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

Problem Set 9

Harvard SEAS - Fall 2022 *Due: FRI. Dec. 2, 2022 (11:59pm)*

Your name: Evan Jolley
Collaborators: Dhrub "The GOAT" Singh
No. of late days used on previous psets: 3
No. of late days used after including this pset: 6

The purpose of this problem set is to practice proving that problems are unsolvable via reduction, and gain more intuition for the kinds of problems about programs that are unsolvable (through examples).

Throughout this problem set, you may use some pseudocode in describing RAM and Word-RAM programs (like for loops), but be sure that the pseudocode can be implemented using actual RAM/Word-RAM commands that satisfy the constraints of the given problem (e.g. having no arithmetic overflows or being write-free).

1. ($\mathsf{P}_{\mathsf{search}} \not\subseteq \mathsf{NP}_{\mathsf{search}}$) Recall that in lecture, we commented that there are artificial problems in $\mathsf{P}_{\mathsf{search}}$ that are not in $\mathsf{NP}_{\mathsf{search}}$. Here you will see one. Specifically, consider the computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ given as follows:

$$\begin{aligned} \mathcal{I} &= \{P : P \text{ is a RAM program}\} \\ \mathcal{O} &= \{0,1\} \\ f(P) &= \begin{cases} \{0,1\} & \text{if } P \text{ halts on } \varepsilon \\ \{0\} & \text{otherwise} \end{cases} \end{aligned}$$

Prove that Π is in $\mathsf{P}_{\mathsf{search}}$ and that Π is not in $\mathsf{NP}_{\mathsf{search}}$.

Proof that Π is in $\mathsf{P}_{\mathsf{search}}$:
To prove that Π is in $\mathsf{P}_{\mathsf{search}}$, we must construct an algorithm that solves Π in polynomial time. We define our algorithm A to take P as an input and return 0 regardless of what P is – the only thing that A does is return 0. We know that 0 is a valid output of Π, and thus, this algorithm correctly solves Π. Of course this algorithm runs in polynomial time, as only one thing is executed (return 0). We have constructed an algorithm that correctly solves Π in polynomial time and proven that Π is in $\mathsf{P}_{\mathsf{search}}$.

Proof that Π is not in $\mathsf{NP}_{\mathsf{search}}$:
To be in $\mathsf{NP}_{\mathsf{search}}$ a computational problem must have solutions of polynomial length that can be verified in polynomial time. We know that our possible outputs are 0 and 1 so solutions will be of polynomial length. We must focus our proof on disproving that these solutions can be verified in polynomial time.

We will accomplish this by reducing from *HaltOnEmpty* to a verifier for Π. We know that *HaltOnEmpty* is an unsolvable problem from lecture, so if we are able to construct this reduction, it would disprove Π being in $\mathsf{NP}_{\mathsf{search}}$. The algorithm is as follows:

1. The algorithm takes as input a RAM program P and outputs *yes* if P halts on ϵ and *no* otherwise. ϵ is an empty array of length 0.

2. Defining our verifier is simple. It will take 0 or 1 as well as the problem P as inputs, 0 and 1 being the possible outputs from our initial definition of Π. The verifier will return *true* if the solution is valid and $false$ if it is not.

3. Now we will run our *HaltOnEmpty* oracle on our verifier with inputs P and 1. We choose to input 1 rather than 0 for correctness reasons – $f(P)$ can be equal to 0 in the case that P halts on ϵ and otherwise as defined in the question. Restricting to only 1 allows us to be certain if P halted or not.

4. If the oracle returns *yes*, then P halted on ϵ. If *no*, then P did not halt.

Runtime:
We can modify our input in constant time, and there is only one call to our oracle. Thus, this reduction runs in constant time.

Correctness:
To prove that our reduction is correct we must prove that our problem Π is solved if and only if P halts on ϵ. Because we restricted the input to our verifier to be only 1, we know that the verifier will only return *yes* if P halts on ϵ. If we had not restricted this, a *yes* return from a 0 input would not have been able to decipher whether P halted or not. In the other direction, if P does halt on ϵ, a *yes* will be returned following the correctness of *HaltOnEmpty*. If P halts on ϵ, this means that our solution by the definition of $f(P)$ given to us in the problem is either 0 or 1, which are both known valid solutins of Π. Thus, we have shown the correctness of our reduction.

Because *HaltOnEmpty*, an unsolvable problem, reduces to the verifier of Π, we know that the verifier is also unsolvable. This proves that solutions to Π can not be verified in polynomial time, and thus proves that Π is not in $\mathsf{NP}_{\mathsf{search}}$.

2. (Undecidability of arithmetic overflows) An *arithmetic overflow* in the execution of a Word-RAM program is when the result of an arithmetic operation (addition or multiplication) results in a value larger than 2^w, where w is the current word size, so the result has to be taken modulo 2^w. In Lecture 24, you will see how SMT Solvers are able to find a bug due to arithmetic overflow in Binary Search truncated to two levels of recursion. In this problem, you will see that finding arithmetic overflow errors in general programs is an unsolvable problem.

   (a) Give an algorithm that converts any RAM program P into an equivalent Word-RAM program P' that never has arithmetic overflow. That is, for all inputs (arrays of natural numbers) x, P' halts on x iff P halts on x, and if they do halt, then $P'(x) = P(x)$, and whenever P' carries out an operation $\texttt{var}_i = \texttt{var}_j \texttt{ op } \texttt{var}_k$, the result is always smaller than 2^w, where w is the current word size. Do not worry about the efficiency of your simulation (in contrast to Theorem 7.1.1 of Lecture 7, which does a fairly involved simulation in order to obtain the runtime as described; something much simpler suffices

here).

To avoid arithmetic overflow, we simply need to dynamically allocate more memory to fit x_i. We will only need to allocate more memory for addition and multiplication, as subtraction and division will always lead to smaller x_i values because variables are natural numbers. Additionally, we might need to allocate more memory if a user tries to assign a variable to be larger than $2^w - 1$ where w is the word length.

Intuitively, the amount of memory we need to $MALLOC$ is determined by x_j, x_k, and the operation in question. We will construct and algorithm to $MALLOC$ the correct amount:

```
for x in 1 to 2^w-1
    for y in 1 to 2^w-1
        MALLOC
var_i = var_j op var_k
```

This algorithm guarantees that our new S after the MALLOC commands is at least $(2^w - 1)^2$. var_j and var_k are both bounded by $2^w - 1$ (nothing greater than this can be stored in a variable as defined in WR), meaning we will not need more memory than $(2^w - 1) * (2^w - 1)$ to store var_i.

It is important to note that for loops do not exist in Word-RAM or RAM. Thus, we will need to use If...GOTOs and counters to simulate our nested for loop.

We will need to traverse our WR program looking for lines that match the pattern $var_i = var_j \ op \ var_k$, and then change that line to our simulated nested for loops followed by our original $var_i = var_j \ op \ var_k$, as shown above. This will guarantee that arithmetic overflow never occurs. Also, we will need to update the line numbers on previously existing GOTO statements to account for the added lines needed to simulate these for loops.

If we correctly construct our nested for loops and add them before all addition and multiplication operations, we will have successfully created a WR program that will not arithmetically overflow.

(b) Using Part 2a and the undecidability of HaltOnEmpty for RAM programs, prove that the following computational problem is unsolvable:

Input	**:** A Word-RAM program P
Output	**:** `yes` if P has an arithmetic overflow when run on input ε, `no` otherwise

Computational Problem ArithmeticOverflow

We can prove that $ArithmeticOverflow$ is unsolvable by showing that $HaltOnEmpty$, an known unsolvable problem, reduces to it. That is, we need to give a reduction that decides whether a program P halts on an empty input using an oracle that solves $ArithmeticOverflow$. The algorithm A is as follows:

i. A(P):
Input : A RAM program P
Output : yes if P halts on ϵ (an empty list), no otherwise

ii. Construct from P and x a Word-RAM program Q_P such that Q_P has arithmetic overflow iff P halts on ϵ;

iii. Run the $ArithmeticOverflow$ oracle on Q_P and return its result;

Constructing Q_P:
The first thing to note is that $ArithmeticOverflow$ inputs a Word-RAM program and $HaltOnEmpty$ inputs a RAM program. Thus, the program P we input will not be restricted by WR restrictions like arithmetic overflow. Our answer to 2a shows that we can construct a WR program P' that can simulate everything that P does while also preventing arithmetic overflow. Additionally, if we did not augment our program to prevent this, our oracle would return yes regardless of whether P halts on ϵ if there was an operation that caused arithmetic overflow present in P.

We now have a Word-RAM program that does not arithmetically overflow on its own, and we want to further augment it to arithmetically overflow if it halts on ϵ so that our oracle returns the correct answer to $HaltOnEmpty$. Word-RAM programs only halt if they execute their last line without it returning to an earlier line – there are no returns or halts in the middle of commands, just GOTOs that could take the program to the last line. Thus, adding something that is guaranteed to arithmetically overflow after the previous last line guarantees that the program overflows iff P halts on x.

To construct something that arithmetically overflows, we can utilize the simulated for loop model we described in 2a to construct something like this:

```
counter = 1
for x in 1 to w
    counter = counter * 2
```

On the loop's final iteration, counter will be set to 2^w, which is greater than the $2^w - 1$ limit imposed on variable assignment by the definition of Word-RAM.

After creating a Word-RAM program simulating P that only overflows if the last line is reached, we can pass the pass this program to our $ArithmeticOverflow$ oracle and it will output the correct answer to $HaltOnEmpty$ (yes if P halts on ϵ and no otherwise).

With this claim, we see that plugging the construction of $Q_{P,x}$ into the template outlined above gives a correct reduction from $HaltOnEmpty$ to $ArithmeticOverflow$. By the unsolvability of the $HaltingOnEmpty$ Problem and the fact that if we can reduce an unsolvable problem to a different problem then that problem is itself unsolvable, we deduce that $ArithmeticOverflow$ is unsolvable.