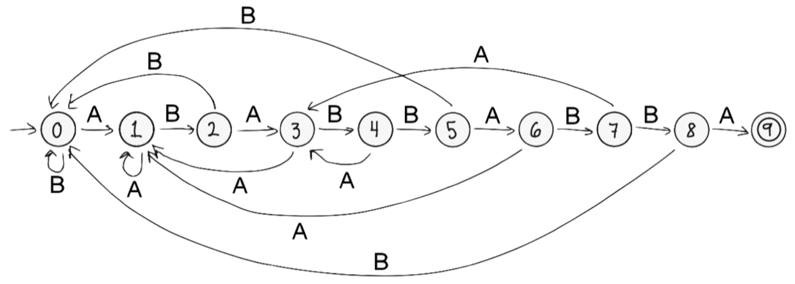
CS–172 Computability & Complexity, Spring 2021

Homework 4 Solutions

Note: This is the updated version of these solutions, including comments on common errors. These solutions are not necessarily model answers. Rather, they are designed to be tutorial in nature, and sometimes contain more explanation than is required for full points. Also, bear in mind that there may be more than one correct solution. The maximum total number of points available is 32.

1. The diagram below shows the DFA. *4pts*



2. Instead of just one counter, we maintain $k-1$ counters, each associated with at most one active element. *6pts* Initially all counters are zero and there are no active elements. If the next element in our stream is x_i, we update the counters and active elements according to the following rule:

- if x_i is active then increase its counter by 1, else
- [x_i not active] if there are fewer than $k-1$ active elements then make x_i active and set its counter to 1, else
- [x_i not active and $k-1$ other elements active] decrement all counters by 1

This algorithm uses $k-1$ counters up to n and their associated active elements, for a total of $O(\log n + \log|\Sigma|)$ bits of memory (since k is assumed to be constant).

To justify the correctness of this algorithm, we generalize the argument for $k=2$ that we saw in class. Suppose x is any element that occurs $m > \frac{n}{k}$ times. We claim that x must be active at the end of the stream. To do this, we assume that it is not active and use this fact to associate each single occurrence of x with $k-1$ occurrences of distinct other elements, for a total of $m(k-1)$ such occurrences. This will be a contradiction since the total number of occurrences of other elements is less than $n - \frac{n}{k} = \frac{n}{k}(k-1) < m(k-1)$.

To achieve the above association, imagine as in our previous argument that the counters are implemented as stacks, where actual occurrences of elements are pushed and popped. (Note that this is for analysis purposes only: the algorithm itself just implements them as numerical counters.) Consider any occurrence of element x. If this occurrence causes the counters to be decremented, then we associate it with the $k-1$ occurrences on the tops of all those stacks. (Note that decremening happens only when all stacks are non-zero.) If the occurrence causes a counter to be incremented, we associate it with the later occurrence of some other element that pops that occurrence of x off its stack, along with the $k-2$ other occurrences that are popped along with it. (Note that this occurrence of x must eventually be popped by our assumption that x is not active at the end.) This completes the association and hence the proof.

Note that the justification is a crucial ingredient of this answer. The algorithm itself is a relatively straightforward generalization of the one for majority (i.e., $k = 2$), but the justification is a bit more delicate in this case. Points were also deducted for outputting k (or more) elements, rather than the specified $k - 1$.

3. Following the framework introduced in Note 3, it suffices to find a set of $2^{\Omega(n)}$ distinguishable inputs for graphs of size n. Here's one way to do this. Let $V = \{1, 2, \ldots, n\}$ denote the vertex set of these graphs, and for simplicity assume n is even. (If n is odd the same idea works with the necessary rounding.) For each subset $U \subseteq V$ of size $n/2$, construct a graph G_U consisting of exactly two trees (minimally connected components), one on U and the other on $V \setminus U$. We claim that the graphs G_U, as U ranges over all such subsets, are distinguishable. If we can prove this then we will be done, because the number of such graphs is $\frac{1}{2}\binom{n}{\frac{n}{2}} = 2^{\Omega(n)}$ (where the factor $\frac{1}{2}$ comes from the symmetry between U and $V \setminus U$). *4pts*

To prove that any pair G_U, G_W with $U \neq W$ are distinguishable, pick a pair of vertices v_1, v_2 such that $v_1, v_2 \in U$ but $v_1 \in W$ and $v_2 \in V \setminus W$. (I.e., v_1, v_2 are on the *same* side of the partition $(U, V \setminus U)$ but on *opposite* sides of the partition $(W, V \setminus W)$.) The existence of such a pair of vertices follows from the fact that $U \neq W$ and $|U| = |W|$. Now if we add the edge $\{v_1, v_2\}$ to the stream G_U, we still get a disconnected graph; but if we add this same edge to the stream G_W the graph becomes connected. Hence this edge distinguishes G_U, G_W, as claimed. This completes the proof.

Note that the above solution is not the only possible one. For example, the trees can be replaced by any connected graphs, and instead of partitioning the vertices into two equal-size subsets U, V, we could allow partitions of any size (in which case the number of distinguishable strings is just 2^{n-1}). Some students found a distinguishable set of size only $\Omega(n)$, rather than $2^{\Omega(n)}$. This only implies that the number of memory bits needed is $\Omega(\log n)$, which is much smaller than claimed.

4. (a) A PDA for this language repeatedly reads a 0 and pushes it onto the stack until it encounters the first 1 in the input. It then repeatedly reads a 0 from the input and pops the stack. The machine accepts iff the stack is empty at the end of the input. It rejects if it encounters any 1s in the popping phase. This PDA is deterministic. *4pts*

A CFG for this language has the single variable S (which is also the start variable) and alphabet $\{0, 1\}$. The production rules are:

$$S \longrightarrow 0S0 \mid 1$$

For the PDA in this part, some students didn't say that the machine rejects if it finds more 1's in the input.

(b) A PDA for this language first uses nondeterminism to guess whether the input string satisfies the first condition ($i = j$) or the second condition ($j = k$). If it guesses the first condition, then it repeatedly reads an a from the input and pushes it onto the stack. When it encounters the first b in the input, it repeatedly reads a b and pops an a from the stack. If the stack is empty when it encounters the first c in the input, it checks that all remaining input symbols are c and then accepts. Any input symbols violating the required syntax cause rejection. If it guesses the second condition, it executes a similar procedure where it first skips over as in the input and then compares the numbers of bs and cs using the stack. *4pts*

A CFG for this language has variables S, S_1, S_2, A, C (where S is the start variable) and alphabet $\{a, b, c\}$. The production rules are:

$$\begin{aligned} S &\longrightarrow S_1C \mid AS_2 \\ S_1 &\longrightarrow aS_1b \mid \varepsilon \\ S_2 &\longrightarrow bS_2c \mid \varepsilon \\ C &\longrightarrow cC \mid \varepsilon \\ A &\longrightarrow aA \mid \varepsilon \end{aligned}$$

The first rule corresponds to the "OR" in the definition of the language. Variable S_1 generates strings of a's followed by an equal number of b's, while variable S_2 does the same for b's followed by c's. Variables A, C simply generate strings of a's and c's, respectively, of arbitrary length.

For the CFG in this part, some students omitted ε from the right-hand side of some production rules, resulting in some strings in the language not being generated.

(c) A PDA for this language works by maintaining a counter that records the "excess" of 1s over 0s seen so far (where the baseline is twice as many 0s as 1s). Thus if x is the portion of the input read so far, we define the counter as $\text{diff}(x) = (2 \times \#1(x)) - \#0(x)$. It's also convenient to record the sign of the counter, which we do via the variable $\text{maj}(x) = 1$ if $\text{diff}(x) \geq 0$ and 0 otherwise. The PDA will maintain the counter by ensuring that, after reading x, the stack contains exactly $|\text{diff}(x)|$ copies of $\text{maj}(x)$ (and is empty iff $\text{diff}(x) = 0$). This invariant is maintained as follows; let $w(1) = 2$ and $w(0) = 1$. On reading the next symbol a, if the stack is empty then push $w(a)$ copies of a onto the stack; otherwise, if $a = \text{maj}(x)$ then push $w(a)$ copies of a onto the stack else pop the stack $w(a)$ times. If the stack is of size 1 and we need to pop twice (which happens when the stack contains only a 0 and we see a 1), then instead of popping twice, do one pop followed by pushing a 1 onto the stack. (Note that $\text{maj}(x)$ can be deduced from the top symbol of the stack.) Finally, accept iff at the end of the input the stack is empty. Note that this PDA is deterministic. *4pts*

One possible CFG for this language has a single variable S (which is also the start variable) and production rules:

$$S \longrightarrow 1S00S \mid 00S1S \mid 0S1S0S \mid \varepsilon$$

It should be clear that all strings generated by this grammar have twice as many 0's as 1's, because every production rule preserves this property. However, we need to explain why the grammar generates *all* such strings. To do this, we use the same counter $\text{diff}(x)$ as we used for the PDA, where we imagine the string x being read left-to-right. Call a string x "balanced" if $\text{diff}(x) = 0$; thus the grammar should generate exactly the balanced strings.

To see that all balanced strings are generated by the above grammar, we consider three exhaustive cases:

Case (i): w begins with 1. Consider the first time at which the counter hits zero. Then the last two symbols read before that point must have been 0's; moreover, the portion of the string between the initial 1 and this pair of 0's takes the counter from $+2$ to $+2$, so it must itself be balanced; finally, any remaining part of the string takes the counter from zero back to zero again, so again is balanced. This case corresponds to the first production rule.

Case (ii): w begins with 0, and the counter hits 0 before becoming positive. In this case, the second symbol must be a 0 since otherwise the counter would jump from -1 to $+1$. And the symbol read immediately before the counter hits zero must be a 1. The portion of the string between the initial 00 and this 1 takes the counter from -2 to -2, so it must be balanced, as must any remaining part of the string. This case corresponds to the second production rule.

Case (iii): w begins with 0, and the counter becomes positive without hitting zero. The symbol that causes the counter to become positive must be a 1, which takes the counter from -1 to $+1$. Thus the portion between the initial 0 and this 1 takes the counter from -1 to -1, so is balanced. After reaching $+1$, consider the first time when the counter hits zero; note that the symbol that causes this to happen must be a 0, and that the intervening portion of the string (taking the counter from $+1$ to $+1$) is balanced. Finally, any remaining portion of the string must be balanced. This case corresponds to the third production rule.

As usual, other grammars are possible for this language. However, any such grammar must ensure that there are twice as many 0's as 1's; that the above grammar does this can be seen immediately from the production rules. We graded the grammar part of this problem leniently because these grammars are hard to check; you should go back and review your grammar in light of the above solution.

5. We first show that for any regular grammar $G = (V, \Sigma, R, S)$, we can construct an NFA M such that $L(G) = L(M)$. Since G is regular, we can assume that all rules in G are of the form $A \to wB$ or $A \to w$. Further, we can assume that $|w| \leq 1$, since if we have a rule $A \to x_1x_2 \ldots x_kB$ for terminals x_i, we can convert this into a new set of rules $A \to x_1A_1$, $A_1 \to x_2A_2, \ldots A_{k-1} \to x_kB$ for new variables A_i. *6pts*

We construct M as follows: We have one state for each variable, with an additional accepting state F. The start state will be the state corresponding to the start symbol S, and F will be the only accepting state. Finally, we define the transition function δ: for each rule of the form $A \to wB$, we add a transition from state A to state B on symbol w. (If w is the empty string, this is a transition on ε.) For each rule of the form $A \to w$, we have a transition from A to F on symbol w.

We now show that $L(M) = L(G)$. First, let w be any string produced by G. Then w was obtained by applying a sequence of productions of the form $S \to w_1A_1$, $A_1 \to w_2A_2, \ldots, A_t \to w_t$. (Some of the w_is may possibly be empty strings.) Corresponding to this is a sequence of transitions in M of the form

$$S \xrightarrow{w_1} A_1 \xrightarrow{w_2} A_2 \xrightarrow{w_3} \cdots \xrightarrow{w_{t-1}} A_t \xrightarrow{w_t} F$$

that leads to the accept state. Conversely, any accepting path on input w in the NFA consists of a sequence of transitions of the above form, which in turn yield a derivation for w in G.

We next show how, given a DFA M, we can write a regular grammar G such that $L(M) = L(G)$. We will have one variable for each state in the DFA, and one terminal for each alphabet symbol of the DFA. The start symbol will be the start state. For each transition in the DFA of the form $\delta(q, a) = q'$, we introduce a rule in the grammar $q \to aq'$. Finally, for each accepting state q_f, we add a rule $q_f \to \varepsilon$. By construction, accepting computations of M are in one-to-one correspondence with strings produced by G.

In the Regular Grammar $\to$ NFA direction, some students assumed that w on the right-hand side of the rules must have length ≤ 1; in fact you have to show that this can be arranged by modifying the grammar, as described above. In the DFA $\to$ Regular Grammar direction, some students omitted the rules corresponding to accepting states q_f.