KØBENHAVNS
UNIVERSITET

Diskret Matematik og Formelle Sprog: Exam April 14, 2021
Problems and (Sketches of) Solutions

Please note that the main purpose of this document is to explain what the correct solutions are and how to arrive at them. Thus, while the text below is certainly intended to provide good examples of how to solve problems and reason about solutions, these examples do not necessarily specify exactly how the handed-in exams were expected to look like—for such examples, see the course notes published on Absalon, which contain many worked-out exercises. Instead, the solution sketches below are sometimes more detailed than would have been expected from the solutions handed in, and sometimes less detailed. An effort has been made to indicate clearly when this is the case, though.

1 (80 p) The DMFS main instructor is so excited by the simplicity of insertion sort and the efficiency of merge sort that he wants to get the best of both worlds. To this end, he has designed a sorting algorithm `MERGE-INSERTION-SORT` for arrays `A` that can be described in pseudocode as follows (where we assume that arrays are indexed from 1 to n and contain elements that can be compared using the operator ">"):

```
MERGE-INSERTION-SORT (A)
    if (size (A) > 1)
        mid := floor ((1 + size (A)) / 2)
        L   := new array of size mid
        R   := new array of size size (A) - mid
        for (i := 1 upto mid)
            L[i] := A[i]
        for (i := mid+1 upto size (A))
            R[i-mid] := A[i]
        MERGE-INSERTION-SORT (L)
        MERGE-INSERTION-SORT (R)
        A := MERGE-INSERT (L, R)
```

The subroutine `MERGE-INSERT` used above is as follows:

```
MERGE-INSERT (L, R)
    s := size (L)
    t := size (R)
    M := new array of size s + t
    for (i := 1 upto s)
        M[i] := L[i]
    for (i := s+1 upto s+t)
        j    := i
        M[j] := R[j-s]
```

```
        while (j > 1 and M[j-1] > M[j])
            tmp    := M[j-1]
            M[j-1] := M[j]
            M[j]   := tmp
            j      := j-1
    return M
```

1a (30 p) Argue that `MERGE-INSERTION-SORT` is in fact a valid algorithm in that it will terminate on any input array `A` and return some result. Then explain what `MERGE-INSERTION-SORT` does to `A`. In particular, does it actually sort the array correctly or not? Argue why the algorithm is correct sorting algorithm, or give a counter-example showing that it is not.

Solution: The main procedure `MERGE-INSERTION-SORT` is identical to the merge sort method that we saw in class. Given an array `A` of size larger than 1, the algorithm splits `A` into two arrays of strictly smaller size, makes two recursive calls, and then combines the results using the method `MERGE-INSERT`. If `MERGE-INSERT` terminates, then the algorithm will terminate.

Just as for the merge sort algorithm that we saw in class, we can now argue by induction that `MERGE-INSERTION-SORT` is a correct sorting algorithm. The base case is for arrays of size 1, which are clearly handled correctly by the algorithm.

For the induction step, suppose that `MERGE-INSERTION-SORT` is correct on arrays of size less than n and consider an array `A` of size n. This array is split into two strictly smaller arrays `L` and `R`, and by our induction hypothesis `MERGE-INSERTION-SORT` will sort these two arrays correctly. What `MERGE-INSERT` does it to copy the first array `L` into a new array `M`. After this, the first `size (L)` elements of `M` are sorted correctly. The nested for and while loops then perform insertion sort of the elements of `R`, placing each element in turn in the right position in `M`. The pseudocode for this is again identical to what we saw in class. Hence, `MERGE-INSERT` will return a sorted array, and this concludes the inductive step in our argument. It follows by the induction principle that `MERGE-INSERTION-SORT` is a correct sorting algorithm.

1b (50 p) Analyze the time complexity of `MERGE-INSERTION-SORT`. Give the best asymptotic upper bound that you can find. Provide a lower bound by constructing a family of instances that will force the algorithm to run as asymptotically slowly as possible. Can you get your upper and lower bounds to match asymptotically? (Please make sure to include all necessary details in your analysis, but note that partial answers will also give credits.)

Solution: First note that any correct time complexity analysis has to take into consideration that this is a *recursive* algorithm. It is *not* sufficient to just observe that parts of the algorithm are similar to merge sort and insertion sort, and that the time complexity should therefore be the sum or product or whatever of the time complexities of these sorting methods. That is very far from a correct, complete analysis.

A second observation is that, just as in the CLRS textbook, we can assume without that the size of the array is a power of 2. The general case can easily be reduced to this case, basically by running the analysis below for the smallest power of 2 that is larger than n. (It is also possible to do all of the analysis below super-formally by rounding up to integers for all divisions, but since it does not really change the underlying math in any significant way this is actually a step that is usually ignored even in scientific research papers, and so it should arguably be OK to ignore it also in an introductory CS course.)

If `size (A)` is $n = 2^m$, then there will be $m = \log n$ levels of recursion, since the array is divided into two equal-size pieces for each recursive call (recall that logarithms are always

base 2 unless specified otherwise). At every level we perform one or more insertion sorts on a total of n elements. A first rough analysis therefore suggests that the time complexity should be something like $O(n^2 \log n)$ (and arguing this correctly already gives some points).

This analysis is overly pessimistic, however. To see this, suppose that insertion sort can sort n elements in time at most Kn^2 for some constant K. Then we obtain the following:

- At the top level of recursion, we sort one array with n elements in time at most Kn^2.
- At the second level, we sort two arrays containing $n/2$ elements each in total time at most $2 \cdot K(n/2)^2 = Kn^2/2$.
- At the third level, we sort four arrays containing $n/4$ elements each in total time at most $4 \cdot K(n/4)^2 = Kn^2/4$, et cetera.

In addition to the sorting, we have at most a linear amount of work at each level of recursion. Therefore, the total running time should be something like $(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots)Kn^2+O(n \log n) \leq 2Kn^2 + O(n \log n) = O(n^2)$.

Arguing as above will already give you quite a lot of points, but since the argument is a bit delicate we would like you to a bit more formal for a full score. More formally, we can first observe that `MERGE-INSERT` on arrays `L` and `R` with `size (L) + size (R)` $= n$ takes time at most $K_1 n^2$ for some large enough constant K_1 (since we have two nested loops that run over $O(n)$ elements, and there is a constant amount of work done in the innermost loop).

Now suppose that `MERGE-INSERTION-SORT` takes time

$$T(m) \leq Km^2 \tag{1}$$

for arrays of size $m < n$, where we note that for small lists we can always make this true by choosing the constant K large enough. Then by analysing the code, we see that the time $T(n)$ needed for `MERGE-INSERTION-SORT` on an array of size n is the time of two `MERGE-INSERTION-SORT` calls on arrays of half the size plus one `MERGE-INSERT` call plus some linear amount of work $K_2 n$ for creating and copying arrays and setting up the function calls. The total time complexity $T(n)$ is therefore

$$T(n) \leq 2 \cdot T(n/2) + K_1 n^2 + K_2 n \leq 2 \cdot T(n/2) + K_3 n^2 \tag{2}$$

if we choose, say, $K_3 = K_1 + K_2$. Plugging our induction hypothesis (1) into (2) yields

$$T(n) \leq 2 \cdot K(n/2)^2 + K_3 n^2 \leq (K/2 + K_3)n^2 \leq Kn^2 \tag{3}$$

if we choose $K \geq 2 \cdot K_3$. Since we are free to choose all constants as we like, and we can choose them in order K_1, K_2, K_3, K to satisfy all of the constraints above, this concludes our inductive step. The upper bound on the running time now follows by the induction principle.

To obtain worst-case instances that lead to quadratic running time, we can simply take arrays sorted in reverse order. Then every `MERGE-INSERT` call will have to shift all elements in `R` to before the elements in `L`, which takes time quadratic in `size (L) + size (R)`. Thinking a bit more, we can realize that it is even sufficient to just pick an array where `A[1]` to `A[n/2]` are the largest elements in the array sorted in any order and `A[n/2+1]` to `A[n]` are the smallest elements sorted in any order. Then the final call at the top level of recursion will take quadratic time. As a side note, we can also observe that even the array is correctly sorted from the start, then `MERGE-INSERTION-SORT` will require time $\Theta(n \log n)$ to verify this. So we are really getting the *worst* of both worlds, not the best. What a terrible idea the instructor had...

2 (40 p) Consider the sequence $(a_n)_{n\in\mathbb{N}^+}$ defined by

$$a_i = \begin{cases} 1 & \text{if } i = 1; \\ 3 & \text{if } i = 2; \\ 2a_{i-1} - a_{i-2} & \text{if } i > 2. \end{cases}$$

Find a closed expression (i.e., a formula) for a_n and prove that your formula is correct.

Solution: By using the recursive expression for a_i we find that $a_3 = 2\cdot 3 - 1 = 5$, $a_4 = 2\cdot 5 - 3 = 7$, and $a_5 = 2 \cdot 7 - 5 = 9$, and the hypothesis naturally arises that $a_i = 2i - 1$. Let us prove this by using (strong) mathematical induction.

The base case is clear—we have already done several of them above. Our induction hypothesis is that $a_i = 2i - 1$ for all $i < n$. For $i = n$ we then obtain, using first the formula $a_i = 2a_{i-1} - a_{i-2}$ and then the induction hypothesis $a_i = 2i - 1$, that

$$a_n = 2a_{n-1} - a_{n-2} = 2(2(n-1) - 1) - (2(n-2) - 1) = 4n - 6 - (2n - 5) = 2n - 1 \quad (4)$$

as desired. The equality follows by the induction principle.

3 (50 p) Computer science exams can only be held virtually, but the Copenhagen Tivoli Gardens amusement park is apparently about to open for physical visits by the general public. This means that students frustrated by Zoom lectures and *Digital eksamen* exams will soon be able to head to the *bumper cars* (*radiobiler*) and let go of some steam by trying to bump into as many other cars (or as few cars) as possible.

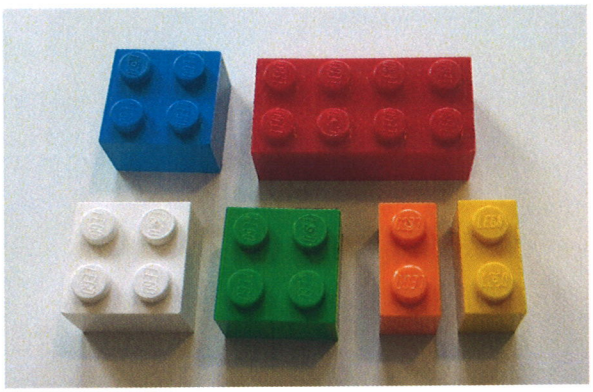
Suppose that there are $n \geq 2$ bumper cars sharing an area of $m > n^2/2$ square meters.[1] An amazing mathematical fact is that after every bumper car round has finished, there are two different bumper cars that have bumped with exactly the same number of other (distinct) cars. Prove this!

Solution: We start by making three quick observations:

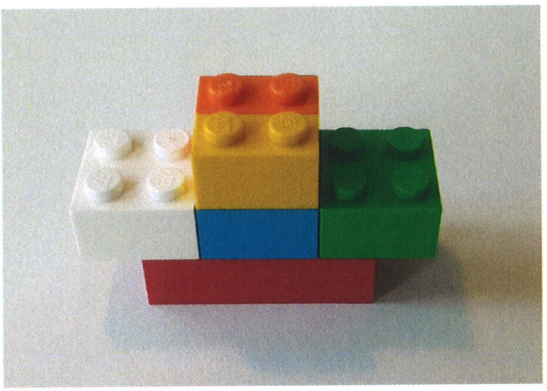
1. The area of course has nothing to do with this, and can just be ignored.
2. "Bumping" is a symmetric relation, in that bumper car A can only have a bump with bumper car B if car B also had a bump with car A. (This is clear from context, since otherwise the claim in the problem statement would be obviously false for two bumper cars.)
3. Each one of the n bumper cars can bump with at most $n - 1$ other cars. If one car bumps with all other cars, then there is no car without bumps, and if there is a car without bumps, then no car can bump with all other cars. (Here we are using the symmetry of the bumping relation.)

Now we can think of each car as being a pigeon (n of them) and each pigeonhole as being labelled by a number of bumps ($n - 1$ possibilities, as discussed above, since we cannot have 0 bumps and $n - 1$ bumps at the same time). Clearly, by the pigeonhole principle there have to be two bumper cars that have the same number of bumps.

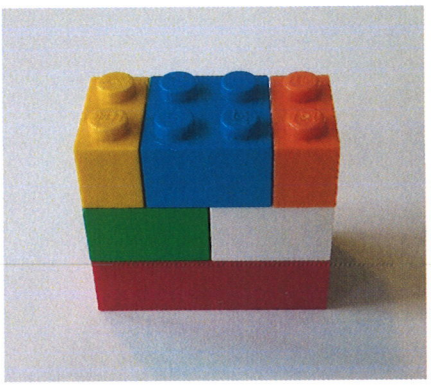
[1] In reality, there are 18 bumper cars in the Copenhagen Tivoli Gardens sharing an area of 280 square meters, but this is not too relevant for this problem.



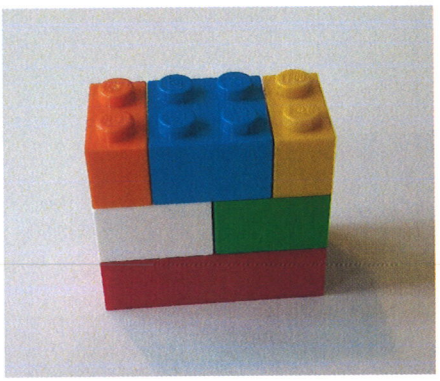
(a) A set of lego pieces.



(b) Invalid construction (not a cuboid).



(c) Valid construction of 3-layer cuboid.



(d) Symmetric version of the same cuboid.

Figure 1: Lego pieces and constructions (and non-constructions) of cuboids for Problem 4.

4 (60 p) Another way in which Danish computer science students have been able to entertain themselves—already during lock-down—is to play with legos. In order to make this kind of activity more relevant from the point of view of discrete mathematics, rumour has it that the students often consider different sets of lego pieces (like the one in Figure 1a) and investigate in how many different ways cuboids (rectangular parallelepipeds) can be built from such pieces.

Inspired by this, let us consider the lego pieces in Figure 1a, which as we observe all have different colours. In this particular problem we want to build cuboids that are three layers high. The pieces should be put together in the standard lego way, with the lego studs pointing upwards and being fitted into the holes of any pieces above. All pieces should be used. Constructions like the one in Figure 1b are not valid—this particular construction does have three layers, and the studs point upwards and are fitted into the holes above when possible, but the resulting geometric shape is not a cuboid.

Two cuboids are considered to be the same if they can be rotated so that they become similar. Thus, the constructions in Figures 1c and 1d are both valid, but are considered to be the same cuboid, since rotating Figure 1c yields Figure 1d.

How many *different* cuboids can you build with the lego pieces in Figure 1a satisfying the restrictions described above? Please make sure to explain clearly how you reason to reach your answer.

Solution: We first note, for the record, that the fact that all lego bricks have to be used was not stated quite so explicitly in the original problem statement, but it was intended to be clear from context and was also explained at an early stage during the exam.

Let us start by computing how many cuboids can be built without considering rotational symmetry (so that, for the moment, we consider Figure 1c and Figure 1d to be different cuboids). Once we have this number, it is clear that we should divide by 2 to get the number of constructions as asked for in the problem statement. This is so since it is not possible to build a cuboid in accordance with the rules that is symmetric under rotation. Hence, every cuboid can be paired up with exactly one other different cuboid so that the two are obtained from each other by rotation.

Counting the number of studs, there are:

- one 4×2 brick;
- three 2×2 bricks;
- two 2×1 bricks.

Let us analyse how the cuboids can be constructed.

Once the 4×2 brick is placed somewhere, all the other pieces have to be used in other layers, since otherwise we cannot get a cuboid. The layer with a single 4×2 brick can be chosen in 3 ways.

Then there will be a layer with two 2×2 bricks. Once we have decided on the 4×2 layer, the position for the layer with two 2×2 bricks can be chosen in 2 ways. Since there is a total of three 2×2 bricks, the left 2×2 brick can be chosen in 3 ways, after which the right 2×2 brick can be chosen in 2 ways.

So far, we have built two layers in our cuboid in a total of $3 \cdot (2 \cdot 3 \cdot 2) = 36$ ways.

For the third layer, we obtain a case analysis as follows, considering the lego bricks from left to right:

1. A 2×1 brick, then a 2×2 brick, and finally a 2×1 brick (as in Figure 1c and Figure 1d). Given the choices already made, there are 2 choices for this alternative, namely in which order the 2×1 bricks appear. (Note that there is only one 2×2 brick left by now, so there is no flexibility there.)
2. As in case 1 above, but with both 2×1 bricks appearing farthest to the left in the same orientation as in Figure 1c. There are again 2 choices for this alternative, depending on in which order the 2×1 bricks appear.
3. As in case 2, but both 2×1 bricks appearing rotated as in Figure 1b. This gives 2 more choices, depending on in which order the 2×1 bricks appear.
4. As in case 2, but with the 2×1 bricks appearing farthest to the right: 2 more choices.
5. Finally, as in case 3 but with the 2×1 bricks appearing farthest to the right: 2 more choices.

We see that we get a total of 10 ways of constructing the third layer, yielding a total number of 360 constructions. Since we do not distinguish between cuboids that are rotationally symmetric, however, we should divide by 2 to get the final answer that there are $360/2 = 180$ different cuboids that can be constructed.

5 (60 p; 20 p per formula) Decide for each of the propositional logic formulas below whether it is a tautology or a contradiction. If neither of these cases apply, then present one satisfying and one falsifying assignment for the formula. For a full score, please present truth tables for all the subformulas analogously to how we did it in class, and also try to *explain* why your answers are correct by interpreting what the formulas mean. Good explanations can compensate for minor slips in the truth tables.

(Note that logical not $\neg$ is assumed to bind harder than the binary connectives, but other than that all formulas are fully parenthesized for clarity. We write $\rightarrow$ for logical implication and $\leftrightarrow$ for equivalence.)

5a $((p \rightarrow q) \wedge (q \rightarrow r)) \leftrightarrow (p \rightarrow r)$

Solution: Let us first fill in the truth table (using $\top$ for *true* and $\bot$ for *false*). We consider all possible combinations of truth value assignments to p, q, and r, and for all assignments we evaluate all subformulas $(p \rightarrow q)$, $(q \rightarrow r)$, $(p \rightarrow q) \wedge (q \rightarrow r)$, and $(p \rightarrow r)$. This yields the following truth table:

p	q	r	$((p \rightarrow q)$	$\wedge$	$(q \rightarrow r))$	$\leftrightarrow$	$(p \rightarrow r)$
$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\bot$	$\top$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\bot$
$\top$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$	$\top$
$\top$	$\top$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

We see that assigning, e.g., $p = q = r = \bot$ makes the formula evaluate to true, but flipping q to true makes the formula evaluate to false.

What the formula claims is that p implying q and q implying r is equivalent to that p should imply r. In one direction this is true—if p implies q and q implies r, then it is easy to verify that it also holds that p implies r. However, the other direction is false for the simple reason that the fact that p implies r tells us absolutely nothing about how these variables are related to q.

5b $(p \wedge (\neg q \vee r)) \rightarrow (q \rightarrow (p \wedge r))$

Solution: The truth table for this formula is as below.

p	q	r	$(p$	$\wedge$	$(\neg q$	$\vee$	$r))$	$\rightarrow$	$(q$	$\rightarrow$	$(p \wedge r))$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$
$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\bot$
$\top$	$\bot$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\bot$	$\top$	$\bot$
$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
$\top$	$\top$	$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$

As we can see from the fourth-to-last column (under the main implication of the formula), this is a tautology.

To explain why this is so, we can reason as follows. Suppose that p is true that $\neg q \vee r$ is also true. We need to argue that in this case it is true that q implies $p \wedge r$. This is what the formula claims, and so if we can prove that this claim always holds, then the formula is a tautology.

Since $\neg q \vee r$ is true, it holds that either q is false or r is true (or both). If q is false, then the implication $q \to (p \wedge r)$ holds since falsity implies anything (by the truth table for the connective $\to$ that we learned in class). And if instead r is true, then the conjunction $p \wedge r$ is true, and so the implication $q \to (p \wedge r)$ is also true since truth is implied by anything (according to the same truth table). We have now explained why the formula is a tautology.

5c $((p \to q) \vee (r \to s)) \to ((p \vee r) \to (q \vee s))$

Solution: We take the liberty here of skipping the truth table (since we are only providing *sketches* of solutions, after all) and instead focus on explaining what is going on.

The answer here is that this formula is neither a tautology nor contradictory.

It is straightforward to verify that the formula evaluates to true if all variables are assigned false, because then the premise of the outermost implication is false, and falsity implies anything.

The reason the formula is not always true is, loosely speaking, that the fact that p implies q or that r implies s does not say much about what the disjunction of p and q implies or does not imply. To see this, let us assign p to true and all other variables to false. Then $p \to q$ is false but $r \to s$ is true, so the disjunction $(p \to q) \vee (r \to s)$, which is the premise of the outermost implication, is true. However, since $p \vee r$ is true (because of p) but $q \vee s$ is false, the implication $(p \vee r) \to (q \vee s)$ is false. We see that the outermost implication as a true premise but a false conclusion, and so it is false.

6 (70 p) In this problem we focus on relations. Suppose that $A = \{e_0, e_1, \ldots, e_5\}$ is a set of 6 elements and consider the relation R on A represented by the matrix

$$M_R = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(where element e_i corresponds to row and column $i + 1$).

6a (20 p) Let us write S to denote the symmetric closure of the relation R. What is the matrix representation of S? Can you explain in words what the relation S is by describing how it can be interpreted?

Solution: Firstly, by studying the matrix M_R we can conclude that a and b are related by R—which we have learned to denote by aRb, or $(a, b) \in R$—precisely when $b \equiv a + 2 \pmod 6$. Taking the symmetric closure means that a and b can switch places, so we get that $(a, b) \in S$ if $b \equiv a \pm 2 \pmod 6$, and

$$M_S = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

is the matrix representation of this relation.

We can also compute directly on the matrix and obtain M_S as the coordinate-wise Boolean or of M_R and $(M_R)^\top$.

6b (20 p) Now let T be the transitive closure of the relation S. What is the matrix representation of T? Can you explain in words what the relation T is by describing how it can be interpreted?

Solution: As just noted, we have $(a, b) \in S$ if $b \equiv a \pm 2 \pmod 6$. Now, if for a, b, c it holds that $b \equiv a \pm 2 \pmod 6$ and $c \equiv b \pm 2 \pmod 6$, then c and a differ by 0, 2, or 4 (mod 6). Or, in other words c and a are both odd or both even. It is not hard to see that this is the transitive closure—nothing more fun happens, and so this is what the relation T is. We have $(a, b) \in T$ if a and b are both odd or both even. The relation T can be represented as

$$M_T = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

in matrix form.

6c (30 p) Suppose that we instead let T' be the transitive closure of the relation R, and then let S' be the symmetric closure of T'. Are S' and T the same relation? If they are not the same, show some way in which they differ. If they are the same, is it true that S' and T constructed in this way from some relation R on a set A will always be the same? Please make sure to motivate your answers clearly.

Solution: Applying the same kind of reasoning as in Problem 6b, we get that if $b \equiv a + 2 \pmod 6$ and $c \equiv b + 2 \pmod 6$, then $c \equiv a \pm 2 \pmod 6$, and one more application of transitivity gives us the relation T. Since we already argued that T is closed under transitivity, we have $T' = T$. This relation is also symmetric, so taking the symmetric closure does not change it. Hence, we indeed have $S' = T' = T$.

This does not hold in general, however. Let $A = \{1, 2\}$ and let R be the relation containing only the pair $(1, 2)$. Then taking the transitive closure does not affect anything, so the relation stays the same, and taking the symmetric closure of the transitive closure yields the relation $\{(1, 2), (2, 1)\}$ with two pairs. However, if we take the symmetric closure first, so that we get $\{(1, 2), (2, 1)\}$, then the transitive closure will also relate 1 and 2 with themselves, so that the final relation after symmetric closure followed by transitive closure is $\{(1, 1), (1, 2)(2, 1), (2, 2)\}$. Expressed in in slightly more fancy language, we have now shown that the two operations of symmetric closure and transitive closure do not commute.

7 (100 p) Assume that we are given the directed graph in Figure 2. The graph is given to us in adjacency list representation, with the out-neighbours in each adjacency list sorted in lexicographic order, so that this is the order in which vertices are encountered when going through neighbour lists. (For instance, the out-neighbour list of a is $\{b, d, e, g\}$ sorted in that order.)

7a (60 p) Run Dijkstra's algorithm by hand on this graph, starting in the vertex a. Use a heap for the priority queue implementation. Assume that in the array representing the heap, the vertices are initially listed in lexicographic order.

During the execution of the algorithm, describe for every vertex which neighbours are being considered and how they are dealt with. Show for the first two dequeued vertices how the heap changes after the dequeueing operations and all ensuing key value updates in the

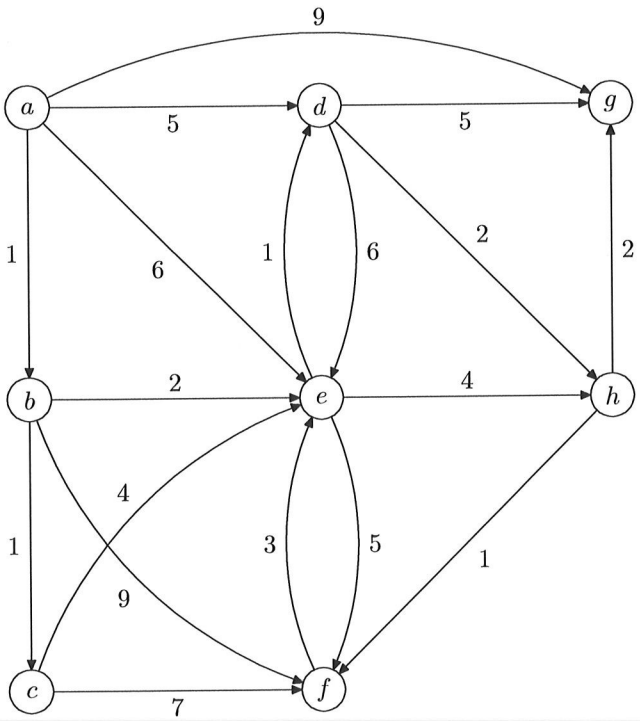


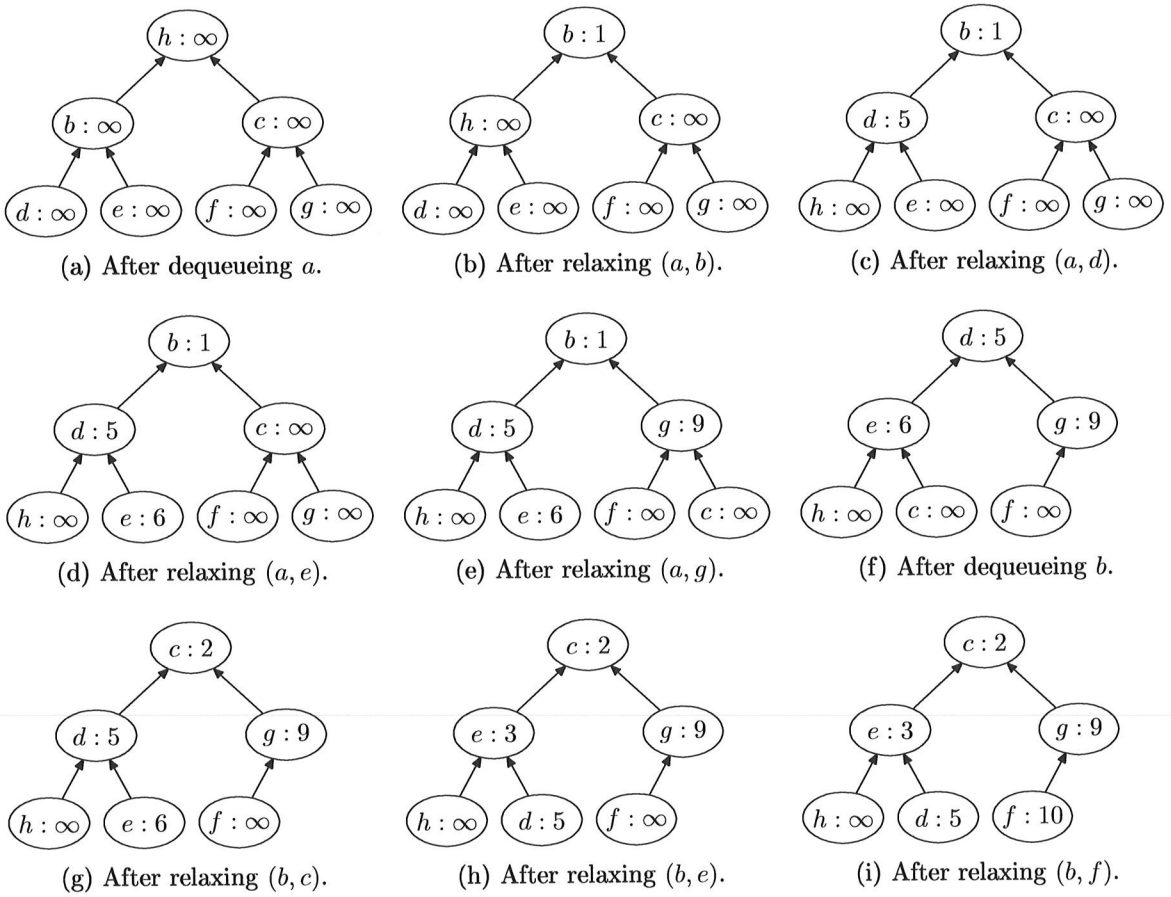
Figure 2: Directed graph for Problem 7a.

priority queue. For the rest of the vertices, it is sufficient to just describe how the key values are updated, without redrawing the heap after each operation, but you still have to describe how the algorithm considers all neighbours of the dequeued vertices. Finally, show the directed tree T produced at the end of the algorithm.

Solution: We illustrate the heap used for the priority queue and how it changes in Figure 3. At the outset, the vertex a has key 0 and all other vertices have key ∞. We will use the notation $v : k$ in the heap when vertex v has key value k.

1. After a has been dequeued, vertex h is moved to the top of the heap and we have the configuration in Figure 3a. Relaxing the edge (a, b) shifts b to the top and pushes h down, yielding Figure 3b. Relaxing (a, d) swaps d and h, yielding Figure 3c. Relaxing (a, e) updates the key of e, but since it is still larger than the key of the parent d nothing moves in the heap (see Figure 3d). Finally, relaxing (a, g) makes g bubble up and c bubble down, so that we have the configuration in Figure 3e when all outgoing edges from a have been relaxed.

2. Since b is now at the top of the heap, it is the vertex dequeued next. This will add the edge (a, b) to the shortest path tree, which we indicate in Figure 4. When b is removed, c is moved to the top. This violates the min-heap property, since the key of c is larger than that of its children. Since d has smaller key than g, we swap d and c. The min-heap property in the subtree rooted at c, i.e., the left subtree of the heap, is now violated since the key of c is still not smaller than or equal to that of its children. Since e has smaller key than h, c and e trade places. The subtree rooted at c is now a single vertex, and so is a legal min-heap, and the full heap after removal of b looks as in Figure 3f.



(a) After dequeueing a. (b) After relaxing (a, b). (c) After relaxing (a, d).

(d) After relaxing (a, e). (e) After relaxing (a, g). (f) After dequeueing b.

(g) After relaxing (b, c). (h) After relaxing (b, e). (i) After relaxing (b, f).

Figure 3: Heap configurations for priority queue in Problem 7a.

Relaxing the edge (b, c) decreases the key value of c to $1 + 1 = 2$. Since the key value of c is now smaller than that of e, c bubbles up and e bubbles down, and since c also has a smaller key than d these two vertices also trade places, yielding the heap in Figure 3g. Relaxing (b, e) swaps d and e, resulting in Figure 3h. Finally, relaxing (b, f) updates the key of f but does not change the structure of the heap, since the parent g of f has a smaller key (see Figure 3i).

3. Since c is now at the top of the heap, it is dequeued next, and the edge (b, c) is added to the shortest paths tree. When we relax (c, e), we see that the distance 2 to c plus the edge weight 4 sum to 6, which is larger than the current key 3 of c, so no update of c is made. Relaxing the edge (c, f) decreases the key of f to $2 + 7 = 9$, however.

4. Vertex e now has the smallest key 3 and is dequeued next. This adds the edge (b, e) to the shortest paths tree, since the key of e was last updated when (b, e) was relaxed. Relaxing (e, d), (e, f), and (e, h) yields updated key values 4, 8, and 7, respectively.

5. Now vertex d has the smallest key 4 and so is dequeued, adding (e, d) to the shortest paths tree. When we relax (d, g), we see that the distance 4 to d plus the edge weight 5 sum to 9, which is not smaller than the current key 9 of g, so no update is made. Relaxing (d, h)

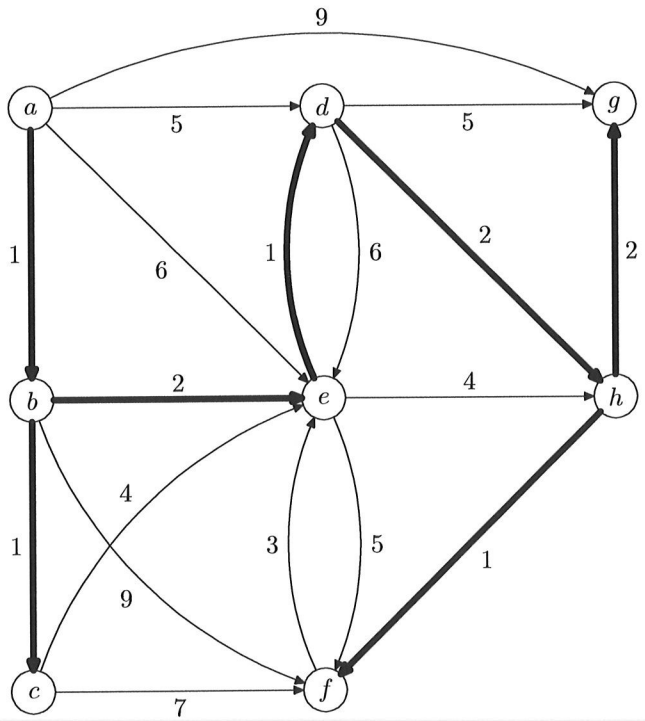


Figure 4: Shortest paths computed by Dijkstra's algorithm for graph in Figure 2.

decreases the key of h to 6, however.

6. Vertex h is dequeued next, adding the just relaxed edge (d, h) to the shortest paths tree. Relaxing (h, g) and (h, f) leads to decreased keys 8 and 7, respectively.

7. Next, f is dequeued, adding the just relaxed edge (h, f).

8. Since f has no out-neighbours, we proceed to dequeueing the final vertex g in the queue, adding the edge (h, g).

The computed shortest paths tree is indicated by the bold edges in Figure 4.

7b (10 p) Consider the directed tree of shortest paths T produced in Problem 7a. Suppose that all edge weights in G are changed by some additive constant $c \in \mathbb{R}^+$. Is it true that T is still a directed tree of shortest paths for the modified graph? Please make sure to motivate your answer clearly.

Solution: No, this is not true. If we choose c to be large enough, then the shortest path between two vertices will always be the one with the fewest edges. This is not the case for the tree T computed in Problem 7a. (Consider, e.g., the shortest path $a \to b \to e \to d \to h \to f$ from a to f, which has many more edges than $a \to b \to f$.) Incidentally, we discussed precisely this question in class during one of the lectures, so for that reason this should hopefully have been a very easy subproblem.

7c (10 p) Consider the directed tree of shortest paths T produced in Problem 7a. Suppose that all edge weights in G are changed by some multiplicative constant $c \in \mathbb{R}^+$. Is it true

that T is still a directed tree of shortest paths for the modified graph? Please make sure to motivate your answer clearly.

Solution: Yes, this is true. Since all weights change by a factor c, the cost of all paths also change by a factor c. Hence, all shortest paths remain shortest paths, and the tree T is still good. (As far as the instructor can remember, this was not discussed in class, but it was an exercise for one of the exercise sessions.)

7d (20 p) Suppose that we want to give an extra bonus to paths with few hops, so that the length of a path is calculated as the sum of the weight of all the edges in the path ***plus*** the number of edges in the path. Describe an algorithm that can solve this problem (for any directed graph G with non-negative edge weights) and analyze its time complexity.

Solution: This problem can be solved in different ways. The easiest way is probably to read in the graph, add 1 to all edge weights, and then run Dijkstra's algorithm as before. The length of any path after this change will clearly be the sum of the weight of all the edges in the path plus the number of edges in the path, and so Dijkstra's algorithm will return the answers we are looking for. Since the preprocessing step can be performed in time $O(|V| + |E|)$ for a graph with vertex set V and edge set E, the running time of Dijkstra's algorithm dominates, and the asymptotic time complexity is the same as for the original version of Dijkstra's algorithm, i.e., $O(|E| \log |V|)$.

8 (60 p) Consider the regular expression $((ab)^*a)^* \mid (ab^*a^*b)$ and determine which of the words below belong to the language generated by this regular expression. Motivate your answers briefly but clearly by explaining for each string how it can be generated or arguing why it is impossible.

1. *abab*
2. *ababab*
3. *abbabbb*
4. *abbbbb*
5. *aaabaaba*
6. *abababa*

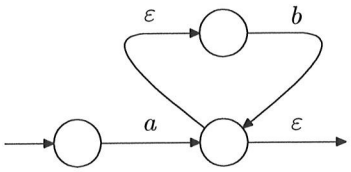
Solution: We give 10 p per correct answer with full motivation.

We start by observing that the regular expression $((ab)^*a)^* \mid (ab^*a^*b)$ accepts words that fit one of two patterns:

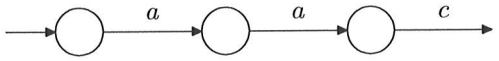
- Zero or more repetitions of strings on the form $ababab \ldots aba$, where we can have zero or more repetitions of the pattern ab but where there has to be an a at the end.
- A single a, followed by zero or more b, followed by zero or more a, terminated by a single b.

With this in mind, we get the following answers:

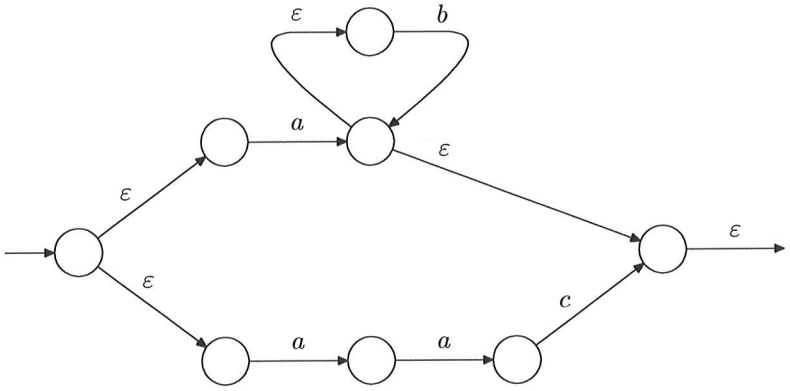
1. $abab = a(b)^1(a)^1b$ fits the second pattern. ✓
2. *ababab* ✗ There are too many alternations of a and b for the second pattern, and for the first pattern the string should end with a.



(a) NFA fragment for regular expression ab^*.



(b) NFA fragment for regular expression aac.



(c) NFA fragment for regular expression $ab^*|aac$.

Figure 5: NFA fragments in translation of regular expression in Problem 9.

3. $abbabbb$ ✗ The first pattern does not match, since the string does not end with a, and since there are two alternations of a and b there should only be a single b the second time to match the second pattern.

4. $abbbbb = a(b)^4(a)^0b$ fits the second pattern. ✓

5. $aaabaaba = ((ab)^0)a)^2((ab)^1a)^2$ fits the first pattern. ✓

6. $abababa = (ab)^3a$ fits the first pattern. ✓

9 (120+ p) In this problem we want to construct a deterministic finite automaton (DFA) that recognizes the language generated by the regular expression $((ab^*)|(aac))^*$.

9a (60 p) Translate this regular expression to a nondeterministic finite automaton (NFA) using the method from Mogensen's notes that we learned in class. Make sure to describe clearly which part of the regular expression corresponds to which part of the NFA, and which states are accepting. Please do not take any shortcuts without explaining what you are doing.

Solution: We follow the procedure outlined in Section 1.3 in Mogesen's notes. Three NFA fragments with corresponding regular (sub)expressions are shown in Figure 5. The full translation of the regular expression to a nondeterministic finite automaton is as in Figure 6, where we have added numbers to the states to be able to refer to them in the next subproblem.

As a side note, we remark that the expression $((ab^*)|(aac))^*$ is overly paranthesized, just to make the intended meaning clear. Since we agreed in class that the star operator binds hardest, followed by concatenation, it would have been fully sufficient to just write $(ab^*|aac)^*$.

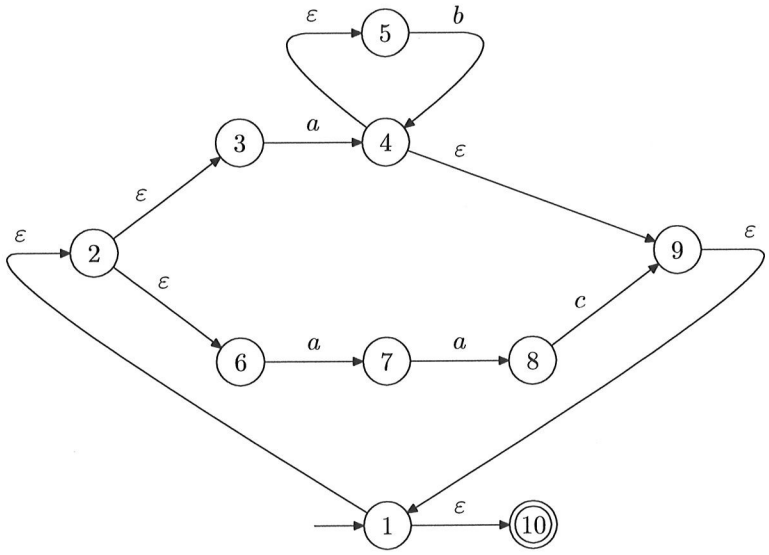


Figure 6: Full translation to NFA of regular expression in Problem 9.

9b (60 p) Translate the nondeterministic finite automaton to a deterministic finite automaton using the method from Mogensen's notes that we learned in class. Make sure to explain in detail how you perform the subset construction, so that it is possible to follow your line of reasoning. Present clearly the resulting DFA, with all states and transitions, and specify which states are accepting.

Solution: We follow Algorithm 1.3 in Section 1.5.2 of Mogesen's notes, except that we use calligraphic letters $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, ... to refer to the constructed states in the deterministic finite automaton (to distinguish them more clearly from the NFA states). Referring in what follows to the numbered states in the NFA in Figure 6, the starting state is

$$\varepsilon\text{-closure}(\{1\}) = \{1, 2, 3, 6, 10\} = \mathcal{A}$$

(where we recall that the *ε-closure* of a set of states S is any state that can be reached from some state in S via zero or more ε-transitions). The possible transitions from $\mathcal{A}$ on characters a, b, and c are

$$\begin{aligned}
&\text{move}(\mathcal{A}, a) = \varepsilon\text{-closure}(\{4, 7\}) = \{1, 2, 3, 4, 5, 6, 7, 9, 10\} = \mathcal{B} && \textbf{[new state]}\\
&\text{move}(\mathcal{A}, b) = \emptyset && \text{[no transition possible]}\\
&\text{move}(\mathcal{A}, c) = \emptyset && \text{[no transition possible]}
\end{aligned}$$

We consider next the new state $\mathcal{B}$, for which we get the transitions

$$\begin{aligned}
&\text{move}(\mathcal{B}, a) = \varepsilon\text{-closure}(\{4, 7, 8\}) = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\} = \mathcal{C} && \textbf{[new state]}\\
&\text{move}(\mathcal{B}, b) = \varepsilon\text{-closure}(\{4\}) = \{1, 2, 3, 4, 5, 6, 9, 10\} = \mathcal{D} && \textbf{[new state]}\\
&\text{move}(\mathcal{B}, c) = \emptyset && \text{[no transition possible]}
\end{aligned}$$

Moving on to state $\mathcal{C}$ we obtain

$$\begin{aligned}
&\text{move}(\mathcal{C}, a) = \varepsilon\text{-closure}(\{4, 7, 8\}) = \mathcal{C}\\
&\text{move}(\mathcal{C}, b) = \varepsilon\text{-closure}(\{4\}) = \mathcal{D}\\
&\text{move}(\mathcal{C}, c) = \varepsilon\text{-closure}(\{9\}) = \{1, 2, 3, 6, 9, 10\} = \mathcal{E} && \textbf{[new state]}
\end{aligned}$$

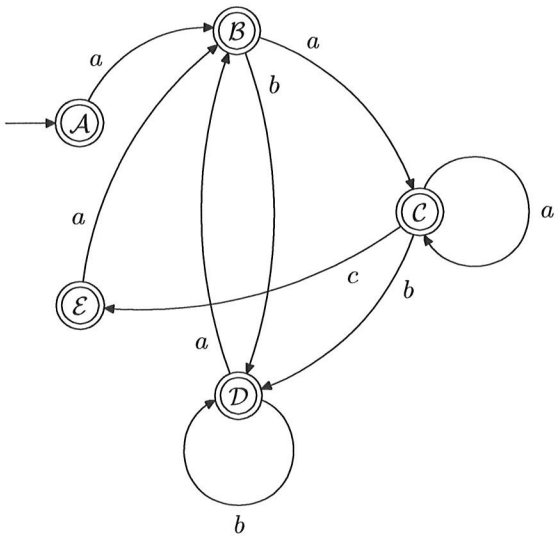


Figure 7: Conversion of NFA in Figure 6 to DFA.

and for state $\mathcal{D}$ we derive

$$\begin{aligned}
\text{move}(\mathcal{D}, a) &= \varepsilon\text{-closure}(\{4, 7\}) = \mathcal{B} \\
\text{move}(\mathcal{D}, b) &= \varepsilon\text{-closure}(\{4\}) = \mathcal{D} \\
\text{move}(\mathcal{D}, c) &= \emptyset \qquad \text{[no transition possible]}
\end{aligned}$$

Finally, for state $\mathcal{E}$ we can observe that it is identical to $\mathcal{A}$ except for the added NFA state 9, the only transition from which is an ε-transition to 1. We therefore should get the same transitions as for state $\mathcal{A}$, and indeed we can compute that

$$\begin{aligned}
\text{move}(\mathcal{E}, a) &= \mathcal{B} \\
\text{move}(\mathcal{E}, b) &= \emptyset \\
\text{move}(\mathcal{E}, c) &= \emptyset
\end{aligned}$$

Since we have now considered all possible transitions from all states in our constructed DFA, we are done. Interestingly, since all subsets of states contain the accepting NFA state 10, all states in our constructed DFA are accepting. Hence, the DFA will never reject a string by ending up in the "wrong state"—what will happen for illegal strings is that we will be in a state where there is no legal transition for the next character. See Figure 7 for an illustration of the constructed automaton.

9c *Bonus problem (worth 40 p extra; might be hard):* How small a DFA can you produce that accepts precisely the language generated by the regular expression above? Can you prove that the size of your DFA is optimal? Note that that you can solve this problem even if you did not solve the other subproblems above.

Solution: Looking at Figure 7, it is not hard to see that the states $\mathcal{A}$ and $\mathcal{E}$ have exactly the same outgoing transitions. We can therefore merge these two states to get a smaller DFA as in Figure 8. Let us now argue why any DFA for the regular expression $(ab^*|aac)^*$ must have at least 4 states, meaning that the construction in Figure 8 is optimal.

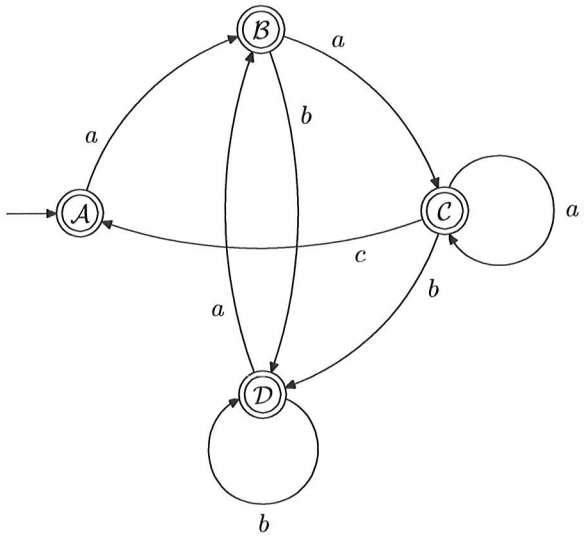


Figure 8: Smallest possible DFA equivalent to that in Figure 7.

Observe first that we need 3 states keeping track of whether we have seen 0, 1, or 2 or more a's since the last read c (since this determines whether we are ready to accept another c or not). These are the states $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$, in Figure 8.

Argued slightly differently (and in more detail), the initial strings (i) ε, (ii) a, and (iii) aa must lead to different states. First note that the states corresponding to these strings must all be accepting, since ε, a, and aa all match the regular expression. Now we can observe that (iii) is the only one of these strings after which we are willing to append a c to obtain aac, whereas c and ac are invalid strings. Hence, string (iii) must end up in a different state from that of strings (i) and (ii). Furthermore, after having seen (ii) we accept ac, resulting in the string aac, but we cannot append the same string to the empty string in (i) since ac is not a valid string. Therefore, strings (i) and (ii) must also end up in different states in the DFA.

To see why at least one more state is needed, consider the state of the DFA after having read ab. After this we are willing to accept another b to get abb. This rules out the initial state corresponding to (i), since if the DFA were in that state it would also have to accept the string b, which is not legitimate. Furthermore, after having read ab it is not acceptable to append ac, since $abac$ does not match the regular expression, and this rules out the state corresponding to (ii). Finally, since appending c to obtain abc is also not acceptable, we cannot be in the state corresponding to (iii). This shows that after having read ab the DFA must be in a different state than after having read strings (i), (ii), or (iii), and so an additional state like $\mathcal{D}$ in Figure 8 is also needed.

We remark that it is not necessary for this subproblem to go the formal route by first constructing an NFA from the regular expression and then converting this NFA to a DFA. Just by studying the regular expression and understanding what it means, it is possible to write down the DFA in Figure 8 directly, and then argue that the size has to be optimal.

10 (90 p) Consider the following context-free grammars, where a, b, c are terminals, B, S are nonterminals, and S is the starting symbol.

Grammar 1:

$$S \to aS \tag{5a}$$
$$S \to B \tag{5b}$$
$$B \to bcB \tag{5c}$$
$$B \to \tag{5d}$$

Grammar 2:

$$S \to aS \tag{6a}$$
$$S \to BS \tag{6b}$$
$$S \to B \tag{6c}$$
$$B \to bcB \tag{6d}$$
$$B \to \tag{6e}$$

Grammar 3:

$$S \to aS \tag{7a}$$
$$S \to BS \tag{7b}$$
$$S \to B \tag{7c}$$
$$B \to bBc \tag{7d}$$
$$B \to \tag{7e}$$

Which of these grammars generate regular languages? For each grammar, write a regular expression that generates the same language (and explain why), or argue why the language generated by the grammar is not regular. In your regular expressions, please use *only* the concatenation, alternative (|) and star (*) operators, and not the syntactic sugar extra operators that we just mentioned in class but never utilized.

Solution: We give 30 p per correct answer with full motivation.

Grammar 1 generates strings that contain zero or more occurrences of a (resulting from the production $S \to aS$) followed by zero or more occurrences of bc (resulting from the production $S \to B$ followed by $B \to bcB$ and finally $B \to$). This languages is captured by the regular expression $a^*(bc)^*$.

Grammar 2 is very similar, except that the production $S \to BS$ allows us to "start over" with S again and generate as many copies as we like of strings in the language generated by Grammar 1. The regular expression corresponding to Grammar 2 is therefore $(a^*(bc)^*)^*$.

The language generated by Grammar 3 is not regular. To see this, note that strings of the form $b^n c^n$ are in the language (generated by $S \to B$ followed by $B \to bBc$ for the desired number of times n and finally $B \to$) while strings of the form $b^n c^{n+1}$ are not (since the terminals b and c are always generated in pairs). In order to distinguish between the two, the regular expression would need to do counting, but as mentioned in Mogesen's notes and during the lectures this is something that regular expressions cannot do. (You do not have to prove that regular expressions cannot count to get full credits for this problem—referring to what is said in the notes or what we covered in class is enough.)