The Theory of Computational Complexity

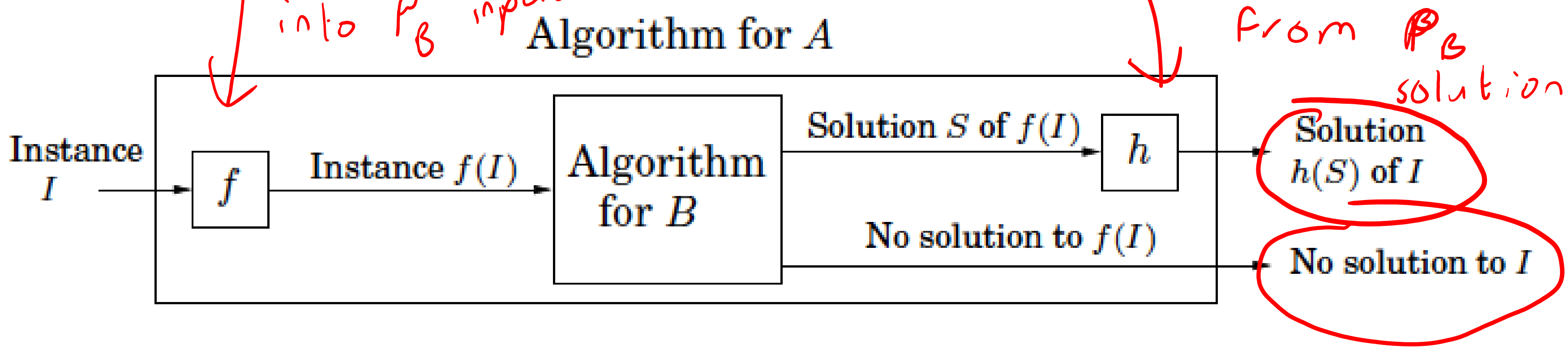
Reductions

$A \to B$

- Problem A reduces to Problem B if you can convert every instance of Problem A to an instance of Problem B, and convert the solution to Problem B back to the original solution, in *polynomial time*.

convert P_A input into P_B input

extract P_A solution from P_B solution



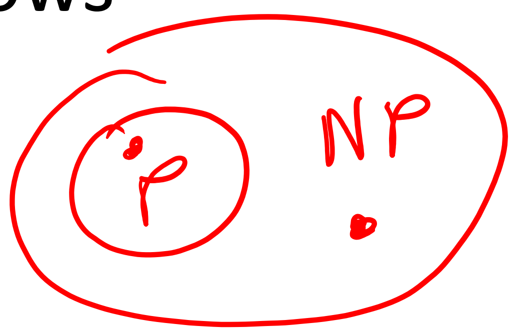
- Is reduction transitive?

Reductions

$A \rightarrow B$

Problem A's complexity class is less than or equal to Problem B's complexity class

- Why are reductions useful?
- If we have an poly. time algorithm for Problem B, and can convert back and forth in polynomial time, then we can solve Problem A in polynomial time!
- Example: Bipartite matching reduces to network flows

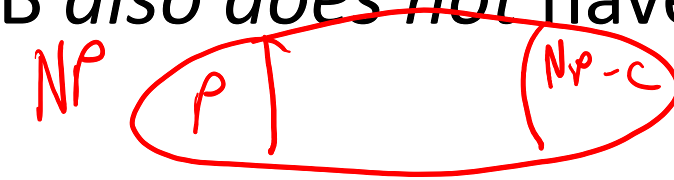


NP-Completeness

$A \leq_c B$

$B \to A$ $A \to B$

- Tricky use of reductions: If Problem A *does not* have a polynomial time solution, and it reduces to Problem B, then Problem B *also does not* have a polynomial time solution!



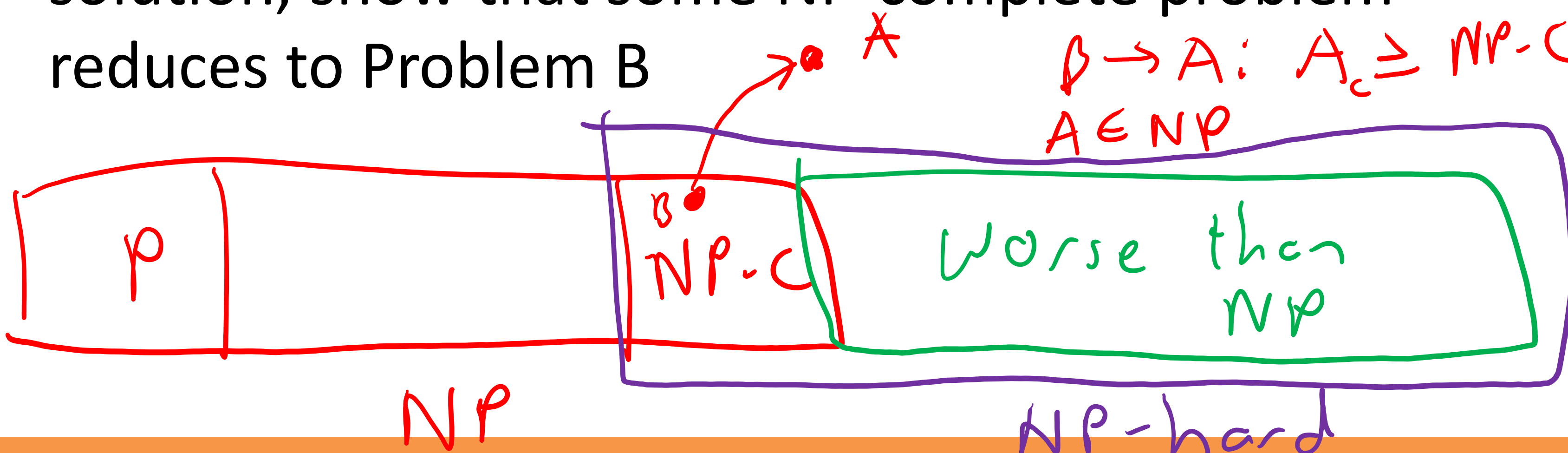
- A search problem is NP-complete if *every other problem in NP reduces to it* can
- In other words: if you can solve an NP-complete problem quickly, you can solve anything in NP quickly!

show that an NP-comp. prob. reduces to it

- A problem is NP-hard if it is at least as hard as the hardest prob. in NP

Reduction Strategy

1. Direction of reduction depends on your goal:
   - If you have a polynomial algorithm for Problem A, and want to find one for Problem B, show that B reduces to A
   - If you want to show that Problem B has no polynomial solution, show that some NP-complete problem reduces to Problem B



Reduction Strategy

2. When reducing Problem A reduces to Problem B, show the following: 0. figure out reduction (f, h)
   1. Any instance of Problem A can be converted to an instance of Problem B in polynomial time
   2. A solution to the converted instance can be converted back to a solution for Problem A in polynomial time
   3. If Algorithm B finds a solution to the converted instance, it corresponds to an actual solution to the Problem A instance

      correctness
   4. If the Problem A instance has a solution, then Algorithm B is able to find a solution to the converted instance

Showing that a Problem is NP-Complete

$A \rightarrow B$

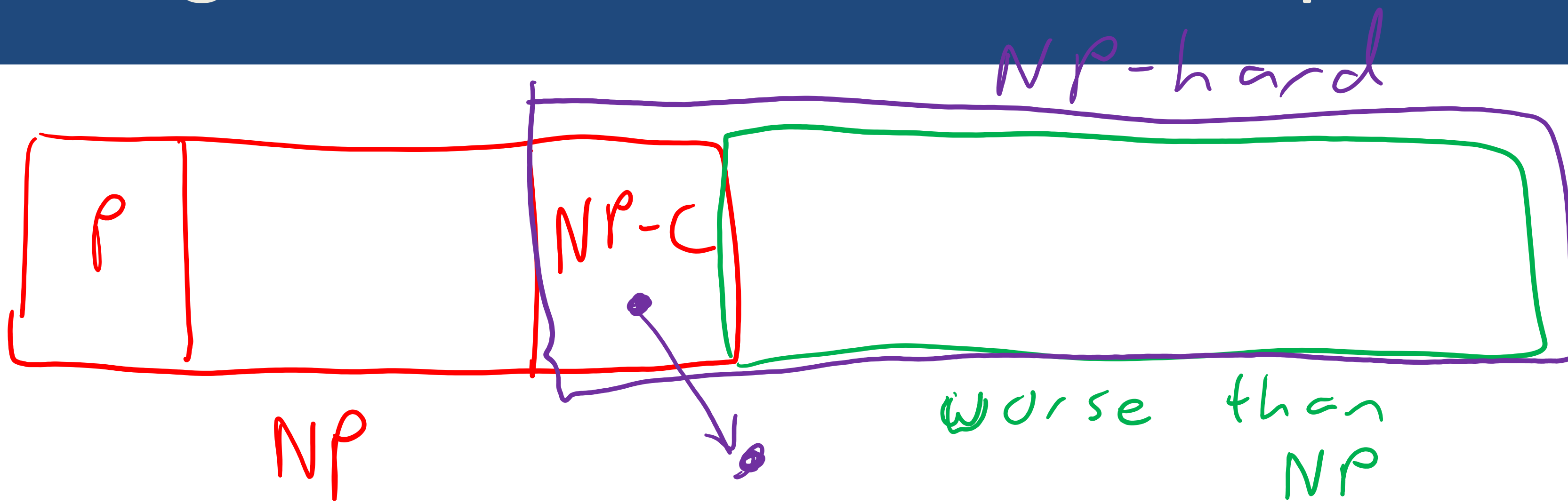
1. Pick a problem that is known to be NP-complete (we have seen some examples- 3SAT is common)
2. Show that the NP-complete problem reduces to your problem A

   3SAT → A : A is NP-hard
3. Show that your problem has a solution that can be verified in polynomial time (problem is in NP)

HP → HC (in class)

HC → HP (you can try!)

Showing that a Problem is NP-Complete

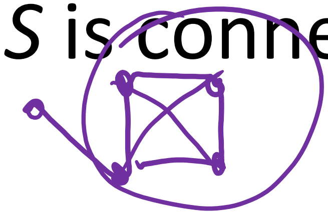


to show problem A is NP-complete, show that it is in NP-hard and in NP

halting problem
↓
given an arbitrary piece of code, will it terminate?

In-Class Exercise: MaxClique

- MaxClique: A clique in a graph is a set of nodes *S* such that every node in *S* is connected to every other node in *S*



- Given a graph, find ~~does it have a~~ clique of size at least *k*?
- Show that MaxClique is NP-complete ~~hard~~ by reducing $\boxed{\text{3SAT}}$ to MaxClique

$$(x \lor y \lor z) \land (\bar{y} \lor \bar{z})$$

$x = T$ $z = T$

$y = F$

In-Class Exercise: MaxClique

3SAT → Max Clique

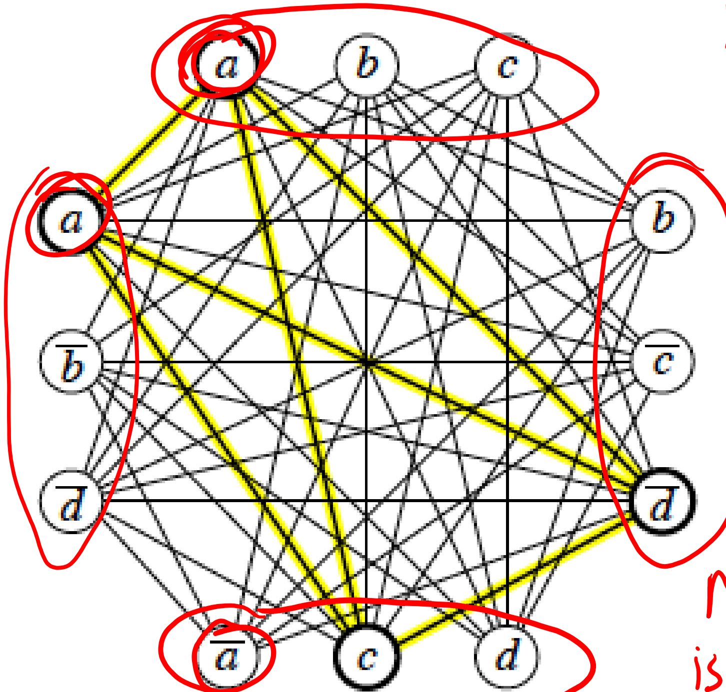
$$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$$

1. show that 3SAT → MC. (Show that MC is NP-hard)

k = #clauses

f: one set of 3 nodes for each clause.

draw an edge between all pairs of nodes except those in same clause, and no connections b/w nodes and their negations



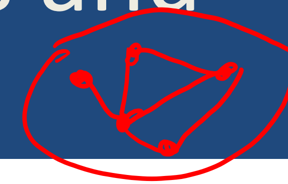
a = T

c = T

d = F

2. show that MC ∈ NP. This is easy - just check that output set is fully-conn. ~~and~~ + has ≥ k nodes

In-Class Exercise: Spanning Trees and Hamiltonian Path



* within the tree

- Degree-restricted spanning tree (DRST): A DRST is a spanning tree such that every node has degree *at most k*,* for some input value of *k*. Your task is to ~~determine whether a graph has~~ Find a DRST, for a given value of *k*.
- Hamiltonian Path: ~~Does the graph contain~~ Find a path that visits every node in the graph exactly once (note: no *s* and *t* this time).
- Show that the DRST problem is NP-Complete by reducing Hamiltonian Path to DRST.

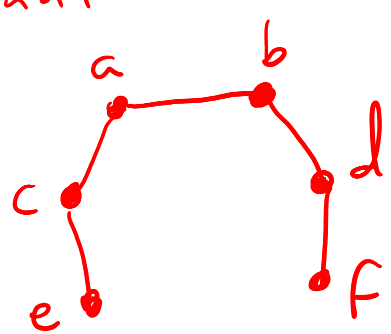
HP → DRST

In-Class Exercise: Spanning Trees and Hamiltonian Path

1. Show that DRST is NP-hard by showing HP → DRST

f: use the same graph, set $k=2$

then the DRST output by black box is itself a HP. If you have a tree where max degree is 2, then that tree is a path! Branching requires degree ≥ 3 for some node.



2. Show that DRST $\in$ NP.

To check output, check that each node has degree $\leq k$, check that all nodes included, check that tree is connected without cycles

In-Class Exercise: Spanning Trees and Hamiltonian Path

To show that DRST is NP-complete, we first show that it is NP-Hard. To do this, we reduce HP → DRST. Suppose we are given graph G as input to HP. Use the same graph as input to DRST and set k=2. Then return the output from DRST algorithm as a sol'n to HP problem. It's obvious that the reduction runs in polynomial, because all we're doing is setting k=2. This reduction works, because if we set k=2, then the DRST is a path - there is no branching possible. Because it doesn't contain

In-Class Exercise: Spanning Trees and Hamiltonian Path

cycles, and spans every node, it is thus a HP.
Thus, HP → DRST, so DRST is NP-hard.

Next, to show that DRST ∈ NP, we must show that solutions can be verified in polynomial time. To check whether a set of edges is indeed a DRST, we have to ~~check~~
(1) iterate over all edges in the set & confirm that each node appears at most k times,
(2) check that all nodes are included,
(3) check that it is connected (DFS) and has no cycles (DFS). This runs in polynomial time.