Theory of Computation: Final Exam

Arjun Chandrasekhar

Due Thursday, 12/16/2021 at 11:59 pm (50 points)

1. (10 points) We prove this in two directions. First, we show that if L is Turing-recognizable, then L can be recognized by a 2DTM. Let M_L be the TM that recognizes L. We will design a 2DTM M_{L2} to recognize L by simply simulating M_L and ignoring all of the non-input rows of the 2D grid. Additionally M_{L2} will place a special marker \$ to the left of the input. If the 2DTM ever encounters this symbol, it will immediately move right, to simulate the fact that M_L is supposed to stay in place if it ever tries to move past the leftmost tape square.

   To prove the backwards direction, we show that if L is recognized by a 2DTM, then it can be recognized by a normal TM. Let M_{L2} be the 2DTM that recognizes L. We will design a machine M_L that simulates M_{L2}. To do this, M_L will keep track of what row/column coordinate M_{L2} is at each point in time. It will have a special tape section that stores the machine's coordinate (we can treat the original starting point as $(0, 0)$). Additionally, M_L will keep track of all of the non-empty rows that are in use by M_{L2}. Each row of input will be tracked in a separate portion of the infinite 1D tape, with the various rows sections separated by a special symbol not part of the original tape alphabet. At every step, the TM will write and change its state as the 2DTM would. Based on whether the 2DTM is supposed to move up, down, left, or right, we will update the stored coordinate, and go to the corresponding section of the tape.

2. (10 points) Suppose L is Turing recognizable. There is a machine M_L that recognizes L. We will design a machine M_D that recognizes DROP-OUT$(()L)$ as follows:

   1. M_D takes w as input
   2. Try all possible ways of inserting a character at some position in w. In particular, try all possible characters σ, and for each character, try all positions i where σ could be inserted into w to create a new string w'.
   3. Run M_L on all possible w'. **Do this in parallel**.
   4. If any w' are accepted, then accept w.
   5. If none of the w' are accepted, then our machine will either reject or loop forever.

   If $w \in$ DROP-OUT(L) then at least one way of inserting a character into w will create a new string w' that is accepted by M_L. If $w \notin$ DROP-OUT(L) then every w' will be rejected or loop forever. This may lead to M_D looping forever, but it will not accept, which is all we need in order to simply recognize (and not decide) DROP-OUT(L).

3. We will decide the language using the following algorithm:

   1. Create a DFA D_3 that recognizes $L(D_1) \cap L(D_2)^c$.
   2. Use a decider for $\mathrm{E}_{\mathrm{DFA}}$ to check if $L(D_3) = \emptyset$
   3. If $L(D_3) = \emptyset$, then accept $\langle D_1, D_2 \rangle$. Otherwise reject.

   If $L(D_1) \subseteq L(D_2)$ then D_1 should not accept any strings that are not accepted by D_2. Therefore $L(D_1) \cap L(D_2)^c$ should be empty. We check for this to decide whether $L(D_1)$ is contained in $L(D_2)$.

4. (10 points) AFSOC ALL_{HALT} is decidable. Let M_A be a machine that decides ALL_{HALT}. We will use M_A to construct a machine M_H that decides HALT. M_H does the following:

   1. M_H takes $\langle M, w \rangle$ as input.
   2. Create a machine P that does the following:

      a. P takes s as input
      b. Ignore s and run M on w (which are hard-coded constants)

   3. Use M_A to check if P halts on all inputs
   4. If M_A accepts $\langle P \rangle$, then M_H accepts $\langle M, w \rangle$ **5.** Otherwise, M_H rejects $\langle M, w \rangle$

   Note that P always does the same thing, namely running M on w. So if M halts on w, then P will always halt. Otherwise, P will always loop. Thus, if we can decide whether P halts on all inputs, we can infer whether M halts on w.

   However, we know that HALT is undecidable! We have arrived at a contradiction. We conclude that ALL_{HALT} must be undecidable.

5. (a) (10 points) $L \in \text{P}$. We can decide it using the following algorithm:

      1. Take $\langle G, k \rangle$ as input.
      2. If $k > 10$, reject $\langle G, k \rangle$
      3. For each combination of vertices $S \subseteq V(G)$ of size k do the following:
         a. Check if every pair of vertices $u, v \in S$ is connected.
         b. If S is a clique of size k, then accept $\langle G, k \rangle$. Otherwise, continue to the next subset.
      4. If no cliques were found, reject $\langle G, k \rangle$

   Checking if $k \leq 10$ can be done in polynomial time. If $k \leq 10$, the number of vertex subsets that we have to check is at most $O(V^{10})$. Checking if a subset of vertices forms a clique takes $O(V^2 \cdot E)$ time. Thus, the overall runtime is polynomial.

   (b) (10 points) L is NP-Complete. First we will show that $L \in \text{NP}$ by constructing a polynomial-time verifier V that does the following:

      1. Take $\langle G, k, I \rangle$ as input, where $\langle G, k \rangle$ are the input to the original problem, and I is a proposed independent set.
      2. If $k < 10$, reject $\langle G, k, I \rangle$
      3. Check if $|I| = k$. If not, reject $\langle G, k, I \rangle$
      4. Check if I is a subgraph of G. If not, reject $\langle G, k, I \rangle$
      5. Check if every pair of vertices $u, v \in I$ is disconnected.
      6. If I is a valid independent set, accept $\langle G, k, I \rangle$. Otherwise, reject $\langle G, k, I \rangle$

   The length of the certificate I is at most $O(k)$. It takes $O(k)$ time to check that I is a big enough independent set. It takes $O(V)$ time to verify that I is a subgraph of G. Finally, it takes $O(V^2)$ time to verify that I is a valid independent set. Thus, L can be verified in polynomial time.

   Next we will show that L is NP-Hard by reducing from independent set. Our reduction will work as follows:

      1. Take $\langle G, k \rangle$ as input
      2. Create a graph G' which is a copy of G along with 10 new nodes that are isolated from the rest of the graph.
      3. Output $\langle G', k + 10 \rangle$

   The reduction involves constructing G' and calculating $k + 10$, both of which can be done in polynomial time. Now we just need to prove the reduction is correct.

Yes maps to yes: Suppose G has an independent set I of size k. Then I along with the 10 new nodes forms an independent set of size $k + 10 \geq 10$ in G'.

No maps to no: Suppose G does not have an independent set of size k. The biggest independent set in G has size $j < k$. Then the biggest independent set in G' will have size $j + 10 < k + 10$.