CS 120: Intro to Algorithms and their Limitations

Problem Set 9

Due: December 6, 2023 11:59pm **Denny Cao**

Collaborators:
No. of late days used on previous psets: 6
No. of late days used after including this pset: 7

§1 $\texttt{P}_{\texttt{search}} \not\subseteq \texttt{NP}_{\texttt{search}}$

Claim — $\Pi \in \texttt{P}_{\texttt{search}}$.

Proof of claim. To show that Π is in $\texttt{P}_{\texttt{search}}$, we show that there exists a Word-RAM program P such that $f(P)$ can be found in polynomial time. Let P be the Word-RAM program that returns whether or not an input $x \in \mathbb{N}$ is at least 1:

```
    Input     : An input x occupying memory location M[0]
    Output    : A value y ∈ {0, 1} in memory location M[0]
    Variables : input_len, output_len, output_ptr, zero, word_len, mem_size
1   zero = 0;
2   output_len = input_len + zero;
3   output_ptr = 0;
4   if M[output_ptr] >= 1 GOTO 6 ;
5   if zero == 0 GOTO 7 ;
6   M[output_ptr] = 1 ;
7   HALT ;
```

Algorithm 1: Word-RAM Program P

We verify that $f(P)$ can be found in polynomial time. We break inputs $\varepsilon \in \mathbb{N}$ for P into two cases:

1. $\varepsilon \geq 1$: The total operations performed in this case is 6 regardless of the amount of bits to represent ε in the Word-RAM program, and thus takes time $O(1)$.

2. $\varepsilon = 0$: The total operations performed in this case is also 6, and thus takes time $O(1)$.

As in both cases, P halts in $O(1)$ and P halts for all $\varepsilon \in \mathbb{N}$, it follows that, for all inputs ε for P, $f(P) = \{\texttt{0}, \texttt{1}\}$ and can be found in $O(1)$ time which is polynomial time, and the proof is complete. □

Claim — $\Pi \notin \texttt{NP}_{\texttt{search}}$.

Proof of claim. To show that $\Pi \notin \texttt{NP}_{\texttt{search}}$, we show that there exists a Word-RAM program P such that a solution x cannot be verified in polynomial time. Let P be the Word-RAM program as follows:

Input : An input x occupying memory location $M[0]$
Variables : input_len, output_len, output_ptr, zero, word_len, mem_size

zero $= 0$;
output_len $=$ input_len $+$ zero;
output_ptr $= 0$;
if zero $== 0$ **GOTO** 4 ;
HALT;

Algorithm 2: Word-RAM Program P

Consider the case when the verifier V must verify if a solution $x = 0$ is correct for P. This will only happen if P does not halt. However, if P does not halt, then the verifier V would run indefinitely, and thus V would not run efficiently in polynomial time, and the proof is complete. □

§2 Undecidability of Arithmetic Overflows

(a) *Proof.* Let P' be a Word-RAM equivalent program for a RAM program P. We form a new program P'' where integer overflows cannot occur by an algorithm $\varphi =$ ConvertNoOverflowP which takes P' as input. We describe φ:

ConvertNoOverflowP $(P'(V', (C'_0, \ldots, C'_{\ell-1})))$
    $C'' = (\ )$;
    foreach $i \in [\ell]$ **do**
        if C'_i *in the form* $\mathtt{var}_i = \mathtt{var}_j + \mathtt{var}_k$ *or* $\mathtt{var}_i = \mathtt{var}_j \times \mathtt{var}_k$ **then**
            $\mathtt{start}_j = \mathtt{len}(C'') + 3$;
            $\mathtt{start}_k = \mathtt{len}(C'') + 5$;
            $\mathtt{end}_j = \mathtt{len}(C'') + 15$;
            $\mathtt{end}_k = \mathtt{len}(C'') + 13$;
            $\mathtt{end}_S = \mathtt{len}(C'') + 10$;
            C''.append($\mathtt{counter}_j$ = 0);
            C''.append($\mathtt{counter}_S$ = 0);
            C''.append(IF $\mathtt{counter}_j$ == $\mathtt{var}_j$ GOTO $\mathtt{end}_j$);
            C''.append($\mathtt{counter}_k$ = 0);
            C''.append(IF $\mathtt{counter}_k$ == $\mathtt{var}_k$ GOTO $\mathtt{end}_k$);
            C''.append(IF $\mathtt{counter}_S$ == S GOTO $\mathtt{end}_S$);
            C''.append($\mathtt{counter}_S$ = $\mathtt{counter}_S$ + one);
            C''.append($\mathtt{counter}_k$ = $\mathtt{counter}_k$ + one);
            C''.append(IF zero == 0 GOTO $\mathtt{start}_k$);
            C''.append(MALLOC());
            C''.append($\mathtt{counter}_k$ = $\mathtt{counter}_k$ + one);
            C''.append(IF zero == 0 GOTO $\mathtt{start}_k$);
            C''.append($\mathtt{counter}_j$ = $\mathtt{counter}_j$ + one);
            C''.append(IF zero == 0 GOTO $\mathtt{start}_j$);
        C''.append(C'_i);
    return $P(V+3, C'')$;

Algorithm 3: ConvertNoOverflowP

For P', integer overflow occurs when an operation $\mathtt{var}_j$ op $\mathtt{var}_k$ results in a value greater than or equal to 2^w, where $w =$ word_len. We bound the greatest value of $\mathtt{var}_j$ op $\mathtt{var}_k$ by the maximum value of $\mathtt{var}_j \times \mathtt{var}_k$, as multiplication will result in the largest value. We can thus bound the number of bits needed to represent an operation without integer overflow with

$$\lceil \log_2(\texttt{var}_j \ \texttt{op} \ \texttt{var}_k) \rceil \leq \lceil \log_2(\texttt{var}_j \times \texttt{var}_k) \rceil$$

We can avoid integer overflow by calling `MALLOC` m times if $\texttt{var}_j \ \texttt{op} \ \texttt{var}_k \geq 2^w$. The amount of times we call `MALLOC` will be

$$m = \texttt{var}_j \times \texttt{var}_k - S$$

As $w = \lfloor \log \max\{S, x[0], \ldots, x[n-1]\} \rfloor$, and thus $w \geq \log S$, where S is the length of memory, if we call `MALLOC` $\texttt{var}_j \times \texttt{var}_k - S$ times prior to $\texttt{var}_j \ \texttt{op} \ \texttt{var}_k$, we will obtain a new value for S, S', such that $S' \geq \texttt{var}_j \ \texttt{op} \ \texttt{var}_k$. This is because, with each `MALLOC` call, we increment S by 1.

From Theorem 7.1 in Lecture 7, for every RAM program P, there exists a Word-RAM Program P' such that P' halts on x if and only if P halts on x, and if they halt, then $P'(x) = P(x)$. It follows then that, if we show that, for every Word-RAM Program P' there exists a Word-RAM Program P'' such that P' halts on x if and only if P' halts on x, and if they halt, then $P''(x) = P'(x)$, by transitivity of biconditional statements, our proof is complete. □

(b) *Proof.* It suffices to show that `HaltOnEmpty` reduces to `ArithmeticOverflow`, or that there is an algorithm A that solves `HaltOnEmpty` given an oracle for `ArithmeticOverflow`. We describe the reduction below:

$A(P)$:
Input : A RAM program P
Output : `yes` if P halts on ε, `no` otherwise
Construct from P a RAM program Q_P such that Q_P overflows on ε if and only if P halts on ε;
Run the `ArithmeticOverflow` oracle on Q_P and return its result;

Algorithm 4: Reduction from `HaltOnEmpty` to `ArithmeticOverflow`

We construct Q_P as follows:

$Q_P(P)$:
$P' =$ `ConvertNoOverflowP(`P`);`
$C'' = (\ );$
`end` $=$ `len(`$P'.C$`);`
C''`.append(x = 2);`
C''`.append(counter = 0);`
C''`.append(IF counter == word_len GOTO end + 7);`
C''`.append(x = x × 2);`
C''`.append(counter = counter + 1);`
C''`.append(IF zero == 0 GOTO end + 3);`
C''`.append(HALT);`
$Q_P = P'$;
foreach $i \in [7]$ **do**
  $Q_P.C$`.append(`C''_i`);`
return Q_P*;*

Algorithm 5: The RAM Program Q_P constructed from P

Claim — Q_P overflows on ε if and only if P halts on ε.

Proof of claim. We construct Q_P by first creating an equivalent Word-RAM program P' for P such that there will be no integer overflows for operations by calling `ConvertNoOverflowP(`P`)` from Question 2.a. As P' can still overflow by setting a variable to a value larger than 2^w, we set a variable $x = 2^{w+1}$ if P' halts. As P' by construction does not result in overflows otherwise, by adding C'', we ensure that it halts if and only if it overflows ■

The claim implies that plugging this construction of Q_P into Algorithm 4 gives a correct reduction from `HaltOnEmpty` to `ArithmeticOverflow`, and thus completes the proof that `ArithmeticOverflow` is unsolvable. □