CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Active Learning Exercise 3: Reading for Receivers

Harvard SEAS - Fall 2021 *Sept. 28, 2021*

The goals of this exercise are:

- to develop your skills at understanding, distilling, and communicating proofs and the conceptual ideas in them
- to build your comfort with simulation arguments that we use to understand and relate different computational models

In class on 9/23, we introduced the RAM Model and asserted that it is fairly similar to assembly language for the CPUs on our computers, with variables in RAM programs corresponding to registers in the CPUs. However, our CPUs have a fixed number of registers (like 8 or 16), while RAM Programs are allowed to have any constant number of variables. In this exercise you will see that restricting the number of variables in a RAM Program to a fixed constant (e.g. 9) does not reduce its power. You will prove it by a simulation argument, like the one we used to show that adding the mod operation to a RAM program does not increase its power.

Theorem 0.1. *Every RAM Program can be simulated by one that uses at most 9 variables.*[1] *That is, for every RAM Program P, there is a RAM Program P' that uses at most 9 variables such that for all inputs x, we have $P'(x) = P(x)$ and the running time of P' on x, and*

$$\mathrm{Time}_{P'}(x) = O\left(\mathrm{Time}_P(x) + |P(x)|\right),$$

where $|P(x)|$ denotes the length of P's output on x, measured in memory locations.

To prepare for the exercise, we recommend reviewing the lecture notes from Thursday 9/23 to refresh your memory on the RAM Model and the (less involved) simulation argument we did during that class.

[1] The constant 9 is not optimized here, and can be reduced.

CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Active Learning Exercise 3: Reading for Senders

Harvard SEAS - Fall 2021 *Sept. 28, 2021*

The goals of this exercise are:

- to develop your skills at understanding, distilling, and communicating proofs and the conceptual ideas in them
- to build your comfort with simulation arguments that we use to understand and relate different computational models

In class on 9/23, we introduced the RAM Model and asserted that it is fairly similar to assembly language for the CPUs on our computers, with variables in RAM programs corresponding to registers in the CPUs. However, our CPUs have a fixed number of registers (like 8 or 16), while RAM Programs are allowed to have any constant number of variables. In this exercise you will see that restricting the number of variables in a RAM Program to a fixed constant (e.g. 9) does not reduce its power. You will prove it by a simulation argument, like the one we used to show that adding the mod operation to a RAM program does not increase its power.

Theorem 0.1. *Every RAM Program can be simulated by one that uses at most 9 variables.*[1] *That is, for every RAM Program P, there is a RAM Program P' that uses at most 9 variables such that for all inputs x, we have $P'(x) = P(x)$ and the running time of P' on x, and*

$$\mathrm{Time}_{P'}(x) = O\left(\mathrm{Time}_P(x) + |P(x)|\right),$$

where $|P(x)|$ denotes the length of P's output on x, measured in memory locations.

In the 20 minutes you have for the exercise, it's not realistic to hope that the receiver will be able to write all the details of the proof (e.g. all of the lines of RAM code). Instead, your goal should be to convey enough of the main ideas that they would be able to fill in the details on their own afterwards. Also, the RAM code in the proof below might have some small bugs—do let us know if you find any—but hopefully is enough to convince you of the theorem.

Proof. The idea is that P' will use some memory locations (immediately after the input) to store the values of the variables of P, and read them into temporary variables whenever needed. This will force some reindexing of the memory accesses of P.

In the construction of P' to make some simplifying assumptions about P:

- P never uses its `input_len` variable after the first step of the computation, which is of the form $\texttt{var}_0 = \texttt{input\_len} + \texttt{var}_1$. This can be done by adding new variables `input_len_copy` and `zero`, adding the line $\texttt{input\_len\_copy} = \texttt{input\_len} + \texttt{zero}$ to the beginning of P, and replacing all other occurrences of `input_len` in the commands of P with `input_len_copy`. This modification only increases the runtime of P by one step.

[1] The constant 9 is not optimized here, and can be reduced.

- Whenever P halts, we have $\texttt{output\_ptr} \geq \texttt{input\_len}$, i.e. its output locations don't overlap its input locations. We can modify P to have this property by modifying each HALT line to be preceded with a loop that copies the output to locations $\texttt{input\_len}, \texttt{input\_len} + 1, \ldots, \texttt{input\_len} + \texttt{output\_len} - 1$ and sets $\texttt{output\_ptr} = \texttt{input\_len}$. This modification increases the runtime of P by at most $O(\texttt{output\_len}) = O(|P(x)|)$.

Now let's get to the main steps of constructing P'. The program P' will have the required variables input_len, output_len, output_ptr, as well as $\texttt{temp}_0$, $\texttt{temp}_1$, $\texttt{temp}_2$, $\texttt{temp}_3$, zero, and one, and add lines $\texttt{zero} = 0$ and $\texttt{one} = 1$ to the beginning as usual. Let $\texttt{var}_0, \ldots, \texttt{var}_{v-1}$ be the variables in P other than input_len. For clarity, we will write M' to denote the memory of P', to distinguish it from the memory M of P that it is simulating. P' will use memory location $M'[\texttt{input\_len} + i]$ to store the value of variable $\texttt{var}_i$ of P.

Suppose P has a line of the form $\texttt{var}_i = \texttt{var}_j \texttt{ op } \texttt{var}_k$. We will replace this with a series of lines as follows:

$$
\begin{array}{rll}
\texttt{0} & \texttt{temp}_0 = \texttt{input\_len} + \texttt{zero} & \\
\texttt{1} & \texttt{temp}_0 = \texttt{temp}_0 + \texttt{one} & \\
 & \vdots & \\
\texttt{j} & \texttt{temp}_0 = \texttt{temp}_0 + \texttt{one} & \texttt{/* now temp}_0 == \texttt{input\_len} + j \texttt{ */} \\
\texttt{j+1} & \texttt{temp}_1 = M'[\texttt{temp}_0] & \texttt{/* now temp}_1 == \texttt{var}_j \texttt{ */} \\
 & & \\
\texttt{j+2} & \texttt{temp}_0 = \texttt{input\_len} + \texttt{zero} & \\
\texttt{j+3} & \texttt{temp}_0 = \texttt{temp}_0 + \texttt{one} & \\
\texttt{j+4} & \texttt{temp}_0 = \texttt{temp}_0 + \texttt{one} & \\
 & \vdots & \\
\texttt{j+k+2} & \texttt{temp}_0 = \texttt{temp}_0 + \texttt{one} & \texttt{/* now temp}_0 == \texttt{input\_len} + k \texttt{ */} \\
\texttt{j+k+3} & \texttt{temp}_2 = M'[\texttt{temp}_0] & \texttt{/* now temp}_2 == \texttt{var}_k \texttt{ */} \\
 & & \\
\texttt{j+k+4} & \texttt{temp}_3 == \texttt{temp}_1 \texttt{ op } \texttt{temp}_2 & \texttt{/* now temp}_3 = \texttt{var}_j \texttt{ op var}_k \texttt{ */} \\
 & & \\
\texttt{j+k+5} & \texttt{temp}_0 = \texttt{input\_len} + \texttt{zero} & \\
\texttt{j+k+6} & \texttt{temp}_0 = \texttt{temp}_0 + \texttt{one} & \\
\texttt{j+k+7} & \texttt{temp}_0 = \texttt{temp}_0 + \texttt{one} & \\
 & \vdots & \\
\texttt{j+k+i+5} & \texttt{temp}_0 = \texttt{temp}_0 + \texttt{one} & \texttt{/* now temp}_0 == \texttt{input\_len} + i \texttt{ */} \\
\texttt{j+k+i+6} & M'[\texttt{temp}_0] = \texttt{temp}_3 & \texttt{/* now var}_i == \texttt{temp}_3 == \texttt{var}_j \texttt{ op var}_k \texttt{ */}
\end{array}
$$

Here we have replaced one line of P with $i + j + k + 7 \leq 3v + 4$ lines of (non-looping) code.

The initial line of P of the form $\texttt{var}_i = \texttt{input\_len} + \texttt{var}_j$ is handled similarly, except that we don't need to read the value of input_len from M'.

A conditional $\texttt{IF var}_i == 0 \texttt{ GOTO } k$ can be similarly replaced with $i + 2 \leq v + 1$ lines of code ending in a line of the form $\texttt{IF temp}_1 == 0 \texttt{ GOTO } k'$. (Note that we will need to change the line numbers in the GOTO commands due to the various lines that we are inserting elsewhere.)

Lines where P reads and writes from M are slightly more tricky, because we need to shift pointers outside the input region by v to account for the locations where M' is storing the variables

of P. Specifically, we can replace a line $\texttt{var}_i = M[\texttt{var}_j]$ with a code block as follows:

```
 0  temp_0 = input_len + zero
 1  temp_0 = temp_0 + one
          ⋮
 j  temp_0 = temp_0 + one                          /* now temp_0 == input_len + j */
j+1 temp_1 = M'[temp_0]                                    /* now temp_1 == var_j */

j+2 temp_2 = input_len - temp_1                    /* zero iff temp_1 ≥ input_len */
j+4 IF temp_2 == 0 GOTO j+6
j+5 IF zero == 0 GOTO j+v+6                  /* don't shift pointers to input */

j+6 temp_1 = temp_1 + one
j+7 temp_1 = temp_1 + one
          ⋮
j+v+5 temp_1 = temp_1 + one                            /* now temp_1 == var_j + v */

j+v+6 temp_2 = M'[temp_1]                              /* now temp_2 == M[var_j] */

j+v+7 temp_0 = input_len + zero
j+v+8 temp_0 = temp_0 + one
j+v+9 temp_0 = temp_0 + one
          ⋮
j+v+i+7 temp_0 = temp_0 + one                     /* now temp_0 == input_len + i */
j+v+i+8 M'[temp_0] = temp_2                /* now var_i == temp_2 == M[var_j] */
```

Here we have replaced one line of P with $j + v + i + 7 \leq 3v + 5$ lines of (non-looping) code. Writes to memory are handled similarly (with the shifting the pointer by v happening after the line $M'[\texttt{temp}_1] = \texttt{temp}_2$) to reads.

Before P' halts, it also needs to set `output_len` and `output_ptr`, by reading the memory locations corresponding to the variables $\texttt{var}_i$ and $\texttt{var}_j$ that represent the output length and output pointer of P, and incrementing the output pointer by v as above. (This is where we use the assumption that the output of P is always in memory locations after the input.)

All in all, we have replaced each line of P by $O(v)$ non-looping lines in P'. Since v is a constant (depending only on P and not the size of the input x), we incur only a constant-factor slowdown in runtime (on top of the additive $O(|P(x)|)$ slowdown we may have incurred in the initial modifications of P). □