UVa - cs302: Theory of Computation Spring 2008

Exam 2 Final Comments

See the preview comments for the grade distribution and answers to the other questions.

Problem 1: Short Answers. (25) For each question, provide a correct, clear, precise, and concise answer from the perspective of a theoretical computer scientist.

b. [10] Explain the essence of the *Church-Turing Thesis* in a way that would be understandable to a typical fifth grader.

Answer: You know how you learn to solve long addition, multiplication, and division problems in math class? Well, what you are really learning is called a "procedure". That means, it is a series of instructions for what to do. The hard problem of adding two many digit numbers is broken down into a series of steps. For example, you start by adding the rightmost digits, writing down the answer at the bottom and if the sum is more than ten, writing the carry at the top of the next column. Then, you move left one position and add the next column. If you follow the rules carefully, the instructions work no matter what the numbers are — they could have hundreds of digits in them, but you can still add them by following the same rules. The good think about the procedures you are learning in math class is that they always finish. There is a fancy name for a procedure that always finishes: it is called an "algorithm".

To do addition, you have rules that involve making decisions (for example, if the sum in one column is higher than 10, you need to carry the tens value), and keeping going (you keep adding the columns from right to left until you are done), as well as a way to keep track of what you are doing (when you add a column, you remember in your head what the sum is, and you can do this because you learned how to add any two digits). It turns out that thow three kinds of rules are *all* you need to do any computation! If you can keep track of what you are doing, make decisions, keep going, and have a big enough sheet of paper to write down everything you want, you can do every calculation. What you are doing is very similar to what a computer does — computers are all around you: in your video game machine, in your cell phone, in your iPod, etc. All of these computers might seem very different, but other than differences in their screens and how much memory they have (this is like the size of the paper you can write things on), what they can do is exactly the same, and it is exactly the same as what you can do following rules on paper. The difference is the computers can carry out the steps much faster, they never get bored, and they hardly ever make mistakes. This was understood way back before even your parents were born (before there were any video games, cell phones, or iPods!) by Alonzo Church and Alan Turing, who started out understanding what computers could do by thinking about how fifth graders do arithmetic.

Problem 2: Turing Machines. (25)

b. [15] Define a *cyclical Turing machine* as a Turing machine with a one-way infinite tape (as in our standard Turing machine definition), except that the left edge behavior is defined differently. Instead of staying in the leftmost square, if a cyclical Turing machine would move left past the left edge of the tape, the head moves to the rightmost non-blank square.

Prove that the class of languages that can be recognized by a cyclical Turing machine is identical to the class of languages that can be recognized by a regular Turing machine.

Answer: To prove two computing models are equivalent, we need to show that each model can simulate the other model. So, we need to show that (1) a cyclical Turing machine (CTM) can simulate a regular Turing machine (RTM), and that (2) an RTM can simulate a CTM.

(1) To simulate an RTM with a CTM, we need to avoid the cycling behavior. We can do this by adding a previously unused symbol, %, to the tape alphabet. To simulate an RTM with a CTM, we add a new start state which walks to the end of the input, and then moves the original input to the right one square by copying each symbol to the square to the right, and then moving two squares left to the next symbol. After this, the original first square is not unused. Write a % in that square. Then, simulate the original RTM, starting on the second square, but with an additional transition rule for whenever the input symbol is % to move right one square. This simulates the normal left edge behavior of the RTM.

(2) To simulate a CTM with an RTM, we need to simulate the cycling behavior. Doing this correctly is a little tricky, since at the beginning we know the leftmost blank square is the end of the input, but there is no way to ensure this is always the case. To avoid this problem, we add a special mark for the rightmost used square. This mark needs to be maintained by the transition rules, so whenever the machine would move right from the marked square, it enters a new state that marks whatever is written on the new square. In the case where the marked square is overwritten with a blank, the rules needs to mark the square to its left as the new rightmost square. So, to simulate the CTM, at the beginning the RTM finds the rightmost non-blank and adds a mark to it. As in case (1), we move all the input to the right one square to make room for the % marker at the leftmost square. A transition rule is added so that whenever the % is encountered, the machine moves to the rightmost square by finding the square with the special mark.

Thus, we can do the simulations in both directions, so the computing models are equivalent.

Problem 4: Decidability. (30) For each part, state clearly whether the described language is *decidable* or *undecidable*. Support your answer with a convincing proof. You may use any of the theorems we have proven in class or in the book, but you may not use Rice's theorem in your proof unless you also include a proof of Rice's theorem.

b. [10] $NOTSUB_{TM} = \{\langle A, B\rangle\}$ where A and B are descriptions of Turing machines and there is some string w which is accepted by A that is not accepted by B (that is, the language accepted by A is not a subset of the language accepted by B).

Answer: We show two different proofs which use reductions from different problems.

Proof by Reduction from EQ_{TM}. We show that $NOTSUB_{TM}$ is undecidable by showing that a decider for $NOTSUB_{TM}$ could be used to build a decider for EQ_{TM}.

Assume $NOTSUB_{TM}$ is decidable. Then, there exists a TM M_{NS} that decides $NOTSUB_{TM}$.

We can construct M_{EQ} that decides EQ_{TM} using M_{NS}:

$M_{EQ}(\langle A, B\rangle)$:

1. Check that the input is a valid representation of two Turing machines. If not, **reject**.
2. Simulate M_{NS} on $\langle A, B\rangle$. If it accepts, **reject**.
3. Simulate M_{NS} on $\langle B, A\rangle$. If it accepts, **reject**.
4. Otherwise, **accept**.

This correctly decides EQ_{TM} since it accepts $\langle A, B\rangle$ only if M_{NS} rejects both $\langle A, B\rangle$ (meaning there is no string w that is accepted by A that is not accepted by B) and $\langle B, A\rangle$ (meaning that there is no string w that is accepted by B and not accepted by A), hence A and B must be the same language.

This proves that M_{NS} cannot exist, since we know M_{EQ} cannot exist. Thus, $NOTSUB_{TM}$ is undecidable.

Proof by Reduction from A_{TM}. We show that $NOTSUB_{TM}$ is undecidable by showing that a decider for $NOTSUB_{TM}$ could be used to build a decider for A_{TM}.

Assume $NOTSUB_{TM}$ is decidable. Then, there exists a TM M_{NS} that decides $NOTSUB_{TM}$.

We can construct M_A that decides A_{TM} using M_{NS}:

$M_A(\langle M, w\rangle)$:

1. Check that the input is a valid representation of a Turing machine and an input pair. If not, **reject**.
2. Construct M_w, a TM that simulates M running on input w (by erasing the given input, and writing w on the tape, and then simulating M).
3. Construct M_{ACCEPT}, a TM that accepts all inputs.
4. Simulate M_{NS} on $\langle M_{ACCEPT}, M_w\rangle$. If it rejects, **accept**. If it accepts, **reject**.

This correctly decides A_{TM} since if M accepts w then M_w accepts all inputs, so $L(M_{ACCEPT})$ is a subset of $L(M_w)$ and M_{NS} will reject $\langle M_{ACCEPT}, M_w\rangle$, and the M_A machine correctly accepts. Similarly, if M does not accept w, then M_w rejects all inputs, so $L(M_{ACCEPT})$ is not a subset of $L(M_w)$ and M_{NS} will accept, and M_A correctly rejects.

c. [10] $L_{BusyBee} = \{\langle M, w, k\rangle\}$ where M describes a Turing machine, and k is the number of different FSM states M enters before halting on input w. (Note that q_{Accept} and q_{Reject}

are counted as states for the number of different states count.) (The same language as in Problem 3c.)

Answer:

Proof by Reduction from $HALT_{TM}$.

Assume $L_{BusyBee}$ is decidable. Then, there exists a TM $M_{BusyBee}$ that decides $L_{BusyBee}$.

We can construct M_{HALT} that decides $HALT_{TM}$ using $M_{BusyBee}$:

$M_{HALT}(\langle M, w \rangle)$:

1. Check that the input is a valid representation of a Turing machine and an input pair. If not, **reject**.
2. Extract the number of states, n, from the description M. (This is easy, just count the states in the description.)
3. For each $i = 1, 2, \ldots, n$, simulate $M_{BusyBee}$ on $(\langle M, w, i \rangle)$. If it accepts, **accept**.
4. If none accept, **reject**.

This works since we know n is finite since the number of states in a TM must be finite, so trying all of the i values must eventually terminate. If M would halt on w, it must halt after entering some number of states that is between 1 and n, so if it would halt one of the $M_{BusyBee}$ simulations must accept.