version of independent set:
  - Find the largest independent set
- But there is also a **decision** version:
  - Given a graph ***G*** and a number ***k***, does ***G*** contain an independent set of size at least ***k***?

Vertex cover

- The vertex cover problem is another graph problem:
  - Given a graph $G = (V, E)$, we say that a set of nodes $S \subseteq V$ is a **vertex cover** if every edge $e \in E$ has at least one end in S
  - In other words, find a set of vertices such that all edges touch at least one
- It's easy to find a big vertex cover: all vertices
- It's hard to find a small one
- Decision version:
  - Given a graph G and a number k, does G contain a vertex cover at size at most k?

Relationship between independent set and vertex cover

- **Claim:** Let $G = (V,E)$ be a graph. S is an independent set if and only if its complement $V - S$ is a vertex cover.
- **Proof:**
  - Suppose that S is an independent set. Consider an edge $e = (u,v)$. Since S is independent, it cannot be the case that both u and v are in S. Thus, one of them must be in $V - S$. It must be the case that every edge has at least one end in $V - S$, so $V - S$ is a vertex cover.

Proof continued

- Suppose that $V - S$ is a vertex cover. Consider any two nodes u and v in S. If they were joined by edge e, then neither end of e would lie in $V - S$, contradicting the assumption that $V - S$ is a vertex cover. Thus, it must be the case that no two nodes in S are joined by an edge, so S must be an independent set.

■

Vertex cover $\leq_P$ independent set

- **Proof:**
  - If we have a black box to solve independent set, we can decide whether ***G*** has a vertex cover of size at most ***k*** by asking the black box whether ***G*** has an independent set of size at least ***n*** – ***k***.
  - ■

SAT and 3-SAT

- Consider a set of ***n*** Boolean variables, $\boldsymbol{x_1}$, $\boldsymbol{x_2}$, …, $\boldsymbol{x_n}$
- Each value is 0 or 1
- A **term** is either a variable x_i or its negation $\overline{x_i}$
- A **clause** is a disjunction (set of logical ORs) of terms, like:

$$x_1 \vee \overline{x_6} \vee \overline{x_5} \vee x_2 \vee \overline{x_3} \vee x_4$$

- A clause has length ***l*** if it has ***l*** terms
- A truth assignment is an assignment of 0 or 1 to every x_i

Satisfiability

- A clause is **satisfied** if a truth assignment evaluates it to true
- A collection of clauses is satisfied if a truth assignment satisfies each clause
- Another way to view satisfiability is that, given clauses C_1, C_2, …, C_k, the following statement evaluates to true with some truth assignment:

$$C_1 \wedge C_2 \wedge \cdots \wedge C_k$$

Satisfiability

- The satisfiability problem (SAT):
  - Given a set of clauses $C_1, C_2, \ldots, C_k$ over a set of variables $\{x_1, x_2, \ldots, x_n\}$, is there a satisfying truth assignment?
- The 3-satisfiability problem (3-SAT) is a special case of SAT in which all clauses have exactly three terms:
  - Given a set of clauses $C_1, C_2, \ldots, C_k$, each of length 3, over a set of variables $\{x_1, x_2, \ldots, x_n\}$, is there a satisfying truth assignment?

What makes a problem NP?

- Is there something that sets apart problems that are NP-complete from other problems that (probably) take exponential time?
- Yes!
- It's easy to prove that you have an answer for one
- In other words, they're easy to **check**

Problems and algorithms

- Input to a problem will be encoded as a finite (binary) string ***s***
- The length of ***s*** is $|s|$
- For a decision problem, an algorithm ***A*** receives an input string and returns "yes" or "no"
  - This output is $A(s)$
- A decision problem ***X*** is the set of strings for which the answer is "yes"
- ***A* solves** the problem ***X*** if for all strings ***s***, $A(s)$ = "yes" if and only if $s \in X$

The class of problems P

- Formally, an algorithm ***A*** has polynomial running time if
  - There is a polynomial function ***p***(***x***)
  - Such that, for every input string ***s***, the algorithm ***A*** terminates on ***s*** in at most O(***p***(|***s***|)) steps
- Thus, **P** is the set of all decision problems ***X*** for which there is an algorithm ***A*** with polynomial running time that solves ***X***

Efficient certification

- ***B*** is an **efficient certifier** for a problem ***X*** if:
  - ***B*** is a polynomial-time algorithm that takes two input arguments ***s*** and ***t***
  - There is a polynomial function ***p***(***x***) such that, for every string ***s***, we have ***s*** ∈ ***X*** if and only if there exists a string ***t*** such that |***t***| ≤ ***p***(|***s***|) and ***B***(***s***,***t***) = "yes"
- ***B*** can evaluate a "proof" ***t*** for input ***s***
- You could use ***B*** as part of a brute force approach, trying lots of strings ***t*** to see if they work for ***s***

The class of problems NP

- **NP** is the set of all problems for which there exists an efficient certifier
- Note that **P** $\subseteq$ **NP**
  - Why?
  - We can make an efficient certifier by simply using an efficient solver
  - Such a certifier could even ignore string ***t*** and check ***s*** on its own
- **NP** is an abbreviation for "nondeterministic polynomial" because, for a machine that can nondeterministically explore all paths at the same time, checking a solution and finding a solution take the same time

NP-Complete Problems

NP-complete problems

- While trying to figure out if **P** = **NP**, computer scientists have considered the hardest problems in **NP**
  - What are those?
- A hardest problem ***X*** in **NP** has the following properties:
  - $X \in \mathbf{NP}$
  - For all $Y \in \mathbf{NP}$, $Y \leq_P X$
- In other words, it's a problem in **NP** that we can reduce all other problems in **NP** to
- The hardest problems in any class are its "complete" problems
- Thus, we call the hardest problems in **NP** the **NP-complete** problems

An important consequence

- **Claim:** Suppose ***X*** is an **NP-complete** problem. ***X*** is solvable in polynomial time if and only if **P** = **NP**.
- **Proof:**
  - If **P** = **NP**, then ***X*** can be solved in polynomial time, since ***X*** ∈ **NP**.
  - Conversely, suppose that ***X*** can be solved in polynomial time. For all other problems ***Y*** ∈ **NP**, $Y \leq_P X$. Thus, all problems ***Y*** can be solved in polynomial time and **NP** ⊆ **P**. Since we already know that **P** ⊆ **NP**, it would be the case that **P** = **NP**.

NP-complete problems

- Circuit satisfiability
- 3-SAT
- Independent set
- Vertex cover
- Set cover
- Traveling salesman problem
- Hamiltonian cycle
- Hamiltonian path
- Graph coloring
- Subset sum
- Knapsack

Approximation Algorithms

Load balancing

- You have m machines $M_1, M_2, \ldots, M_m$
- You have n jobs
- Each job j has a processing time t_j
- We can assign jobs $A(i)$ to machine M_i
- The total time that M_i needs to work is:

$$T_i = \sum_{j \in A(i)} t_j$$

- We want to minimize the **makespan**, which is just the longest T_i
- In other words, we want the last machine running to stop running as early as possible
- Unfortunately, doing so in **NP-hard**

Improved approximation algorithm

- We use a greedy algorithm
- However, we first sort all the jobs in descending order
- Now, $t_1 \geq t_2 \geq \ldots \geq t_n$
- If there are m jobs or fewer, our algorithm will be optimal, since each machine will get at most one job
- If there are more than m jobs, $T^* \geq 2t_{m+1}$
  - Consider the first $m + 1$ sorted jobs.
  - Each takes at least t_{m+1} time. Since there are at least $m + 1$ jobs and only m machines, one machine will get at least two of these jobs.
  - That machine will have processing time at least $2t_{m+1}$

Sorted greedy algorithm gets a makespan $T \leq \frac{3}{2}T^*$

- **Proof:**
  - Let M_i be the machine that get the maximum load T in the greedy assignment
  - Let j be the last job assigned to M_i, and assume that M_i has at least 2 jobs
  - When j was assigned to M_i, it had the smallest load of any machine, namely $T_i - t_j$
  - Thus, every machine had load at least $T_i - t_j$

$$\sum_{k=1}^{m} T_k \geq m(T_i - t_j)$$

$$T_i - t_j \leq \frac{1}{m}\sum_{k=1}^{m} T_k$$

Proof continued

- Since $\sum_{k=1}^{m} T_k = \sum_{i=1}^{n} j_i$ and $\frac{1}{m}\sum_{i=1}^{n} j_i \leq T^*$

$$T_i - t_j \leq T^*$$

- Note that $\boldsymbol{j \geq m + 1}$, since the first $\boldsymbol{m}$ jobs will be put on $\boldsymbol{m}$ different machines
- Thus, $t_j \leq t_{m+1} \leq \frac{1}{2}T^*$
- But the optimal makespan must be at least as big as any job, thus $t_j \leq T^*$, thus:

$$T_i = (T_i - t_j) + t_j \leq T^* + \frac{1}{2}T^* = \frac{3}{2}T^*$$

- Since our makespan $T = T_i$, the proof is done.

■

Set cover (optimization version)

- Given:
  - Set U of n elements
  - Collection of sets $S_1, S_2, \ldots, S_m$ of subsets of U
  - Each subset S_i has a weight $w_i \geq 0$
- Find the subsets with smallest total weight whose union is equal to all of U

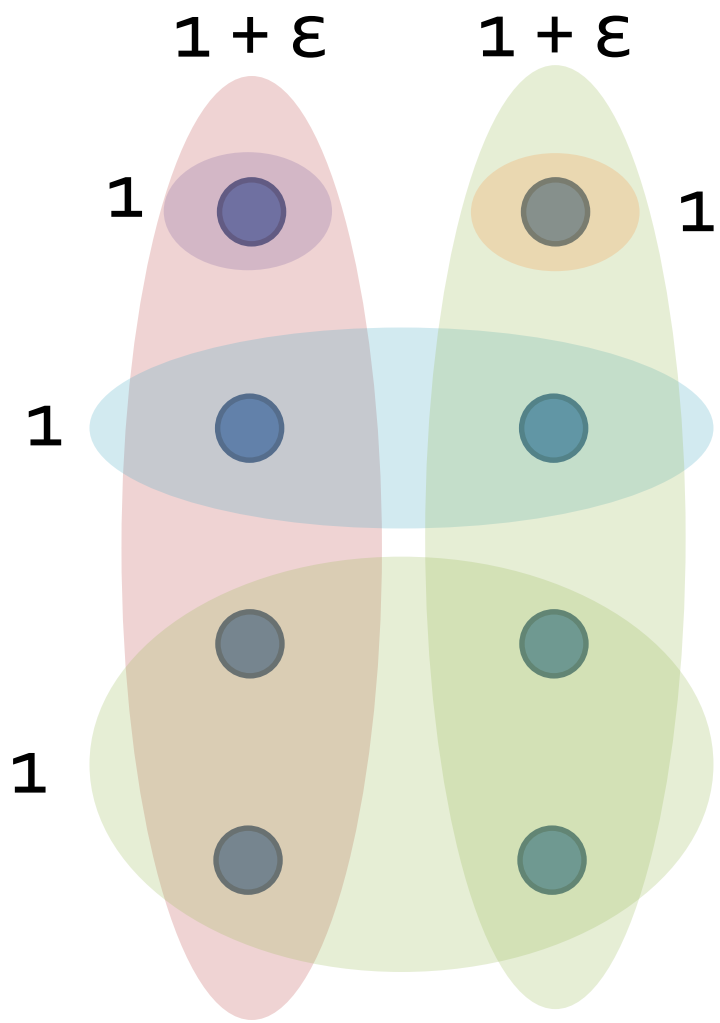
Algorithm design

- We want the most bang for our buck
- We want small weight sets with a lot of elements
  - In other words, low cost per element
- So, we look at the value $w_i/|S_i|$ for each set, and pick the lowest such value set
- We keep doing that, but we only "count" the elements in each set that still aren't covered

Greedy set cover algorithm

- Start with $\boldsymbol{R} = \boldsymbol{U}$ and no sets selected
- While $\boldsymbol{R} \neq \emptyset$
  - Select set $\boldsymbol{S_i}$ with minimum $\boldsymbol{w_i}/|\boldsymbol{S_i} \cap \boldsymbol{R}|$
  - Delete set $\boldsymbol{S_i}$ from $\boldsymbol{R}$
- Return the selected sets

Set cover example



Algorithm finds a total weight of 4

Optimal is a total weight of $2 + 2\varepsilon$

Analysis

- How good (or bad) is our set cover approximation in the worst case?
- Let's think about the cost per item incurred by each set we add:
  - $c_s = w_i/|S_i \cap R|$ for all $s \in S_i \cap R$
  - Imagine we assign that cost in the algorithm when we cover those elements
- Clearly, these c_s values end up being the total weight of our solution C:

$$\sum_{s_i \in C} w_i = \sum_{s \in U} c_s$$

Unfortunately: math

- To bound our analysis, we will use the idea of the harmonic function:

$$H(n) = \sum_{i=1}^{n} \frac{1}{n} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

- This function grows…slowly but infinitely
- We will not prove it here, but it turns out that $H(n)$ is $\Theta(\log n)$

Final approximation bound

- Let d^* be the size of the largest set
- **Claim:**
  - Set cover C found by our greedy algorithm has weight at most $H(d^*)$ times the optimal weight w^*
- **Proof:**
  - The optimal set cover C^* has weight $w^* = \sum_{S_i \in C^*} w_i$
  - By our previous proof:

$$w_i \geq \frac{1}{H(d^*)} \sum_{s \in S_i} c_s$$

Approximation bound continued

- Since C^* is a set cover

$$\sum_{S_i \in C^*} \sum_{s \in S_i} c_s = \sum_{s \in U} c_s$$

- Putting it all, insanely, together:

$$w^* = \sum_{S_i \in C^*} w_i \geq \sum_{S_i \in C^*} \frac{1}{H(d^*)} \sum_{s \in S_i} c_s \geq \frac{1}{H(d^*)} \sum_{s \in U} c_s = \frac{1}{H(d^*)} \sum_{S_i \in C} w_i$$

■

Log approximation

- All of that madness means that our approximation algorithm to set cover might return a set cover costing O(log ***d****) times the true optimal
- Worse, ***d**** could be some constant fraction of ***n***, making the approximation an O(log ***n***) times worse than optimal
- This approximation is worse than **any** constant approximation, since our approximation actually will degrade as ***n*** gets larger
- To top it off, there's even a proof that this is the best you can approximate set cover, unless **P** = **NP**

Knapsack

- We've seen knapsack in dynamic programming (but with a pseudo-polynomial running time)
- We've seen knapsack as an NP-complete problem
- Now, can we approximate it in fully polynomial time?
- Recall:
  - We have n items
  - Each item has a weight w_i and a value v_i
  - We want to maximize total value without going over our weight capacity W

The best approximation yet!

- Our algorithm will take those items and the capacity ***W*** as well as a parameter ε
- We will find a set of items ***S*** within the weight capacity whose value is at worst $\frac{1}{1+\varepsilon}$ of the optimal!
- And the algorithm will be polynomial for any **particular** choice of ε
  - But it will not be polynomial in ε, if that makes sense
- This kind of algorithm is called a **polynomial-time approximation scheme** (PTAS)

Algorithm design

- We had a pseudo-polynomial algorithm for knapsack that ran in time $O(\boldsymbol{nW})$
- The book gives details on how we can flip around weights and values to get a dynamic programming knapsack algorithm that runs in time $O(\boldsymbol{n}^2\boldsymbol{v}^*)$ where $\boldsymbol{v}^*$ is the largest value of any item
- Let's assume that algorithm is correct and build our approximation algorithm out of it

Rounding notation

- We use a rounding factor ***b***
- Each rounded value $\tilde{v}_i = \lceil v_i/b \rceil b$
- Note that $v_i \leq \tilde{v}_i \leq v_i + b$
- To get small values, we can scale the rounded values down by ***b***:

$$\hat{v}_i = \frac{\tilde{v}_i}{b} = \lceil v_i/b \rceil$$

- Note that the knapsack problem with values $\tilde{v}_i$ has the same optimum solution as the problem with $\hat{v}_i$, if you scale the answers by ***b***

Approximate knapsack algorithm

- Knapsack-Approx(ε)
  - Set $\boldsymbol{b} = (\varepsilon/(2\boldsymbol{n}))\ \max_i \boldsymbol{v}_i$
  - Solve the Knapsack problem with values $\hat{v}_i$
  - Return the set $\boldsymbol{S}$ of items found

Approximation running time

- We only rounded the values, not the weights, so the answer we get is legal
- The algorithm we use as a subroutine runs in time $O(\boldsymbol{n}^2\boldsymbol{v}^*)$ where $\boldsymbol{v}^*$ is the biggest value
- Since $\boldsymbol{b} = (\varepsilon/(2\boldsymbol{n}))\max_i \boldsymbol{v}_i$, the biggest value $\boldsymbol{v}_j$ will also have the biggest rounded value:

$$\hat{v}_j = \lceil v_j/b \rceil = \left\lceil \frac{v_j}{v_j\varepsilon/(2n)} \right\rceil = \left\lceil \frac{2n}{\varepsilon} \right\rceil = c \cdot n\varepsilon^{-1}$$

- So our algorithm on rounded values runs in time $O(\boldsymbol{n}^3\varepsilon^{-1})$

Approximation bound continued

- We established that $\sum_{i \in S} v_i \geq \sum_{i \in S} \tilde{v}_i - nb$
- Since $\sum_{i \in S} \tilde{v}_i \geq \tilde{v}_j = 2\varepsilon^{-1}nb$,

$$\sum_{i \in S} v_i \geq 2\varepsilon^{-1}nb \; - nb = (2\varepsilon^{-1} - 1)nb$$

- For $\varepsilon \leq 1$, $2 - \varepsilon \geq 1$, thus,

$$nb \leq (2 - \varepsilon)nb \; \leq \varepsilon \sum_{i \in S} v_i$$

- Leading finally to

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S} v_i + nb \leq (1 + \varepsilon) \sum_{i \in S} v_i$$

Approximation algorithms

- Some NP-hard problems can be approximated within a constant factor
- Some (like knapsack) can be approximated even better
  - Within a factor of $1 + \varepsilon$ where we can pick the value of ε
- Some can't be approximated within even a constant factor
  - Unless P = NP

Upcoming

Next time…

- There is no next time!

Reminders

- Finish Assignment 7
  - **Due tonight by midnight!**
- Review chapters 1 – 8 and 11
- Final exam:
  - Wednesday, April 24, 2024
  - 8:00 – 10:00 a.m.