CSE 421
Introduction to Algorithms

Lecture 22: Hardness and Reductions

Last time: Duality for Linear Programs

primal

Maximize $\boldsymbol{c^\top x}$

subject to

$$\boldsymbol{Ax \leq b}$$

$$\boldsymbol{x \geq 0}$$

dual

Minimize $\boldsymbol{b^\top y}$

subject to

$$\boldsymbol{A^\top y \geq c}$$

$$\boldsymbol{y \geq 0}$$

Theorem: The dual of the dual is the primal.

Theorem (Weak Duality): Every solution to primal has a value that is at most that of every solution to dual.

Theorem (Strong Duality): If the primal has a solution of finite value, then that value is equal to optimal solution of dual.

Solving Linear Programs (LPs)

- We will discuss algorithms for solving linear programs in next Wednesday's class.

- They can be solved in worst-case polynomial time. There are also is are practical algorithms (variants on the Simplex Algorithm) that solve many of them efficiently but have bad worst-case behavior. (See Math 407.)

- For us, the primary goal is to be able to express problems as LPs, when suitable.

The power of linear programming

In a sense, every polynomial-time algorithm can be expressed as polynomial-size linear program!

Theorem: [See CSE 431] Every polynomial-time algorithm can be expressed as a polynomial-size Boolean circuit with $\wedge, \vee, \neg$ gates.

Theorem: For any Boolean circuit C with output gate g can define an LP with one variable per input and one variable per gate, and objective function g that computes the output of C.

The power of linear programming

Idea: Have variables $x_1, \ldots, x_n$ for the input bits plus variables $g_1 = x_1, \ldots, g_n = x_n, g_{n+1}, \ldots, g_t$ for the gate values
Have the objective function just be the value g_t of the output gate.

NOT gate

$$g_i = \neg g_j$$

$$g_i + g_j = 1$$

OR gate

$$g_i = g_j \vee g_k$$

$$g_i \geq g_j$$
$$g_i \geq g_k$$
$$g_i \leq g_j + g_k$$
$$g_i \leq 1$$

AND gate

$$g_i = g_j \wedge g_k$$

$$g_i \leq g_j$$
$$g_i \leq g_k$$
$$g_i \geq g_j + g_k - 1$$
$$g_i \geq 0$$

Q: Does every problem have a polynomial time algorithm?

A: NO. The Halting problem is undecidable so it doesn't have an algorithm at all [Turing]

Q: If there is an algorithm for a problem is there is always one that runs in polynomial time?

A: NO. There are problems that require exponential time to solve. (See CSE 431)

Q: What about some of the problems we've seen so far?

How do we know that a problem is hard?

At this point in the quarter, you've probably at least once been banging your head against a problem…

… for so long that you began to think "there's no way there's actually an efficient algorithm for this problem."

That wasn't true for any of the problems we have assigned you to solve (so far).

- But we think that it **is** true for certain types of problems, including one where you showed how some algorithms failed to work.

- Over the next week we will look at how you can figure out that some problem you encounter is just as hard as those.

Some definitions

Defn: A problem is a set of inputs and their associated correct outputs.

- "Find a Minimum Spanning Tree" is a problem.
- Input is a graph, output is the MST.

- "Tell whether a graph is bipartite" is a problem.
- Input is a graph, output is "yes" or "no"

- "Find the 'maximum subarray sum'" is a problem.
- Input is an array, output is the number that represents the largest sum of a subarray.

Some definitions

Defn: An instance is a single input to a problem.

- A single, particular graph is an instance of the MST problem
- A single, particular graph is an instance of the bipartiteness-checking problem.
- A single, particular array is an instance of the maximum subarray sum problem.

Relative Hardness of Problems

- Want to ***compare*** the hardness of problems
  - Want to be able to say

"Problem **B** is solvable in polynomial time
⇒ problem **A** is solvable in polynomial time"

"Problem **B** is at least as hard as problem **A**"

Polynomial Time Reduction

Defn: We write $A \leq_P B$ iff there is an algorithm for A using a 'black box' (subroutine or method) that solves B that

- uses only a polynomial number of steps, and
- makes only a polynomial number of calls to a method for B.

Theorem: If $A \leq_P B$ then a poly time algorithm for $B \Rightarrow$ poly time algorithm for A

Proof: Not only is the number of calls polynomial but the size of the inputs on which the calls are made is polynomial!

Corollary: If you can prove there is no fast algorithm for A, then that proves there is no fast algorithm for B.

Intuition for "$A \leq_P B$": "B is at least as hard* as A" *up to polynomial-time slop.

Now the weird part…

We read "$A \leq_P B$" as "A is polynomial-time **reducible** to B" or "A can be **reduced** to B in polynomial time"

- It means "we can solve A using at most a polynomial amount of work on top of solving B."
- But word reducible seems to go in the opposite direction of the $\leq$ sign.

The general motivation for the terminology is:

- "To solve A we can reduce our attention from all possible things just to solving B."

- Often we have easy problem $\leq_P$ harder problem. (e.g. bipartite matching $\leq_P$ flow)
- Sometimes we can show general case $\leq_P$ special case (e.g. stable matching)
  - In this case we really use the extra polytime work we're allowed.

Some Previous Examples

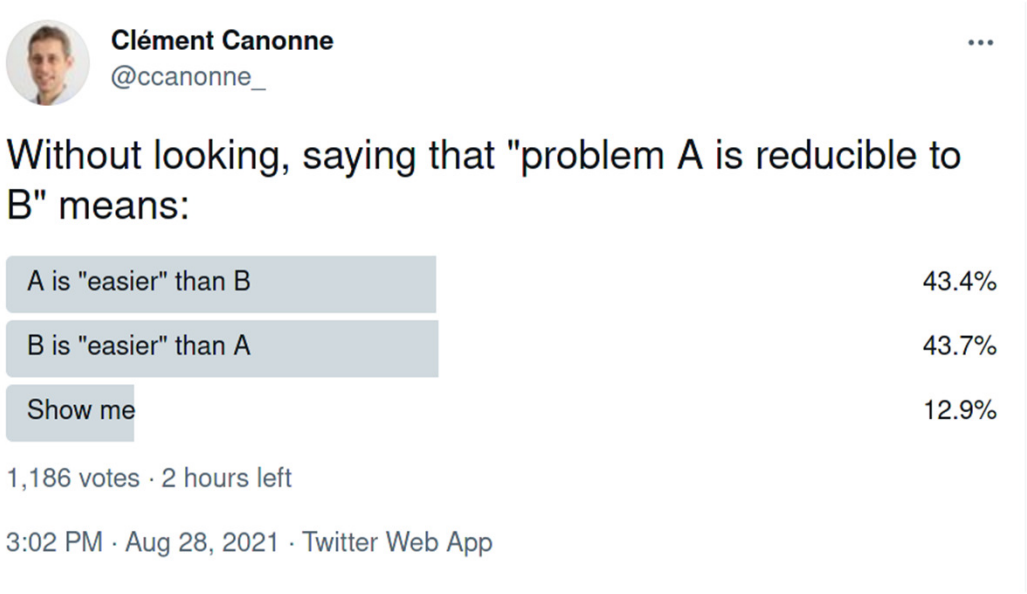
- On Homework 1, you reduced “stable matchings with different numbers of applicants and jobs with only some unacceptable“ to “[standard] stable matching”.

- On Homework 2, you (might have) reduced
  “labelling bear photographs” to “2-coloring”.

- We reduced “Bipartite Matching” to “Network Flow”.

Getting the wording right

Lots of people mess this up!



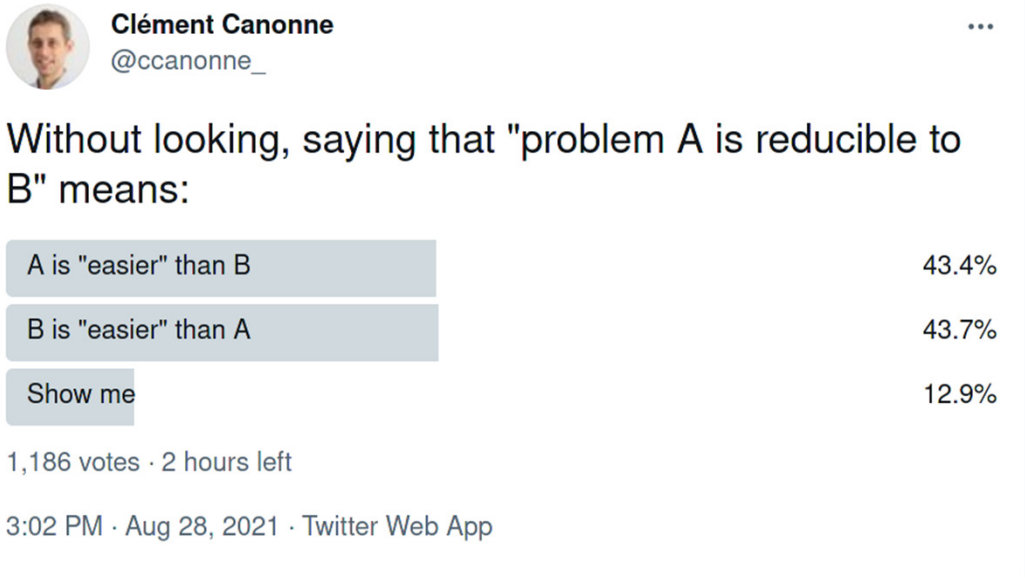
A Reduced Joke: Engineers vs Mathematicians

- An engineer
  - is placed in a kitchen with an empty kettle on the table and asked to boil water
    - she fills the kettle with water, puts it on the stove, turns on the burner and boils water.
  - she is next confronted with a kettle full of water sitting on the counter and asked to boil water
    - she puts it on the stove, turns on the burner and boils water.

- A mathematician
  - is placed in a kitchen with an empty kettle on the table and asked to boil water;
    - she fills the kettle with water, puts it on the stove, turns on the burner and boils water.
  - she is next confronted with a kettle full of water sitting on the counter and asked to boil water
    - she empties the kettle, places the empty kettle on the table and says,
      "I've reduced this to an already solved problem".

Getting the wording right

Lots of people mess this up!



Tl;dr check the direction you're going every time. It's going to take a while to be intuitive.

Decision Problems

Defn: A decision problem is a problem that has a "**YES**" or "**NO**" answer.

- A correct algorithm has a Boolean return type

Example: Is this polytope empty?

- Problems can be rephrased in terms of very similar decision problems.
  - Instead of "Find the shortest path from s to t"
    ask "Is there a path from s to t of length at most k?"
  - Can do binary search to find exact value.
  - If a problem is easy then all of its individual output bits must be easy
  - If a problem is hard then at least one of its output bits must be hard.

A Special Kind of Polynomial-Time Reduction

We will often use a restricted form of $A \leq_P B$ often called a Karp or many-one reduction...

Defn: $A \leq_P^1 B$ iff there is an algorithm for A given a black box solving B that on input x that

- Runs for polynomial time computing $y = f(x)$
- Makes 1 call to the black box for B on input y
- Returns the answer that the black box gave

We say that the function f is the reduction.

Let's do a reduction

4 steps for reducing (decision problem) A to problem B

1. Describe the reduction itself
   - i.e., the function that converts the input for A to the one for problem B.
2. Make sure the running time would be polynomial
   - In lecture, we'll sometimes skip writing out this step.
3. Argue that if the correct answer (to the instance for A) is **YES**, then the input we produced is a **YES** instance for B.
4. Argue that if the correct answer (to the instance for A) is **NO**, then the input we produced is a **NO** instance for B.

Let's do a reduction

4 steps for reducing (decision problem) A to problem B

1. Describe the reduction itself
   - i.e., the function that converts the input for A to the one for problem B.
2. Make sure the running time would be polynomial
   - In lecture, we'll sometimes skip writing out this step.
3. Argue that if the correct answer (to the instance for A) is **YES**, then the input we produced is a **YES** instance for B.
4. Argue that if the input we produced is a **YES** instance for B then the correct answer (to the instance for A) is **YES**.

Contrapositive

Reduce 2Color to 3Color

Defn: A undirected graph $G = (V, E)$ is k-colorable iff
we can assign one of k colors to each vertex of V s.t. for $(u, v) \in E$, their colors, $\chi(u)$ and $\chi(v)$, are different.
"edges are not monochromatic"

2Color: Given: an undirected graph G
Is G 2-colorable?

3Color: Given: an undirected graph G
Is G 3-colorable?

2Color $\leq_P$ 3Color

- Given a graph G figure out whether it can be 2-colored, by using an algorithm that figures out whether it can be 3-colored.

Usual outline:

- Transform G into an input for the **3Color** algorithm
- Run the **3Color** algorithm
- Use the answer from the **3Color** algorithm as the answer for G for **2Color**

Reduction

If we just ask the **3Color** algorithm about G, if G is 3-colorable but not 2-colorable it will give the wrong answer because it has the 3rd color available.

Idea: Add extra vertices and edges to to **G** to force the 3rd color to be used there but not on G

Reduction f: Add one extra vertex v and attach it to **everything** in G.

Write $H = f(G)$.

(f is clearly polynomial time computable.)

Let's do a reduction

4 steps for reducing (decision problem) A to problem B

1. Describe the reduction itself
   - i.e., the function that converts the input for A to the one for problem B.
2. Make sure the running time would be polynomial
   - In lecture, we'll sometimes skip writing out this step.
3. Argue that if the correct answer (to the instance for A) is **YES**, then the input we produced is a **YES** instance for B.
4. Argue that if the input we produced is a **YES** instance for B then the correct answer (to the instance for A) is **YES**.

Correctness

Two statements to prove (two directions):

If G is a **YES** for **2Color** (G is 2-colorable) then H is a **YES** for **3Color** (H is 3-colorable)

Suppose G is 2-colorable: G has a 2-coloring χ so edges of G have different colored endpoints. We get a 3-coloring of H by using χ for all the copies of original vertices of G and a 3rd color for the extra vertex v: Original edges of G in H have different colored endpoints; the extra edges too. So H is 3-colorable.

If H is a **YES** for **3Color** (H is 3-colorable) then G is a **YES** for **2Color** on (G is 2-colorable)

Suppose H is 3-colorable: Consider a 3-coloring χ' of H. Consider the extra vertex v in H that was added to G. For every vertex u of G, we have an edge (u, v) so $\chi'(u) \neq \chi'(v)$. This means that every vertex u of G is colored with one of the two colors other than $\chi'(v)$. So we can use χ' as a 2-coloring of G since all those edges had different colored endpoints in H. So G is 2-colorable.

Write two separate arguments

The two directions we covered actually prove an if and only if.

To make sure you handle both directions, I **strongly** recommend:

- Always do two separate proofs! (Don't try to prove both directions at once, don't refer back to the prior proof and say "for the same reason". There are usually subtle differences.)
- Don't use contradiction! (It's easy to start from the wrong spot and accidentally prove the same direction twice without realizing it.)

Another proof of 2Color $\leq_P$ 3Color

We had an $O(n + m)$ time algorithm for **2Color** based on BFS.

Simply solve the **2Color** problem without making any calls to a **3Color** method!

Two Simple Reductions

Independent-Set:

Given a graph $G = (V, E)$ and an integer k

Is there a $U \subseteq V$ with $|U| \geq k$ such that no two vertices in U are joined by an edge? (U is called an independent set.)

Clique:

Given a graph $G = (V, E)$ and an integer k

Is there a $U \subseteq V$ with $|U| \geq k$ such that every pair of vertices in U is joined by an edge? (U is called a clique.)

Claim: **Independent-Set $\leq_P$ Clique**

Independent-Set $\leq_P$ Clique

Given:

- (G, k) as input to **Independent-Set** where $G = (V, E)$

Use function f that transforms (G, k) to (G', k) where

- $G' = (V, E')$ has the same vertices as G but E' consists of **precisely** those edges on V that are not edges of G.

graph complement

From the definitions, U is an independent set in G

$\Leftrightarrow$ U is a clique in G'

Easy to check both directions given this...

Clique $\leq_P$ Independent-Set

Given:

- (G, k) as input to **Clique** where $G = (V, E)$

Use function f that transforms (G, k) to (G', k) where

- $G' = (V, E')$ has the same vertices as G but E' consists of **precisely** those edges on V that are not edges of G.

From the definitions, U is an clique in G

$\Leftrightarrow$ U is an independent set in G'

Easy to check both directions given this...

Another Reduction

Vertex-Cover:

Given a graph $G = (V, E)$ and an integer k

Is there a $W \subseteq V$ with $|W| \leq k$ such that every edge of G has an endpoint in W? (W is a vertex cover, a set of vertices that covers E.)

Claim: **Independent-Set $\leq_P$ Vertex-Cover**

Lemma: In a graph $G = (V, E)$ and $U \subseteq V$

U is an independent set $\Leftrightarrow$ $V - U$ is a vertex cover

Reduction Idea

Lemma: In a graph $G = (V, E)$ and $U \subseteq V$

U is an independent set $\Leftrightarrow$ $V - U$ is a vertex cover

Proof:

($\Rightarrow$) Let U be an independent set in G

Then for every edge $e \in E$,
U contains at most one endpoint of e

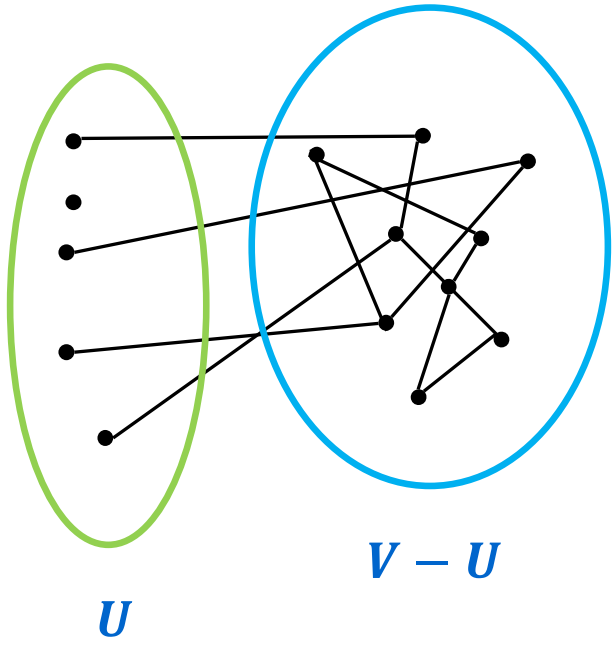
So, at least one endpoint of e must be in $V - U$

So, $V - U$ is a vertex cover

($\Leftarrow$) Let $W = V - U$ be a vertex cover of G

Then U does not contain both endpoints of any edge
(else W would miss that edge)

So U is an independent set ■



Reduction for Independent-Set $\leq_P$ Vertex-Cover

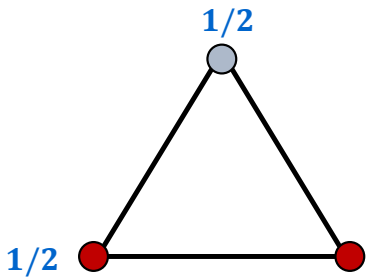
- Map (G, k) to $(G, n - k)$
  - Previous lemma proves correctness
- Clearly polynomial time
- Just as for Clique, we also can show
  - **Vertex-Cover $\leq_P$ Independent-Set**
    - Map (G, k) to $(G, n - k)$

Vertex Cover

Given: Undirected graph $G = (V, E)$

Find: smallest set of vertices touching all edges of G.

Doesn't work: To define a set we need $x_v = 0$ or $x_v = 1$



LP minimum $= 3/2$

Vertex Cover minimum $= 2$

Natural Variables for LP:

x_v for each $v \in V$

Minimize $\sum_v x_v$

subject to

~~$0 \leq x_v \leq 1$~~ for each node $v \in V$

$x_u + x_v \geq 1$ for each edge $\{u, v\} \in E$

This LP optimizes for a different problem: "**fractional vertex cover**". x_v indicates the fraction of vertex v that is chosen in the cover.