CS/Math 240: Intro to Discrete Math	4/21/2011

Solutions to Homework 9

Instructor: Dieter van Melkebeek

Problem 1

A tree is a connected graph with out simple cycles. Since the given graph G is connected, we only need to prove that it cannot contain any cycles. Let's use induction on the number of vertices to prove that the graph cannot contain any cycles and hence is a tree.

For the base case, consider a connected graph with one vertex. The number of edges is 0. This graph is does not contain any cycles. Therefore, it is a tree.

Now consider a connected graph of $n+1$ vertices and n edges. Since the graph is connected, therefore each vertex has at least one edge incident on it. We claim that there is at least one vertex which has exactly one edge incident on it. Suppose there is no such vertex. This implies that each vertex has a degree of at least 2. From lecture 18, we know that $\sum_{v \in V} deg(v) = 2|E|$. Since $deg(v) \geq 2$ for all the vertices, therefore $\sum_{v \in V} deg(v) \geq 2(n+1)$. This means that $|E| \geq n+1$ but the graph has only n edges. Hence, there is at least one vertex with degree exactly 1. Let's label this vertex as v and its only neighbor as u. We claim that the graph $G' = (V \setminus \{v\}, E \setminus \{(u,v)\})$ is connected. Since v has degree one in the original graph, therefore the only vertex that could have possibly been disconnected is u. But then no path between u and any other vertex w could have used the edge (v,u). Hence G' is still connected. Note that G' has n vertices and $n-1$ edges. By induction hypothesis, G' does not contain any cycles. If we now add back the vertex v and the edge (u,v), the graph will not contain any cycle, v has degree 1. Hence, G does not contain any cycles and hence is a tree. This completes the proof.

Problem 2

Let's first work out a couple of example trees so as to understand how the given program, TreeRepresentation, behaves.

For the graph in Figure 1, on each iteration of the loop, the leaf with smallest label will be selected. So, on the first iteration leaf labeled 1 will be selected, on the second iteration leaf with label 2 will be selected and so on. For each of these leaves, the neighbor is labeled 9. Hence, $A[i] = 9$ for all $1 \leq i \leq n-2$.

For the graph in Figure 2, on the first iteration vertex labeled 1 will be selected. Its neighbor is the vertex labeled 2. When the vertex with label 1 is removed, its neighbor will become a leaf which is selected in the second iteration. Its neighbor is the vertex with label 3. We continue with this process until we are left with only two vertices in the tree. Hence, $A[i] = i+1$ for all $1 \leq i \leq n-2$.

We make some observations about the TreeRepresentation algorithm.

- First, notice that in each iteration of the for loop, one leaf and its incident edge is removed from G. Since G starts out as the given tree T on n vertices, it follows by induction that at

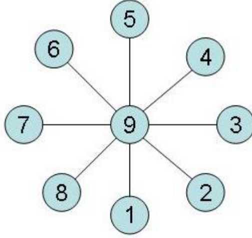


Figure 1

Figure 2

the start of the ith iteration, G is a subgraph of T that is a tree on $n+1-i$ vertices. After the last iteration, there is a subtree with 2 vertices left, i.e., a single edge of the original tree T.

- In the first iteration, the smallest leaf ℓ of T and its incident edge e are removed. The other endpoint of e is $A[1]$. The remaining tree T' has vertex set $\{1, 2, \ldots, n\} - \{\ell\}$, and TreeRepresentation(T) is just the concatenation of $A[1]$ with $A' \doteq$ TreeRepresentation(T').
- Every vertex v of T appears exactly $\deg(v) - 1$ times in $A[1..n-2]$. This follows by induction on n, based on the decomposition of the previous bullet. In particular, the leaves of T are exactly those vertices that do not appear in $A[1..n-2]$.

Part a

The second bullet suggests the following recursive approach for recovering T from A: Determine the edge e that is first deleted, recover T' from $A' = A[2..n-2]$, and add e to T' to obtain T. We implement this approach in an iterative way by determining the edges that TreeRepresentation removes in order, collecting them in a set E, and adding the single remaining edge at the end.

In order to do so, we initialize E to the empty set, and maintain the set L of leaves of G throughout the algorithm. By the third bullet, L needs to be initialized to $\{1, 2, \ldots, n\} - \{A[1], A[2], \ldots, A[n-2]\}$. Once we the intial value of L, we know the leaf ℓ that is removed in the first step of TreeRepresentation, namely $\ell = \min(L)$. The unique vertex to which ℓ is connected is $A[1]$, so the edge e that is removed in the first step is $\{b, A[1]\}$. We add e to E. After we remove ℓ and its incident edge e from G, the set of leaves L needs to be updated. The vertex ℓ is removed from L, and the other endpoint $A[1]$ is added to L if it became a leaf. By the last two bullets, the latter is the case iff $A[1]$ does not appear in $A' = A[2..n-2]$.

We repeat this step $n-2$ times. The only edge that still needs to be added to E is the single edge that remains at the end. The endpoints of this edge form L, i.e., the edge is just L.

The leads to the following algorithm.

Function ConstructTree(A, n)

Input: n: integer, $n \geq 2$,
A: an array of length $n-2$ with entries in $V = \{1, 2, 3, \cdots, n\}$
Output: a tree $T = (V, E)$ such that TreeRepresentation$(T, n) = A$

```
(1) E ← ∅
(2) L ← {1, 2, ··· , n} − {A[1], A[2], ··· , A[n − 2]}
(3) for i = 1 to n − 2 do
(4)     ℓ ← min(L)
(5)     E ← E ∪ {{ℓ, A[i]}}
(6)     L ← L − {ℓ}
(7)     if A[i] ∉ {A[i + 1], ··· , A[n − 2]} then
(8)         L ← L ∪ {A[i]}
(9) E ← E ∪ {L}
```

Part b

Given any array A with $n-2$ entries in the range $\{1, 2, 3, \cdots, n\}$, we can always construct a tree whose representation is the array A. In order to prove this, we first need to argue that ConstructTree always outputs a tree. Let us name the obtained output as $T = (V, E)$. We have n vertices, therefore $|V| = n$. The loop in the algorithm adds one edge to E in each iteration and $n-2$ is number of loop iterations. We also add one more edge to the graph after the execution of the loop is over. So in all we add $n-1$ edges, therefore $|E| = n-1$. We know that an acyclic graph with n vertices and $n-1$ edges is a tree. So, we only need to argue that T does not contain any cycles.

To argue the latter, observe that when we add the edge $\{\ell, A[i]\}$ during the ith iteration, no later iteration can add edges that are incident with ℓ. This is because all subsequent edges consist of vertices from $L - \{\ell\}$ and from $A[i+1..n-2]$, and neither contains ℓ. Thus, if we look at the list of edges in reverse order of being added to E, each edge introduces a new vertex, which implies there can be no cycles.

This completes the proof that ConstructTree always outputs a tree T. That TreeRepresentation(T) equals the given array A follows because in every step of ConstructTree we add the edge that TreeRepresentation removes (or has left at the end). This also implies that the tree T is unique.

Since the construction is always possible for any A and it is unique as well, we can conclude that the number of distinct trees on V is same as the number of different possible A. Since array A has $n-2$ positions each of which can be filled with any of the n possible vertex labels, therefore the number of distinct trees on V is n^{n-2}.

Problem 3

Part a

We begin by examining P_T for small values of n, where $n = |V|$. When $n = 1$, we see that there are x possible colorings of G, which correspond to the x possible colors the solitary vertex can have. When $n = 2$, we see that once we fix a vertex to a color (of which there are x to choose from), the second vertex can only be set to one of the remaining $x - 1$ colors. So P_t for $n = 2$ is $x(x-1)$.

We can extend this reasoning to a tree of arbitrary size by noting that for this tree, once we set a single vertex to a color (of which there are x to choose from), this constrains the neighbors of the vertex to have one of the remaining $x - 1$ colors. When these neighbors have their colors set, they will in turn constrain their neighbors (those vertices that are a distance of two from the original vertex) to have a different color than themselves (again, $x - 1$ colors to choose from since these new vertices may now have the same color as the original vertex).

To justify our argument, we formalize the above procedure as follows:

(1) Designate an arbitrary vertex of the tree as the root.

(2) Direct all edges away from the root. We now have a directed acyclic graph.

(3) Perform a topological sort on the tree. We know this is possible for any DAG.

(4) Color vertices from left to right. Each vertex v but the first will have in-degree 1 (otherwise, this would imply two paths from the root to v - a cycle in the original graph). As such, these vertices will have a single coloring constraint on them.

So each vertex in the tree besides the start vertex will be constrained to take on one of $x - 1$ colors. Then, our total number of colorings P_T for a tree with n vertices is $x(x-1)^{n-1}$.

Part b

We will argue this statement in two parts; first, we will show that any valid coloring of G'_e will have a unique mapping to a valid coloring of $G \setminus e$ with u, v as the same color. Then, we will show that any valid coloring of $G \setminus e$ with u, v as the same color has a unique mapping to a valid coloring of G'_e. This will show that the number of valid colorings of each graph is exactly equal.

Take a valid coloring of G'_e. If we apply the same coloring (besides the colors of u and v) to $G \setminus e$, this coloring will certainly be valid for all vertices excepting u and v, since these are unchanged between graphs. In G'_e, the vertex (uv) has a number of constraints from other vertices which restrict its color. We see that the individual constraints on u and v in $G \setminus e$ from other vertices must be as or less strict. So u and v can be set to the same color as uv; and thus any valid coloring of G'_e can be translated into a valid coloring of $G \setminus e$ with u and v the same color. This mapping is unique, as it retains all the colors from the original G'_e.

Now, take a valid coloring of $G \setminus e$, where u and v have the same color (say, c). This means that in this coloring, any neighbor of u or v do not have the color c. If we apply this same coloring to G'_e, we see that any neigbor of u or v is now a neighbor of the contracted uv. So none of these neighbors of uv have the color c; as such, uv can be validly assigned the color c. In this way, any valid coloring of $G \setminus e$, where u and v have the same color can be translated into a valid coloring of G'_e. This mapping is unique, as it retains all the colors from the original $G \setminus e$.

We have shown a bijection between the valid colorings of G'_e and $G \setminus e_{u=v}$. As such, the cardinality of each set is equal.

Part c

We will perform an induction on the number of edges in G. Then, $P(0)$ is the following: *A graph with no edges has a polynomial* P_G. We see this is true; there are no constraints on any vertex in G, so P_G is simply x^n, where n is the number of vertices in the graph.

Now, assume $P(m)$: any graph with m edges has a polynomial P_G. We will show that any graph with $m+1$ edges also has a polynomial P_G.

Taking any graph G with $m+1$ edges, we can remove an arbitrary edge $e = (u, v)$ to form $G \setminus e$; this graph has m edges, and as such has a polynomial $P_{G\setminus e}$. This polynomial is certainly larger than P_G, since G has the additional constraint that u and v must not have the same color.

Note that if we contract the same edge e to form G'_e, this graph has m edges and as such, a polynomial $P_{G'_e}$. From (b), we know that this is the same as the number of valid colorings of $G \setminus e$ with at most x colors in which u and v have the same color. In other words, these are exactly the colorings of $G \setminus e$ that are invalid in G. We see that

$$P_G = P_{G\setminus e} - P_{G'_e} \tag{1}$$

Since the difference of two polynomials is a polynomial, it follows that P_G is polynomial. So $P(m+1)$ holds, and our proof is complete.

Part d

Extending our observations from (a), we note that in any graph, if we set the colors of each vertex in an iterative process, each time we set the color of a vertex it introduces a constraint on that vertex's neighbors. Each vertex, then, will have a number of possible colors conditional on its neighbors that have already been colored. This number falls in the range $x...(x-n)$, where n is the highest degree of a vertex in G. The total number of colorings of G will be the product of these possibilities (or zero, if any term falls below 1).

In this way, we see immediately that the degree of P_G should equal the number of vertices in G (since each vertex will have its own term that leads with x). Since the coefficient of each of these terms is 1, we note that the coefficient of the highest-order term is necessarily 1 as well.

We can formalize this argument as a proof by induction on the number of edges in G. Let $P(0)$ be the statement: *A graph with 0 edges and n vertices has P_G with degree equal to n; the coefficient of $x^n = 1$.* We recall that $P_G = x^n$ for a graph with no edges; so $P(0)$ holds.

Now, assuming $P(m)$, we show $P(m+1)$; a n-vertex graph with $m+1$ edges has P_G with degree equal to n, and leading coefficient 1. To prove this, we refer to our equation from (c); specifically, we refer to equation (1).

Note that both $P_{G\setminus e}$ and $P_{G'_e}$ have m edges; so by the IH the degree of $P_{G\setminus e}$ is n (and further, the x^n term has coefficient 1. $P_{G'_e}$ has one less vertex than our original graph due to contraction; so its degree is $n-1$. So $P_{G\setminus e} - P_{G'_e}$ will have degree n (and further, leading coefficient 1). From equation (1), then, we see that $P(m+1)$ holds.

It remains to be seen what the coefficient of the second-highest term represents. Taking our examples from (a), and examining a number of other small examples (for example, $P_{K_3} = (x)(x-1)(x-2) = x^3 - 3x^2 + 2x$), we note that this coefficient is equal to $-|E|$, the number of edges in the graph G. Again, this can be seen more clearly from our inductive proof in (c).

Originally, in a no-edge graph, there is no second degree term; so the second-highest coefficient is 0. We assume that in a m-edge graph, the second-highest coefficient is the number of edges. So for $P_{G\setminus e}$ in our proof for (c), the coefficient of the second highest degree term of $P_{G\setminus e}$ is $-m$.

We note that $P_{G'}$ is a graph with one less vertex than $P_{G\setminus e}$; so it has degree one less. The coefficient of this highest-degree term is 1. So $P_{G\setminus e} - P_{G'}$ results in a polynomial whose second highest degree term has a coefficient one less than the coefficient of $P_{G\setminus e}$. Then, this coefficient is

$-1 * (m) - 1 = -1 * (m + 1)$. Thus, in an $m + 1$-edge graph, the second-highest coefficient is the number of edges as well.

Problem 4

Part 1

The language D_0 consists of all multiples of zero. Thus, the only string in the language is the string 0. We designed a finite automaton that accepts only one string of length 1 in Lecture 22. We show the automaton in Figure 3.

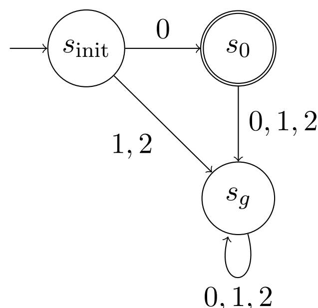


Figure 3: The finite state automaton M_1 for the language D_0.

Part 2

The language D_1 consists of all multiples of one. Therefore, any string that starts with a 1 or a 2 is in the language. The only string in the language that starts with 0 is the string 0. Any other string starting with zero violates the requirement that there be no leading zeros.

We design a finite state automaton M_2 for this language. We start in state s_{init}, and never come back to this state. The machine M_2 goes to state s_0 on input 0. This is an accepting state because 0 is in the language. From s_0, M_2 transitions to the garbage state s_g on all inputs, and this state is not accepting because all such inputs are part of a string with a leading zero. If M_2 gets to state s_g, it can't get out of it. On input 1 or 2 in state s, M_2 goes to state s_1, and never leaves that state afterwards because any string that starts with a 1 or a 2 is in the language.

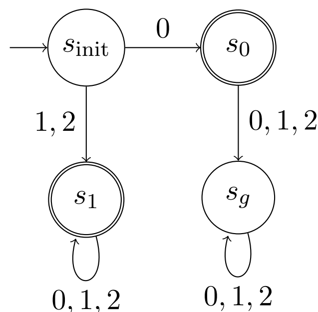


Figure 4: The finite state automaton M_2 for the language D_1.

Part 3

You could try to write down the first few even numbers in ternary and see if there is a pattern. The first few even numbers are 0, 2, 11, 20, 22, 101, 110, 112, 121, 200, ... It is a little tricky to see the pattern, so let's try a different approach.

Using the notation from Lecture 22, the string $x = x_1x_2 \dots x_N$ is the ternary representation of the number $\text{Val}(x) = \sum_{i=1}^{N} a_i 3^{N-i}$. In Lecture 22 we designed an automaton that accepts binary representations of multiples of an integer k. We use the same strategy to design an automaton that accepts ternary representations of multiples of 2.

Let's start by ignoring the issue of leading zeros and design a machine M_3' that accepts ternary representations of multiples of 2 with leading zeros (and also the empty string). For each possible remainder after dividing by 2, there is one state. The state s_0' indicates that the first n digits represent an even integer (in other words, the remainder after dividing $\text{Val}(x_1x_2 \dots x_n)$ by 2 is 0), and s_1' indicates that they represent an odd integer (the remainder is 1). We make the state s_0' accepting and the state s_1' is rejecting.

Now consider what happens when M_3' reads the $(n+1)$st digit. It is in state s_i' where $\text{Val}(x_1x_2 \dots x_n) \bmod 2 = i$. Now $\text{Val}(x_1x_2 \dots x_nx_{n+1}) = 3 \cdot \text{Val}(x_1x_2 \dots x_n) + x_{n+1}$. We would like to know what this is modulo 2 as that will give us the transition function. This is a problem we left as an exercise in Lecture 22, and now solve it in full generality.

Lemma 1. *For any $a, n \in \mathbb{Z}$ and any integer $c \geq 2$, $cn + a \bmod c = a \bmod c$.*

Proof. Write $a = cq + r$ where $c \in \mathbb{Z}$ and $0 \leq r < c$. Then r is the remainder of a after dividing by c. We have $cn + a = cn + cq + r = c(n + q) + r$, and we see that the remainder of $cn + a$ after dividing by c is the same as the remainder after dividing a by c. □

Lemma 2. *For any $a, b, n \in \mathbb{Z}$ and any integer $c \geq 2$,*

$$(b(n \bmod c) + a) \bmod c = (bn + a) \bmod c \tag{2}$$

Proof. Write $a = cq + r$, $b = cs + t$, and $n = cu + v$ where $q, s, u \in \mathbb{Z}$ and $0 \leq r, t, v < c$.

Then $n \bmod c = v$, so $b(n \bmod c) + a = (cs + t)v + cq + r = csv + tv + cq + r = c(sr + q) + tv + r$, so $(b(n \bmod c) + a) \bmod c = (tv + r) \bmod c$ by the previous lemma.

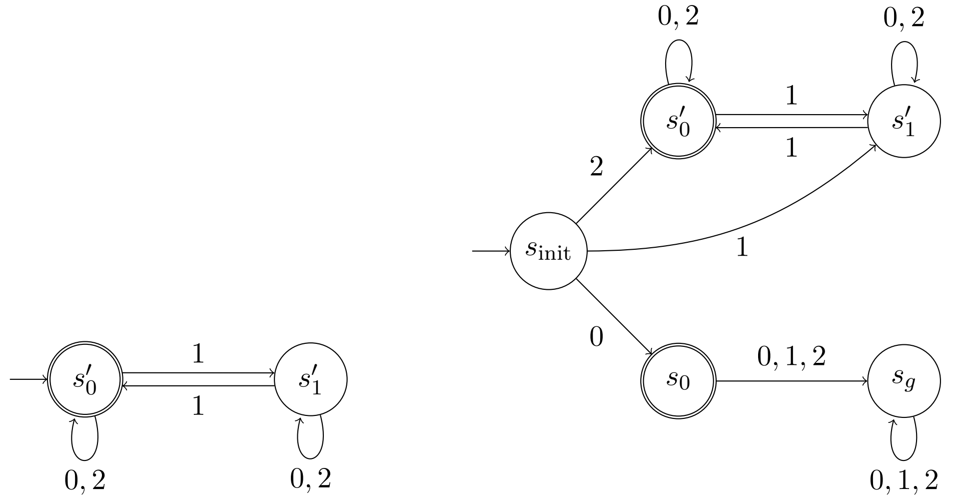
Now let's manipulate the right-hand side. We have $bn + a = (cs + t)(cu + v) + cq + r = c^2su + csv + ctu + tv + cq + r = c(csu + sv + tu + q) + tv + r$, and we can apply the previous lemma to get $(bn + a) \bmod c = (tv + r) \bmod c$.

We have shown that both sides of (2) are the same, and we are done. □

We can now use Lemma 2 to design the transition function the same way as in Lecture 22. When M_3' is in state s_i', it goes to state $s'_{(3i + x_{n+1}) \bmod 2}$ after it reads x_{n+1}.

When the machine starts, it hasn't read anything, and we declare that the part of the input read so far is even. Thus, the machine starts in state s_0'. We show the machine in Figure 5a. Using the same strategy as in Lecture 22, we convert the machine from Figure 5a to the machine M_3 for D_2, which is shown in Figure 5b.

We offer an alternative description of strings in D_2 that will be used later. Note that 3^{N-i} is odd for all $i \in \{1, \dots, N\}$. Now whenever x_i is even, that is, either 0 or 2, the term $x_i 3^{N-i}$ is even. If $x_i = 1$, the term $x_i 3^{N-i}$ is odd, and this is the only way how it can be odd. Then the sum



(a) An automaton M'_3 that accepts ternary representations of multiples of 2, including those with leading zeros.

(b) The finite state automaton M_3 for the language D_2.

Figure 5: Designing a finite state automaton for the language D_2.

$\sum_{i=1}^{N} x_i 3^{N-i}$ is even if and only if an even number of the coefficients x_i is odd. Therefore, a string x is in D_2 if and only if it contains an even number of ones, and doesn't start with 0 unless $x = 0$.

Observe that the automaton M'_3 accepts exactly those strings that contain an even number of ones. We saw a similar automaton in Lecture 21 for binary strings.

Part 4

The language L_1 is a modification of D_2. The empty string is now allowed, and the string 0 is not allowed.

Let's design a finite state automaton M_4 for this language by modifying M_3. Since 0 was the only string in D_2 that starts with a zero, and this string is now disallowed, we can drop the state s_0 of M_3, and send M_4 to the garbage state s_g right after it reads the initial 0. Everything else stays the same. We show the automaton in Figure 6.

Observe that the only difference between D_2 and L_1 is the fact that no nonempty string in L_1 starts with a zero, and that unlike D_2, L_1 contains the empty string. So L_1 consists of all nonempty strings that don't start with a zero and contain an even number of ones, together with the empty string.

Part 5

The language $L_2 = L_1 D_2$ consists of all concatenations of two strings, one from L_1 and one from D_2, i.e., $L_2 = \{xy \mid x \in L_1, y \in D_2\}$. We show that $L_2 = D_2$.

First, we argue that $L_2 \subseteq D_2$. The language L_1 contains no strings that start with a zero, and contains the empty string ϵ. Consider a string $z = xy$ in L_2 with $x \in L_1$ and $y \in D_2$. If z starts with zero, $x = \epsilon$ because no string in L_1 starts with a zero, and $y = z$, so $z \in D_2$. If z does not start

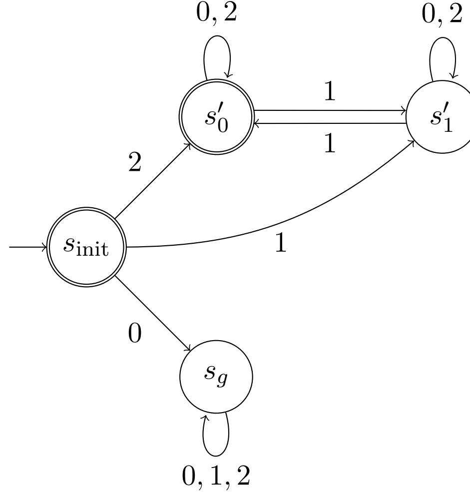


Figure 6: The finite state automaton M_4 for the language L_1.

with a zero, both x and y contain an even number of ones, so their concatenation also contains an even number of ones. Because the string z does not start with a zero and contains an even number of ones, our alternative characterization of D_2 implies that $z \in D_2$.

Conversely, consider any string $w \in D_2$. Then the string $z = \epsilon w$ where ϵ is the empty string belongs to L_2 because $\epsilon \in L_1$ and $w \in D_2$. But $\epsilon w = w$, so $w \in L_2$, and it follows that $L_2 \subseteq D_2$.

We have shown that $L_2 = D_2$. The finite state automaton for this language is M_3 in Figure 5b.

Part 6

We show that $L_1^* = L_1$. To get some intuition, consider our alternative characterization of L_1. This language contains all strings that don't start with a zero and contain an even number of ones. Concatenating any number of such strings results in another string that does not start with a zero and has an even number of ones. The only string we did not consider is the empty string $\epsilon \in L_1^0$, and the empty string is in L_1 by definition. We argue more formally in the paragraphs that follow.

The containment $L_1 \subseteq L_1^*$ follows by the definition of Kleene closure. We argue the other containment by induction. In particular, we show that $L_1^k \subseteq L_1$ for all $k \in \mathbb{N}$.

We have $L_1^0 = \{\epsilon\}$, and $\epsilon \in L_1$, so $L_1^0 \subseteq L_1$, which proves the base case. We also have $L_1^1 = L_1$, so $L_1 \subseteq L_1$ also holds.

Now assume $L_1^i \subseteq L_1$ for some $i \geq 1$. We can write $L_1^{i+1} = L_1 L_1^i$. Consider any string $z \in L_1^{i+1}$, which we can write as $z = y_1 y_2 \dots y_i y_{i+1}$ where $y_k \in L_1$ for $k \in \{1, \dots, i+1\}$. Now the string $y' = y_2 y_3 \dots y_{n+1}$ is a concatenation of n strings from L_1, so it's in L_1 by the induction hypothesis. It follows that we can write $z = y_1 y'$ where $y_1, y' \in L_1$. There are two cases to consider.

Case 1: $y_1 = \epsilon$ or $y' = \epsilon$. Then $z = y'$ or $z = y_1$, and both of those are in L_1, so $z \in L_1$.

Case 2: $y_1, y' \neq \epsilon$. In that case y_1 consists of an even number of ones by our alternative characterization of D_2. The same holds about y', so the concatenation of y_1 and y' also contains an even number of zeros. Furthermore, the concatenation does not start with a zero and is nonempty, so it's in L_1.

Thus, $L_1^k \subseteq L_1$ for all $k \in \mathbb{N}$. This implies that $L_1^* = \bigcup_{k=0}^{\infty} L_1^k \subseteq \bigcup_{k=0}^{\infty} L_1 = L_1$, which completes

the proof that $L_1^* = L_1$.

Now L_3 is the concatenation of a string in L_1 followed by a zero. Since $\epsilon \in L_1$ and no other string in L_1 starts with a zero, the only string in L_3 that starts with a zero is the string 0.

Consider a nonempty string in L_1. Such a string contains an even number of ones, so it's a multiple of 2. Also, since it ends with a zero, $\mathrm{Val}(x_1x_2\ldots x_N) = \sum_{i=1}^{N} x_i 3^{N-i} = \sum_{i=1}^{N-1} x_i 3^{N-i} + 0 = 3\sum_{i=1}^{N-1} x_i 3^{N-1-i}$, which means that the string represents a number that is divisible by 3. Any integer that is divisible by 2 and 3 is divisible by 6, so we have just shown that every nonempty string in L_1 followed by a zero is in D_6. Since 0 is also a multiple of 6, we actually get $L_3 \subseteq D_6$.

Now consider any multiple of 6, and let $x = x_1x_2\ldots x_N$ be its ternary representation. Since $\mathrm{Val}(x)$ is an even number, $x \in D_2$. Furthermore, any multiple of 6 is divisible by 3, so the last digit in its ternary representation is a zero. To see that, consider

$$\mathrm{Val}(x_1x_2\ldots x_N) = \sum_{i=1}^{N-1} x_i 3^{N-i} + x_N = 3\sum_{i=1}^{N-1} x_i 3^{N-1-i} + x_N,$$

and note that the only way the right-hand side is divisible by 3 is if $x_N = 0$. This also implies that all occurrences of 1 in x appear in $x' = x_1x_2\ldots x_{N-1}$. Since x contains an even number of ones, so does x' by the previous sentence. Also note that x' does not start with a zero because otherwise x would be a string with leading zeros, which is something that cannot happen for a string in D_6. It follows that x' is a string with an even number of ones that does not start with a zero (the empty string is a possibility, and it also satisfies this characterization), so $x' \in L_1$. It follows that $x \in L_3$. Hence, $D_6 \subseteq L_3$, and $L_3 = D_6$.

We can design the automaton M_6 for this language exactly the same way as we designed M_3. The automaton is in Figure 7.

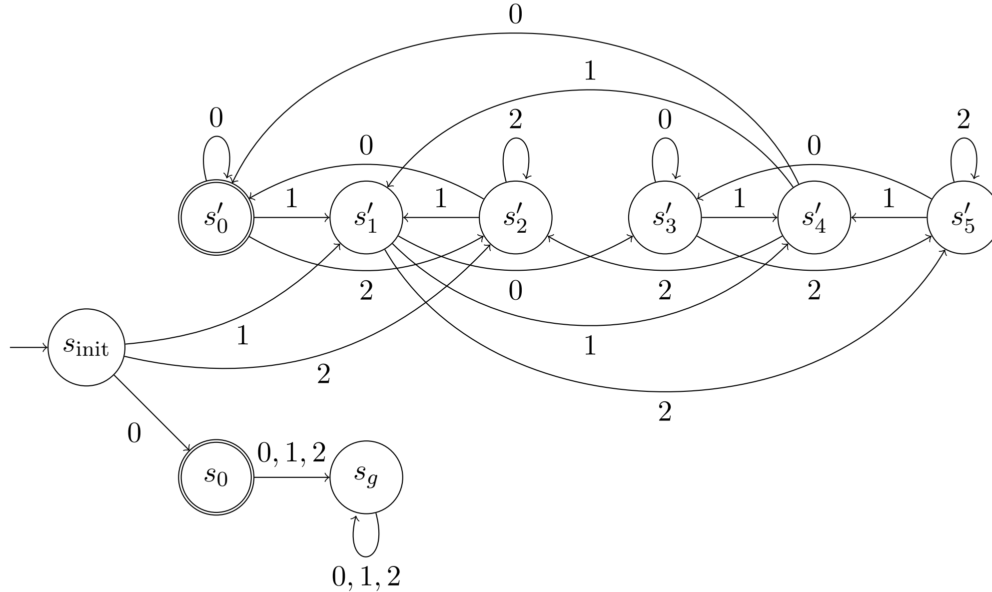


Figure 7: The finite state automaton M_6 for the language L_3.

Part 7

Now let's analyze the language $L_4 = ((D_2 \setminus \{0\})\{xx \mid x \in \{0,1,2\}^*\}) \cup \{0\}$. To simplify notation, let $R = \{xx \mid x \in \{0,1,2\}^*\}$.

First let's focus on the set $R = \{xx \mid x \in \{0,1,2\}^*\}$. To get a string in this set, take any sequence x of zeros, ones, and twos, and write it down twice. Such a sequence has an even number of ones because it contains twice the number of ones in x. Thus, if we attach it at the end of a nonzero string from D_2, we get another nonempty string that does not start with zero and has an even number of ones. This string is in D_2 by our alternative characterization of D_2 from Part 3. Since the string doesn't start with zero, it is in $D_2 \setminus \{0\}$, and we get that $(D_2 \setminus \{0\})R \subseteq D_2 \setminus \{0\}$. The other containment also holds since if $x \in D_2 \setminus \{0\}$, then $x\epsilon \in (D_2 \setminus \{0\})R$, so $(D_2 \setminus \{0\})R = D_2 \setminus \{0\}$.

Now we have $L_4 = ((D_2 \setminus \{0\})R) \cup \{0\} = (D_2 \setminus \{0\}) \cup \{0\} = D_2$. The finite state automaton for this language is M_3 in Figure 5b.

Problem 5

Part a

We model the situation the monk is facing using a finite state machine.

The states are tuples (r, g) where r is the number of red beads and g is the number of green beads the monk has in his bowl. The monk starts in the state $(15, 12)$.

The input alphabet is the set $\{\text{exchange}, \text{swap}\}$.

On exchange, the monk goes from state (r, g) to state $(r-3, g+2)$. On swap, the monk goes from state (r, g) to state (g, r).

The definition of a finite state machine requires that the number of states be finite, but there are infinitely many pairs (r, g) with $r, g \in \mathbb{N}$. We argue that only a finite number of those is necessary in order to characterize all states the monk can be in. For that, we analyze how the quantity $s = r + g$ changes during the monk's stay at the monastery.

Let the monk's state after i days be (r_i, g_i), and let $s_i = r_i + g_i$. We have $r_0 + g_0 = 27$. The transitions change the value of s as follows. If the monk performs an exchange on day $i+1$, the new sum is $s_{i+1} = r_{i+1} + g_{i+1} = r - 3 + g + 2 = r + g - 1 = s_i - 1$, and if the monk performs a swap, the new sum is $s_{i+1} = s_i$ because he just changes all beads one for one. Thus, s can either stay the same or decrease by one on any given day, which means that s never exceeds its initial value of 27.

Using the previous paragraph, we can give a rough upper bound on the number of states, thus showing that this number is finite. Since the monk cannot have a negative number of beads of any color, $r, g \geq 0$, and there are at most 28 ways to set r and g so that $r + g = s$ for any value of s. Because there are at most 28 possible values of s, we need at most 28^2 out of the infinitely many pairs (r, g) to describe all the states a monk can be in.

Part b

To show that the monk never leaves the monastery, it suffices to show that he can't reach state $(5, 5)$ from the start state $(15, 12)$. For that, we analyze the quantity $d = r - g$.

Let the monk's state after i days be (r_i, g_i), and let $d_i = r_i - g_i$. If the monk performs an exchange on the next day, the difference becomes $d_{i+1} = (r_i - 3) - (g_i + 2) = r_i - g_i - 5 = d_i - 5$. After a swap, the difference becomes $d_{i+1} = g_i - r_i = -(r_i - g_i) = -d_i$.

Let's examine the possible values of d. At the beginning, we have $d = 15 - 12 = 3$. If the monk keeps performing exchanges, the difference keeps decreasing by 5 so some possible differences are $3, -2, -7, -12, -17, \ldots$. In any state, the monk can perform a swap, which flips the sign of the difference. Thus, more possible differences are $\ldots, 17, 12, 7, 2, -3$. If the monk performs exchanges in those states, he can reach states $\ldots, 17, 12, 7, 2, -3, -8, -13, \ldots$. Performing an exchange in one of those states takes the monk to one of the states $\ldots, 13, 8, 3, -2, -7, -12, -17, \ldots$, and we have seen this situation already. It looks like we have found all the possibilities for the difference $r - g$, and zero is not among them. This indicates that the monk should not be able to leave the monastery.

We now give a formal proof. Let's start with an invariant.

Invariant 1. *After i days, $d_i = 2 + 5x_i$ or $d_i = 3 + 5x_i$ for some $x_i \in \mathbb{Z}$.*

Proof. We argue by induction. For the base case, we have $d_0 = r_0 - g_0 = 15 - 12 = 3$, so $d_0 = 3 + 5x_0$ for $x_0 = 0$.

Now assume that the invariant holds after day i. There are four cases to consider, depending on which form d_i takes and which action the monk takes on day $i + 1$.

Case 1.1: $d_i = 3 + 5x_i$, action = exchange. Then we get $d_{i+1} = d_i - 5 = 3 + 5x_i - 5 = 3 + 5(x_i - 1)$, so the invariant is maintained with $x_{i+1} = x_i - 1$.

Case 1.2: $d_i = 3 + 5x_i$, action = swap. Then $d_{i+1} = -d_i = -3 - 5x_i = 2 - 5 - 5x_i = 2 - 5(x_i + 1)$, so the invariant is maintained with $x_{i+1} = -(x_i + 1)$.

Case 2.1: $d_i = 2 + 5x_i$, action = exchange. Then we get $d_{i+1} = d_i - 5 = 2 + 5x_i - 5 = 2 + 5(x_i - 1)$, so the invariant is maintained with $x_{i+1} = x_i - 1$.

Case 2.2: $d_i = 2 + 5x_i$, action = swap. Then $d_{i+1} = -d_i = -2 - 5x_i = 3 - 5 - 5x_i = 3 - 5(x_i + 1)$, so the invariant is maintained with $x_{i+1} = -(x_i + 1)$. □

Note that no integer x satisfies $0 = 2 + 5x$ or $0 = 3 + 5x$, so Invariant 1 implies that the monk can never reach the state $(5, 5)$ where the difference $r - g$ is zero. It follows that no monk can leave the monastery.

Extra Credit

Instead of describing the states in the finite state machine of part (a) of Problem 5 using the number of red and green beads, we can describe them using the quantities s and d defined in parts (a) and (b) of Problem 5, respectively. This is because the system of equations

$$\begin{aligned} s &= r + g \\ d &= r - g \end{aligned}$$

has a unique solution $r = \frac{s+d}{2}$, $g = \frac{s-d}{2}$, so the pair (s, d) uniquely defines a pair (r, g). From now on, we use s and d to describe the monk's state.

We saw in part (a) of Problem 5 that the value of s can either stay the same or decrease by 1. The only way it can stay the same is if the monk performs a swap, and in that case d gets multiplied by -1 by part (b) of Problem 5. For a given state (s, d), the only other state with the same value of s the monk can reach in his life is, therefore, the state $(s, -d)$. Thus, for each value of s, the monk can be in at most two different states with that value of s during his life.

For each value of s with $s \geq 3$, the monk has a chance of performing an exchange unless his actions in his early days in the monastery prevent him from doing so (for example, he cannot make an exchange with 2 red beads and 1 green bead). When s gets below 3, the monk can only perform swaps, and cannot decrease s any further. This means that during his lifetime, the monk can be in states with at most $27 - 2 + 1 = 26$ different values of s, and for each such value he can be in two different states, for a total of 52 states in his lifetime.

To complete the proof, we demonstrate a sequence of actions the monk can take in order to reach this maximum number of states. In particular, we show the following.

Lemma 3. *For each value of i with $0 \leq i \leq 25$, the monk can reach the state $(27-i, 2)$ or $(27-i, 3)$ and visit a total of $2(i+1)$ different states on the way.*

Proof of Lemma 3. We argue by induction on i.

The initial state with is $(s, d) = (27, 3)$, which corresponds to the case $i = 0$. The monk can perform two swaps to go to state $(27, -3)$ and back to $(27, 3)$. Now the monk is in state $(27, 3)$ and visited 2 different states, so the base case is proved.

So suppose that for some value of i, the monk can reach either state $(27 - i, 2)$ or $(27 - i, 3)$ with a total of $2(i + 1)$ states visited. There are two cases to consider. In the arguments below, we define $s = 27 - i$ to simplify notation. Note that if $s = 2$, there is nothing to prove in the induction step because i has the maximum value of 25 in that case. Thus, we assume $s \geq 3$ in the inductive step.

Case 1: The monk is in state $(s, 3)$. In this case $r \geq 3$ because $g \geq 0$, $d = 3$, and $r = g + d$. Therefore, the monk can perform an exchange to get to the state $(s - 1, 3 - 5) = (s - 1, -2)$. After that, he performs a swap to reach the state $(s - 1, 2)$. The monk could not have visited the states $(s - 1, -2)$ and $(s - 1, 2)$ prior to visiting $(s, 3)$ because the value of the first component cannot increase. Thus, at the point the monk reaches the state $(s - 1, 2) = (27 - (i + 1), 2)$, he visited $2(i + 1) + 2 = 2(i + 2)$ different states.

Case 2: The monk is in state $(s, 2)$. Because we assume $s \geq 3$, we have $r + g \geq 3$. Furthermore, $r - g = 2$, so the monk must have at least 3 red beads (having 2 red beads would mean he has no green beads, and a total of only two beads, thus violating $s \geq 3$). Then he can perform an exchange and get to state $(s - 1, 2 - 5) = (s - 1, -3)$. After a swap, he reaches the state $(s - 1, 3)$. Since the value of the first component of the state only decreases, the states $(s - 1, -3)$ and $(s - 1, 3)$ were visited for the first time, and the monk visited $2(i + 1) + 2 = 2(i + 2)$ states total. This completes the proof. □

By Lemma 3, the monk can reach the state $(2, 0)$. Now $2 = 27 - 25$, so Lemma 3 also tells us that he can do this in a way that takes him through $2 \cdot (25 + 1) = 52$ different states.