November 22, 2022

CS120: Intro. to Algorithms and their Limitations Hesterberg & Vadhan

Problem Set 8

Harvard SEAS - Fall 2022 *Due: FRI. Nov 18, 2022 (11:59pm)*

Your name: Evan Jolley
Collaborators:
No. of late days used on previous psets: 0
No. of late days used after including this pset: 0

The purpose of this problem set is to to reinforce the definitions of $\mathsf{P}_{\mathsf{search}}$, $\mathsf{EXP}_{\mathsf{search}}$, $\mathsf{NP}_{\mathsf{search}}$, and $\mathsf{NP}_{\mathsf{search}}$-completeness and practice NP-completeness proofs.

1. (Positive Monotone SAT) A boolean formula is *positive monotone* if there are no negations in it. Restricting SAT to Positive Monotone formulas makes it trivial; setting all variables to 1 is always a satisfying assignment.

   However, the following variant of Positive Monotone SAT is more interesting:

Input	: A positive monotone CNF formula $\varphi(x_0, \ldots, x_{n-1})$ and a number $k \in \mathbb{N}$
Output	: A satisfying assignment $\alpha \in \{0,1\}^n$ in which at least k variables are set to 0 (if one exists)

   Computational Problem k-False PositiveMonotoneSAT

   (a) Prove that k-False PositiveMonotoneSAT is $\mathsf{NP}_{\mathsf{search}}$-complete, even when $k = n/2$. (Hint: reduce from SAT, replacing negated variables with new ones and adding additional clauses.)

   Proof

   i. Positive Monotone SAT is in $\mathsf{NP}_{\mathsf{search}}$: Our verifier can check if an assignment α satisfies the PM formula (the same verifier as for SAT). Because we know that SAT is in $\mathsf{NP}_{\mathsf{search}}$, and we use the same verifier in PMSAT, it follows that PMSAT is in $\mathsf{NP}_{\mathsf{search}}$.

   ii. Positive Monotone SAT is $\mathsf{NP}_{\mathsf{search}}$-hard: Since every problem in NP reduces to SAT, all we need to show is $SAT \leq_P PMSAT$ (since reductions are transitive).

We will follow the common reduction template to prove 2. First, we will transform the problem from what we want to solve (SAT) to what we have an oracle for (PMSAT). Next, we will attempt to find a satisfying assignment α' for PMSAT. If such an assignment exists, we must turn α' into a satisfying assignment α that satisfies our original SAT. If not, then $\perp$ will be returned by the oracle and thus by the entire reduction as well.

Pre-processing:

Logically, we will want to replace negated variables, $\neg x_i$, with dummy variables, y_i, within their previous clauses to make the formula positive monotonous. All dummy variables represent the negation of their original counterpart. Then, we can add clauses corresponding to every variable (negated or otherwise) of simply the variable and its dummy counterpart $(x_i \vee y_i)$. This ensures that the inputted SAT problem is preserved. For example:

$$(x_0 \vee \neg x_1 \vee \neg x_2)$$

becomes

$$(x_0 \vee y_1 \vee y_2) \wedge (x_0 \vee y_0) \wedge (x_1 \vee y_1) \wedge (x_2 \vee y_2)$$

All clauses of size two will be solved, as x_i and y_i will be different values by how we defined y_i. Additionally, our new dummy variables preserve the negations in the original SAT clause.

Oracle:

We can set k in the PMSAT oracle equal to the total number of variables in the SAT problem, as we know that one of two variables in each (x_i, y_i) pair will be 0. We will then call the PMSAT algorithm, resulting in either $\perp$ if no solution exists or the assignment α' if a solution does.

Post-processing:

In terms of post-processing, if $\perp$ is returned by the oracle, the we can simply return it as well for our whole reduction. If a solution is returned, a very simple trick can change it to an equivalent solution to SAT. All we must do is simply remove all assignments for dummy variables and return only the assignments for our original set.

We can look to our above example to see how something this simple is possible. Either x_0, y_1, or y_2 is 1 if a solution is returned. If x_0 is 1, then simply returning this assignment would satisfy our original SAT clause. If either y variable is 1, then returning its x counterpart's assignment of 0 will satisfy our original SAT clause, as the x variables are negated.

Runtime:

Editing clauses to replace negated variables will run in time $O(m)$ where m is the number of clauses in our initial CNF formula.

Adding clauses with pairs of original variables and their dummy counterparts will run in time $O(n)$ where n is the number of variables in our initial CNF formula.

Our Oracle runs in constant time as defined by the definition of an Oracle.

Our post-process will either run in constant time or in time $O(n)$ to delete the n dummy variable assignments if a solution exists.

Overall, this reduction runs in $O(n+m)$ which is of course polynomial time.

If ϕ is satisfiable, then $\phi' = R(\phi)$ is satisfiable:

Assume that ϕ is satisfiable. Let $\phi = \phi_0, \phi_1, ..., \phi_m = R(\phi)$ as our algorithm loops through the original clauses, editing them to be PM and adding our pairs. Notice how this algorithm will loop m times to visit all m clauses. We will prove this through induction

Base Case $(i = 0)$**:** $\phi = \phi_0 = R(\phi)$ so this is of course a correct reduction.
Hypothesis: Assume that ϕ_{i-1} is satisfiable.
Step: α_{i-1} is a satisfying assignment to ϕ_{i-1}. We obtain ϕ_i by running our algorithm on the i-th clause, replacing negations with dummy variables and adding the size-2 clauses for variables we have not reached yet. We know these clauses of size 2 will be satisfied no matter as was previously explained.

As for the edited clause, we have 3 possibilities. First, the variables within are not in any previous clauses. In this case, the variables do not have an assignment in ϕ_{i-1}, and just one of them needs to be assigned 1 to satisfy the clause. The others, if more exist, can be either 1 or 0.

Second, some variables are within previous clauses and some are not. Similar to the first case, any of the new variables can be set to 1 to satisfy the clause, or if a previously visited variable already has an assignment of 1, that would suffice as well.

Third, all variables are within previous clauses. Because we assumed that the overall ϕ is satisfiable, no original clauses will be in direct conflict with each other. This means that it is impossible for all of these previously visited variables to be set to 0. At least one variable will be 1, and the clause will be satisfied.

We have proven that if ϕ is satisfiable, then $\phi' = R(\phi)$ is satisfiable through induction.

If α' satisfies $R(\phi)$, then $\alpha'|_x$ also satisfies ϕ, where $\alpha'|_x$ is the restriction of the assignment α' to the x variables:

This was previously explained during pre-processing when it became apparent that a solution for the original SAT problem would come from simply dropping the assignments of dummy variables in $alpha'$. We can prove this through backwards induction.

Base Case ($i = m$)**:** This is equivalent to α' satisfying $R(\phi)$.
Hypothesis: Assume that ϕ_i is satisfiable by α'.
Step: ϕ_i was created from ϕ_{i-1} by using our algorithm on the i-th clause. We are assuming that α' satisfies the newly generated/edit clauses, so then by the soundness of the resolution rule, α' also satisfied the clause that they were generated from.

Thus, we have proven that If α' satisfies $R(\phi)$, then $\alpha'|_x$ also satisfies ϕ through reverse induction.

After following the "standard structure" of proving $\mathsf{NP}_{\mathsf{search}}$-completeness from Lecture 20, we have proven PMSAT to be $\mathsf{NP}_{\mathsf{search}}$-complete.

(b) Prove that if we fix $k = 3$, then k-False PositiveMonotoneSAT is in P. (Hint: show that it suffices to consider assignments in which exactly 3 variables are set to 0.)

First, we only need to consider assignments where exactly 3 variables are set to 0 because if there were more than 3 0s, say four total, one of the 0s could be changed to 1 with no effect on the satisfiability. If there is an assignment with 100 0s, 97 of them could be changed to 1 and the PMSAT problem would still be satisfied.

Given that we only need to consider cases where there are three 0s for $k = 0$, we can brute force a solution that runs in polynomial time. If there are n total variables, there are $\binom{n}{3}$ total possible assignments with three 0s. For each of these assignments, we simply need to check to see if it is a satisfying solution, meaning an $O(n^3)$, polynomial runtime. Thus, if we fix $k = 3$, then k-False PMSAT is in P.

2. (Reductions and complexity classes)

(a) Prove that if a problem Π is in $\mathsf{P}_{\mathsf{search}}$, then $\Pi \leq_p \Gamma$ for all computational problems Γ.

If Π is in $\mathsf{P}_{\mathsf{search}}$, then there exists and algorithm that solves the problem in polynomial time.

Γ is any computational problem. Even though we know nothing about Γ let's set up a reduction from Π to Γ to help illustrate our proof.

Pre-process: In our pre-process step, we can solve Π in polynomial time.

Oracle: Even though Γ is our oracle, we can call it zero times if we so choose. Thus, we will not call our oracle Γ.

Post-process: Now, we can simply return what we returned by solving Π in pre-processing.

As illustrated above, we were able to reduce Π to Γ in polynomial time by simply solving Π is pre-processing and never calling the oracle.

(b) Show that if $\mathsf{NP}_{\mathsf{search}} \subseteq \mathsf{P}_{\mathsf{search}}$, then all problems in $\mathsf{NP}_{\mathsf{search}}$ are $\mathsf{NP}_{\mathsf{search}}$-complete. (The converse of this statement was proved in section, so it is actually an iff.)

There are two things that must be true for a problem Π to be $\mathsf{NP}_{\mathsf{search}}$-complete:

i. Π is in $\mathsf{NP}_{\mathsf{search}}$
ii. Π is $\mathsf{NP}_{\mathsf{search}}$-hard: For every computational problem $\Gamma \in \mathsf{NP}_{\mathsf{search}}$, $\Gamma \leq_P \Pi$

As proven in part A, if a problem is in $\mathsf{P}_{\mathsf{search}}$ it can be reduced to any other computational problem in polynomial time. If $\mathsf{NP}_{\mathsf{search}} \subseteq \mathsf{P}_{\mathsf{search}}$ then all problems in $\mathsf{NP}_{\mathsf{search}}$ are in $\mathsf{P}_{\mathsf{search}}$. Thus, all $\mathsf{NP}_{\mathsf{search}}$ can reduce to any other problem in polynomial time, meaning they are all $\mathsf{NP}_{\mathsf{search}}$-hard and have satisfied the two requirements for $\mathsf{NP}_{\mathsf{search}}$-complete.

(c) Prove that if $\Pi \leq_p \Gamma$ and $\Gamma \in \mathsf{EXP}_{\mathsf{search}}$, then $\Pi \in \mathsf{EXP}_{\mathsf{search}}$. (In other words, $\mathsf{EXP}_{\mathsf{search}}$ is closed under polynomial-time reductions.)

Γ is the oracle of the reduction, entire reduction takes $P + exp + post\ process$, entire reduction is exp, Π is exp

In our reduction to solve problem Π, there are several runtimes to consider.

First, $\Pi \leq_p \Gamma$ and thus our pre-processing will take polynomial time, or some $O(n^c)$ with some constant c.

Our oracle is Γ and is defined to run in exponential time, or some $O(b^{n^c})$ for some base b and constant c. Additionally, there can be a total of a polynomial number calls to the oracle, so $O(p * b^{n^c})$.

Finally, the size of our input/output to the oracle is bounded polynomially, and so our post-process will run in time $O(w * n^c)$ for some constant c and some polynomial

w representing a possible size change occurring in the oracle.

Summing these runtimes, we can see that the entire reduction is solved in exponential time, and thus, $\Pi \in \mathsf{EXP}_{\mathsf{search}}$.

3. (Variant of VectorSubsetSum) In the Sender–Receiver Exercise on November 15, you will see that the following problem is $\mathsf{NP}_{\mathsf{search}}$-complete.

Input	: Vectors $\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_{n-1} \in \{0,1\}^d$, $\vec{t} \in \mathbb{N}^d$
Output	: A subset $S \subseteq [n]$ such that $\sum_{i \in S} \vec{v}_i = \vec{t}$, if such a subset S exists.

Computational Problem VectorSubsetSum

Assuming that result, prove that the following variant is also $\mathsf{NP}_{\mathsf{search}}$-complete.

Input	: Vectors $\vec{v}_0, \vec{v}_1, \ldots, \vec{v}_{n-1} \in \mathbb{N}^d$, $t_0 \in \mathbb{N}$
Output	: A subset $S \subseteq [n]$ such that $\sum_{i \in S} \vec{v}_i = (t_0, t_0, \ldots, t_0)$, if such a subset S exists.

Computational Problem VectorSubsetSumVariant

The two differences from the VectorSubsetSum problem is that the vectors are no longer restricted to have $\{0,1\}$ entries, but now all entries of the target vector are required to be equal. (Hint: reduce from the standard VectorSubsetSum problem. Add an additional vector and an additional coordinate.)

Proof

(a) VectorSubsetSumVariant is in $\mathsf{NP}_{\mathsf{search}}$: A verifier can sum the returned vectors and compare to to the t_0 vector in polynomial time. Because we know that VectorSubsetSum is in $\mathsf{NP}_{\mathsf{search}}$, and we use the same verifier in VSSV, it follows that VSSV is in $\mathsf{NP}_{\mathsf{search}}$.

(b) VectorSubsetSumVariant is $\mathsf{NP}_{\mathsf{search}}$-hard: We must show that VectorSubsetSum (an $\mathsf{NP}_{\mathsf{search}}$-complete problem) can reduce to VectorSubsetSumVariant.

Let's try to reduce VSS to VSSV to prove that VSSV is $\mathsf{NP}_{\mathsf{search}}$-hard

Pre-processing:

Let's say we have a set of vectors $\vec{a}, \vec{b}, \vec{c}, \vec{d}$, and

$$\vec{a} + \vec{b} = \vec{t}$$

When we reduce to VSSV, we must find a way to add some vector $\vec{v}$ to these two vectors to return the vector where every coordinate set to t_0. Let's set up and equation and see if we can solve for what we need:

$$\vec{a} + \vec{b} + \vec{v} = \vec{t_0}$$

Substituting in:

$$\vec{t} + \vec{v} = \vec{t_0}$$

So the vector we need to add to the set during pre-processing is:

$$\vec{v} = \vec{t_0} - \vec{t}$$

Simply adding this vector to the set is not enough to guarantee that VSSV returns the solution of our desired vectors and our added vector. Let's consider the case where $\vec{c} + \vec{d} = \vec{t_0}$. A solver could then either return $(\vec{c}, \vec{d})$ or $(\vec{a}, \vec{b}, \vec{v})$ and both would be correct for VSSV but

not necessarily VSS. Thus, we must add a coordinate to all vectors. This coordinate will be 0 in all of our original vectors and t_0 in our new vector $\vec{v}$. This ensures that any solution to VSSV **must** contain $\vec{v}$, as the added coordinate will of course be equal to t_0 in $\vec{t_0}$. Thus, $\vec{v}$ must be present in correct solutions $(0 + 0 + ... + t_0 = t_0)$.

To summarize, in pre-processing we will add a vector $\vec{v} = \vec{t_0} - \vec{t}$ and add a new coordinate to all vectors: 0 for our original vectors and t_0 for $\vec{v}$.

Oracle:

Because of how we constructed $\vec{v}$, our call to the oracle VectorSubsetSumVariant will either return $\vec{v}$ and a set of vectors that solve our original VectorSubsetSum problem or $\perp$ if no such set of vectors exists.

Runtime:

Constructing our new vector takes constant time, and adding a new coordinate to all n vectors runs in time $O(n)$.

From the SRE on the VectorSubsetSum problem, we know that our oracle will run in polynomial time.

Returning the vectors in our post-processing step will take constant time.

Summing these individual runtimes shows us that the overall runtime for this reduction is polynomial.

If $\exists S \subseteq [n]$ such that $\sum_{i \in S} \vec{v_i} = \vec{t}$, then $\exists S \subseteq [n]$ such that $\sum_{i \in S} \vec{v_i} = (t_0, t_0, \ldots, t_0)$:
(In english: if there is a set of vectors that sum to $\vec{t}$, then there is a set of vectors that sum to $\vec{t_0}$):

This can be proved through exploration of how we constructed our added vector in VSSV.

There is known to be a set S of vectors that sum to $\vec{t}$. We defined $\vec{v}$ to be:

$$\vec{v} = \vec{t_0} - \vec{t}$$

Rearranging we can see that:

$$\vec{t} + \vec{v} = \vec{t_0}$$

We can clearly see that the set of vectors that VSSV will return is the set S that sums to $\vec{t}$ from VSS as well as our created vector.

If there is a set of vectors that sum to $\vec{t_0}$, then there is a set of vectors that sum to $\vec{t}$:

As previously described, a solution to VSS will be within our solution to VSSV, just with our created vector removed. This can be seen here:

$$\sum vectors = \vec{t_0}$$

$$\sum original\ vectors + \vec{v} = \vec{t_0}$$

$$\sum original\ vectors = \vec{t_0} - \vec{v}$$

$$\sum original\ vectors = \vec{t}$$

Thus, there exists a summation of vectors from the original set that equals $\vec{t}$.

After following the "standard structure" of proving $\mathsf{NP}_{\mathsf{search}}$-completeness from Lecture 20, we have proven VectorSubsetSumVariant to be $\mathsf{NP}_{\mathsf{search}}$-complete.