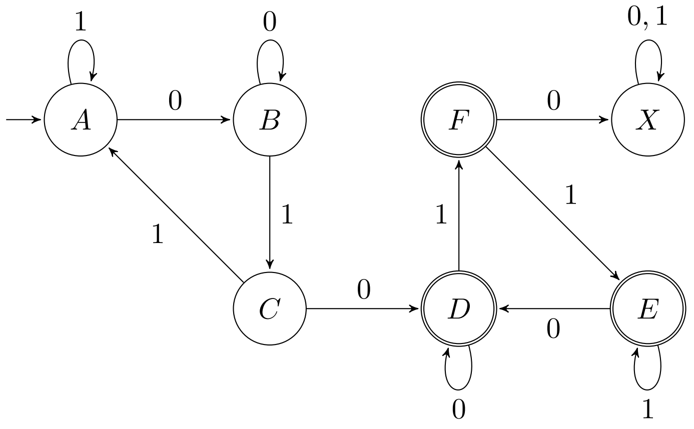
U.C. Berkeley — CS172: Computability and Complexity Answers to Final Exam
David Richerby 11th May, 2016

Answers to Final Exam

General grading notes. Full marks for any correct proof, in all cases. Subtract around one mark for fairly minor mistakes that don't affect the overall validity of the answer.

Question 1



To verify that the automaton is minimal, we construct the relations $\equiv_i$. The following lists their equivalence classes.

$$\begin{aligned}
&\equiv_0\colon \{A,B,C,X\}, \{D,E,F\}\\
&\equiv_1\colon \{A,B,X\}, \{C\}, \{D,E\}, \{F\}\\
&\equiv_2\colon \{A,X\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F\}\\
&\equiv_3\colon \{A\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F\}, \{X\}\,.
\end{aligned}$$

And $\equiv_i = \equiv_3$ for all $i > 3$, since it is not possible to break up any more classes. No states are equivalent so the automaton is minimal.

Grading notes. 7 marks for a correct automaton, 3 marks for the minimality proof. Subtract 2 if the automaton accepts the correct language but isn't minimal; subtract 2 if the automaton is incorrect but close.

Comment. This advice will probably never be useful to you again but I only realised it when grading this question. If you want to prove minimality of some automaton, it's safer to show that the states are all $\equiv$-inequivalent, rather than using Myhill–Nerode. A few students produced a six-state automaton, found six Myhill–Nerode classes and figured they were done so didn't look for any more. That confirms that an automaton for the language must have at least six states but it turns out that you do need seven.

Question 2

Let $M = (Q, \Sigma, \delta, q_0, A)$ be a collector automaton. We define a DFA $D = (Q', \Sigma, \delta', q_0', A')$ such that $L(D) = L(M)$. Each state of D will track the current state of M and which accepting states have been visited on the run so far.

So, let $Q' = Q \times \mathcal{P}(A)$. The initial state is

$$q_0' = \begin{cases} (q_0, \emptyset) & \text{if } q_0 \notin A \\ (q_0, \{q_0\}) & \text{if } q_0 \in A\,. \end{cases}$$

(Actually, we could assume that $q_0 \notin A$, since the automaton visits state q_0 for every possible input, so including q_0 in A gives no discriminatory power.)

The transition function is as follows, where $a \in \Sigma$ and $q' = \delta(q, a)$.

$$\delta'((q, S), a) = \begin{cases} (q', S) & \text{if } q' \notin A \\ (q', S \cup \{q'\}) & \text{if } q' \in A\,. \end{cases}$$

The accepting set is $A' = \{(q, A) \mid q \in Q\}$, so D accepts whenever every accepting state has been visited.

There are regular languages that are not accepted by any collector automaton. This is because, if a CA accepts a string w, it also accepts every string wx such that $x \in \Sigma^*$. So, for example, the language of strings containing an even number of 1s is not decided by any collector automaton.

Grading notes. 12 marks for proving that collector automata accept regular languages; 3 marks for showing that some regular languages aren't accepted.

Comment. A student answer that particularly impressed me was to consider the regular expressions $R^Q_{q_0,a}$ (recall the proof of Theorem 17, translating DFAs to regular expressions) for each $a \in A$ and observe that strings matching $R^Q_{q_0,a}\Sigma^*$ are exactly the strings that visit state a at least once. Then the language accepted by the collector automaton is given by $\bigcap_{a\in A}(R^q_{q_0,a}\Sigma^*)$, which is regular because the regular languages are closed under finite intersections.

By the way, collector automata are just something I made up for the exam. The acceptance criterion was inspired by Muller automata. Those are like DFAs but the input is infinite and acceptance is based on the set of states that are visited infinitely often. In particular, the acceptance criterion can be "The set of states visited infinitely often is exactly A."

Question 3

Suppose that $L = \{x \mid x\#y \in L'\}$ for some decidable L', and let M be a TM that decides L'. Then we can semi-decide L by trying all possible values for y. More formally, we semi-decide L with the following Turing machine.

```
for each $i = 1, 2, \ldots$
    for each string $y$ of length at most $i$
        if $M$ accepts $x\#y$ within $i$ steps, accept
```

If $x \in L$ then the machine will find a value of y that proves this; if $x \notin L$, then the machine will loop for ever, which is allowed behaviour for a machine that semi-decides a language.

Conversely, suppose that L is semi-decidable and some Turing machine T accepts it. Then let L' be the set of all strings $x\#\text{bin}(t)$ such that L accepts x in at most t steps. We need to show that L' is decidable, which is true because we can decide whether $x\#\text{bin}(t) \in L'$ by simulating T for t steps. We accept if T accepts in this time, and reject, otherwise.

Grading notes. 5 marks for each part.

Question 4

Suppose that M halts for infinitely many inputs and outputs $K(x)$ for every input x for which it halts. The set H of strings for which M halts is recursively enumerable, so is enumerated by some Turing machine N, which produces outputs $x_1, x_2, x_3, \dots$. Define the sequence $y_1 = x_1$, $y_{i+1} = x_j$, where $j = \min\{i \mid |x_i| > |y_{i+1}|\}$. The set $Y = \{y_1, y_2, \dots\}$ is decidable. Note that M halts when given any input in Y.

Now, for $i \geq 0$, define $z(n) = \min\{i \mid K(y_i) > n\}$. By construction, $K(y_{z(n)}) > n$ for every n. The function z is computed by some Turing machine M_z, so $\langle M_z\rangle\text{bin}(n)$ is a description of $y_{z(n)}$, with length $c + \log n$ for some constant c. But, for all large enough n, $c + \log n < n$, contradicting the fact that $K(y_{z(n)}) > n > c + \log n$.

Grading notes. Score around 8–10 if the logic a bit off, e.g., not going via description of length $\log n$, not showing how to enumerate strings of high Kolmogorov complexity.

Comment. I was very impressed by a couple of your answers, which could be paraphrased as follows. An M that outputs $K(x)$ for infinitely many x gives proofs of infinitely many statements of the form "$K(x) \geq t$". But we know that only finitely many such statements have proofs (Theorem 57).

Question 5

The problem is clearly in **NP**: a candidate kernel contains at most $|V|$ vertices, which is at most linear in the input size, and we can check in polynomial time that there are no edges within a claimed kernel and that every other vertex receives an edge from it.

For **NP**-hardness, we reduce from CNF-SAT. Let ϕ be a CNF formula in variables $X_1, \dots, X_k$, with clauses $C_1, \dots, C_\ell$. We produce a digraph G_ϕ with vertices $\{X_1, \dots, X_k, \neg X_1, \dots, \neg X_k, C_1, \dots, C_\ell\}$. For each i, there are edges $(X_i, \neg X_i)$ and $(\neg X_i, X_i)$ and, for each clause $C_i \equiv L_1 \vee \cdots \vee L_m$, there are edges from each of the vertices L_j to C_i, along with an edge (C_i, C_i).

Now, suppose that K is a kernel of G_ϕ. K cannot contain any C_i because K must be an independent set and every C_i has an edge to itself. For any $i \in \{1, \dots, k\}$, K must contain exactly one of X_i and $\neg X_i$. It cannot contain both of them, since they are joined by edges and a kernel must be an independent set. It cannot contain neither of them, because X_i and $\neg X_i$ only receive edges from each other so, if neither is in K, neither receives an edge from K.

Further, we claim that setting every literal in K to true is a satisfying assignment for ϕ. First, this is well-defined since, as we have shown, K contains exactly one of X_i and $\neg X_i$ for each variable. Second, every clause C_j must receive an edge from some $L_i \in K$, so every C_j contains a true literal.

Conversely, suppose that ϕ is satisfiable. Then, by essentially the same argument above, the set of true literals in any truth assignment is a kernel of G_ϕ. This establishes that G_ϕ has a kernel if, and only if, ϕ is satisfiable.

Finally, the reduction from ϕ to G_ϕ can be performed in polynomial time, since it requires only simple analysis of the formula and $|V(G_\phi)|$ is at most the length of the formula ϕ.

Grading notes. 3 marks for membership in **NP**, 9 marks for giving a correct reduction, 3 marks for proving it works. Partial credit for an attempted reduction that looks plausible.

Comment. The self-loops on clause vertices are required. If they are omitted, it's no longer true that ϕ is satisfiable iff G_ϕ has a kernel. For example, consider the formula $\phi = X \wedge \neg X$. This has clauses $C_1 = X$ and $C_2 = \neg X$. ϕ is unsatisfiable but $\{X, C_2\}$ and $\{\neg X, C_1\}$ are both kernels of the graph without the loops.

Note that reducing from independent set, vertex cover and similar problems is likely to be problematic, since those problems ask about the existence of an object of at least/at most some given size, whereas KERNEL doesn't specify the size. And the size constraint is necessary for these problems to be hard. For example, while it's hard to find an independent set of size at least k, it's trivial to find an independent set if you don't care how big it is: $\emptyset$ is an independent set.

A few answers proposed that a maximum-sized independent set is always a kernel but this isn't the case. Consider a directed 3-cycle, with all edges going clockwise, say. A maximum-size independent set is a single vertex but no vertex of the graph is a kernel.

Also, note the distinction between a maximum independent set and a maximal one. A maximum I.S. is that has the biggest possible size (i.e., it's a "global maximum"); a maximal I.S. is one that can't have any more vertices added to it while remaining independent (i.e., it's a "local maximum"). For example, consider the graph with vertices $\{c, x_1, \dots, x_k\}$ and edges $\{cx_i \mid 1 \leq i \leq k\}$. $\{c\}$ is a maximal independent set but it isn't maximum; $\{x_1, \dots, x_k\}$ is both maximal and maximum; $\{x_1\}$ is independent but neither maximal nor maximum. Finding maximal independent sets isn't hard: start with $\emptyset$ and greedily add vertices until you can't add any more.

Question 6

We will show that $\overline{F} \in \mathbf{NL}$ and the required result follows by the Immerman–Szelepscényi theorem.

First, observe that a DFA M accepts an infinite language if, and only if, its state diagram contains a cycle that includes at least one state on a path from the initial vertex to some accepting state. If there is such a cycle in M, then there are infinitely many walks from the initial state to some final state, since a walk can go around the cycle any number of times. Conversely, if there are no such cyles, then every walk from the initial state to any accepting state is a simple path (it does not repeat any state). Any finite graph contains a finite number of simple paths, so M accepts only finitely many strings.

Now, we show that we can detect the existence of a cycle in **NL**. We do this by guessing a path from the initial state of length at most $|Q|$, then guessing a path of length at most $|Q|$ from the endpoint of the first path back to itself, then guessing a path of length at most $|Q|$ from there to an accepting state. The algorithm is as follows:

```
$q := q_0$
$n := |Q|$
$i := 0$

while true do
    nondeterministically choose whether to break out of the loop
    nondeterministically choose a new $q$ from $\{\delta(q,a) \mid a \in \Sigma\}$
    $i := i + 1$
    if $i = n$, reject

$c := q$, $i := 0$
repeat
    nondeterministically choose a new $q$ from $\{\delta(q,a) \mid a \in \Sigma\}$
    $i := i + 1$
    if $i = n$, reject
until $q = c$

$i := 0$
while $q \notin A$ do
    nondeterministically choose a new $q$ from $\{\delta(q,a) \mid a \in \Sigma\}$
    $i := i + 1$
    if $i = n$, reject

accept
```

We can compute n because the encoding of DFAs as Turing machines includes the number of states in unary. The algorithm always terminates because every loop makes at most n iterations. It uses logarithmic space because the only explicit variables are c, i, n and q, which take $\log n \leq \log|\langle M\rangle|$ bits each. We also need some variables to check what transitions are available. The transition function is represented as a $|Q| \times |\Sigma|$ matrix so, again, we only need logspace to loop through the entries.

Grading notes. 12 marks for a working algorithm; 3 marks for showing that it uses logarithmic space. Subtract 2 marks for implicitly using **NL** = **co-NL** without saying so. Subtract 3 marks if the answer detects any cycle in the graph, rather than just cycles from which an accepting state can be reached.

Question 7

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be a Turing machine. We will produce a Turing machine $M' = (Q', \Sigma, \Gamma', \delta', q_0, q_{\text{acc}}, q_{\text{rej}})$ which stores $k > 1$ symbols from Γ in a single tape cell, for some value of k which we will choose later. Because M' will read and write characters in blocks, it will simulate M while making approximately one-kth the number of transitions. M' will have one more tape than M. We will use the state to remember the positions of the simulated tape heads within their current blocks of k characters.

Thus, $\Gamma' = \Gamma^k$. M' will first scan its input $x = x_1 \dots x_n$ and write the string $(x_1, \dots, x_k), (x_{k+1}, \dots, x_{2k}) \dots, (x_{n-k+1}, \dots, x_n)$ to its first work-tape. (If n is not a multiple of k, add blanks to the end.) It then returns the head of that work-tape to the left end. This takes a total of $|x| + \lceil |x|/k \rceil \leq 2|x|$ steps.

Now, we simulate M, using M''s first work-tape as the input tape and the other work-tapes to store the contents of M's work-tapes. Each step of the simulation works as follows. First, we move each tape head left, right, right, left, to read the block to the left of the head, the current block and the block to the right, on each tape. We store all the symbols we see in the state. We now know every character within distance k of every head of M, so we can simulate k steps of M's computation immediately. These steps change at most two blocks of each tape, and it takes at most three more steps to write all those changes to the tapes (write current block, write a block to the left/right, and possibly move back to the current block). Thus, we have simulated k steps of M in at most seven steps of M'.

Therefore, the total time needed to simulate M is at most $t = 2|x| + \lceil 7f(|x|)/k \rceil$. Setting $k = 15$ gives $t \leq f(|x|)/2$ for all sufficiently long inputs x, since $|x| = o(f(|x|))$.

The relevance to the definition of time-bounded complexity classes is that we don't lose anything by using big-O notation. The set of languages we can decide in at most $O(f(|x|))$ steps is exactly the set of languages we can decide in at most $f(|x|)$ steps for all large enough inputs.

Grading notes. 17 marks for the proof and 3 marks for explaining the relevance. Baseline score of around 8/17 for realising that you need to store at least two characters per tape cell. Subtract 2 marks for answers that don't account for the cost of finding out what's on adjacent tape cells or updating those cells, when it's necessary to do so.

A common situation is the following. Suppose we've encoded two characters per tape cell and the machine we're simulating moves repeatedly back and forth between its second and third tape cells. In this case, if we're not careful, the simulating machine will move repeatedly back and forth between its first and second tape cells and won't actually save any time. Subtract 4 marks if the answer has this problem.

Comment. Of course, the "2" in $f(|x|)/2$ is not significant. It's usually stated that, for every $\epsilon > 0$, we can produce an M' as in the question that halts after at most $\epsilon f(|x|)$ steps for every input x. This is known as the linear speed-up theorem.