Bram Schuijff
CSCI 387, §S01/02
Homework 5
March 18th, 2025 by 10:30am

Before I answer any questions, I want it to be clear that all of my counter variables are integers. Therefore, when I say 'for $i \in [0, j]$' in the description of a Turing machine, that means 'for all $i \in \mathbb{Z}$ such that $0 \leqslant i \leqslant j$'.

1. (a) We can produce an algorithm in $\mathbf{P}$ by recognizing the following identity is true for any $a, b \in \mathbb{Z}$ and $p \in \mathbb{Z}_{\geqslant 1}$:

$$[a \cdot b \mod p] = [a \mod p] \cdot [b \mod p] \mod p \tag{1}$$

Returning to our original input, in the example where $b = 2^i$ for some $i \in \mathbb{Z}^+$, we can first find $a \mod p$, then use that to find $a^2 \mod p$, then use that to find $a^4 \mod p$, etc... until we arrive at $a^b \mod p$. However, we can just as easily do this with an increment of 1 in the exponent at each step. More formally, this looks like:

1. On input $\langle a, b, c, p \rangle$:
3. Initialize a temporary variable a' such that $a' = a \mod p$.
4. For $i \in [0, b-1]$:
5. Let $a' = (a \cdot a') \mod p$.
7. If $a' = c$, accept. Otherwise, reject.

This Turing machine decides MODEXP because of Equation 1 and the fact that we multiply a by itself b times mod p and store our result in a'. If $a' = b$ after b iterations, then $a^b = c \mod p$ and we accept. Otherwise, we reject. Thus, this Turing machine decides MODEXP.

To demonstrate that MODEXP $\in \mathbf{P}$, we also need to show that this Turing machine runs in polynomial time. Steps 2 and 3 are clearly polynomial. We have a loop over b which runs in $O(b)$ time. Then, calculating $a' = (a \cdot a')$ is polynomial in a and p. Incrementing a variable is also polynomial. Finally, we accept or reject in polynomial time. Therefore, MODEXP $\in \mathbf{P}$.

(b) No, this extension does not work because we end up producing a variable that has length in $\Theta(2^b)$ and scrolling across it can only be done in exponential time. The aforementioned identity is the only reason that MODEXP can be run in polynomial time because the length of a' is upper bounded by $O(p)$.

2. I will first demonstrate that $\mathbf{P}$ is closed under union. Let $L_1, L_2 \in \mathbf{P}$ be languages Then there exist polynomial-time deciders D_1, D_2 for L_1, L_2 respectively. Select $w \in L_1 \cup L_2$. Then, we can produce a new two-tape Turing machine D' such that D' takes as input $\langle w \rangle$ and behaves as follows:

1. On input $\langle w \rangle$:
2. Copy $\langle w \rangle$ onto the second tape.
3. Run D_1 on the first tape. If it accepts, accept. Otherwise, go to step 4.
4. Run D_2 on the second tape. If it accepts, accept. Otherwise, reject.

D' decides $L_1 \cup L_2$ because if $w \in L_1$, then D_1 on the first tape will accept it and therefore D' will accept. If $w \in L_2$, then D_2 on the second tape will accept it and therefore D' will accept. Because D_1, D_2 are also deciders for L_1, L_2, this means that D' is a decider for $L_1 \cup L_2$.

Copying $\langle w \rangle$ takes $O(n)$ time. Clearly D_1 runs on $\langle w \rangle$ in polynomial time by definition of L_1 being in $\mathbf{P}$ and similarly with D_2. Furthermore, by Theorem 7.8, any multitape Turing machine that runs in $O(t(n))$ time has a single-tape equivalent that runs in $O(t^2(n))$ time. Because the square of any polynomial time algorithm still runs in polynomial time, we have there exists a conversion from D' into a single-tape Turing machine D'' that decides $L_1 \cup L_2$ in polynomial time and that $L_1 \cup L_2 \in \mathbf{P}$.

To prove that $L_1^c \in \mathbf{P}$, let $w \in L_1^c$. Then, we have a simpler construction for a Turing machine D_1^c that decides L_1^c in polynomial time:

1. On input $\langle w \rangle$:
2. Run D_1 on input $\langle w \rangle$. If it accepts, reject. Otherwise, accept.

D_1^c clearly decides L_1^c because if D_1 accepts w, then $w \in L_1$ and therefore $w_1 \notin L_1^c$ and vice versa. D_1^c runs in the same time as D_1 and that runs in polynomial time. Thus, $L_1^c \in \mathbf{P}$.

Finally, to prove that $L_1 L_2 \in \mathbf{P}$, let $w_1 w_2 \in L_1 L_2$. First, I employ the convention of Python-like string slicing: for a string w, let $w[i, j]$ be all characters in the indices $[i, j)$. This strategy employs zero-indexing like Python. We let $w[i, i] = \epsilon$ by convention. We construct a three-tape Turing machine C that decides $L_1 L_2$ as follows:

1. On input $\langle w_1 w_2 \rangle$:
2. For each index $i \in [0, |w_1 w_2|]$:
3. Copy $w[0, i]$ to the second tape and $w[i, |w_1 w_2|]$ to the third tape.
4. Run D_1 on $w[0, i]$ and D_2 on $w[i, |w_1 w_2|]$.
5. If both accept, accept. Otherwise, erase tapes two and three.
6. If we have tried slicing at every index, reject.

C decides $L_1 L_2$ because it exhaustively checks each possible delineation point between w_1 and w_2 and runs the respective Turing machines on the two substrings which decide L_1 and L_2 by assumption. We know that the multitape Turing machine C can be converted to a single-tape Turing machine C' that also runs in polynomial time by Theorem 7.8 so long as C runs in polynomial time. Iterating over each index can be done in $O(n)$ time. Copying and erasing the slices also takes $O(n)$ time. Because $L_1, L_2 \in \mathbf{P}$, running D_1 and D_2 on the slices also takes polynomial time. Accepting and rejecting takes polynomial time. Therefore, $L_1 L_2 \in \mathbf{P}$. Thus, $\mathbf{P}$ is closed under union, complement and concatenation.

3. We can employ dynamic programming to help us by keeping track of all of the substrings that we have tried so far. I will use the same Python-like string slicing convention from the previous question. Let $A \in \mathbf{P}$ be our language and let D be a polynomial-time decider for A. Then, we want to construct a Turing machine D^* such that D^* decides A^* and that D^* runs in polynomial time, thereby proving $A^* \in \mathbf{P}$. We give the following description of D^*:

   1. On input $\langle w \rangle$:
   2. If $w = \epsilon$, accept.

3. Construct an $n \times n$ table where $n = |w|$ where each cell is initially blank.
4. Let the rows be zero-indexed and the columns be one-indexed.
5. Cross out every cell below the main diagonal.
6. For $l \in [1, n]$:
7. For $i \in [0, n - l]$:
8. Let $j = i + l$.
9. For $k \in [i + 1, j - 1]$:
10. If cell (i, k) is True and (k, j) is True, fill in (i, j) as True and break.
11. If $k = j - 1$, run $D(\langle w[i, j]\rangle)$.
12. If D accepts $w[i, j]$, fill (i, j) as True. Otherwise, fill False.
13. Otherwise, continue.
14. If $(0, n)$ is filled as True, accept. Otherwise, reject.

Our table thusly indicates membership of a substring in A^*. For instance, if $(i, j) =$ True in our table, then either we ran D on $w[i, j]$ and it accepted—thereby indicating that $w[i, j] \in A \subseteq A^*$—or there existed some way to slice $w[i, j]$ such that it was split into two substrings $w[i, k], w[k, j] \in A^*$. We start at the base case $l = 1$—checking each character w_i of the input string w to see if $w_i \in A$—and can extrapolate from there via dynamic programming. Eventually, we reach $l = n$ which is the whole input string. If there exists some way to decompose w into substrings in A^*—or if $w \in A$—we accept. Otherwise, we reject. Thus, D^* decides A^*.

To prove that $A^* \in \mathbf{P}$, we can analyze the runtime of D^*. Step 2 takes constant time. Constructing—and indexing—an $n \times n$ table can be done in polynomial time, and thus step 3 is polynomial time. Steps 4 and 5 are just bookkeeping and are clearly polynomial time. Step 6 introduces a loop in $O(n)$ which is polynomial and similarly with step 7. Setting a variable in step 8 can be done in polynomial time. Our loop in step 9 indexes at most $n-2$ elements and is therefore polynomial time. Step 10 involves indexing our $n \times n$ table which is polynomial time. Step 11 runs D and we know that D is a polynomial time decider for A and that $A \in \mathbf{P}$ and thus we can run step 11 in polynomial time. Step 12 is also polynomial time because we are just indexing an $n \times n$ table again. Step 14 is again just indexing. Thus, D^* runs in polynomial time and decides A^*. Thus, $A^* \in \mathbf{P}$ and $\mathbf{P}$ is closed under the star operation.

4. (a) I am going to assume that G is undirected for the sake of simplicity. We can produce a polynomial time nondeterministic Turing machine D_1 that goes as follows:

1. On input $\langle G\rangle$:
2. For each vertex $v_i \in V$:
3. Let $C_i = \varnothing$ be a set (no duplicate values).
4. For each vertex v_j adjacent to v_i:
5. If v_j is colored, let c_j be the color of v_j.
6. Insert c_j into C_i.

7. If $|C_i| = 3$, reject.

8. Nondeterministically select a color $c_i \notin C_i$ for v_i.

9. Accept once each vertex has been assigned a color.

D_1 decides 3COLOR. This is because by looping through each vertex in V, we assure that either each vertex will get colored or we will reject. Then, we also assure that no two adjacent vertices will have the same color by forbidding a vertex to be colored similarly to its neighbor. Because we nondeterminsitically assign colors, we also guarantee we will find a solution if one exists.

To demonstrate that D_1 runs in polynomial time, we can go through each step. Looping through the vertices takes polynomial time in the length of the graph encoding. If we have $\langle G \rangle$ implemented in a sane way like an adjacency matrix, then for each vertex, we can find each vertex adjacent to it in polynomial time. Marking off colors takes polynomial time and so does assignment. Thus, D_1 runs in polynomial time and 3COLOR $\in$ **NP**.

(b) We can produce the following polynomial time verifier D_2 that goes as follows:

1. On input $\langle G, w \rangle$:

2. For each vertex v_i:

3. If v_i is uncolored, reject.

4. Let c_i be the color of v_i

5. For each vertex v_j adjacent to v_i:

6. Let c_j be the color of v_j.

7. Let $c_j = c_i$, reject.

8. Accept once we have checked every vertex.

This verifies the witness w because we check every vertex in the graph and ensure that (1) it is colored, and (2) it does not share a color with any of its adjacent vertices. If every vertex is colored and no vertices share colors with their neighbors, it is a valid coloring. Thus, if D_2 accepts, w is a valid coloring and therefore D_2 verifies 3COLOR.

D_2 clearly runs in polynomial time for the same reasons that D_1 runs in polynomial time. Thus, D_2 is a poylnomial time verifier for 3COLOR and thus 3COLOR $\in$ **NP**.

5. (a) We can produce the following polynomial-time nondeterminsitic Turing machine N_1 that decides IND as follows:

1. On input $\langle G, k \rangle$:

2. For $i \in [0, k-1]$:

3. Nondeterministically select $v_i \in V$ where none of its neighbors have marks.

4. If no such v_i exists, continue.

5. Mark v_i.

6. If the set of marked vertices is cardinality k, accept. Otherwise, reject.

N_1 decides IND. This is because if $\langle G \rangle$ is implemented in a sane way—say, an adjacency matrix—then because G is undirected, we know all of a vertex v's neighbors just by looking at its associated row and know that we can simply 'blacklist' any vertices adjacent to v for the purposes of marking. Additionally, our loop ensures that exactly k vertices will be marked if there exists an independent set of size k because we perform k marks and we nondeterministically select our marked vertices. If there is not an independent set of size k, the set of marked vertices will have cardinality less than k and we will reject. Thus, N_1 decides IND.

To prove that N_1 runs in nondeterminstic polynomial time, selecting and marking a vertex can be done in polynomial time in the number of vertices. We also perform at most k loops and thus we remain in polynomial time. Thus, N_1 runs in nondeterministic polynomial time and IND $\in$ **NP**.

(b) To give a polynomial-time verifier N_2 for IND, we can do the following:

1. On input $\langle G, k, w \rangle$:
2. If the number of marked vertices is less than k, reject.
3. For each marked vertex v_i:
4. For each vertex v_j adjacent to v_i:
5. If v_j is marked, reject.
6. Accept.

This verifies a solution for IND because if there are fewer than k marked vertices, clearly the witness is not a solution. Then, we loop through each marked vertex to make sure that the k marked vertices truly form an independent set and thus that none of them are adjacent. If we find any adjacent marked vertices, this is not an independent set and we reject. If we make it through every marked vertex and find no adjacent marked vertices, we accept because the witness provides an independent set of size k.

N_2 clearly runs in polynomial time because the set of marked vertices is upper bounded by k. We also know that the number of vertices adjacent to any given vertex is at most $|V| - 1$. Thus, we find that N_2 runs in polynomial time. Thus, IND $\in$ **NP**.