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

Problem Set 9

Harvard SEAS - Fall 2023 *Due: FRI. Dec. 6, 2023 (11:59pm)*

Your name: Cory Zimmerman
Collaborators: None
No. of late days used on previous psets: 0
No. of late days used after including this pset: 0 (Limit 2 late days for problem set 9.)

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, Anurag 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 $\mathsf{NP}_{\mathsf{search}}$.

Prove Π *is in* $\mathsf{P}_{\mathsf{search}}$:
To be in $\mathsf{P}_{\mathsf{search}}$, the problem Π must be solvable in $O(n^c)$ time where c is a constant. Consider this algorithm solving the problem: Given any input program P, return 0.

Any input program P either halts or doesn't halt. If P does halt, the problem allows a return value of 0. If P doesn't halt, the problem requires a return value of 0. Thus, 0 is always a correct output. Because this algorithm returns valid output for every possible input, this solution must be correct. Returning 0 is a constant time operation, so this algorithm runs in time $O(n^c)$.

Because the algorithm above solves P in polynomial time, it must be the case that P is in $\mathsf{P}_{\mathsf{search}}$.

Prove Π *is not in* $\mathsf{NP}_{\mathsf{search}}$:
To be in $\mathsf{NP}_{\mathsf{search}}$, there must exist an algorithm to solve Π such that all solutions are of polynomial length and solutions are verifiable in polynomial time. Because

the problem already has a constant-size solution set, I'll prove that it can't be verified:

Assume for purposes of contradiction that Π is in $\mathsf{NP}_{\mathsf{search}}$. If this is the case, there must exist some verifier algorithm v that runs in polynomial time. Say we run the algorithm solving Π on some program P and get output 1. Now, v must verify if 1 is a correct output for Π on P. Because Π can only return 1 if a program halts on ϵ, the verifier v must determine if P halts on ϵ. This computational problem is well defined, and we proved in class that it is unsolveable. This implies that the verifier v is capable of solving an unsolveable problem, which is a contradiction. Because of this contradiction, it must be the case that Π actually 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 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).

   First, assert that certain higher-level constructs like *for* and *while* loops can be produced using GOTOs and temp variables in Word RAM. Throughout the algorithm, I'll assume for simplicity that such abstractions do not need to be explicitly expanded or proven.

   In this algorithm, begin by copying all lines of the RAM program P into P'. Then, add MALLOC loops before memory reads and writes to ensure that referenced addresses in memory are always valid. With this established, consider the following adaptation of RAM arithmetic to prevent overflow:

Begin with a macro I'll call safeMult. Defining a multiplication operation $z = x \cdot y$, safeMult expands to the following:

```
z = 0;
for 0 to y - 1 {
    z = z + x;
}
```

By this definition, safeMult adds x to z for y times, which simulates multiplication using addition.

Consider a second macro called safeSum. Given an arithmetic sum $z = x + y$, safeSum expands to the following:

```
z = x;
for 0 to y - 1 {
    while S - 1 <= z {
        MALLOC;
    }
    z = z + 1;
}
```

By this definition, safeSum adds 1 to x for y times, which simulates addition using single-digit operations.

This construction of safeSum guarantees that S, the global variable tracking memory size, is always large-enough to accomodate z. This is simple but wasteful because we only need the word size to accomodate z, not necessarily that much memory. However, because MALLOC increments w when S exceeds 2^w, this guarantees the word size is always large-enough without requiring us to separately track S, w, and 2^w.

Now, make the following operations on the program P', which has already been modified so that memory accesses are in-bounds: (1) replace every instance of multiplication with safeMult, and (2) replace every instance of addition with safeSum. The order is essential, as the addition operations in safeMult must be expanded into instances of safeSum.

Before exiting the "compilation", update all GOTO line numbers that were disrupted by memory operations and macro expansions. Specific values will be dependent on the program and can be calculated by line counts of the specific substitution rules.

Consider why this Word-RAM simulation of RAM is correct. By Theorem 7.1 in the Lecture Notes, assert that the initial memory operations enable P' to correctly model the memory access patterns from P.

For preventing arithmetic overflow, begin with safeSum. By definition of the

word RAM model, the values of x and y are guaranteed to fit in memory. By iteratively adding one to the sum variable z beginning at x, MALLOC is performed as soon as the maximum allocated memory address limits the next value of z. Because the word size is always incremented to accomodate the maximum allocated memory address, this transitively implies that the word size always accomodates the next value of z.

Because multiplication is built on the same principle and expands in part to safeSum itself, safeMult must also guard against arithmetic overflow.

Because neither RAM nor Word RAM natively supports negative numbers, there is no concern about translating those. Since the result of unsigned integer subtraction and division is never greater than the inputs, neither of those operations are at risk of generating overflow.

Thus, this transformation of P into P' must correctly simulate a RAM program in Word RAM without any integer 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

I'll prove that ArithmeticOverflow is undecidable by reducing from HaltOnEmpty to ArithmeticOverflow. Consider the following pseudocode. (Note that I'm assuming the word size is available and a suitable abstraction for exponentiation in Word-RAM can be devised):

```
HaltOnEmpty(RAM program P) {
    Word-RAM program P' = withoutOverflow(P);

    Qp() {
      P'();
      M[0] = 2^w + 2^w; // or anything else that overflows
    }

    return ArithmeticOverflow(Qp);
}
```

The first line constructs a Word-RAM program P' using a function withoutOverflow. Assume withoutOverflow implements the algorithm from $2a$ in which a RAM program is translated into an equivalent Word-RAM program without any arithmetic overflow. The output program is P'.

Skipping Qp for now, at the bottom, I call ArithmeticOverflow on Qp and return it as the result from HaltOnEmpty. Thus, for my reduction to be correct, any Qp that overflows must halt, and any Qp that does not overflow must not halt.

Consider Qp: If $P'()$ halts, the line below is executed. That line causes overflow. That overflow causes ArithmeticOverflow to return true, which causes HaltOnEmpty to return true. That means if P' halts on empty, then ArithmeticOverflow and HaltOnEmpty return true, which is correct.

If P' does not halt on empty, then the line below $P'()$ is never executed. If that line is never executed, then Qp never overflows. This can be guaranteed because P' is constructed such that it never overflows and if P' runs forever in Qp then there can never be integer overflow in Qp. If Qp never overflows, then ArithmeticOverflow returns false, and HaltOnEmpty subsequently returns false.

Because these two cases for the halting behavior of P cover all possible input programs and the output is correct for both cases, it must be true that my reduction returns a correct output for every input, making it correct.

Because there exists a correct reduction from HaltOnEmpty to ArithmeticOverflow and HaltOnEmpty is unsolveable, it must be the case that ArithmeticOverflow is unsolveable.

3. (optional:[1] Memory-Free RAM Programs)

   (a) A *memory-free RAM Program* $P = (V, C_0, \ldots, C_{\ell-1})$ has no commands to read or write to memory, has a fixed set of input variables $x_0, x_1, \ldots, x_{n-1}$, and a fixed set of output variables $y_0, \ldots, y_{m-1}$. It begins its computation with the input x placed in the variables $(x_0, \ldots, x_{n-1})$ and if it halts, its output $P(x)$ is defined to be the values in $(y_0, \ldots, y_{m-1})$. Using the unsolvability of Diophantine Equations (as stated without proof in Lecture 24), prove that the Halting Problem for Memory-Free RAM Programs is unsolvable.

   (b) A memory-free *Word*-RAM is defined similarly to a memory-free RAM, except it has a finite word size, which is initialized to be $w = \lceil \log_2 \max\{x_0, x_1, \ldots, x_{n-1}\} \rceil$ and never changes throughout the computation. Show that, in contrast to memory-free RAMs, the Halting Problem for memory-free *Word*-RAMs is solvable. That is, there is an algorithm that given a memory-free Word-RAM Program $P = (V, C_0, \ldots, C_{\ell-1})$ an input x, decides whether $P(x)$ ever halts. (Hint: The entire state of a computation of a memory-free (Word-)RAM program is determined by the current line number and the values of all variables. Use this to compute a threshold $T(P, x)$ such that if P runs for more than $T(P, x)$ steps, then P must be in an infinite loop.)

[1] This problem is meant to be done based on your enjoyment/interest and only if you have time. It won't make a difference between N, L, R-, and R grades, and course staff will deprioritize questions about this problem at office hours and on Ed.

4. (optional challenge:[1] Artemis' Party)

   Adam is hosting a birthday party for his cat, Artemis. In a shocking turn of events, Artemis' cat-rival Simetra is also having a birthday party on the same day! Adam gathered some intel and found out that Simetra's party will have k guests. So as not to be shown up, Artemis is now feeling the pressure to have k guests as well. The only problem as that a lot of Artemis' friends are enemies with each other. To plan out his invite list, Artemis represents these relationships in a graph, where vertices are his cat-friends and an edge represents an enemy relationship. Given this graph and k, will Artemis be able to invite at least k non-hostile friends to his party, or will his party lose out to his nemesis Simetra?

Input	**:** A graph G of cats where edges are enemy relationships and k is the number of cats Artemis must invite (at least) in order to retain his honor.
Output	**:** `yes` if Artemis can successfully create an invite list of at least k invitees; `no` otherwise

   Computational Problem ArtemisBirthdayParty

   Prove or disprove: ArtemisBirthdayParty is in $\mathsf{P}_{\mathsf{search}}$.