CSCE-433 Formal Languages & Automata
CSCE-627 Theory of Computability

Spring 2022

Instructor: Dr. Jianer Chen
Office: PETR 428
Phone: 845-4259
Email: `chen@cse.tamu.edu`
Office Hours: MWF 10:30–11:30am

Senior Grader: Avdhi Shah
Office: N/A
Phone: tba
Email: `avdhi.shah@tamu.edu`
Office Hours: tba

Solutions to Assignment #6

1. Let A and B be languages and $A \leq_m B$.
(a) If B is context-free, does that imply that A is also context-free? Why or why not?
(b) If A is context-free, does that imply that B is also context-free? Why or why not?

Solutions.
(a) Not necessary. For example, let $A = \{a^n b^n c^n \mid n \geq 0\}$ and $B = \{a^n b^n \mid n \geq 0\}$. As we studied in class, A is not context-free but B is context-free. Consider the following function:

$$R_a(x) = \begin{cases} ab & \text{if } x = a^n b^n c^n \text{ for some } n \geq 0 \\ aab & \text{otherwise} \end{cases}$$

Obviously, we can construct a Turing machine M_a that computes the function $R_a(x)$ such that the Turing machine M_a halts on all inputs. Moreover, note $ab \in B$ and $aab \notin B$. Thus, $R_a(x)$ is a yes-instance of B if and only if x is a yes-instance of A. Therefore, the function $R_a(x)$ is a mapping reduction from A to B, i.e., $A \leq_m B$. However, B is context-free but A is not context-free.

(b) Not necessary. For example, let $A = \{a^n b^n \mid n \geq 0\}$ and $B = \{a^n b^n c^n \mid n \geq 0\}$. Now A is context-free but B is not context-free. Consider the following function:

$$R_b(x) = \begin{cases} abc & \text{if } x = a^n b^n \text{ for some } n \geq 0 \\ abcc & \text{otherwise} \end{cases}$$

Again $R_b(x)$ can be computed by a Turing machine M_b that halts on all inputs. Moreover, note $abc \in B$ and $abcc \notin B$. Thus, $R_b(x)$ is a yes-instance of B if and only if x is a yes-instance of A. Therefore, the function $R_b(x)$ is a mapping reduction from A to B, i.e., $A \leq_m B$. However, A is context-free but B is not context-free. $\square$

2. Let $L = \{\langle M \rangle \mid M \text{ is a Turing machine that accepts } w^R \text{ whenever it accepts } w\}$. Show that L is undecidable.

Proof. We show a mapping reduction (i.e., an algorithm) R that reduces the halting problem HALT to the language L given in the question, i.e., HALT $\leq_m L$. Since HALT is undecidable, this mapping reduction

R will show the undecidability of the language L. The algorithm R on an instance (M, w) of HALT will produce the encoding of a Turing machine M' such that if (M, w) is a yes-instance of HALT, then the language accepted by M' is $\{001, 100\}$ (thus $\langle M' \rangle$ is a yes-instance of L), while if (M, w) is a no-instance of HALT, then the language accepted by M' is $\{001\}$ (thus $\langle M' \rangle$ is a no-instance of L).

Here is a detailed description of the algorithm R: on an input (M, w) that is an instance of HALT, the algorithm R outputs the encoding $\langle M' \rangle$ of a Turing machine M', which is given as follows:

Turing Machine $M'(x)$
1. if $x = 011$ then accept x;
2. if $x \neq 110$ then reject x;
3. run M on w (i.e., call the subroutine M on input w);
4. accept x

Note that on the input (M, w), the algorithm R *produces* the above code and makes it its output. In particular, R *does not run* the Turing machine M' (especially R does not run step 3 of the Turing machine M'). Therefore, the algorithm R always halts.

First note that the Turing machine M' rejects all strings x if x is not 011 and 110. Moreover, M' always accepts 011. Finally, on input 110, which is the reverse of 011: $110 = 011^R$, the Turing machine M' will reach step 3 and run the Turing machine M on input w, where (M, w) is the input to the algorithm R and is an instance of HALT, and accept 110 if and only if the Turing machine M halts on w. In summary, if M halts on w, i.e., if (M, w) is a yes-instance of HALT, then the language accepted by M' is $\{011, 110\}$, so $\langle M' \rangle$ is a yes-instance of L, while if M does not halt on w, i.e., if (M, w) is a no-instance of HALT, then on input 110, the Turing machine M' will be trapped in step 3 so will not accept 110, so the language accepted by M' in this case is $\{011\}$ and $\langle M' \rangle$ is a no-instance of L.

Therefore, the algorithm R on an instance (M, w) of HALT produces an instance $\langle M' \rangle$ of the language L such that (M, w) is a yes-instance of HALT if and only if $\langle M' \rangle$ is a yes-instance of L. Moreover, R halts on all inputs. In conclusion, R is a mapping reduction from the undecidable problem HALT to the language L. As a consequence, this proves that the language L is undecidable. $\square$

3. A *useless state* in a Turing machine is one that is never entered on any input string. Consider the problem of determining whether a Turing machine has any useless states. Formulate this problem as a language and show that it is undecidable.

Proof. We formulate the problem as the following language:

$$\text{USELESS} = \{\langle M, q \rangle \mid q \text{ is a useless state of the Turing machine } M\}$$

Recall that the complement of the halting problem HALT:

$$\text{NOT-HALT} = \{\langle M, w \rangle \mid \text{The Turing machine } M \text{ does not halt on input } w\}$$

is undecidable (in fact, as we showed in class, NOT-HALT is not even Turing-recognizable). To prove the undecidability of the language USELESS, we construct a mapping reduction R from the undecidable problem NOT-HALT to the problem USELESS, as follows: on an instance (M, w) of NOT-HALT, the mapping reduction R constructs and outputs an instance (M', q_{acc}) of USELESS, where q_{acc} is the unique accepting state of the Turing machine M'. The Turing machine M' works as follows: on any input x, M' first runs the Turing machine M on input w, then enters the accepting state q_{acc} of M' to accept its own input x. Again, we emphasize that the mapping reduction R only *produces* the pair (M', q_{acc}), i.e., the encoding of the Turing machine M' and its accepting state q_{acc}, not running M' on its input x. Thus, the mapping reduction R can be computed by a Turing machine that halts on all inputs. It is easy to see that the Turing machine M' cannot reach its accepting state q_{acc} on any input x (i.e., q_{acc} is a useless state of M' so (M', q_{acc}) is a yes-instance of USELESS) if and only if the Turing machine M does not halt on w (i.e., (M, w) is a yes-instance of NOT-HALT). This verifies that R is indeed a mapping reduction from NOT-HALT to USELESS. Since NOT-HALT is undecidable, we conclude that the language USELESS is also undecidable. □

4. (a) (CSCE 433 students only) Show that P is closed under union, concatenation, and complement.
(b) (CSCE 627 students only) Show that NP is closed under union and concatenation.

Proof.
(a) We first prove that the class P is closed under union and concatenation. Let L_1 and L_2 be two languages in P. Thus, there are (deterministic) algorithms (i.e., Turing machines) M_1 and M_2 that accept L_1 in time $O(n^c)$ and L_2 in time $O(n^d)$, respectively, where c and d are fixed constants. Now consider the following algorithm $M_\cup$:

Turing Machine $M_\cup(x)$
1. run M_1 on x;
2. if M_1 accepts x then accept x;
3. run M_2 on x;
4. if M_2 accepts x then accept x;
5. reject x

It is easy to see that $M_\cup$ accepts x if and only if either M_1 accepts x (i.e., $x \in L_1$) or M_2 accepts x (i.e., $x \in L_2$), that is, if and only if $x \in L_1 \cup L_2$. Thus, the algorithm $M_\cup$ accepts the language $L_1 \cup L_2$. Moreover, let $a = \max\{c, d\}$, then a is also a fixed constant and the algorithm $M_\cup$ runs in time $O(n^c + n^d) + O(1) = O(n^a)$ (where the time $O(1)$ is for the execution of steps 2, 4, and 5), i.e., $M_\cup$ runs in polynomial time. Finally, since both algorithms M_1 and M_2 are deterministic, the algorithm $M_\cup$ is also deterministic. Therefore, the language $L_1 \cup L_2$ is accepted by the deterministic polynomial-time algorithm $M_\cup$, i.e., $L_1 \cup L_2$ is in the class P. This proves that the class P is closed under union.

Now consider the concatenation $L_{cat} = \{x \mid x = x_1x_2, x_1 \in L_1, x_2 \in L_2\}$ of L_1 and L_2. The difficulty here is that we do not know where to break the input x into x_1 and x_2 so that we can get $x_1 \in L_1$ and $x_2 \in L_2$. To resolve this, we simply try all possible ways of breaking. The algorithm is given as follows (note that if $i = 0$ then $a_1a_2 \cdots a_i = \varepsilon$ and if $i = n$ then $a_{i+1}a_{i+2} \cdots a_n = \varepsilon$):

Turing Machine $M_{cat}(x)$
\\ assume $|x| = n$ and $x = a_1a_2 \cdots a_n$
1. for $(i = 0;\ i \le n;\ i{+}{+})$
1.1. run M_1 on $a_1a_2 \cdots a_i$;
1.2. if M_1 accepts $a_1a_2 \cdots a_i$
1.3. then run M_2 on $a_{i+1}a_{i+2} \cdots a_n$;
1.4. if M_2 accepts $a_{i+1}a_{i+2} \cdots a_n$
1.5. then accept x;
2. reject x

If $x = a_1a_2 \cdots a_n$ is in L_{cat}, then there must be an index i_0 such that $a_1a_2 \cdots a_{i_0} \in L_1$ and $a_{i_0+1}a_{i_0+2} \cdots a_n \in L_2$. Thus, when the for-loop of the algorithm M_{cat} reaches $i = i_0$, step 1.5 of the algorithm will accept x. On the other hand, if $x = a_1a_2 \cdots a_n$ is not in L_{cat}, then for any index i, at least one of the conditions $a_1a_2 \cdots a_i \in L_1$ and $a_{i+1}a_{i+2} \cdots a_n \in L_2$ will fail, so the algorithm must reach step 2 and reject x. In conclusion, the algorithm M_{cat} accepts the language L_{cat}. The algorithm M_{cat} is deterministic because both the algorithms M_1 and M_2 are deterministic. Finally, the for-loop in the algorithm M_{cat} runs for n times, in each time it runs the algorithms M_1 and M_2 on inputs of length bounded by n, thus taking $O(n^a)$ time, where $a = \max\{c, d\}$. As a result, the algorithm M_{cat} runs in time $O(n \cdot n^a) = O(n^{a+1})$, which is a polynomial of n. Summarizing the above discussion, we conclude that the language L_{cat} is accepted by the deterministic polynomial-time algorithm M_{cat}, so L_{cat} is in the class P. This proves that the class P is closed under concatenation.

The case for complement is simple. Let $\overline{L}_1$ be the complement of the language L_1, where L_1 is in the class P and accepted by a deterministic polynomial-time Turing machine M_1. We simply swap the accepting states and the rejecting states of the Turing machine M_1 to get a new Turing machine $\overline{M}_1$. Thus, the new Turing machine $\overline{M}_1$ accepts an input x if and only if the Turing machine M_1 rejects x, i.e., $\overline{M}_1$ accepts exactly the complement $\overline{L}_1$ of L_1. Because M_1 is a deterministic polynomial-time algorithm, $\overline{M}_1$ is also a deterministic polynomial-time algorithm that accepts $\overline{L}_1$. This proves that the complement $\overline{L}_1$ of the language L_1 is also in the class P, thus, completing the proof that the class P is closed under complement.

(b) Let L_1 and L_2 be two languages in the class NP. Thus, there are nondeterministic Turing machines M_1 and M_2 that accept L_1 in time $O(n^c)$ and L_2 in time $O(n^d)$, respectively, where c and d are fixed constants.

To show that the class NP is closed under union, consider the following Turing machine $M_\cup$:

```
Turing Machine M∪(x)
1.  nondeterministically pick one of steps (a) and (b):
(a)   run M1 on x;
(b)   run M2 on x.
```

The Turing machine $M_\cup$ is nondeterministic because of step 1 and because the Turing machines M_1 and M_2 are nondeterministic. Moreover, the running time of the Turing machine $M_\cup$ is bounded by the sum of that of M_1 and M_2, i.e., by $O(n^c + n^d) = O(n^a)$, where $a = \max\{c, d\}$ is a fixed constant. Thus, $M_\cup$ is a nondeterministic polynomial-time Turing machine.

We must carefully verify that the Turing machine $M_\cup$ accepts the union $L_1 \cup L_2$ of L_1 and L_2. For this, we must verify that for any $x \in L_1 \cup L_2$, there is a computational path of $M_\cup$ that accepts x, while for $x \notin L_1 \cup L_2$, all computational paths of $M_\cup$ reject x.

Let $x \in L_1 \cup L_2$. Then either $x \in L_1$ or $x \in L_2$. Without loss of generality, suppose $x \in L_1$. Since M_1 accepts L_1, on the input x, there must be a computational path P_1 of M_1 that accepts x. Now the computational path $P_\cup$ of $M_\cup$ on input x that in step 1 (nondeterministically) takes step (a) to run M_1 on x then follows the computational path P_1 of M_1 will accept x. Therefore, for any $x \in L_1 \cup L_2$, there is a computational path of $M_\cup$ that accepts x.

On the other hand, suppose $x \notin L_1 \cup L_2$, i.e., $x \notin L_1$ and $x \notin L_2$. Then no computational path of M_1 and M_2 on input x would accept x. Thus, for the Turing machine $M_\cup$ on input x, no matter which of step (a) or step (b) is taken, and no matter which computational path of M_1 or M_2 is followed, the corresponding computational path of $M_\cup$ will reject x. Thus, all computational paths of the Turing machine $M_\cup$ will reject x.

This verifies that the nondeterministic polynomial-time Turing machine $M_\cup$ accepts the language $L_\cup$, i.e., the language $L_\cup$ is in the class NP. In conclusion, the class NP is closed under union.

To show that NP is closed under concatenation, consider the following Turing machine M_{cat}:

```
Turing Machine Mcat(x)
\\ assume |x| = n and x = a1a2···an
1.  nondeterministically pick an integer i, 0 ≤ i ≤ n;
2.  run M1 on a1a2···ai;
3.  if the computational path of M1 rejects a1a2···ai then reject x;
4.  run M2 on ai+1ai+2···an;
5.  if the computational path of M2 rejects ai+1ai+2···an then reject x;
6.  accept x
```

The Turing machine $M_\cup$ is nondeterministic because of step 1 and because the Turing machines M_1 and M_2 are nondeterministic. Moreover, the running time of the Turing machine $M_\cup$ is bounded by the sum of that of M_1 and M_2 (note that M_{cat} runs each of M_1 and M_2 only once), i.e., by $O(n^c + n^d) = O(n^a)$, where $a = \max\{c, d\}$ is a fixed constant. Thus, $M_\cup$ is a nondeterministic polynomial-time Turing machine.

To verify that Turing machine M_{cat} accepts the language $L_{cat} = \{x \mid x = x_1x_2, x_1 \in L_1, x_2 \in L_2\}$, which is the concatenation of L_1 and L_2, let $x = a_1a_2 \cdots a_n$ be in L_{cat}. Then there must be an index i_0 such that $a_1a_2 \cdots a_{i_0} \in L_1$ and $a_{i_0+1}a_{i_0+2} \cdots a_n \in L_2$. Now for the computational path of M_{cat} that takes the index $i = i_0$ in step 1, the Turing machine M_1 in step 2 will accept $a_1a_2 \cdots a_{i_0}$ so it will reach step 4 to run M_2 that will accept $a_{i_0+1}a_{i_0+2} \cdots a_n \in L_2$. Thus, this computational path of M_{cat} will eventually reach step 6 and accept x. On the other hand, if $x = a_1a_2 \cdots a_n$ is not in L_{cat}, then for any index i (nondeterministically picked at step 1 of the Turing machine M_{cat}), at least one of the conditions $a_1a_2 \cdots a_i \in L_1$ and $a_{i+1}a_{i+2} \cdots a_n \in L_2$ will fail, so the algorithm M_{cat} will either reject at step 3 or reject at step 5, no matter which computational path of M_1 and M_2 is followed. That is, all computational paths of the Turing machine M_{cat} will reject x. This proves that the nondeterministic polynomial-time Turing machine M_{cat} accepts the concatenation L_{cat} of L_1 and L_2, thus, L_{cat} is in the class NP. This completes the proof that the class NP is closed under concatenation. □

5. (a) (CSCE 433 students only) Let COMPOSITE $= \{N \mid N > 0$ is an integer but not a prime$\}$. Prove that the language COMPOSITE is in NP.

(b) (CSCE 627 students only) Two graphs G and H are *isomorphic* if the vertices of G may be renamed so that G becomes identical to H. Prove that the following language is in NP:

$$\text{ISOMORPHISM} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are isomorphic}\}.$$

Proof.

(a) For this question, we need some more detailed and careful understanding of the representation of instances of the problem COMPOSITE. If an integer $N > 0$ is given as a binary number, then it has $\lfloor \log_2 N \rfloor + 1$ bits. If N is given as an instance of COMPOSITE, then its length is $n = \lfloor \log_2 N \rfloor + 1 \approx \log_2 N$. Therefore, when we say that an algorithm solves the problem COMPOSITE in polynomial time, we really mean that the algorithm runs in time that is bounded by a polynomial of the length $n = |N| \approx \log_2 N$ of the input integer N. Thus, to prove that the problem COMPOSITE is in NP, we need to present a nondeterministic algorithm that solves the problem COMPOSITE in time polynomial of $\log_2 N$ on an input integer N.

The idea of the algorithm is simple: to prove that N is not a prime, we need to find an integer N' such that $1 < N' < N$ and that N' divides N. Because our algorithm is nondeterministic, we can simply "guess" the integer N'. The algorithm is given as follows:

```
NotPrime(N)
\\ N is an integer, and n = |N|
1.  nondeterministically guess an integer N' of at most n bits;
2.  if (N' ≤ 1) or (N' ≥ N) then reject N;
3.  If (N' does not divide N) then reject N;
4.  accept N.
```

We give explanations for the above algorithm, prove its correctness, and analyze its complexity. If N is not a prime, i.e., if N is a yes-instance of COMPOSITE, then there must be an integer N_0 such that $1 < N_0 < N$ and that N_0 divides N. In this case, the computational path of the algorithm NoPrime that correctly guessed this N_0 in step 1 will not reject N in steps 2-3 so will reach step 4 and accept N. Thus, for a yes-instance of COMPOSITE, there is at least one computational path of `NotPrime`(N) that accepts N. On the other hand, if N is a no-instance of COMPOSITE, i.e., if N is a prime, then each computational path of `NotPrime` that picks an integer N' in step 1, will either find out that N' is not a proper integer (i.e., N' does not satisfy $1 < N' < N$) then reject N in step 2, or get a proper N' (i.e., $1 < N' < N$) but find out that N' does not divide N (because N is a prime) so reject N in step 3. In conclusion, if N is a no-instance of COMPOSITE, then all computational paths of `NotPrime`(N) will reject N. This verifies that the nondeterministic algorithm `NotPrime` accepts the language COMPOSITE.

What that still remains is to show that the algorithm `NotPrime` runs in time polynomial of $n = \log_2 N$, where n is the number of bits of the binary representation of the integer N. Step 1 takes time $O(n)$ because we can guess a binary bit 0 or 1 in constant time. Step 2 also takes time $O(n)$ because we can compare two binary numbers of at most n bits in time $O(n)$. Step 3 can be implemented using the division algorithm we learned in elementary school, which takes time $O(n^2)$ (students: please verify this). Therefore, each computational path of the algorithm `NotPrime` runs in time $O(n^2)$. Thus, `NotPrime` is a nondeterministic polynomial-time algorithm that accepts the language COMPOSITE.

This completes the proof that the language COMPOSITE is in the class NP.

(b) Assume that the vertices of the graph G are labeled a_1, a_2, ..., a_n, while the vertices of the graph H are labeled b_1, b_2, ..., b_n (note that if G and H have different numbers of vertices, then $\langle G, H \rangle$ is obviously a no-instance of ISOMORPHISM). What we need is a one-to-one mapping h from the vertex set $\{a_1, a_2, \ldots, a_n\}$ of the graph G to the vertex set $\{b_1, b_2, \ldots, b_n\}$ of the graph H that relabels the vertex a_i of G by the vertex $f(a_i)$ of H so that G becomes identical to H. Again, this mapping h can be "guessed" using nondeterminism. The algorithm is given as follows:

```
ISOM(G, H)
\\ the vertex set of the graph G is {a1, a2, ..., an}, and
\\ the vertex set of the graph H is {b1, b2, ..., bn}
1.  for (i = 1; i ≤ n; i++)
        nondeterministically guess an integer k, 1 ≤ k ≤ n, and let h(i) = k;
2.  if {h(1), h(2), ..., h(n)} ≠ {1, 2, ..., n} then reject (G, H);
3.  for (i = 1; i ≤ n; i++)
        for (j = 1; j ≤ n; j++)
           if (ai and aj are adjacent in G but bh(i) and bh(j) are not adjacent in H) or
              (ai and aj are not adjacent in G but bh(i) and bh(j) are adjacent in H)
           then reject (G, H);
4.  accept (G, H).
```

We give explanations for the above algorithm, prove its correctness, and analyze its complexity. If the graphs G and H are isomorphic, i.e., if (G, H) is a yes-instance of ISOMORPHISM, then there is a one-to-one mapping h that maps each vertex a_i in G to its corresponding vertex $b_{h(i)}$ in H, such that for any i and j, the vertices a_i and a_j in G are adjacent if and only if the vertex $b_{h(i)}$ and $b_{h(j)}$ in H are adjacent. Therefore, for the computational path of `ISOM`(G, H) that for every i has guessed the correct $h(i)$ in step 1, the algorithm `ISOM`(G, H) will pass all the tests in steps 2-3, and reach step 4 and accept the input (G, H). Thus, for a yes-instance of ISOMORPHISM, there is at least one computational path of `ISOM`(G, H) that accepts (G, H). On the other hand, if (G, H) is a no-instance of ISOMORPHISM, i.e., if the graphs G and H are not isomorphic, then each computational path of `ISOM` that picks a mapping h in step 1, will either find out that h is not a one-to-one mapping (i.e., $\{h(1), h(2), \ldots, h(n)\} \neq \{1, 2, \ldots, n\}$) then reject (G, H) in step 2, or get a one-to-one mapping h in step 1 but find out that h cannot keep the adjacency relations in the graphs G and H (i.e., for some i and j, either a_i and a_j are adjacent in G but $b_{h(i)}$ and $b_{h(j)}$ are not adjacent in H, or a_i and a_j are not adjacent in G but $b_{h(i)}$ and $b_{h(j)}$ are adjacent in H) so reject (G, H) in step 3. In conclusion, if (G, H) is a no-instance of ISOMORPHISM, then all computational paths of `ISOM`(G, H) will reject (G, H). This verifies that the nondeterministic algorithm `ISOM` accepts the language ISOMORPHISM.

For the complexity of the algorithm `ISOM`, first note that guessing an integer k between 1 and n takes time $O(\log_2 n)$ because the integer k has at most $\log_2 n$ bits (see the discussion in the solution to (a) of this question). As a result, step 1 of the algorithm `ISOM` takes time $O(n \log_2 n)$. Step 2 of the algorithm `ISOM` can be implemented by sorting the integers in $\{h(1), h(2), \ldots, h(n)\}$ to find out if all numbers are distinct (recall that by step 1, we know that $1 \leq h(i) \leq n$ for all i). Thus, step 2 takes time $O(n \log_2 n)$. The loop-body of the double loop in step 3 is executed n^2 time, and each execution of the loop-body takes time $O(1)$ (assume that the graphs G and H are given in their adjacency matrices so that vertex adjacency can be tested in time $O(1)$). Thus, step 3 of the algorithm `ISOM` takes time $O(n^2)$. In summary, every computational path of the algorithm `ISOM` runs in time $O(n \log_2 n + n \log_2 n + n^2) = O(n^2)$, which is a polynomial of n. Thus, `ISOM` is a nondeterministic polynomial-time algorithm that accepts the language ISOMORPHISM.

This completes the proof that the language ISOMORPHISM is in the class NP. $\square$