Problem Set for Tutorial 8 — TDDD14/TDDD85

1 Turing Machines

Exercise 1. Construct a Turing machine which, for input alphabet $\Sigma = \{0, 1\}$, computes the *complement* of a Boolean input string, i.e., each bit $b \in \{0, 1\}$ is replaced by its complement $\bar{b} = 1 - b$. After having computed the complement the Turing machine should return to the leftmost position of the tape and accept. For this exercise you should give a *formal* definition of your machine which uses the definition of a Turing machine as a 7-tuple (see the course book or the lecture notes for details).

Exercise 2. Construct Turing machines for solving each of the following problems:

1. Accept the language $\{0^m 1^n \mid 0 \leq m \leq n\}$.
2. Accept the language $\{x \in \{a, b\}^* \mid x$ contains the same number of a's and b's$\}$.
3. If the string 1^n is placed on the tape, the TM generates the string $(01)^n$.
4. If the string $1^m \sharp 1^n$ is placed on the tape, the TM generates the string 1^{mn} (i.e. multiplication).

High-level descriptions are sufficient but you should clearly specify how your machines operates on the tape, and when they accept.

Exercise 3. For the following statements, indicate whether they are true or false and give a justification:

1. The intersection of two decidable languages is decidable.
2. The union of two Turing-recognizable languages is Turing-recognizable.
3. If the complement $\bar{L}$ of a Turing-recognizable language L is decidable, then L is decidable.
4. Let L_1 and L_2 be Turing-recognizable languages. Then $L_1 L_2$ (recall the notation for set concatenation) is Turing-recognizable.

Solutions

Solution to Exercise 1. The high-level idea is simple: we go to the right of the tape and replace each bit by its complement. When we reach the first blank symbol B we have reached the end of the tape and should then go to the left without altering anything. Our machine is then a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where:

- $Q = \{q_s, q_l, q_r, q_{\text{accept}}, q_{\text{reject}}\}$ is the finite set of states, where q_s is the start state, q_l is the state which instructs the machine to go from right to left, q_r the state which goes to the right and flips each input bit, and q_{reject} a reject state (which we do not really need but is still part of the formal definition), and q_{accept} the accept state.
- $\Sigma = \{0, 1\}$ is the input alphabet,
- $\Gamma = \{0, 1, B\}$ is the tape alphabet,
- $\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function.

We continue by specifying the transition function. Here, we realize that we somehow have to recognize when we have reached the initial position of the input tape, since the machine otherwise would get stuck in the leftmost position, repeatedly trying to move to the left, without accepting. There are several ways to handle this and we consider a simple approach where the tape alphabet is augmented with two special symbols $\underline{0}$ and $\underline{1}$. In the start state we then replace 0 with $\underline{1}$, or 1 with $\underline{0}$, and when we later on return to this cell of the tape we replace it 0 and 1 and accept. The most important entries of δ are then as follows (we leave the remainding specification of δ as an exercise).

- $\delta(q_s, 0) = (q_r, \underline{1}, R)$,
- $\delta(q_s, 1) = (q_r, \underline{0}, R)$,
- $\delta(q_r, 0) = (q_r, 1, R)$,
- $\delta(q_r, 1) = (q_r, 0, R)$,
- $\delta(q_r, B) = (q_l, B, L)$,
- $\delta(q_l, 0) = (q_l, 0, L)$,
- $\delta(q_l, 1) = (q_l, 1, L)$,
- $\delta(q_l, \underline{0}) = (q_{\text{accept}}, 0, L)$,
- $\delta(q_l, \underline{1}) = (q_{\text{accept}}, 1, L)$.

Hence, to simplify the task we expanded the tape alphabet so that one could mark symbols in a convenient way and return to the leftmost position of the tape. Can you think of a solution to this problem where the tape alphabet is not altered, i.e., is $\{0, 1, B\}$?

Solution to Exercise 2. We construct Turing machines for each specified task as follows.

1. We consider two descriptions, one which requires few additional assumptions, and one which is more complicated, but is closer to how one would solve the problem in a programming language. The first TM begins by scanning the tape from left to right and makes sure that the input is of the form 0^*1^*, and the returns to the beginning of the tape. It then proceeds to the right of the tape until it encounters the first 1. It replaces 1 by a blank symbol and enters a new state in which it goes back to the left of the tape until it reaches the first 0, which it replaces by a blank symbol. It continues in this fashion and goes back and forth, continously erasing a 0, followed by a 1, and aborts if it sees an unexpected symbol, or if the tape has run out of 0's. After exhausting all 0's it accepts.

   The second TM uses a counting strategy instead. The basic idea is simply to count the number of 0's and make sure that this number does not exceed the number of 1's. How can this be accomplished? We begin by checking that the input is well-formed (as before), and then partitions the tape into two parts, Left and Right, where Left contains the input, and where Right will be used to keep track of the number of symbols. We could, for example, represent this as a binary number, and introduce states for incrementing and decrementing this number by 1 (reaching a trash state if one tries to decrement 0). Whenever the machine reads a 0 it then increments the count, and then continues with 1's in which case it instead decrements the count.

2. We introduce a special symbol $\star$ which we will use to remove symbols from the tape (we avoid writing a blank since we could then struggle to find the rightmost position, which is always blank, initially). Starting from the left, it replaces the first a with $\star$, and then scans the tape to the left/right until it finds a b which it replaces by $\star$. It goes back to the beginning of the tape and repeats this process until no more a can be found. The TM accepts if the tape is empty, i.e., every symbol is blank or the special symbol, otherwise it rejects.

3. We interpret this exercise as follows: if the input is well-formed (1^n for some $n \geq 1$) then the TM should produce $(01)^n$, and otherwise its behaviour does not matter (so we could simply accept without altering the tape). This initial check can easily be accomplished, so we henceforth assume that the input is 1^n for some $n \geq 1$. We then start at the leftmost 1, change it to 0, and move right. Then we shuffle the remaining tape one symbol to the right, and write 1 in the newly freed cell. We repeat this process until no more 1's are found, and accept.

4. The answer is left as an exercise.

Solution to Exercise 3.

1. Assume that we have two decidable languages L_1, L_2. We show that $L_1 \cap L_2$ is decidable.

   L_1, L_2 are defined by some total TM's (Turing machines that halt on every input) $L_1 = L(M_1)$ and $L_2 = L(M_2)$. We can construct a TM M that, on an input string x, behaves in the following way:

- M simulates step-by-step the computation of M_1 on input x until M_1 halts,
- if M_1 rejects x then M halts and rejects,
- otherwise M simulates the computation of M_2 on input x,
- if M_2 accepts x then M halts and accepts,
- otherwise M halts and rejects.

We need x available when starting simulation of M_2; for this purpose a copy of x can be kept at the beginning of the tape. Obviously, x is accepted by M iff it is accepted by M_1 and by M_2. Thus $L(M) = L(M_1) \cap L(M_2)$. Notice that if M_1 and M_2 are total TM's then M is total, thus $L(M)$ is decidable.

2. Assume that we have two Turing-recognizable languages L_1, L_2 defined by two TM's; $L_1 = L(M_1)$ and $L_2 = L(M_2)$. We show that $L_1 \cup L_2$ is Turing-recognizable.

   We can construct a TM M that simulates the computations of M_1 and M_2 on the same input string "in parallel", performing one step of M_1, one step of M_2, and so on. M can be a two tape TM, one tape would be the tape of M_1, the other of M_2. If one of the simulated machines accepts, then M accepts. Clearly, $L(M) = L(M_1) \cup L(M_2)$.

   Notice that we cannot first perform the computation of M_1 and then of M_2; the first computation may not halt and thus the computation of M_2 may never begin.

3. Construct a total TM for L out of TM's for L and $\bar{L}$. The construction is similar to the previous one.

4. A TM M for L_1L_2 would first split the input string x into two ($x = y_iz_i$) in all $|x|+1$ possible ways ($i = 1, \ldots, |x|+1$). Then for each of them M checks whether $y_i \in L_1$ and $z_i \in L_2$. The checks have to be done in parallel.[a] M accepts when some pair of checks succeeds. (This implies that $x \in L_1L_2$. Conversely, if $x \in L_1L_2$ then at least one pair of checks succeeds.)

   We cannot use $2(|x|+1)$ tapes, as the number of tapes has to be fixed (independent of x). So we split a tape into $2(|x|+1)$ sections, each section would play the role of the tape of a TM performing one of the checks. If the TM needs more tape cells than assigned, the tape section is enlarged by shifting all the tape sections that are to the right of the given one. The second tape of M would be used for bookkeeping (which section is dealt with now, etc.).

[a] More precisely, for a given splitting $x = x_1x_2$ the two checks for $x_1 \in L_1$ and $x_2 \in L_2$ can be performed sequentially.

2 Advanced and Exam Like Exercises

Exercise 4. Which of the following statements are always true, respectively, false?

1. All decidable languages are mapping reducible to each other (if A and B are decidable languages then $A \leq_m B$ and $B \leq_m A$). If the statement is false, can you identify any conditions under which it holds?

2. If A is a decidable language and B an undecidable language then $A \leq_m B$.

Exercise 5. Let A, B and C be languages (over the same alphabet Σ) where we know that $A \leq_m B$ and $A \leq_m C$ (recall that $L \leq_m L'$ denotes that there exists a mapping reduction from L to L').

1. Prove or disprove that $A \leq_m B \cup C$ always holds.

2. Prove or disprove that $A \leq_m B \cap C$ always holds.

Solutions

Solution to Exercise 4.

1. Hint: try to find a counter example consisting of the two "extremes": the smallest language $\emptyset$ and the largest language Σ^* over an alphabet Σ. If A and B are decidable, but distinct from $\emptyset$ and Σ^*, try to construct a mapping reduction which for any input string $x \in \Sigma^*$ decides whether $x \in A$ or $x \notin A$ (which can be done since A is decidable) and uses this to decide which string to output.

2. Hint: this can be answered similarly to the previous question. First, if B is undecidable, can it be the case that $B = \emptyset$ or $B = \Sigma^*$? Use this insight to construct a mapping reduction.

Solution to Exercise 5. First recall that for any two languages L and L', $L \leq_m L'$ means that L is "easier than" L' (modulo Turing computability). Hence, for a question like this, if we know that A is easier than B, and A is easier than C, can we then conclude that A is easier than $B \cup C$, respectively, $B \cap C$?

1. Constructing a mapping reduction from A to $B \cup C$ seems difficult under the given assumptions (the underlying problem is that we do not even know that the languages B and C are Turing-recognizable, so any attempt of "simulating" machines for B and C is going to fail). Can we come up with an example where A is not necessarily easier than $B \cup C$? This could, for example, be the case if $B \cup C$ is a very simple language. Note that a single counter example to the claim is sufficient to disprove it since the statement is that $A \leq_m B \cup C$ *always* holds. Here, the answer to Exercise 4 is useful since we have already established that $A \leq_m \Sigma^*$ is in general not true.

   With this in mind, let $A = \{a\}$ be a language consisting of a single string $a \in \Sigma^*$, and let B be any language such that $\emptyset \subset B \subset \Sigma^*$, i.e., B is not empty but is not equal to the full language Σ^* either. Note that this implies that $\emptyset \subset \bar{B} \subset \Sigma^*$ (recall that $\bar{B}$ is the set of all strings not included in B). We claim that $A \leq_m B$ and $A \leq_m \bar{B}$. First, define $f(a) = b$ for some arbitrary $b \in B$. Second, choose a string $c \notin B$ and then for any string $a' \notin A$ define $f(a') = c$. Then, for any string $x \in \Sigma^*$, $f(x) \in A$ if and only if $f(x) \in B$. A mapping reduction from A to $\bar{B}$ can be constructed in the same way. But then $B \cup \bar{B} = \Sigma^*$, and we conclude that A does not admit a mapping reduction to $B \cup \bar{B} = \Sigma^*$ and that the proposed statement is false.

2. This question is similar to the previous one and the answer is left as an exercise.