CSE 103 Computational Models
Fall 2021 Prof. Fremont

Practice 1

Midterm Practice Problems

This set of problems tries to cover most of the material we've seen so far, so it is large. Feel free to jump around and find (sub)problems which you think will be most helpful. Each subproblem is a reasonable question to have on the midterm.

1. Notation for sets and languages

Let $\Sigma = \{0,1\}$. For each of the following statements, decide whether it is true or false.

(a) $\Sigma^* - \{x_1 \ldots x_n \mid n \geq 1, x_i \in \Sigma\} = \emptyset$

Solution: This is false: the set on the right consists of all words over Σ of length at least 1, whereas Σ^* contains all words over Σ of any length, so their difference is the set $\{\varepsilon\}$, not the empty set.

(b) $\{x \in \Sigma^* \mid \exists k \geq 0.\ |x| = 2k\} = \{00,01,10,11\}^*$

Solution: This is true: the set on the left consists of all strings over Σ of even length, which are exactly the strings that can be broken up into a sequence of zero or more pairs from $\{00,01,10,11\}$.

(c) $|\Sigma^*| = |\{S \subseteq \Sigma\}|$

Solution: This is false: the left-hand size is infinite, since there are infinitely-many strings over Σ, but the right-hand size is the number of subsets of Σ, which is $2^{|\Sigma|} = 4$ (the power set $\mathscr{P}(\Sigma)$ consisting of all subsets of Σ is $\{\emptyset, \{0\}, \{1\}, \{0,1\}\}$).

(d) $|\Sigma^3 \times \Sigma^5| < |\Sigma^8|$

Solution: This is false: $\Sigma^3 \times \Sigma^5$ is the set of pairs (x,y) where $x \in \Sigma^3$ and $y \in \Sigma^5$, so $|\Sigma^3 \times \Sigma^5| = |\Sigma^3| \cdot |\Sigma^5| = 2^3 \cdot 2^5 = 2^8 = |\Sigma^8|$. Another way to see this is that a string of length 3 followed by a string of length 5 make up a string of length 8.

(e) $\{xyz \mid x,y,z \in \Sigma \cup \{\varepsilon\}\} = \Sigma^{\leq 3}$

Solution: This is true: if $x,y,z \in \Sigma$ then $xyz \in \Sigma^3$, while if $x = \varepsilon$ but $y,z \in \Sigma$ then $xyz = yz \in \Sigma^2$, and so forth down to $x = y = z = \varepsilon$, in which case $xyz = \varepsilon \in \Sigma^0$. So we can get all strings over Σ of length up to 3 in this way.

(f) $|\mathscr{P}(X)| > |X|$ for all finite sets X.

Solution: This is true: the power set $\mathscr{P}(X)$ has $2^{|X|}$ elements, since when building a subset of X we can choose for each element of X whether or not to include it. Since $2^n > n$ for all $n \geq 0$, the inequality is true.

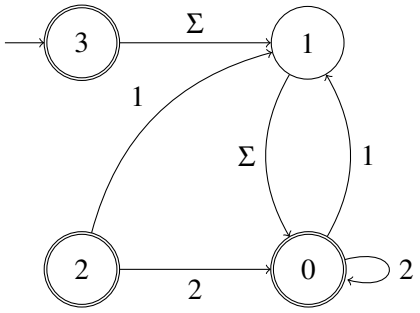
(*Note:* This inequality is actually also true for infinite sets, if you use an appropriate notion of size: this result is called "Cantor's theorem". It establishes that there are different infinities, and kicked off the development of set theory in the late 19th century.)

2. Working with a DFA (including converting to a regular expression)
Consider the DFA $M = (Q, \Sigma, \delta, q_0, F)$ where:

- $Q = \{0,1,2,3\}$
- $\Sigma = \{1,2\}$
- $\delta(q,s) = \begin{cases} 1 & q = 0 \text{ and } s = 1 \\ 0 & s > q \\ 1 & q = 3 \text{ and } s = 1 \\ q - s & \text{otherwise} \end{cases}$
- $q_0 = 3$
- $F = \{0,2,3\}$

(a) Draw M as a graph.

Solution:



(b) Which of the following strings does M accept: $\varepsilon, 1, 112, 111, 2221, 1112$? Give an accepting path for each accepted string.

Solution:

ε**:** Accepted, via the path (3).

1**:** Rejected (the path $(3,1)$ ends in a rejecting state).

112**:** Accepted, via the path $(3,1,0,0)$.

111**:** Rejected; the path is $(3,1,0,1)$.

2221**:** Rejected; the path is $(3,1,0,0,1)$.

1112**:** Accepted, via the path $(3,1,0,1,0)$.

(c) What is $\hat{\delta}(q_0,x)$ for each of the strings x from part (b)? What about $\hat{\delta}(2,x)$?

Solution:

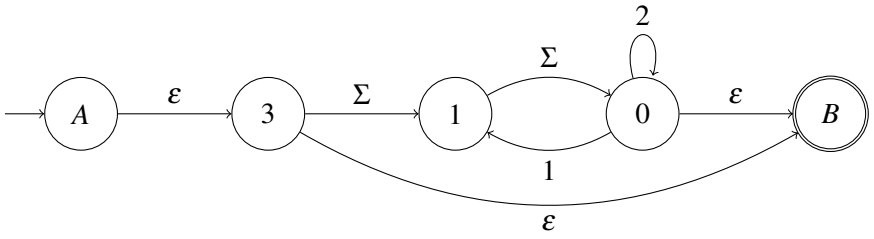
- $\hat{\delta}(q_0,\varepsilon) = 3, \quad \hat{\delta}(2,\varepsilon) = 2.$
- $\hat{\delta}(q_0,1) = 1, \quad \hat{\delta}(2,1) = 1.$
- $\hat{\delta}(q_0,112) = 0, \quad \hat{\delta}(2,112) = 0.$
- $\hat{\delta}(q_0,111) = 1, \quad \hat{\delta}(2,111) = 1.$
- $\hat{\delta}(q_0,2221) = 1, \quad \hat{\delta}(2,2221) = 1.$
- $\hat{\delta}(q_0,1112) = 0, \quad \hat{\delta}(2,1112) = 0.$

(d) What is the language $L(M)$ of M? (You may give your answer as a regular expression.)

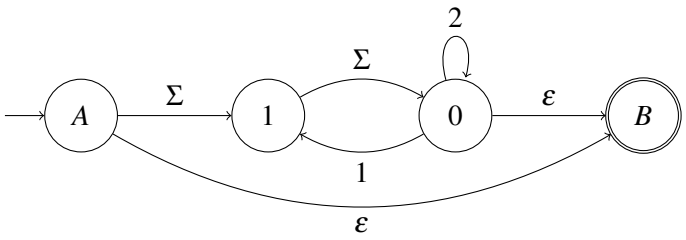
The only reachable accepting states are 3 and 0, and the only string whose path ends in state 3 is the empty string ε. To get to state 0 we need any two symbols, and we can then stay there by either reading a 2 or reading a 1 followed by another symbol. So $L(M) = \varepsilon \mid \Sigma\Sigma(2 \mid 1\Sigma)^*$.

(e) Convert M into an equivalent regular expression, showing the sequence of GNFAs you get along the way. Confirm that the expression you get agrees with your answer to part (d).

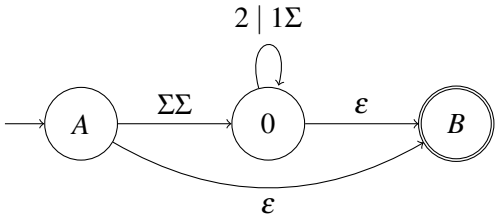
Solution: We obtain our first GNFA by adding new start and accepting states, with ε-transitions to the latter from the original accepting states. We'll also drop state 2 since it is unreachable and so will not affect our result. This gives us:



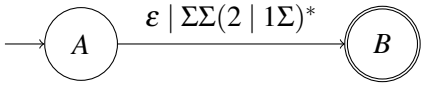
Let's eliminate state 3 first. To take into account paths from state A to 1 through 3, we add a transition from A to 1 labeled with $\varepsilon\Sigma = \Sigma$; similarly, for paths from A to B through 3 we add a transition from A to B labeled with $\varepsilon\varepsilon = \varepsilon$. So we get:



Next, we'll eliminate state 1. To take into account paths from A to 0 through 1, we'll add a transition from A to 0 labeled with $\Sigma\Sigma$; for paths from 0 back to itself through 1, we'll add 1Σ to its self-loop. So we get:



Finally, to eliminate state 0 we have to take into account paths from A to B through 0: since the self-loop on 0 is labeled $2 \mid 1\Sigma$, we add $\Sigma\Sigma(2 \mid 1\Sigma)^*\varepsilon = \Sigma\Sigma(2 \mid 1\Sigma)^*$ to the existing transition from A to B. This yields our final GNFA:



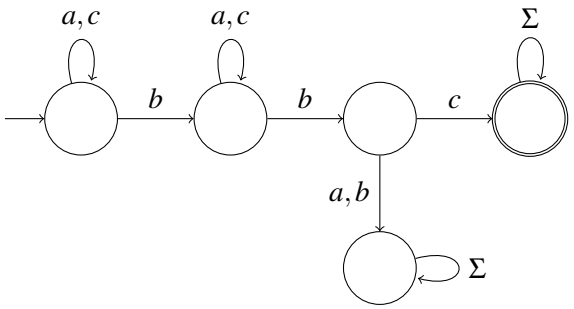
From this, we can read off the regular expression $\varepsilon \mid \Sigma\Sigma(2 \mid 1\Sigma)^*$, agreeing with the language we found in part (d).

3. Designing DFAs

Draw DFAs recognizing each of the following languages:

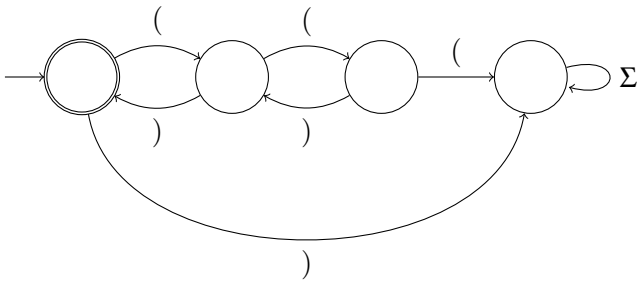
(a) All strings over $\Sigma = \{a, b, c\}$ which contain at least 2 bs, and where the second b is immediately followed by a c.

Solution:



(b) All strings of correctly-nested parentheses whose nesting depth never exceeds 2 (so for example $(()())()$ is fine but $(()(()))$ is not).

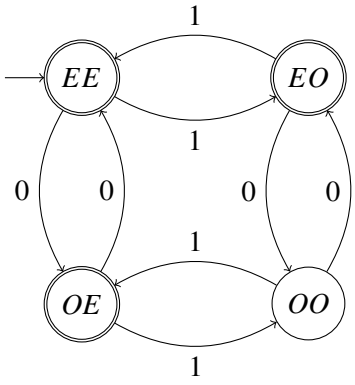
Solution:



(c) All binary strings which contain an even number of 0s or an even number of 1s (or both).

Solution:

In the DFA below, state EO for example indicates that we have read an *even* number of 0s and an *odd* number of 1s.



4. Algorithms for DFAs

Give short descriptions of algorithms for each of the following problems. In each case, try to make your algorithm as fast as possible, and state its asymptotic runtime in terms of the size of its input (for a DFA D, measure its size $|D|$ as the total number of transitions, i.e. the number of states times the size of the alphabet).

(*Note:* You should not need to look up any algorithms; all of these problems can be solved using variations of graph algorithms and/or algorithms we've discussed in class or on the homework.)

(a) Decide whether there is a word $w \in L(D)$ containing the symbol s.

Solution: Viewing D as a graph, we can use DFS or another graph traversal to find all states reachable from the start state. Delete all states which were not reachable, along with the transitions leading to them. For each of the remaining states which still have an incoming transition on input s, use another DFS to check if it can reach an accepting state. If so, then there is an accepting path in D that uses an s-transition, so some word in $L(D)$ contains s; otherwise, no accepting path can use an s-transition, so no word in $L(D)$ contains s.

This procedure does one DFS to start, plus one DFS for every state of D in the worst case. Each DFS takes linear time, so the total runtime is $O(|D|^2)$.

Alternate Solution: Since writing this problem, I realized you can actually do it in linear time using Tarjan's strongly-connected components algorithm, which I don't expect you to know. But if you're interested, read on.

Use Tarjan's algorithm to find the strongly-connected components of D (as a directed graph), as well as a topological order on them. Go through the components in reverse topological order, labeling them as *good* if they contain an accepting state or one of their successors (which we've already looked at) does. Now do a forward traversal starting from the component of the start state, and throw out all components which are not good or which are not reachable from the start state. At this point we are left with only the states which are both reachable from the start state and able to reach an accepting state. So we can now simply go through the graph and check if there are any s-transitions left. Tarjan's algorithm takes linear time, and so do our various traversals through the graph, so the total runtime is $O(|D|)$.

(b) Given a DFA D, find the longest word in $L(D)$ (if there is a tie, return any one of the longest words) or determine that $L(D)$ contains arbitrarily long words.

Solution: First, do a DFS from every state as in part (a) above to find all states which are both reachable from the start state and able to reach an accepting state, throwing out all other states. If the remaining graph contains a cycle (which you can tell during DFS), then $L(D)$ contains arbitrarily long words, since we can repeat the cycle like in the pumping lemma. Otherwise, the graph is acyclic, and the longest words in $L(D)$ correspond to the longest paths in the graph starting from the start state. We can use breadth-first search (BFS) to find such a longest path.

This procedure does one DFS to start, plus one DFS for every state of D in the worst case, plus a final BFS. Each DFS and BFS takes linear time, so the total runtime is $O(|D|^2)$. (N.B. You can get this down to linear using Tarjan's algorithm to do the initial pruning as in the alternate solution to part (a).)

(c) Given a DFA D and a length ℓ, decide if $L(D)$ contains any word of length ℓ. (You should express the runtime of your algorithm in terms of ℓ as well as $|D|$.)

Solution: Do a BFS-like traversal of the DFA, maintaining a set S_i of all states reachable after exactly i transitions. That is, start with $S_0 = \{q_0\}$ (the initial state) and compute S_{i+1} by following all outgoing transitions from every state in S_i. If S_ℓ contains an accepting state, then there is a path to that state consisting of ℓ transitions, so $L(D)$ contains a word of length ℓ; otherwise, every path of ℓ transitions ends in a rejecting state, so $L(D)$ contains no word of length ℓ.

In the worst case, at each iteration we must consider all transitions of D (if every state is in S_i), so each iteration takes $O(|D|)$ time. There are $O(\ell)$ iterations, so the total runtime is $O(\ell|D|)$.

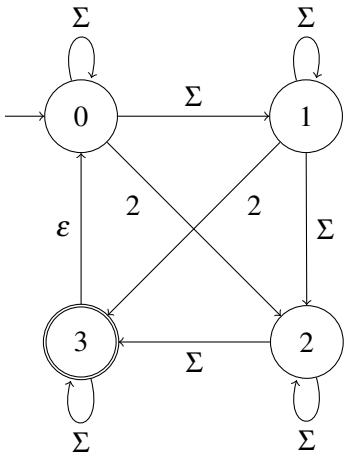
5. Working with an NFA

Consider the NFA $M = (Q, \Sigma, \delta, q_0, F)$ where:

- $Q = \{0,1,2,3\}$
- $\Sigma = \{1,2\}$
- $\delta(q,s) = \begin{cases} \{0\} & q = 3 \text{ and } s = \varepsilon \\ \{q' \in Q \mid 0 \leq q' - q \leq s\} & s \in \Sigma \\ \emptyset & \text{otherwise} \end{cases}$
- $q_0 = 0$
- $F = \{3\}$

(a) Draw M as a graph.

Solution:



(b) What are the ε-closures $E(\{0\})$, $E(\{3\})$, $E(\{0,3\})$, and $E(\{1,2,3\})$?

Solution:

- $E(\{0\}) = \{0\}$
- $E(\{3\}) = \{0,3\}$
- $E(\{0,3\}) = E(\{0\}) \cup E(\{3\}) = \{0,3\}$
- $E(\{1,2,3\}) = E(\{1\}) \cup E(\{2\}) \cup E(\{3\}) = \{0,1,2,3\}$

(c) Which of the following strings does M accept: $\varepsilon, 1, 21, 11, 111, 1112$? Give an accepting path for each accepted string.

Solution:

ε**:** Rejected; the start state 0 is not accepting, and it has no outgoing ε-transitions.

1**:** Rejected; we can stay in state 0 or move to state 1, but neither is accepting.

21**:** Accepted, via the path $(0,2,3)$.

11**:** Rejected; we can stay in state 0, move to state 1 and stay there, or move to state 1 and then 2, but none of these are accepting states.

111**:** Accepted, via the path $(0,1,2,3)$.

1112**:** Accepted, via various paths including $(0,0,1,2,3)$ and $(0,1,2,3,3)$.

(d) What is $\hat{\delta}(q_0, x)$ for each of the strings x from part (b)? What about $\hat{\delta}(3,x)$?

Solution:

- $\hat{\delta}(q_0, \varepsilon) = \{0\}, \quad \hat{\delta}(3, \varepsilon) = \{0,3\}$
- $\hat{\delta}(q_0, 1) = \{0,1\}, \quad \hat{\delta}(3,1) = \{0,1,3\}$
- $\hat{\delta}(q_0, 21) = \{0,1,2,3\}, \quad \hat{\delta}(3,21) = \{0,1,2,3\}$
- $\hat{\delta}(q_0, 11) = \{0,1,2\}, \quad \hat{\delta}(3,11) = \{0,1,2,3\}$
- $\hat{\delta}(q_0, 111) = \{0,1,2,3\}, \quad \hat{\delta}(3,111) = \{0,1,2,3\}$
- $\hat{\delta}(q_0, 1112) = \{0,1,2,3\}, \quad \hat{\delta}(3,1112) = \{0,1,2,3\}$

(e) What is the language $L(M)$ of M?

Solution: The only accepting state is state 3, and once we get there we can remain, so we just need to find all ways to reach state 3. As we saw above, ε is rejected, so we need at least one symbol. If the first symbol is 1, then we can advance to state 1, whereupon we can either read a 2 to go directly to 3 or read a 1 to go to 2 and then another symbol to go to 3. If instead the first symbol is 2, then we can go to state 2 and then read another symbol to reach 3. So $L(M) = (12 \mid 11\Sigma \mid 2\Sigma)\Sigma^*$.

Alternate Solution: Clearly any string of length ≥ 3 is in $L(M)$, because of the Σ-transitions from 0 to 1 to 2 to 3. Checking all strings of length ≤ 2, we find that only ε, 1, 2, and 11 are rejected. So $L(M)$ consists of all strings except these four (which is in fact equivalent to the regular expression above).

(f) Use the subset construction to build a DFA equivalent to M (only drawing the states reachable from the start state). Confirm that it has the same language you found in part (e).

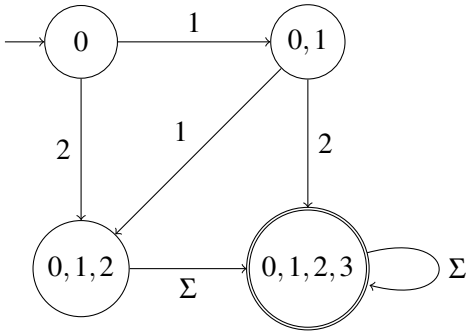
Solution: The initial state will be $E(\{q_0\}) = \{0\}$. On input 1, since $\delta(0,1) = \{0,1\}$ we will transition to state $\{0,1\}$. On input 2, since $\delta(0,2) = \{0,1,2\}$ we will transition to state $\{0,1,2\}$.

Consider state $\{0,1\}$ next. On input 1, since $\delta(0,1) = \{0,1\}$ and $\delta(1,1) = \{1,2\}$ we will transition to state $\{0,1,2\}$. On input 2, since $\delta(0,2) = \{0,1,2\}$ and $\delta(1,2) = \{1,2,3\}$ we will transition to state $\{0,1,2,3\}$.

Consider state $\{0,1,2\}$ next. On input 1, since $\delta(0,1) = \{0,1\}$, $\delta(1,1) = \{1,2\}$, and $\delta(2,1) = \{2,3\}$, we will transition to $\{0,1,2,3\}$. On input 2, since $\delta(0,2) = \{0,1,2\}$, $\delta(1,2) = \{1,2,3\}$, and $\delta(2,2) = \{2,3\}$, we will transition to $\{0,1,2,3\}$.

Finally, at state $\{0,1,2,3\}$, since $q \in \delta(q,s)$ for all $q \in Q$ and $s \in \Sigma$, on any input we will transition to $\{0,1,2,3\}$.

We have not reached any new states. Of the four we found above, only $\{0,1,2,3\}$ is accepting, since it is the only state containing the accepting state 3. So we obtain the DFA:



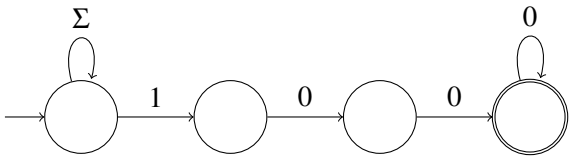
This DFA only accepts if we read a prefix of the form 11Σ, 2Σ, or 12, followed by zero or more symbols. This agrees with the language we found in part (e) above.

6. Designing NFAs

Draw NFAs recognizing each of the following languages:

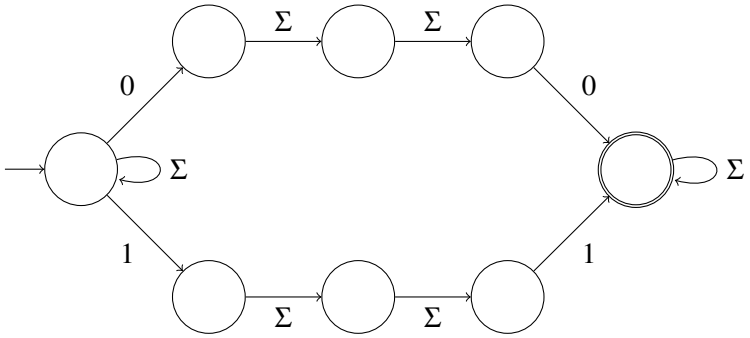
(a) Binary strings where the last 1 is followed by at least two 0s.

Solution:



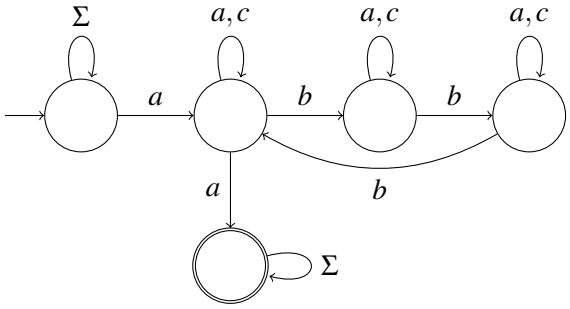
(b) Binary strings containing either a pair of 0s with two symbols in between, or a pair of 1s with two symbols in between (e.g. 0100 and 1110011 are fine, but 101 and 0001110 are not).

Solution:



(c) Strings over $\Sigma = \{a, b, c\}$ containing a pair of as where the number of bs in between is a multiple of 3 (including zero). For example, *caca*, *ababcbba*, and *abababa* are fine (in the last case the pair of as is the first and last a), but a, *aba*, and *abbbbaba* are not.

Solution:



7. Closure properties of regular languages

For each of the following statements, give a short argument for why it is true or false.

(a) For any infinite sequence of regular languages $L_1, L_2, \ldots$ (i.e. we have a regular language L_i for each $i \geq 0$), the language $L = \cup_{i \geq 0} L_i$ is regular.

Solution: This is false. If we let $L_i = \{0^i 1^i\}$, then each L_i is finite (containing only a single string) and so is regular. But $\cup_{i \geq 0} L_i = \{0^i 1^i \mid i \geq 0\}$, which we've proved is not regular.

(*Note:* This does not contradict the fact that regular languages are closed under union. We proved that if L and R are regular, so is $L \cup R$; repeating this shows that the union of any *finite* number of regular languages is regular, but it doesn't follow that the union of *infinitely-many* regular languages is regular.)

(b) If L is a regular language, then so is any subset $R \subseteq L$.

Solution: This is false. The language Σ^* is regular, and is a superset of every language over Σ. So if the claim were true, then every language would be regular, but we have proved that some languages are not regular.

(c) If X, Y, and Z are regular languages over an alphabet Σ, then so is $\{w \in \Sigma^* \mid w \text{ is in exactly two of } X, Y, \text{ and } Z\}$.

Solution: The set of strings over Σ which are in X and Y but not Z is $X \cap Y \cap \overline{Z}$ (recalling that $\overline{Z}$ is the complement $\Sigma^* - Z$). So the language of strings which are in exactly two of X, Y, and Z is $(X \cap Y \cap \overline{Z}) \cup (X \cap \overline{Y} \cap Z) \cup (\overline{X} \cap Y \cap Z)$. Since regular languages are closed under complement, intersection, and union, this language is regular.

(d) If L is a regular language, so is $\{w \in L \mid w \text{ contains every symbol of } \Sigma\}$.

Solution: The language R of strings which contain every symbol of Σ is regular: we can build a DFA with one state for every subset of Σ to remember which symbols have been seen so far (or simply recall that on HW 2 Problem 5c we built an NFA for the complement, namely the strings which do *not* contain every symbol of Σ, then apply closure under complement). The language we want is $L \cap R$, which is regular since regular languages are closed under intersection.

8. More algorithms for finite automata

Give short descriptions of algorithms for each of the following problems. In each case, try to make your algorithm as fast as possible, and state its asymptotic runtime in terms of the size of its input (for an automaton M, measure its size $|M|$ as the total number of transitions, i.e. the number of states times the size of the alphabet).

(*Note:* You should not need to look up any algorithms; all of these problems can be solved using variations of graph algorithms and/or algorithms we've discussed in class or on the homework.)

(a) Given DFAs A and B, decide if there is any $w \in \Sigma^*$ such that both $w \in L(A)$ and $w \in L(B)$.

Solution: Using the product construction we studied in HW 2 Problem 4, we can build a DFA D of size $O(|A| \cdot |B|)$ such that $L(D) = L(A) \cap L(B)$. Then we can apply the DFA emptiness algorithm to check if $L(D) = \emptyset$: if so, then there is no word in both $L(A)$ and $L(B)$; otherwise, there is.

Building the product automaton takes $O(|A||B|)$ time, and since the emptiness algorithm is linear, the total runtime is also $O(|A||B|)$.

(b) Given DFAs A and B, decide whether $L(A) \subseteq L(B)$.

Solution: Observe that $L(A) \subseteq L(B)$ if and only if $L(A) \cap \overline{L(B)} = \emptyset$ (a Venn diagram can help visualize this). So we can simply complement B, then apply the algorithm from part (a) to check if the intersection is empty. The complement takes $O(|B|)$ time, which is dominated by the runtime of the algorithm above, which is $O(|A||B|)$.

Alternate Solution: Observe that $L(A) \subseteq L(B)$ if and only if $L(A) - L(B) = \emptyset$. We can use the modified product construction from part (c) of HW 2 Problem 4 to compute a DFA D such that $L(D) = L(A) - L(B)$, then apply the emptiness algorithm to D. As in part (a) above, the total runtime will be $O(|A||B|)$.

(c) Given a DFA D, decide whether there is any $w \in L(D)$ containing every symbol of Σ.

Solution: As in part (d) of Problem 7 above, we can build a DFA R of size $2^{|\Sigma|}$ accepting all strings which contain every element of Σ. We can then apply the intersection algorithm from part (a) above to check if there is any $w \in L(D) \cap L(R)$. This will take time $O(2^{|\Sigma|} \cdot |D|)$, which is $O(2^{|D|} \cdot |D|)$ in the worst case.

(d) Given a DFA D, decide whether $L(D)$ contains any word of even length.

Solution: We can build a DFA E with 2 states accepting all words of even length. We can then apply the intersection algorithm from part (a) above to check if there is any $w \in L(D) \cap L(E)$. This will take time $O(|D|)$, since E has constant size.

(e) Given an NFA N and a word $w \in \Sigma^*$, decide whether *every* possible path for w in N ends in an accepting state.

Solution: We can run the NFA simulation algorithm to find the set S of all possible states that N can end up in on input w. If every state in S is accepting, then every path for w is an accepting path; otherwise, some path for w ends in a rejecting state and so is not an accepting path. The NFA simulation algorithm takes $O(s^2|w|)$ time for an NFA with s states, so the total time required will be $O(|N|^2|w|)$.

9. Reading regular expressions

For each of the following pairs of regular expressions over $\Sigma = \{a, b, c\}$, state whether their languages are equal ($=$), one is a proper subset of the other ($\subset$), or if they are incomparable. If $L(R_1) \subset L(R_2)$, give an example of a string in $L(R_2)$ that is not in $L(R_1)$, or vice versa if $L(R_2) \subset L(R_1)$; if the languages are incomparable, give an example string from each language that is not in the other.

(a) $R_1 = (a^*b^*)^*$
$R_2 = (a^*b^*a^*)^*$
Solution: $L(R_1) = L(R_2)$, since both expressions match all strings over $\{a, b\}$.

(b) $R_1 = aa^*$
$R_2 = \emptyset^*$
Solution: R_1 matches 1 or more as, while R_2 matches only the empty string (the concatenation of zero or more words from the language containing no words = the concatenation of zero words = ε), so the languages are incomparable.

(c) $R_1 = (a \mid b \mid c \mid \varepsilon)^*$
$R_2 = (a \mid b \mid c \mid \emptyset)^*$
Solution: $L(R_1) = L(R_2)$, since both expressions match all strings over $\{a, b, c\}$.

(d) $R_1 = (a \mid b)(b \mid c)$
$R_2 = \Sigma$
Solution: R_1 matches the strings $\{ab, ac, bb, bc\}$, while R_2 matches the strings $\{a, b, c\}$, so the languages are incomparable.

(e) $R_1 = (a \mid b \mid \varepsilon)(a \mid b \mid \varepsilon)$
$R_2 = aa \mid ab \mid ba \mid bb$
Solution: R_1 matches the 4 strings listed in R_2, but also matches a, b, and ε, so $L(R_2) \subset L(R_1)$.

10. Writing regular expressions

Write regular expressions for each of the following languages over $\Sigma = \{a, b, c\}$:

(a) Strings that begin with two of the same symbol (e.g. *aabc*).

Solution: $aa\Sigma^* \mid bb\Sigma^* \mid cc\Sigma^*$, or equivalently $(aa \mid bb \mid cc)\Sigma^*$

(*Note:* We don't allow exponents in our simple definition of regular expressions, but if we did, the expression $(a \mid b \mid c)^2\ \Sigma^*$ would *not* work: it's equivalent to $(a \mid b \mid c)(a \mid b \mid c)\Sigma^*$, which doesn't force the first two symbols to be the same.)

(b) Strings containing two *a*s which have the substring *cc* somewhere in between (e.g. *aabccac* is fine but *acbca* is not).

Solution: $\Sigma^* a \Sigma^* cc \Sigma^* a \Sigma^*$

(c) Strings where the number of *b*s is a multiple of 3 (including zero).

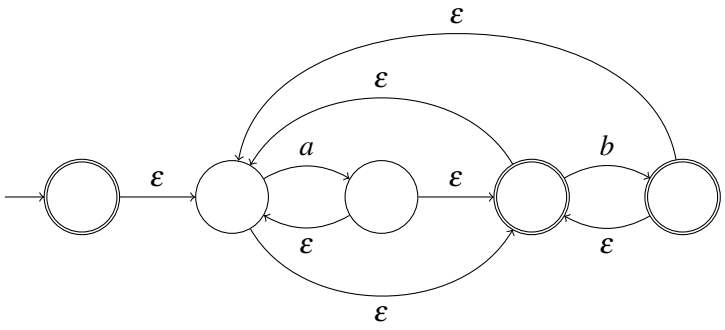
Solution: $(a \mid c)^* (b\ (a \mid c)^*\ b\ (a \mid c)^*\ b\ (a \mid c)^*)^* (a \mid c)^*$

11. Converting regular expressions to NFAs

Convert the following regular expressions over the alphabet $\Sigma = \{a, b, c\}$ to NFAs:

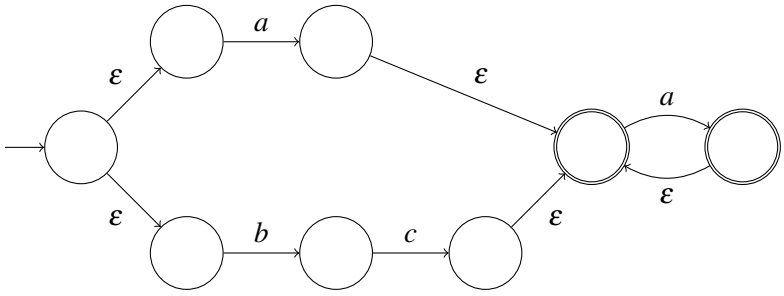
(a) $(a^*b^*)^*$

Solution: Here and in the subsequent solutions, I will follow the procedure we discussed in class, which doesn't necessarily yield the smallest possible NFA but works for all regular expressions. On the homework and exams, any equivalent NFA is fine.



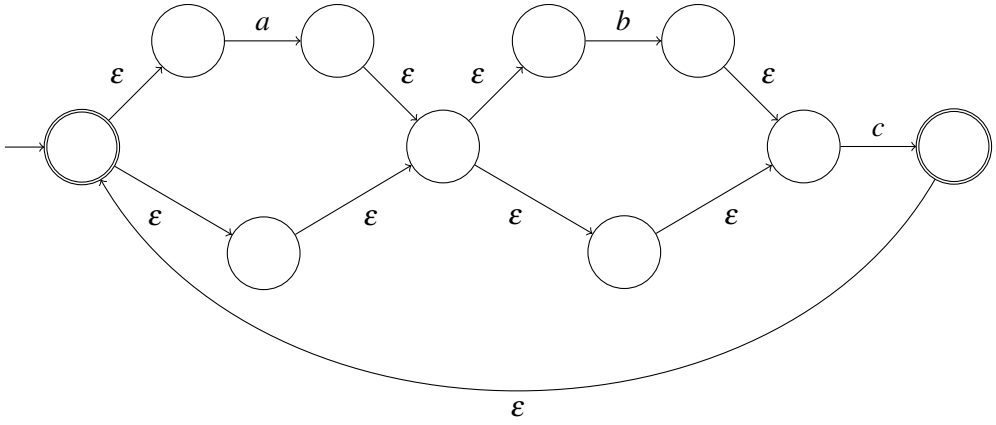
(b) $(a \mid bc)a^*$

Solution:



(c) $((a \mid \varepsilon)(b \mid \varepsilon)c)^*$

Solution:



12. Non-regular languages

Prove that the following languages are not regular. If you like, you may refer to other languages shown to be non-regular in class or on the homework.

(a) $\{0^n 1^m 0^n \mid n, m \geq 0\}$

Solution: We will use the pumping lemma. Given $n \geq 1$, let $w = 0^n 10^n \in L$. For any decomposition $w = xyz$ with $|xy| \leq n$ and $y \neq \varepsilon$, both x and y consist only of 0s, with y containing at least one. So picking $k = 2$, the string xy^2z is of the form $0^m 10^n$ with $m > n$, so xy^2z is not in L, which would contradict the pumping lemma if L were regular.

(b) All binary strings which are not palindromes (i.e., which are not equal to their reversal).

Solution: Let L be the language of binary strings which *are* palindromes. We'll prove that L is not regular using the pumping lemma. Given $n \geq 1$, let $w = 0^n 10^n \in L$. For any decomposition $w = xyz$ with $|xy| \leq n$ and $y \neq \varepsilon$, both x and y consist only of 0s, with y containing at least one. So picking $k = 2$, the string xy^2z is of the form $0^m 10^n$ with $m > n$, so it is not a palindrome and therefore not in L. This would contradict the pumping lemma if L were regular, so L is not regular. Since regular languages are closed under complement, the language of binary strings which are *not* palindromes is not regular either.

Alternate Solution: If the language of (binary) non-palindromes were regular, the language of palindromes would be too since regular languages are closed under complement. Since the language of even-length strings is regular, and regular languages are closed under intersection, the language L of even-length palindromes would be regular as well. But if a palindrome has even length, it is of the form ww^R for some word $w \in \{0,1\}^*$, and we proved in HW 3 Problem 4b that the language of such words is not regular.

(c) All binary strings with an unequal number of 0s and 1s.

Solution: We'll prove that the language L of binary strings with an *equal* number of 0s and 1s is not regular, using the pumping lemma — since regular languages are closed under complement, the desired result will then follow. The argument is exactly the same as for the language $\{0^n 1^n \mid n \geq 0\}$ we looked at in class: given $n \geq 1$, let $w = 0^n 1^n \in L$. For any decomposition $w = xyz$ with $|xy| \leq n$ and $y \neq \varepsilon$, both x and y consist only of 0s, with y containing at least one. So picking $k = 2$, the string xy^2z has strictly more 0s than 1s and is not in L. This would contradict the pumping lemma if L were regular, so L is not regular.

13. Working with a CFG (including parsing using the CYK algorithm)
Consider the CFG G with terminal symbols $\Sigma = \{a, b, c\}$, nonterminal symbols $V = \{S, T\}$, start symbol S, and the rules

$$S \to aSbS \mid \varepsilon \mid T$$
$$T \to c \mid \varepsilon$$

(a) For each of the following strings, state whether it can be derived from G, giving a leftmost derivation if so: *ab*, *aab*, *aacbcb*, *acbab*.

Solution:

ab: This can be derived by the leftmost derivation $S \Rightarrow aSbS \Rightarrow abS \Rightarrow ab$.

aab: This cannot be derived, since *a*s and *b*s are always created together and so we must have equal numbers of them.

aacbcb: This can be derived by $S \Rightarrow aSbS \Rightarrow aaSbSbS \Rightarrow aaTbSbS \Rightarrow aacbSbS \Rightarrow aacbTbS \Rightarrow aacbcbS \Rightarrow aacbcb$.

acbab: This can be derived by $S \Rightarrow aSbS \Rightarrow aTbS \Rightarrow acbS \Rightarrow acbaSbS \Rightarrow acbabS \Rightarrow acbab$.

(b) Is G ambiguous? If so, draw two different parse trees for some string in $L(G)$; otherwise argue why every string has a unique leftmost derivation.

Solution: It is ambiguous. The string *ab* has another leftmost derivation besides the one we gave above, namely $S \Rightarrow aSbS \Rightarrow aTbS \Rightarrow abS \Rightarrow ab$.

(c) Here is an equivalent grammar G' in Chomsky normal form:

$$S \to XY \mid \varepsilon \mid c$$
$$X \to AZ \mid a$$
$$Y \to BZ \mid b$$
$$Z \to XY \mid c$$
$$A \to a$$
$$B \to b$$

Use the CYK algorithm to parse the string $w = aacbbc$ according to G'. Show the resulting table, and explain how it shows whether or not $w \in L(G')$.

Solution: For the first row of the table, we need to check which nonterminals can derive each of the 6 substrings of length $\ell = 1$, using a rule of the form $V \to t$. Checking the grammar, we find that X and A can derive a, Y and B can derive b, and S and Z can derive c. So the table so far looks like:

	a	a	c	b	b	c
$\ell = 1$	X, A	X, A	S, Z	Y, B	Y, B	S, Z

Moving on to $\ell = 2$, we are now in the recursive case of the algorithm. For each substring, we have to consider all possible ways of splitting it into two nonempty parts, then look at the earlier rows of the table to see how those parts can be derived. For $\ell = 2$, the only way to split is into two strings of length 1. For example, the substring starting at position 2 is *ac*, which we split into *a* (the substring of length 1 starting at position 2) and *c* (the substring of length 1 starting at position 3). According to the first row of the table, *a* can be derived from X and A, while *c* can be derived from S and Z. So we need to look for all rules whose right-hand side is XS, XZ, AS, or AZ. There is only one rule that works, namely $X \to AZ$, so the table at row 2, column 2 will only store X.

As another example, consider the substring of length 2 starting at position 1, namely *aa*. We split it into *a* and *a*, and check the first row of the table to find that both parts can be derived from X and A. So we need to look for rules whose right-hand side is XX, XA, AX or AA. There are no such rules, so it isn't possible to derive *aa* from any of the variables, and we put $\emptyset$ into the table at row 2, column 1. Filling out the rest of the row in the same way, we get:

	a	*a*	*c*	*b*	*b*	*c*
$\ell = 1$	X,A	X,A	S,Z	Y,B	Y,B	S,Z
$\ell = 2$	$\emptyset$	X	$\emptyset$	$\emptyset$	Y	

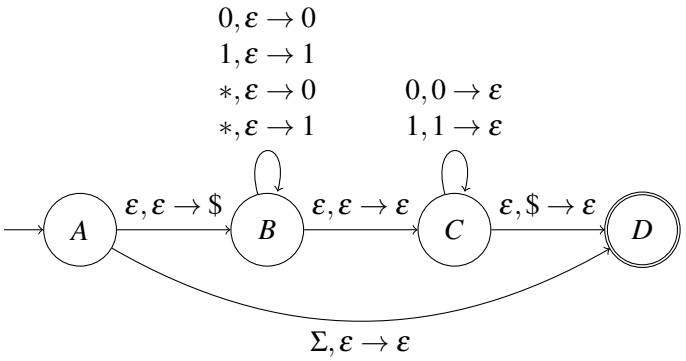
For substrings of length $\ell = 3$, there are two possible splits: for example, the length-3 substring starting at position 2, *acb*, can be split into *a* and *cb* or *ac* and *b*. To see how *a* can be derived, we check row 1, column 2: X or A. For *cb*, we check row 2, column 3 (since it's the substring of length 2 starting at position 3): it's the empty set, so there's no way to derive that string, and this split won't work. Let's look at the other possible split, into *ac* and *b*: for *ac* we check row 2, column 2, and find X; for *b* we check row 1, column 3 and find Y and B. So we need to see if there are any rules with right-hand side XY or XB. There are two such rules, $S \to XY$ and $Z \to XY$, so we put S and Z into row 3, column 2. Continuing in the same way to fill out the entire table, we get:

	a	*a*	*c*	*b*	*b*	*c*
$\ell = 1$	X,A	X,A	S,Z	Y,B	Y,B	S,Z
$\ell = 2$	$\emptyset$	X	$\emptyset$	$\emptyset$	Y	
$\ell = 3$	$\emptyset$	S,Z	$\emptyset$	$\emptyset$		
$\ell = 4$	X	$\emptyset$	$\emptyset$			
$\ell = 5$	S,Z	$\emptyset$				
$\ell = 6$	S,Z					

Finally, to check whether $w \in L(G')$, we look at the bottommost cell of the table, which lists all variables which can derive the substring of length $\ell = 6$ starting at position 1: w itself. The cell contains S, which is the start variable, so w can be derived from G' and $w \in L(G')$.

14. Working with a PDA

Consider the following PDA with input alphabet $\Sigma = \{0, 1, *\}$ and stack alphabet $\Gamma = \{0, 1\}$:



What is the language of this PDA?

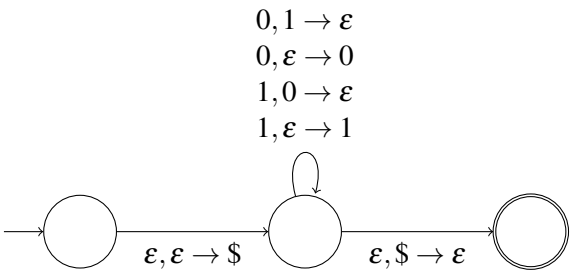
Solution: There are two ways we can get to the accepting state D: the direct transition reading in any single symbol from Σ, or passing through the two intermediate states B and C. In the latter case, notice the first and last transition push a \$ onto the stack and pop it off respectively: this is our usual trick for detecting when the stack is empty. So we will only be able to get to D along this path if the symbols we push onto the stack at state B get popped off at state C. At state B, whenever we read a 0 or a 1 we push it onto the stack, whereas when we read a $*$ we can nondeterministically push either a 0 or a 1 onto the stack. At any point, we can nondeterministically decide to go to state C without reading any symbols or modifying the stack. We then can pop 0s and 1s off the stack by reading the corresponding input symbols. Therefore, the only way to reach D through B and C is if the input string can be divided into two equally-long parts $x \in \Sigma$ and $y \in \{0, 1\}$, where y is the reverse of x except that any $*$ in x can correspond to either a 0 or 1 in y: that is, the input is an even-length palindrome where the first half can contain wildcard $*$ symbols that can correspond to either 0 or 1 in the second half. So the language of the PDA consists of all such even-length palindromes-with-wildcards, together with strings of length 1.

(*Note:* I originally intended this PDA to accept all palindromes-with-wildcards, but I made two mistakes: can you fix them? You need to move the transition from A to D somewhere else to allow *odd*-length palindromes (not just palindromes of length 1), and change the self-loop at C to allow wildcards in the second half of the string.)

15. Designing a PDA

Let L be the language of binary strings with an equal number of 0s and 1s (so for example $\varepsilon, 01, 1010 \in L$ but $0, 11, 1000 \notin L$). Draw (as a graph) a PDA recognizing L.

Solution: In a similar way to the PDA we saw in class for 0^n1^n, we can use the stack to remember how many 0s we've seen. The difference now is that the 0s and 1s can come in any order; so we'll instead only remember how many *extra* 0s or 1s we've seen, i.e. how many 0s or 1s do not match earlier opposite digits. We can do this as follows: whenever we see a 0, pop a 1 off the stack if possible; otherwise push the 0 on the stack (and similarly whenever we see a 1). Combining this with the \$ trick to detect an empty stack, we get the following PDA:



For any input string, it is always possible to take a path through this PDA where if we have read n 0s and m 1s so far, then the stack contains $n-m$ 0s if $n \geq m$ and $m-n$ 1s if $m \geq n$. So if the number of 0s and 1s is equal, then there is a path where we end up with an empty stack after reading the entire string (except for the \$ marker), and can then move to the accepting state. Conversely, since we cannot reach the accepting state without popping all 0s and 1s off the stack, and popping each symbol requires reading the opposite symbol, this PDA only accepts strings with equal numbers of each symbol.