CS 21 Decidability and Tractability **Winter 2023**

Solution Set 3

Posted: February 1

If you have not yet turned in the Problem Set, you should not consult these solutions.

1. (a) A 2-NPDA is a 7-tuple $(Q, \Sigma, \Gamma_1, \Gamma_2, \delta, q_0, F)$, where $Q, \Sigma, \Gamma_1, \Gamma_2$ and F are all finite sets, and
   - Q is the set of states.
   - Σ is the input alphabet.
   - Γ_1 is the alphabet of stack 1.
   - Γ_2 is the alphabet of stack 2.
   - $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma_1 \cup \{\epsilon\}) \times (\Gamma_2 \cup \{\epsilon\}) \to \mathcal{P}(Q \times (\Gamma_1 \cup \{\epsilon\}) \times (\Gamma_2 \cup \{\epsilon\}))$ is the transition function.
   - $q_0 \in Q$ is the start state.
   - $F \subseteq Q$ is a set of accept states.

   A 2-NPDA $M = (Q, \Sigma, \Gamma_1, \Gamma_2, \delta, q_0, F)$ accepts string $w \in \Sigma^*$ if w can be written as

   $$w_1, w_2, \ldots, w_m \in (\Sigma \cup \{\epsilon\})^*,$$

   and there exist states

   $$r_0, r_1, \ldots r_m$$

   and pairs of strings

   $$(s_0, t_0), (s_1, t_1), \ldots, (s_m, t_m) \in (\Gamma_1 \cup \{\epsilon\})^* \times (\Gamma_2 \cup \{\epsilon\})^*$$

   such that
   - $r_0 = q_0$, and
   - $(s_0, t_0) = (\epsilon, \epsilon)$, and
   - $(r_{i+1}, c, d) \in \delta(r_i, w_{i+1}, a, b)$, where $s_i = au$ and $s_{i+1} = cu$ for some $u \in \Gamma_1^*$, and $t_i = bv$ and $t_{i+1} = dv$ for some $v \in \Gamma_2^*$, and
   - $r_m \in F$.

   (b) The 2-NPDA pushes "\$" onto stack 1 and stack 2. The machine operates in three phases. In phase one, it reads 0 or more a's from the tape, pushing an equal number of a's onto stack 1. In phase two it reads 0 or more b's from the tape, popping an a from stack 1 and pushing a b onto stack 2 for each b it reads from the tape. If it runs out of a's to pop, it rejects. In phase three it reads 0 or more c's from the tape, popping a b from stack 2 for each c it reads from the tape. If the machine enters phase three and there is not a "\$" on stack 1, it rejects. If the machine runs out of b's to pop from stack 2, it rejects. If the machine reaches the end of the string and there is not a "\$" on stack 2, it rejects.

(c) To prove the two types of machines are equivalent, we need to show that given any Turing Machine, we can construct an equivalent 2-NPDA, and vice-versa.

First we will simulate a Turing Machine with a 2-NPDA. The main idea is two use S_1 (the first stack) to represent everything to the $left$ of the Turing Machine head. Use S_2 (the other stack) to represent the spot under the head and everything to its $right$. Given this representation, we can now simulate the operations of the Turing Machine. Reading is equivalent to popping from S_2 and writing is equivalent to pushing the new symbol onto S_2. Moving right is equivalent to popping from S_2 and pushing the popped symbol onto S_1. Moving left is simply the reverse operation of moving right.

There are some details we have to note here. First, we need to initialize the 2-NPDA. In the Turing Machine, the head begins at the start of the input string. In our 2-NPDA representation, this is the same as the input string residing in S_2 with the start of the string at the top of the stack. To get this, we read the input string and push each symbol onto S_1. Then we pop each symbol from S_1 and push it onto S_2 yielding the desired starting point and string orientation. We can now begin executing the simulated Turing Machine. Another point to note is that the Turing Machine can move over an arbitrary length of tape left or right of the input string. This could result in an empty stack after enough moves. So if during a move, a pop from a stack indicates an empty stack, we would push a blank symbol onto the other stack to maintain the head positioning.

Now we will show that given a 2-NPDA, we can construct an equivalent Turing Machine. From lecture we know that multi-tape Turing Machines are equivalent to single tape Turing Machines. Therefore, we can construct a 3-tape turing machine, where the input tape is identical to the 2-NPDA input tape, and the two work tapes simulate the two stacks of the 2-NPDA. Call the tapes in the Turing Machine T_0, T_1, T_2, and the stacks in the 2-NPDA S_1 and S_2. Write a special symbol on each of the work tapes to simulate the bottom of the 2-NPDA stacks. To simulate pushing a onto S_1 we move right in T_1 and then write a. To simulate pushing a onto S_2 we move right in T_2 and then write a. To pop from S_1 read from T_1 and move left. If the read yields the special symbol, then we have reached the bottom of the stack so we do not move left. Popping from S_2 is identical. With this construction, we have simulated the use of the two stacks of the 2-NPDA, so can fully simulate execution of the 2-NPDA.

2. (a) A Queue Automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ and F are all finite sets, and

- Q is the set of states.
- Σ is the input alphabet.
- Γ is the queue alphabet.
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \to \mathcal{P}(Q \times (\Gamma \cup \{\epsilon\}))$ is the transition function.
- $q_0 \in Q$ is the start state.
- $F \subseteq Q$ is a set of accept states.

A Queue Autmomaton $M = (Q, \Sigma, \Gamma_1, \Gamma_2, \delta, q_0, F)$ accepts string $w \in \Sigma^*$ if w can be written as

$$w_1, w_2, \ldots, w_m \in (\Sigma \cup \{\epsilon\})^*,$$

and there exist states

$$r_0, r_1, \ldots r_m$$

and strings

$$s_0, s_1, \ldots, s_m \in (\Gamma \cup \{\epsilon\})^*$$

such that

- $r_0 = q_0$, and
- $s_0 = \epsilon$, and
- $(r_{i+1}, c) \in \delta(r_i, w_{i+1}, a)$, where $s_i = au$ and $s_{i+1} = uc$ for some $u \in \Gamma_1^*$,
- $r_m \in F$.

(b) First, we show how to simulate a Turing Machine with a Queue Automaton (QA). Our queue will contain the contents of the Turing Machine tape at all times, with the symbol currently being read at the head of the queue. We will use "$" to mark the beginning of the tape. Thus at all times the queue looks like:

$$\text{head of queue} \rightarrow \quad ax\$y \quad \leftarrow \text{tail of queue},$$

where $a \in \Sigma$ is the symbol currently under the head of the Turing Machine, $x \in \Sigma^*$ is the contents of the tape to the right of the head, and $y \in \Sigma^*$ is the contents of the tape to the left of the head.

We will describe a queue "primitive" that we will use in simulating the TM.

Cyclically shift right. We perform this operation is three phases. We will keep a running example to illustrate. Suppose we start out with queue contents

$$ab\$c$$

In Phase 1 we place a # marker at the end of the queue, and then replace each queue symbol x with a new symbol (w, x), where w is the symbol immediately to x's left in the queue. For this we use a special set of separate states that "remember" the last symbol shifted. That is, for each $w \in \Sigma$, after having shifted symbol w, we are in a special state q_w. Then when we encounter the next symbol x, we enqueue not x, but the new symbol (w, x). To start the process we enqueue # and move to state $q_\#$. We then repeat the following: From a given state q_w, dequeue a symbol x, enqueue the new symbol (w, x), and move to state q_x. When we dequeue # and enqueue the final new symbol we complete Phase 1. In our example the queue will now look like this:

$$(\#, a)(a, b)(b, \$)(\$, c)(c, \#)$$

In Phase 2, we repeat the following: dequeue (w, x), enqueue (w, x) until we dequeue (w, x) where $x = \#$. We then enqueue #, enqueue w, and dequeue whatever symbol is at the head of the queue. In our example the queue will now look like this:

$$(a, b)(b, \$)(\$, c)\#c$$

In Phase 3, we repeatedly dequeue (w, x) and enqueue w, until we dequeue #, at which point we complete Phase 3. In our example the queue will now look like this:

$$cab\$$$

which is our original queue cyclically shifted right one step.

Now to simulate a given step of the Turing Machine, we dequeue the symbol a at the head of the queue and then:

- if the TM transition that would be taken on reading a writes symbol b and moves right, then our QA enqueues b at the tail of the queue.
- if the TM transition that would be taken on reading a writes symbol b and moves left, then our QA enqueues b and then performs two cyclic shifts to the right.

At all points, the following sequence of transitions are available to the QA: dequeue \$, enqueue $-$\$ and perform 2 cyclic shifts to the right. This amounts to adding a blank "$-$" at the end of the tape. As with 2-NPDAs, we initialize the QA by copying the contents of the input into the queue, followed by a \$.

Finally, we need to show how to simulate a QA with TM. Since 2-tape TM is equivalent to a single tape TM, as shown in lecture, it is sufficient to show how to simulate a QA with a 2-tape TM. The first tape will simply contain the input string, and the second tape will contain the queue. The tape alphabet of the QA contains special symbol \$ that denotes the beginning of the queue. At first, one \$ is written on the tape corresponding to the queue. When a symbol is pushed onto the queue, the head finds the first blank space on the tape and writes the symbol there. When a symbol is popped, the tape finds the first symbol on the tape other than \$, reads the symbol and substitutes the symbol with \$. The queue shifts on the tape to the right as the TM pops, but this is acceptable since the tape is infinite.

3. Suppose we have a language expressible as:

$$L = \{x : \text{there exists } y \text{ for which } (x, y) \in R\},$$

where R is decidable. Then L is RE, because given an input x, we can simply enumerate all y's in lexicographic order, and for each one decide whether $(x, y) \in R$. If the answer is ever "yes" then we accept x; otherwise, we will continue on forever. Clearly the language accepted is exactly L.

In the other direction, suppose we have a RE language L. Fix an enumerater E for L. That is, E is a machine that writes a (potentially infinite) sequence of strings on its tape with the guarantee that the set of these strings is exactly L. Define R as follows (y is treated as an integer):

$$R = \{(x, y) : x \text{ is output within the first } y \text{ steps of } E\text{'s execution}\}.$$

Clearly R is decidable, because we can just simulate machine E for y steps to decide whether $(x, y) \in R$. Also, it is clear that L is exactly the set of x for which there *exists* some y for which $(x, y) \in R$ – this is just another way of saying that a string x is in language L iff it is eventually output by the enumerater E.