CS–172 Computability & Complexity, Spring 2021

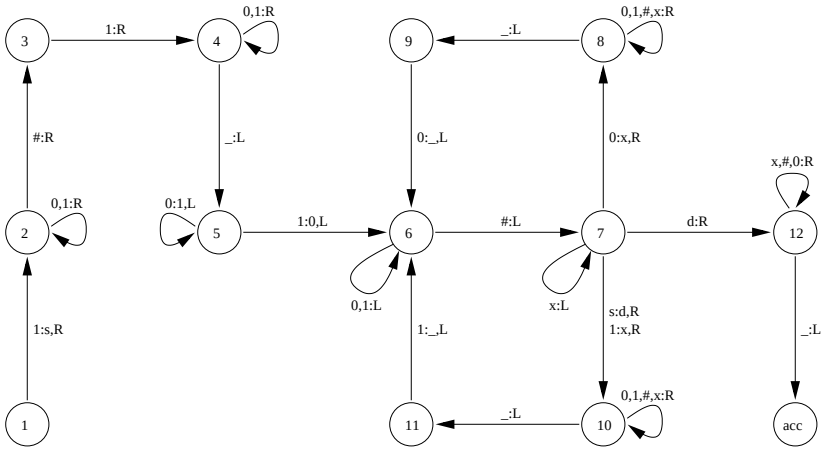
Homework 5 Solutions

Note: These solutions are not necessarily model answers. Rather, they are designed to be tutorial in nature, and sometimes contain more explanation than is required for full points. Also, bear in mind that there may be more than one correct solution. The maximum total number of points available is 31.

1. The following state diagram describes a TM that accepts this language. The TM has twelve states (plus accept and reject states), the input alphabet $\Sigma = \{0, 1, \#\}$ and tape alphabet $\Gamma = \{0, 1, \#, x, s, d, \_\}$, where $\_$ denotes the blank symbol. The start state is 1. It follows some Sipser-induced state diagram conventions: *8pts*

- Any transition not explicitly specified on the diagram is assumed to go to the reject state.
- Any transition of the form $\sigma : \sigma', L$ means "if the read head is over character σ, write σ' to the tape and move left".
- Any transition of the form $\sigma_1, \sigma_2, ... : L$ means "if the read head is over any of the characters $\sigma_1, \sigma_2, ...$ write that character back to the tape and move left".

The basic idea behind the TM is to subtract 1 from the second number in the input, and then compare the resulting numbers for equality. (Obviously other designs are possible, such as comparing the two numbers directly without first doing the subtraction.)



States 1-4 make sure the input is well-formed. State 1 ensures that the most significant bit of the first number is a 1, and replaces it with an s to indicate the beginning of the tape. State 2 verifies that there is one hash mark. State 3 ensures that the first digit of the second number is a 1. State 4 moves the head to the end of the tape, implicitly verifying that there aren't any additional hash marks.

State 5 performs the subtraction, flipping all the least significant bits of the second number up to the first 1, which is then flipped to 0. So 1010 becomes 1001, and 10000 becomes 01111 (this special case is handled in state 12 below).

The rest of the machine simply compares the resulting numbers by zig-zagging back and forth, starting with the least significant bits. States 6 and 7 move the head left, finding the least significant bit of the first number that hasn't yet been crossed off. If that bit is a 0, we travel to states 8 and 9, which cross it off, find the LSB

of the second number, make sure it is a 0 as well, and erase it before returing to state 6. If the bit is a 1, we go to states 10 and 11, which do the equivalent thing for 1.

This process is repeated until either (a) the digits don't match at some point, in which case the TM (implicitly) rejects; or (b) we reach the most significant bit of the first number, denoted by s. In this case, we replace it with a d, verify that it matches the final bit of the second number and, upon going back to the d again, we proceed to state 12, which travels along the entire input from left to right one last time, making sure that there aren't any stray 1s left over. It allows for extra 0s to account for the $10^n \rightarrow 01^n$ case, but note that this is the *only* case in which a stray 0 could occur without any stray 1s occuring as well.

If we find nothing out of the ordinary (that is, a bunch of xs, a #, and perhaps a 0 left over), we accept.

[*For full credit on this problem, you need to provide a clear description of your machine, and give evidence that you have tested it on the sample inputs. Turing machine diagrams with no description of how the machine operates received no credit. (This is analogous to submitting undocumented code in a programming class.) The most common omission was forgetting to check that the input is well-formed.*]

2. (a) We claim that a TM with arbitrary jumps is equivalent to the standard TM. It is immediate that the TM with arbitrary jumps is at least as powerful as the standard model. We only need to show how a TM with arbitrary jumps can be simulated by a standard TM. The basic idea is the following. A jump of size k can be simulated on a standard TM by going through a sequence of k simple moves (where we use $(k-1)$ new states to implement such a jump). Since the TM to be simulated contains only a finite number of jump sizes, we need only add a fixed number of new states to our control. More formally, the new TM replaces each transition of the form $\delta(q,a) = (q', b, (D,k))$ by adding $k-1$ new states $q_1^a \ldots q_{k-1}^a$ and adding transitions $\delta(q,a) = (q_1^a, b, D), \delta(q_i^a, ?) = (q_{i+1}^a, ?, D)$ and $\delta(q_{k-1}^a, ?) = (q', ?, D)$. This completes the proof that the two models are equivalent. *5pts*

[*Some students used a standard TM to simulate a jump of size k by self-looping on a* single *state $k-1$ times. This is not valid since the state itself cannot memorize how many times it has self-looped. Some students only stated that intermediate states are added but they did not specify the transitions of these intermediate states.*]

(b) We claim that a TM with left reset is equivalent to a standard TM. To simulate a given TM M with left reset, we first shift the entire input one square to the right, place a special new symbol, say $, at the left end of the tape, and position the head on the first symbol of the input. This concludes the preprocessing phase, and we now proceed to simulate M. Right moves are simulated directly. A left reset move is simulated by going into an intermediate state, which moves the head left until it hits the $ sign, and then moving to the appropriate state of M. More formally, for each left reset move $\delta(q,a) = (q', b, \perp)$ in the transition function of M, we introduce the transitions $\delta(q,a) = (\hat{q'}, b, L)$, $\delta(\hat{q'}, ?) = (\hat{q'}, ?, L)$ and $\delta(\hat{q'}, \$) = (q', \$, R)$. *5pts*

The other direction is slightly more involved. Let M be a standard TM that we wish to simulate by a TM with left reset. A first thought would be to simulate a left move of M by marking the current cell, resetting, and then moving right until we reach the marked cell. However, at the end of this sequence we would be at the original cell, not the one to the left as desired!

One solution, then, is to copy the entire contents of the tape on every left move. We assume that the simulating TM keeps track of the non-blank portion of M's tape, and that this portion is always bounded on the left by a marker \$ and on the right by a marker &. We call this part of the simulating tape the *valid* part. The valid part is the rightmost non-blank portion of the tape. To the left of the valid part there may be other (invalid) symbols; we can always tell whether a symbol is valid or invalid because the invalid symbols have been "crossed." To simulate a left move, we first mark the cell we

are at and then reset to the left. We then search right until we find the first valid symbol (which must in fact be \$), cross it, and copy it to the position immediately to the right of the &. We proceed in the same fashion with all the valid symbols, copying each one to the right-hand end of the tape using a left-reset followed by a right search. As we do this, we check to see if the right neighbor of the cell currently being copied is marked; if so, we know that the current cell should be the new head position, so when we copy it we mark it. This copying process continues until we encounter the right marker &, which we change to a \$ (it becomes the new left marker) and copy to the right-hand end. Finally, we reset once more and move the head right until we encounter the marked square, which is the new head position. This completes the simulation of one left move! Right moves are simulated directly, with the minor detail that we may have to move the right end marker & one space to the right.

[*When simulating a standard TM by a left-reset TM, some students used left-reset then scrolled right directly; this leads to the original tape cell instead of the one to the left. Some students did not properly justify how to simulate a left-reset TM with a standard TM.*]

(c) We show that this restricted one-way TM is less powerful than the standard TM. We will do this by showing that a one-way TM can be simulated by a DFA. The first step is to show that a one-way TM as defined in the question can be simulated by a *right-only* TM, that moves right on every transition. To see this, consider the one-way TM's behavior at a given cell of the tape. Suppose that on state q_0 and tape symbol a_0, it writes a_1, transitions to state q_1, and does not move the head. Consider the sequence of configurations $uq_0a_0v, uq_1a_1v, \ldots$. The head either (i) eventually moves right after k steps (for some k) and enters configuration ua_kq_kv; or (ii) loops at this cell forever (after making at most $|Q| \times |\Sigma|$ transitions, as this is the maximum possible number of state-symbol pairs, after which time looping must occur); or (iii) eventually goes into a halting state at this cell. In case (i), we can just simulate this entire sequence of transitions with a single right move by setting $\delta(q_0, a) = (q_k, a_k, R)$. In case (ii) we can go directly to the reject state. And in case (iii) we can just go directly to the appropriate halting state. Moreover, this case analysis can be hard-wired into the transition function of the right-only TM. *5pts*

Now note that a right-only TM can never read any symbol it writes itself, and hence its tape is effectively read-only. Thus it can be simulated by a DFA. Since we know that DFAs are less powerful than TMs, we can conclude that one-way TMs cannot simulate standard TMs.

[*It's not enough to just say that a one-way TM is "clearly less powerful" than a standard TM for the simple reason that it cannot move its head left on its tape. This is just not enough of a reason. (For example, other restrictions to the standard model, such as limiting the number of states or the tape alphabet, do not result in a less powerful model.) To show that the one-way TM is less powerful, you actually have to give an example of a language it cannot recognize (but the standard model can). The easiest way to do this is to show that it's actually equivalent in power to a DFA. Finally, some students simply claimed that the one-way TM can be simulated by a DFA (or a PDA) without showing how the transitions of the TM can actually be simulated by a DFA (or PDA).*]

3. (a) **No**. Such a RAM, together with its program, is actually a finite automaton. To see this, note that the set of possible configurations of the RAM is finite. (Recall that a configuration of a RAM consists of its *state* — i.e., the contents of all the registers — and its program counter.) Specifically, if m is the number of instructions in the program, and n is the number of registers, then the number of distinct configurations is $m(2N+1)^n$, since there are m possible values for the program counter and $2N+1$ values for each of the n registers. We can think of these configurations as the states of the FA; in each configuration, a transition to another configuration is made according to the action of the program instruction pointed to by the program counter. (If the instruction is not a `read` instruction, this will *4pts*

actually be an ϵ-transition.) Hence such a RAM is in fact only a FA, so can only accept a regular language. [NOTE: It is not enough here to simply say that the technique we used in class to simulate a TM on a RAM no longer works: this only shows that a particular method can't be used. The above argument rules out *any* possible simulation.]

[*Quite a lot of students argued that the limited RAM can't accept certain recognizable languages because it doesn't have enough space to store the whole of its input in memory. But this argument isn't valid because the RAM model treats its input as a stream, and there is no requirement that it all be stored in memory at the same time. Make sure you read the definition of the RAM model! Another common mistake in this problem was essentially the same as in 2(c) above: it is* **not** *enough to just say that the RAM with finitely many registers is "obviously" less powerful than the RAM with infinitely many.*]

(b) **No**. The argument is essentially the same as in part (a). The key additional observation is that, if the registers are bounded, then it is only possible to access a finite number of them, so really this RAM model is the same as that in part (a). To justify this claim, as in part (a) let m be the number of instructions in the RAM's program. Then the RAM can write to at most m different registers by direct addressing (one for each program instruction), and at most $2N + 1$ registers by indirect addressing (because of the bound on the size of registers). Hence the total number of registers that can ever contain a non-zero value is at most $m + 2N - 1$. This means that the RAM is again a FA, with at most $m(2N + 1)^{m+2N+1}$ states. [Note the contrast between restricting the number of registers (even when there are only two, the model can still simulate any TM) and restricting their size (which allows us to only accept regular languages).] *4pts*

[*Quite a few people incorrectly answered 'yes' to this part. The most common misconception was to forget the fact that, whenever a RAM wants to use a register, the index of that register must first be explicitly written down somewhere (either in a program instruction, or as the result of an arithmetic operation which must then be written to a register).*]