CS 3510: Design & Analysis of Algorithms (Section B)

Exam 4 (Complexity Theory) Study Guide

Professor Abrahim Ladha

1 Recommended Problems:

8.1, 8.3, 8.4, 8.8, 8.9, 8.10, 8.13, and 8.14

2 Topic List:

2.1 Introduction to NP

Decision Problem: A decision problem is a problem where given an input I, we output either True or False (yes/no) if there is a solution that matches its requirements.

Suppose that we have a witness W - that is, some entity that can observe the decision problem running and can see a solution that the decision problem uses to output True. Then, we divide the decision problems into two classes:

Verifiable decision problems: We can verify that the solution given from W is correct in polynomial time relative to input I.

Hard Verifiable decision problems: There is no known polynomial time verification to the solution from W; our current understanding tells us that we need exponential time for verification.

We denote **NP** as the class of all verifiable decision problems. In the next section we'll look at what we do with the "Hard" verifiable problems.

Note: There's also the concept of a "Search" problem, which is defined as follows: A problem is a search problem if it meets the following conditions: (1) for an input I, its algorithm outputs a solution S or specifies that there is no solution, and (2) for a candidate solution S, it is possible to verify its correctness in polynomial time relative to the size of input I.

We could also define the class **NP** as the set of all search problems. Both definitions yield the same result! The main concept to understand here is that NP doesn't tell us anything about the complexity to *solve* the problem, but rather to verify the solution for that problem is correct.

2.1.1 MST Decision Problem

Given input $I : G(V, E), r$, output whether there exists a spanning tree of G with weight $\leq z$.

Suppose we find a candidate subgraph S for this problem. We can verify it using the following method:

1. Traverse the subgraph and ensure that all vertices are spanned. This takes $O(|V| + |E|)$ time.
2. Sum the weights of all edges in G and check that it is $\leq z$.

Summing all the runtimes, we get $O(|V| + |E|)$, which is polynomial time. Therefore MST is in NP!

The search problem variant of MST would be the following: *Output the MST for the Graph G.*

We can utilize a similar approach here: check if the solution spans all vertices, and then run Kruskal's algorithm to verify the cost of the candidate solution is the same as Kruskal's output MST.

2.1.2 K-coloring

Given input $I : G(V, E)$ and $k \in \mathbb{N}$, output whether there exists a k-coloring of G. That is, output "YES" if there exists an assignment σ for all vertices such that $\forall e = (u, v) \in E$, $\sigma(u) \neq \sigma(v)$ or output that it doesn't exist.

Suppose we find a candidate assignment S. Then, we could iterate through each edge (u, v) in G and check that $\sigma(u) \neq \sigma(v)$. This takes $\mathcal{O}(|E|)$, so we've just shown that K-coloring is in NP.

2.1.3 Knapsack 0/1

Given as input a set of items $\{1 \ldots n\}$ with corresponding values $\{v_1 \ldots v_n\}$ and weights $\{w_1 \ldots w_n\}$, and a capacity B, output "YES" if there exists a collection of items that maximizes total value and keeps total weight less than or equal to B, or output that it does not exist.

Suppose we find a candidate collection S. It's easy enough to show that the total weight is $\leq B$ by iterating through the items and summing their weights. However, how can we show that the total value is maximized? We'd have to run Knapsack-search and find if the total value is maximized. However, Knapsack-search is only *pseudo-polynomial* time - it is in reality exponential time! The runtime of Knapsack-search is $\mathcal{O}(nB)$, but since it takes $\log_2(B)$ bits as the input size to represent the number B, the runtime is actually $\mathcal{O}(n2^{\log_2(B)})$ which is exponential on input size. As a result, we cannot conclude Knapsack 0/1 is in NP.

2.1.4 SAT

Given as input a function $f : \{x_1, \ldots, x_n\} \to \{0, 1\}$ in conjunctive normal form (CNF), output "YES" if there exists an assignment $\{x_1, \ldots, x_n\}$ such that $f = 1$ or output that it doesn't exist.

Let's say we have a function f in CNF (disjunctions inside the clauses, and conjunctions joining the clauses):

$$(x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_4 \vee x_5 \vee x_6) \wedge (x_3 \vee \bar{x}_5 \vee x_7)$$

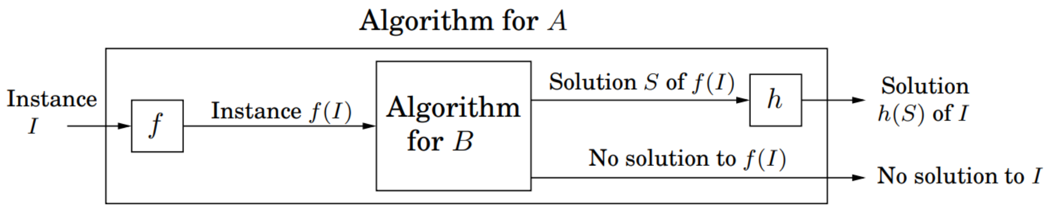
and an candidate assignment S: $\{x_1 \ldots x_7\}$. To verify S is correct, we just have to plug in the values into f and verify that the result is 1. This takes $O(nm)$ time where n is the number of literals and m is the number of clauses.

The subset of NP problems where we know of a solution that *solves* the problem in polynomial time is known as **P**. We have that P $\subseteq$ NP. For example, we know that MST is in P because of the existence of Kruskal's and Prim's algorithms.

2.2 Reductions

How do we classify problems that are not in NP? That is, what can we do with search problems that are not verifiable in polynomial time, or with hard verifiable decision problems? Before we get there, let's review the concept of *reductions*.

A **reduction** from problem A to problem B shows that B is *at least as hard* of a problem to solve as A. We take an input, or *instance* I in problem A and convert it to an input I' of problem B using a polynomial time function f. Then, we retrieve a solution S' by running any algorithm for B, and convert it back to a solution S of problem A using a polynomial time algorithm h. The below figure from the textbook visualizes this process:



We must show that there is a solution to input I for A if and only if there is a solution to input I' for B to complete the reduction. Essentially, we show that A is just a special case of B, and so therefore B must be at least at hard as A.

There are three steps to complete a reduction from A to B (denoted as $A \leq_p B$):

1. Describe the function f, which is transformation of the instance I for problem A to instance $f(I) = I'$ for problem B and show that it runs in polynomial time.

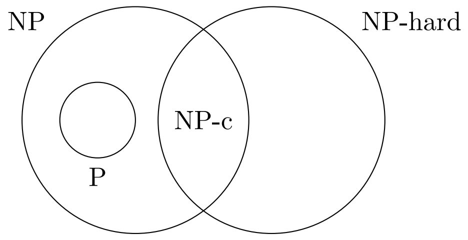
2. Describe the function h, which is the conversion of the solution to instance I' for problem B back into a solution for instance I in problem A.

3. Show that there is a solution for I in A if and only if there is a solution for I' in B. This step can be done multiple ways. To prove that $a \leftrightarrow b$, you must show that $a \rightarrow b$ and that $b \rightarrow a$. However, you can choose to substitute the contrapositive for either statement. For example, proving that $a \rightarrow b$ and $\neg a \rightarrow \neg b$ is sufficient as the latter is the contrapositive of $b \rightarrow a$. This does't need to be a very rigorous proof; showing a 1-to-1 correspondence both ways is sufficient, as long as we understand that you are showing correctness in both directions!

A problem C is **NP-hard** if there exists a reduction $A \leq_p C$ from *all* problems $A \in$ NP. Therefore, C is at least hard as all problems in NP. Therefore, you can prove a problem D is NP-hard by finding a reduction $H \leq_p D$ for some known NP-hard problem H.

A problem C is **NP-complete** if it is in NP *and* NP-hard. This means that that NP-complete problems are the hardest known problems in NP, as all other NP problems reduce to it.

Let's look at a visualization of all these classes: P, NP, NP-hard, NP-complete (NP-c).



P ? NP
One of the longest still unsolved problems in computer science is the following: is P = NP, or can we conclude that P $\subset$ NP? As it happens, if you can prove that just one NP-hard problem is solvable in polynomial time, then you actually show P = NP! This is from our definition of NP-hard, that all problems in NP are reducible to any problem in NP-hard. If we show that one problem in NP-hard is actually in the class P, then all problems in NP are reducible to a P problem, proving that all NP problems would actually be solvable in polynomial time.

To show that a problem is NP-complete, we just have to add one step before the reduction steps: we should show the problem is in NP, which we reviewed in the previous section. The reduction from an existing NP-hard problem will prove that the problem is NP-hard. So if we have that a problem is both in NP and NP-hard, it must be NP-complete.

We use the following important fact in the rest of the unit: **SAT is NP-complete**. In the next sections, we'll be reviewing the reductions to other NP-problems introduced in lecture. At the end of section 2.4, we'll have a set of NP-hard problems that you can use on the exam to come up with a reduction that proves that a given problem is NP-hard.

2.3 3-SAT

Let's introduce a variant of SAT, **3-SAT**, which is also NP-complete. 3-SAT is a simple modification of SAT. Instead of clauses in having any number of literals, we restrict the number of literals in each clause to *at most 3*. Let's formulate a reduction from SAT to 3-SAT to prove that 3-SAT is in fact NP-complete. Here's an example of an input function to 3-SAT:

$$(x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_2 \vee x_4 \vee x_5) \wedge (\bar{x}_3 \vee \bar{x}_4 \vee x_6)$$

Before the reduction, let's show that 3-SAT is in NP. Given an assignment S of 3-SAT, we can plug in the assignment for each literal and check that the result is equal to 1. This takes just $\mathcal{O}(3m) = \mathcal{O}(m)$ time where m is the number of clauses, as we know that we compute at most three logical operations per clause, which is a constant. As this is polynomial time in input size, 3-SAT is in NP.

Reduction:

1. Let's reduce SAT to 3-SAT. Let's look at an example input of SAT:

$$(x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_4 \vee x_5 \vee x_6)$$

   To convert this to an input of 3-SAT, we can actually add boolean variables that do not affect the satisfiability!

$$(x_1 \vee x_2 \vee y_1) \wedge (\bar{y}_1 \vee x_3 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_4 \vee y_2) \wedge (\bar{y}_2 \vee x_5 \vee x_6)$$

   In the general case, for each clause in SAT $x_1 \vee \cdots \vee x_k$, we add new variables $y_1 \dots y_{k-3}$. A larger clause, for example $x_1 \vee \cdots \vee x_7$ would look like this:

$$(x_1 \vee x_2 \vee y_1) \wedge (\bar{y}_1 \vee x_3 \vee y_2) \wedge (\bar{y}_2 \vee x_4 \vee y_3) \wedge (\bar{y}_3 \vee x_5 \vee y_4) \wedge (\bar{y}_4 \vee x_6 \vee x_7)$$

   The addition of these variables does not affect the satisfiability! This means that no matter the assignment of y_i $\forall i$, the assignments of $x_1 \dots x_n$ will result in the same truth value. To convince yourself of this fact, try this on a smaller example such as a clause of size 4. No matter what the truth value of the new variable is, the result will always be the same as the original clause because of he properties of conjunctions and disjunctions (ANDs and ORs). This is our function f - converting each clause of the input to a group of 3-SAT clauses. This function takes $\mathcal{O}(mn)$ time where m and n represent the number of clauses and number of literals respectively. Therefore this input conversion function runs in polynomial time.

2. After we run 3-SAT on our transformed input $f(I)$, we receive an assignment S of both $x_1 \dots x_n$ as well as the new variables y, or an output that no valid assignment exists. If no such assignment exists, we return that no assignment exists for SAT. If there is a valid assignment, we simply delete the assignment of the new variables y and return the assignment of the original literals.

3. $a \rightarrow b$. There is a solution for input I for SAT if there is a solution for $f(I)$ in 3-SAT. This is because the truth values of the additional variables do not affect the assignment of the original values.

   $\neg a \rightarrow b$ (contrapositive of $b \rightarrow a$). There is no solution for input I for SAT if there is no solution for $f(I)$ in 3-SAT. This is because, if there is no valid assignment for the combined variables $x_1 \dots x_n$ and y, then, since our variables y were constructed to maintain the same truth values as the original clauses, there cannot be a valid assignment for the original clauses.

   Therefore, there is a solution for I of SAT if and only if there is a solution for $f(I)$ of 3-SAT. As a result, since SAT is NP-hard, 3-SAT is NP-hard.

And we've shown 3-SAT is NP-Complete!

It may seem a bit redundant to include correctness in all 3 parts of your reduction. As a result, we do not require that much explanation in all three parts! As long as an analysis of correctness is present in your reduction, you're good to go. Additionally, your if-and-only-if proof does not need to be very rigorous - just remember to explain it in both directions, $a \to b$ and $b \to a$, substituting contrapositives as you see fit.

2.4 Independent Set, Clique, Vertex Cover, Subset-Sum

2.4.1 Independent Set

Now let's review the graph problems we covered in lecture. Let's start with Independent Set and show how we can reduce one of the NP-Hard problems we know of (SAT and 3-SAT so far!) to it.

Independent Set: Given as input a graph $G = (V, E)$ and an integer goal g, output "YES" if there exists a subset of the vertices S such that $|S| \geq g$ and that for all pairs of vertices in S, there exists no edge connecting them. Output "NO" otherwise.

Independent Set is in NP, since we can check that $|S| \geq g$ and iterate through the edges in G and confirm that the source and destination of the edge are both not in S in $\mathcal{O}(|V| + |E|)$ time.
Now let's show that Independent Set is NP-hard:

1. We will reduce 3-SAT to Independent Set. Given as input a set of clauses $C_1, C_2, \ldots C_m$ to 3-SAT, build a graph G as follows:

   (a) Represent each literal in the function as a vertex in G, including the negation of literals that appear in the function.

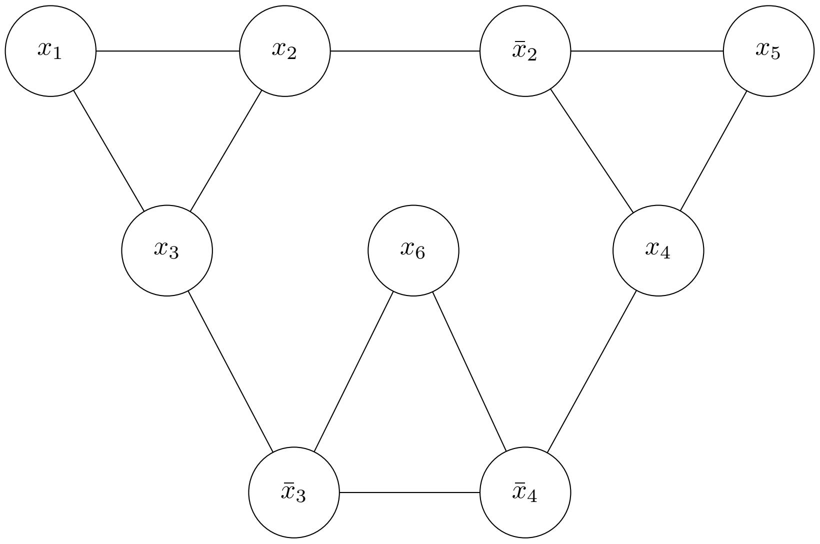
   (b) For each clause C_i, add edges between each pair of literals in the clause.

   (c) Add edges between each literal and its negation, if it exists in the function.

   Let's see an example of this in an input to 3-SAT:

$$(x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_2 \vee x_4 \vee x_5) \wedge (\bar{x}_3 \vee \bar{x}_4 \vee x_6)$$

   Our graph will look like this:



   We take this graph as input into Independent Set, setting goal $g = m$, where m is the number of clauses. Our function f converts input I of 3-SAT into graph $f(I)$ for Independent Set. This takes $O(3m) = O(m)$ time, as we have at most three literals per clause to include as vertices into our graph. Therefore our function is polynomial time relative to input size.

2. Given a solution S to our graph G in Independent Set, we have a set of m vertices for which there exists no edge connecting them, or no solution. If there is no solution, we output no solution for 3-SAT. If there is a solution, we can simply assign the literals represented by the vertices in S to true, and set the rest to false. This takes $O(n)$ time, as we would have to iterate through all literals to make the assignment.

3. $a \to b$: If there is a solution for input I to 3-SAT, there must be a solution to input $f(I) = G$ to Independent Set. This is because a valid assignment of 3-SAT must have at least one literal evaluating to True in each clause by the definition of conjunction. Therefore, if we represent the graph where all literals in clauses are vertices with edges connecting each other, there must be a valid way to select at least one vertex from each clause that is not connected to the others. This is why we set $g = m$, since there must be at least m independent literals that evaluate to True.

   $\neg a \to \neg b$: If there is no solution for input I to 3-SAT, there must be no solution to input $f(I) = (G, g = m)$ to Independent Set. If there is no solution for I, then there cannot be m literals in different clauses that evaluate to True. Therefore there is no way that there can be an independent set of size m if we connect literals in the same clause using edges. Additionally, to account for the existence of inverse literals in other clauses, we add edges between them, to show that we cannot assign True to both literals. As a result, it is impossible for Independent Set to have a solution in this case.

This proves Independent Set is NP-hard. We've shown that Independent Set is also in NP, so we conclude Independent Set is NP-complete.

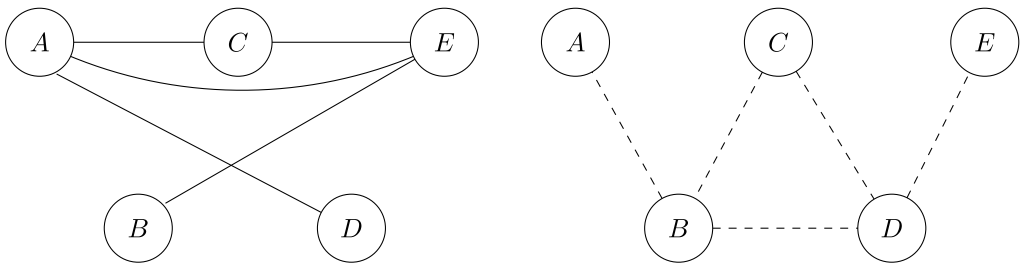
2.4.2 Clique

Clique: Given as input a graph $G = (V, E)$ and an integer goal g, output "YES" if there exists a subset of the vertices S such that $|S| \geq g$ for every pair of vertices in S, there exists an edge connecting them. Otherwise, output "NO."

Clique is in NP, since we check the size of S is at least g, and can iterate through the edges and check that both the source and destination vertices of each edge are either in or not in G in $\mathcal{O}(|V| + |E|)$ time.
The definition of Clique is pretty much the exact opposite of the definition of Independent Set! Let's use that to make a reduction:

Reduction:

1. Given as input a graph G and goal g to Independent Set, convert this to graph $G' = (V, \bar{E})$ where $\bar{E} = \{(x, y) | (x, y) \notin E\}$. Essentially, G' is a graph where all edges connecting vertices are those *not present* in the original graph. The below figures visualize this transformation:

   Additionally, we can keep the same goal g as input to Clique. Our function f to delete and add new edges takes $\mathcal{O}(|E|)$ time, which is polynomial.

2. Given a solution S of Clique, or no solution, we can restore the original solution of Independent Set by returning the exact same result! If there is no solution, we return no solution. Otherwise, we return the same subset S from Clique as the solution to Independent Set.

3. $a \to b$: If there is a solution for input $I = (G, g)$ of Independent Set, there is a solution for input $f(I) = (G', g)$ for Clique. If there is a solution S for I, then all vertices in S have no edges connecting them by defintion. This means, that by construction of G', each pair of those vertices must have edges connecting them. Therefore this subset must be a solution of Clique with goal g.

$b \to a$: If there is a solution for input $f(I) = (G', g)$ for Clique, there must be a solution for input $I = (G, g)$ of Independent Set. By the same logic as the previous step, if we have a clique in the modified graph, then once we remove the edges in the modified graph and replace with those in the original graph, it cannot be that any of the vertices in the clique have an edge connecting each other, by our construction of the modified graph. Therefore this is an independent set.

We can now conclude that Clique is NP-complete.

2.4.3 Vertex Cover

Vertex Cover: Given a graph $G = (V, E)$ and goal g, output "YES" if there exists a subset of vertices S such that $|S| \leq g$ all vertices not in S are accessible by one edge from a vertex in S. In other words, all edges in E have one endpoint in S. Output "NO" otherwise.

Vertex Cover is in NP, as we can (1) confirm that the size of S meets the goal g in $\mathcal{O}(|V|)$ time, and (2) iterate through each edge in E and ensure every edge has at least one endpoint in S in $\mathcal{O}(|V||E|)$ time. Let's prove that Vertex Cover is NP-hard. An important observation that will greatly help us with the reduction is that if you remove the Vertex Cover subset from a graph, you're left with an independent set!

Reduction:

1. Given as input a graph G and goal g to Independent Set, $f(I) = (G, |V| - g)$ where $|V| - g$ is the new goal g' for Vertex Cover!
2. Given a solution S for Vertex Cover, we can return $V \setminus S$ (the set of vertices in V not included in S) as a solution to Independent Set, or no solution if there exists no solution for the modified input to Vertex Cover.
3. $a \to b$: If there is a solution for I in Independent Set, there is a solution for $f(I)$ in Vertex Cover. If we have a set of g vertices for which there exists no edge connecting them, then all other $|V| - g$ vertices must share edges that connect the g vertices. This is the definition of a Vertex Cover, so it must be that there is a solution for $(G, |V| - g)$ for Vertex Cover.

   $b \to a$: If there is a solution to $f(I)$ in Vertex Cover, then there must be a solution to I in Independent Set. If there are $|V| - g$ vertices which cover all other vertices, then the g vertices not in S must not be connected to each other by an edge, as it is required that all edges in E have at least one endpoint in S. Therefore this must be an independent set of size g, so Independent Set will have a solution.

Vertex Cover is NP-hard and in NP, so it is NP-complete.

2.4.4 Subset Sum

Subset Sum: Given a set S of integers and a target t, is there a subset S' of S such that the sum of the elements of S' is exactly t?

Given a 3SAT boolean expression input with n variables and m clauses, create a set S with target t such that some subset of S adds to t if and only if F is satisfiable. We can now show the reduction from 3SAT to Subset Sum to show that Subset Sum is also NP-Complete!

Reduction:

1. We will construct S by having $2(m + n)$ elements where each variable/clause will have two elements in the set S.

$$S = \{\underbrace{v_1, \neg v_1, v_2, \neg v_2, \ldots, v_n \neg v_n}_{\text{corresponds to variables}}, \underbrace{z_1, \neg z_2, \ldots, z_m, \neg z_m}_{\text{corresponds to clauses}}\}$$

2. In terms of the reduction, if x_i is true, we want to include v_i in the sum (and never v_i') and if xi is false, we want to include v_i' and never v_i.

$$\left[\begin{array}{c|cccccc}
v_1 & 1 & 0 & 0 & 0 & \cdots & 0 \\
v_1' & 1 & 0 & 0 & 0 & \cdots & 0 \\
v_2 & 0 & 1 & 0 & 0 & \cdots & 0 \\
v_2' & 0 & 1 & 0 & 0 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \cdots & 0 \\
v_n & 0 & 0 & 0 & 0 & \cdots & 1 \\
v_n' & 0 & 0 & 0 & 0 & \cdots & 1 \\
\hline
t & 1 & 1 & 1 & 1 & \cdots & 1
\end{array}\right]$$

3. We can set each of the values for the various v_i, v_i', z_i, z_i' as $n+m$ digit numbers. The table on the right shows how we can define the first n dig- its. Notice that v_i and v_i' have the same 1 in each column, so Subset-Sum would not pick both v_i and v_i' since t for that column is only 1. However, we need to add additional restrictions on these numbers such that each of the clauses are satisfied as well. We have m additional digits to use, so we will encode information with them. To describe the m digits, here is an example:

$$(\neg x_1 \vee \neg x_2 \vee \neg x_3) \vee (\neg x_1 \vee \neg x_2 \vee x_3) \vee (x_1 \vee \neg x_3 \vee x_3)$$

4. In this, we have $n = 3$ variables and $m = 3$ clauses. Hence, we will have variables $v_1, v_1', v_2, v_2', v_3, v_3'$ and $z_1, z_1', z_2, z_2', z_3, z_3'$ for a total of $2(m + n) = 12$ variables.

$$\left[\begin{array}{c|cccccc}
v_1 & 1 & & & & & 1 \\
v_1' & 1 & & & 1 & 1 & \\
v_2 & & 1 & & & & \\
v_2' & & 1 & & 1 & 1 & 1 \\
v_3 & & & 1 & & 1 & 1 \\
v_3' & & & 1 & 1 & & \\
z_1 & & & & 1 & & \\
z_1' & & & & 1 & & \\
z_2 & & & & & 1 & \\
z_2' & & & & & 1 & \\
z_3 & & & & & & 1 \\
z_3' & & & & & & 1 \\
\hline
t & 1 & 1 & 1 & 3 & 3 & 3
\end{array}\right]$$

5. We define the first n columns the same as before for all of the v_i but have now introduced additional columns for the last m digits. How do we fill these values? Simply put, in the $n + j$ column for any v_i, we put a 1 if $x_i \in c_j$ and for any v_i', we put a 1 if $\neg x_i \in c_j$. For example, since $x_1 \in c_3$ and $\neg x_1 \in c_1$, c_2 and therefore the first two rows are 100001 and 100110. The remaining rows have these buffer variables z_i and z_i' for c_i. If the possible sum can be less than 3, we add 1s in these buffer positions such that we reach 3. These have been filled in as well on the table at the left. The table on the left also has 0s omitted so you can see which entries have a 1, but the remaining entries are 0.

6. In the above, we set $t = 111333$ which ensures that we are picking one of v_1 or v_1' (the first 3 digits). This is equivalent to setting each literal to either true or false. The remaining digits are 3 to ensure that the clauses are satisfied. If it is not possible to make the sum exactly 3, it would not be possible to satisfy the original boolean expression. Hence, we have shown how we can convert from SAT to Subset-Sum.

We have shown that Subset Sum is NP-Hard! Can you prove that it's NP and confirm that it's NP-Complete?

2.4.5 3-Coloring

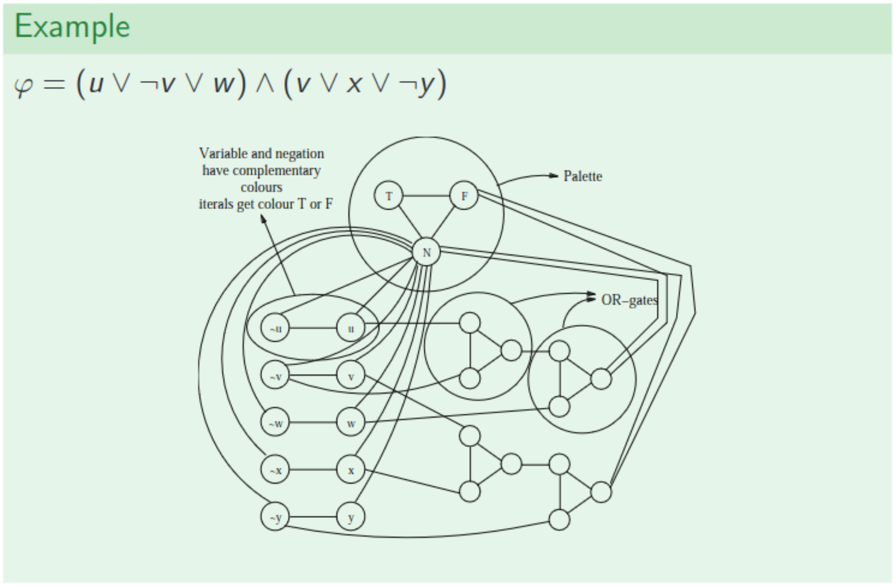
3-Coloring: Given a graph $G(V, E)$, can we color the graph in such a way that no two adjacent vertices share the same color?

3-Coloring $\in NP$: Given a coloring assignment C, we can verify that it is indeed a 3-coloring if C contains three colors total, and that every vertex's neighboring vertices share a different color. This can take $\mathcal{O}(|V|^2)$ time which is polynomial.

For this reduction, we summarize a source from https://cgi.csc.liv.ac.uk/ igor/COMP309/3CP.pdf. To see how the professor covered the reduction, please refer to the lecture notes which will be posted soon. For the reduction, we show that 3-SAT reduces to 3-coloring. Consider the input to 3-SAT with m clauses of n literals $x_1 \dots x_n$. We construct a graph G as follows:

1. Create a triangle of nodes labeled `True, False, Base`.
2. For each variable x_i, construct nodes labeled x_i and $\overline{x}_i$ connected in a triangle with the node `Base`. When G is three-colored, either x_i or $\overline{x}_i$ will get a True coloring, which will determine the assignment.
3. For each clause $C_i = (x_i \vee x_j \vee x_k)$, add an OR-gadget graph with input nodes x_i, x_j, x_k and connect the output node of this gadget graph to the `False` node and `Base`. The gadget graph works as follows:
   (a) If inputs x_i, x_j, x_k are colored False, then the output node of the OR-gadget has to be colored False.
   (b) If at least one of the inputs is colored True than the OR-gadget must be colored True.

Let's look at an example of this construction:


Example

$\varphi = (u \vee \neg v \vee w) \wedge (v \vee x \vee \neg y)$

Variable and negation have complementary colours iterals get colour T or F
Palette
OR-gates
T
F
N
~u
u
~v
v
~w
w
~x
x
~y
y

Let's show that the input to 3-SAT is satisfiable if and only if our graph G is 3-colorable:

1. If the input to 3-SAT is satisfiable, G is 3-colorable. This is because if x_i is assigned True, then we can color x_i True and $\overline{x}_i$ False in G, and for each clause $C_i = (x_i, x_j, x_k)$, at least one of the nodes x_i, x_j, x_k can be colored True. Additionally, the OR-gadget for C_i can be 3-colored such as the output is True, based on the defined properties.

2. If G is 3-colorable, then the input to 3-SAT is satisfiable. This is because we can determine truth assignment from the coloring of x_i and $\overline{x}_i$, and because it cannot be that any OR-gadget has all inputs as False. Assume to the contrary all inputs to an OR-gadget are false. Then, the output of the OR-gate would be False, but it is connected to both the `False` and `Base` nodes, which would not allow a three-coloring, as the **False** node must be colored False!

As a result, 3-coloring is at least as hard as 3-SAT, so it is NP-hard. Because 3-coloring is NP-hard and in NP, it is NP-complete.

Those are all the NP-complete algorithms we've covered in lecture! You are free to use any of these: **SAT, 3-SAT, Independent Set, Clique, Vertex Cover, Subset Sum,** and **3-coloring** when formulating reductions during the exam. Review all the above reductions to get a better understanding of how we can relate problems to each other, modify inputs to get a 1-to-1 relation with a new problem, and transfer solutions back to the original problem.

2.5 2-SAT

Another variant of SAT is 2-SAT, which is in P. Similar to 3-SAT, 2-SAT is a modification of SAT in which the clauses have at most 2 literals. Here's an example of an input function to 2-SAT:

$$(\bar{x_1}) \wedge (x_2 \vee x_3) \wedge (x_3 \vee \bar{x_4}) \wedge (\bar{x_2} \vee x_4) \wedge (x_5)$$

First let's show that 2-SAT is in NP. Given an assignment S of 2-SAT, we can plug in the assignment for each literal and check that the result is equal to 1. This takes just $\mathcal{O}(2m) = \mathcal{O}(m)$ time where m is the number of clauses, as we know that we compute at most two logical operations per clause, which is a constant. As this is polynomial time in input size, 2-SAT is in NP.

Now let's show that 2-SAT is in P. To do this, we must come up with an algorithm to solve 2-SAT in polynomial time.

Algorithm:

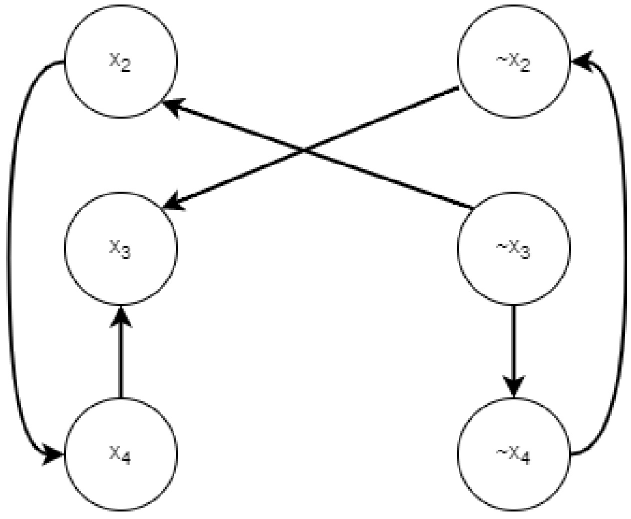
1. All clauses with exactly one literal must hold true. For the input

$$I = (\bar{x_1}) \wedge (x_2 \vee x_3) \wedge (x_3 \vee \bar{x_4}) \wedge (\bar{x_2} \vee x_4) \wedge (x_5)$$

   we set x_1 to 0 and x_5 to 1. We simplify the solution until we have a contradiction, solution, or clauses with exactly 2-literals. By doing this we are either solving the problem, or converting it into an Exact-2-SAT problem where each clause has exactly 2 literals. In example, we will transform our input I into the following CNF:

$$I' = (x_2 \vee x_3) \wedge (x_3 \vee \bar{x_4}) \wedge (\bar{x_2} \vee x_4)$$

2. Given an input in CNF with two literals on each clause, we build a directed graph $G = (V, E)$ where V represents the set of literals $\{x_1, \bar{x_1}, x_2, \bar{x_2}, x_3, \bar{x_3}, ...x_n, \bar{x_n}\}$. For each clause: $(a \vee b)$ create two edges: $\neg a \rightarrow b$ and $\neg b \rightarrow a$. The logic we are trying to represent in this graph is that if $\neg a$ is true, then it is implied that b must also be true and if $\neg b$ is true, then it is implied that a must also be true. This condition is represented by our constructed edge. For our input I', our constructed graph would look like:



   In our constructed graph, for every edge $u \rightarrow v$, if u is true, v must also be true and if u is false, v must also be false. Such a graph is known as a graph of implications.

3. Run SCC on the graph of implications. In our example, there are no strongly connected components, so the algorithm will return the input graph itself.

4. We start iterating from the sink component S (in this example the sink component is the vertex x_3). For $a \in S$, set a to be true. Check if $\neg a$ belongs to S. If it does, there is no valid 2-SAT assignment possible. Delete S and $\neg S$ from G where $\neg S$ is the strongly connected component which can be represented by a reversed and negated version of S (edges are reversed and vertices are negated).

5. Repeat until all variables are assigned a value or a contradiction occurs.

Why this algorithm works:

1. If x_i and $\neg x_i$ are in the same SCC then the corresponding input CNF is not satisfiable. This is because, if such an SCC were to exist then there will paths $x_i \to l_1 \to l_2 \to ...l_m \to \neg x_i$ and $\neg x_i \to l_m \to l_{m-1} \to ...l_1 \to \neg x_i$. Through the implications in the first statement, x_i being true would imply that $\neg x_i$ is also true (which is impossible). Similarly, through the implications in the second statement, $\neg x_i$ being true would imply that x_i is also true (which is also impossible).

2. If S is a strongly connected component of G, the set $\neg S = \neg s, s \in S$ is also a strongly connected component. Because of the way we constructed our graph, if S is a sink component, $\neg S$ must be a source component and vice versa. In step 4 of the algorithm, when we remove a sink S, we also remove this corresponding source component $\neg S$ from the graph.

Time Complexity Analysis:

The construction of this graph takes linear time or $O(V + E)$. Running the SCC algorithm also takes linear time or $O(V + E)$. Hence this algorithm runs in polynomial time thereby making 2-SAT a problem in P.

So that's the content you need to know! To best prepare for the exam, review the lectures and ensure you have a good understanding of all the concepts and algorithms we've covered in class. Going through the practice quiz and textbook questions will really help. Feel free to attempt NP textbook problems we haven't listed in the recommended problems section; other problems will definitely be more involved and/or harder, but will really help you be prepared for the quiz. Please ask any questions you have on Piazza or in TA office hours, and we'd be glad to help. Best of luck on the exam :-)!