KØBENHAVNS UNIVERSITET

Diskret Matematik og Formelle Sprog: Exam April 9, 2022
Problems and 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. In general, the solutions provided below tend to be much more detailed than what would be expected for the exam submissions. As communicated during the course, the course notes published on Absalon, which contain many problems that we worked out in class, are a much better indicator of what was expected from the exam solutions.

1 (70 p) In the following snippet of code `A` is an array indexed from 1 to `A.size` that contains elements that can be compared, and `B` is intended to be an auxiliary array of the same type and size. The functions `floor` and `ceiling` round down and up, respectively, to the nearest integer.

```
m := A.size
for i := 1 upto m
    B[i] := A[i]
while (m > 1)
    for i := 1 upto floor(m / 2)
        if (B[2 * i - 1] >= B[2 * i])
            B[i] := B[2 * i - 1]
        else
            B[i] := B[2 * i]
    if (m mod 2 == 1)
        B[ceiling(m / 2)] := B[m]
    m := ceiling(m / 2)
return B[1]
```

1a (30 p) Explain in plain language what the algorithm above does (i.e., do not just rewrite the pseudocode word by word, so that your text is just a more verbose version of the pseudocode, but interpret what the code is doing and why). In particular, how can you describe the element `B[1]` that is returned at the end of the algorithm?

Solution: Let us write M to denote the value of m at the start of the algorithm, i.e., the size of the array A. The algorithm starts by copying all of the elements in the array A to the array B.

In the first iteration of the while loop, each pair of elements $B[2i-1]$ and $B[2i]$ are compared, and the largest of the two is copied to $B[i]$. If the size of the array $m = M$ is odd, the final element $B[m]$ is copied to $B[\lceil m/2 \rceil]$. Finally, the size of the array is adjusted to $m := \lceil M/2 \rceil$ and a new iteration of the while loop is initiated.

As long as there are two or more elements left in $B[1], \ldots, B[m]$—i.e., as long as $m > 1$—each pair of elements $B[2i-1]$ and $B[2i]$ for $i = 1, \ldots, \lfloor m/2 \rfloor$ are compared and the largest one is kept by being copied to $B[i]$, and any final uncompared element is also copied as in the first iteration. After the size of the array has been adjusted to $m := \lceil m/2 \rceil$ at the end of an iteration, it holds for every element in $B[m+1], \ldots, B[M]$ that it is smaller than or equal to than some element in $B[1], \ldots, B[m]$. If $m > 1$, a new iteration of the while loop is initiated.

When the while loop ends we have $m = 1$, and it follows from the description above that $B[1]$ is a largest element in the array (since every element $B[2], \ldots, B[M]$ is smaller than or equal to $B[1]$). That is, the code snippet finds and returns a largest element in the array A.

1b (30 p) Provide an asymptotic analysis of the running time as a function of the array size m.

Solution: As before, let us write M to denote the size of the input, i.e., the array A. The initialization code

```
m := A.size
for i := 1 upto m
    B[i] := A[i]
```

clearly runs in time $\mathrm{O}(M)$.

Inside the while loop, there is a for loop

```
for i := 1 upto floor(m / 2)
    if (B[2 * i - 1] >= B[2 * i])
        B[i] := B[2 * i - 1]
    else
        B[i] := B[2 * i]
```

that does a constant amount of work per element $B[1]$, $B[2]$, …, $B[\,2 \cdot \lfloor m/2 \rfloor\,]$, namely, it compares the elements pairwise and copies the largest one in each pair, and the final lines of code inside the while loop

```
if (m mod 2 == 1)
    B[ceiling(m / 2)] := B[m]
m := ceiling(m / 2)
```

then just copy the final element if m is odd, after which m is divided by 2 and rounded up. All of this work in an iteration of the while loop can be upper-bounded by $K \cdot m$ for some (small) constant K. The final return call after the while loop is just a constant amount of work and can be ignored.

The while loop is run for values $m = M, \lceil M/2 \rceil, \lceil \lceil M/2 \rceil / 2 \rceil, \ldots$ all the way down to $m = 2$. If we round up M to the nearest power of 2, which cannot decrease our estimate of the running time, we can think of the while loop being run for $m = 2^i$ for $i = \lceil \log M \rceil, \lceil \log M \rceil - 1, \ldots, 1$, and we get that the total amount of work in all iterations of the while loop is

$$\sum_{i=1}^{\lceil \log M \rceil} K \cdot 2^i \;\leq\; K \cdot \sum_{i=0}^{\lceil \log M \rceil} 2^i \;=\; K \cdot \frac{2^{\lceil \log M \rceil} - 1}{2 - 1} \;\leq\; K \cdot 2M\,, \tag{1}$$

which is linear in M, i.e., $\mathrm{O}(M)$.

Switching back to denoting the array size by m as in the problem statement, we see that the algorithm runs in time $\mathrm{O}(m)$, i.e., in time linear in the size of the array. It is not hard to verify that all of the estimates above also have lower bounds matching up to constant factors, and so if we want to be really precise we can state the running time as $\Theta(m)$, but this is not needed for a full score.

1c (10 p) Can you improve the code to run asymptotically faster while retaining the same functionality? How much faster can you get the algorithm to run? Analyse the time complexity of your new algorithm. Can you prove that it is asymptotically optimal?

Solution: In order to find a largest element in an array we have to go over the whole array, and so linear time is asymptotically optimal. Hence, the answer to the first question in Problem 1c is that no, we cannot solve the same problem asymptotically faster.

However, it is certainly possible to construct a simpler algorithm that would probably also have a better constant factor in the linear running time, for instance, something like this:

```
max := A[1]
for i := 2 upto A.size
    if ([A[i] > max)
        max := A[i]
return max
```

In case you did not get a tight analysis in Problem 1b, you could get a full score on Problem 1c by presenting a linear time algorithm and noting why this is optimal. Otherwise, an explanation why the original algorithm is already asymptotically optimal was also sufficient for a full score on Problem 1c.

2 (100 p) After the less than stellar performance by Jakob at the DIKU 50th anniversary outreach event, only 12 brave children registered for a follow-up event that was arranged earlier this semester with the purpose of conveying the excitement of computer science to the younger generation. In view of this, the head of the Algorithms & Complexity Section Mikkel Thorup decided to take charge of the organization. In this problem we will study how Mikkel's follow-up event was arranged.

2a (40 p) Since Mikkel is an avid mushroom picker, he took the 12 children to the forest to collect mushrooms. Being a good host, he of course made sure that every child found at least one mushroom. When everybody returned to DIKU, it turned out that the children had collected exactly 77 mushrooms together. Mikkel explained to the children that this meant there had to be at least two kids who had collected the same number of mushrooms. Can you describe in detail how he could have proven such a claim?

Solution: Let us argue by contradiction. Suppose child number i picked m_i mushrooms, where we sort the children in increasing order with respect to the number of mushrooms picked, i.e., so that $m_1 \leq m_2 \leq \ldots \leq m_{12}$. Since all children found mushrooms we have $m_1 \geq 1$, and if no pair of children picked the same number of mushrooms, this means that $m_1 < m_2 < \ldots < m_{12}$. In particular, this implies for all i that $m_i \geq i$. Summing up, we get that the total number of mushrooms has to be

$$\sum_{i=1}^{12} m_i \;\geq\; \sum_{i=1}^{12} i \;=\; \frac{12 \cdot 13}{2} \;=\; 78\,, \tag{2}$$

where for the next to last equality we can use the arithmetic sum formula $\sum_{i=1}^{n} i = n(n+1)/2$ that we have learned in class.

However, according to the problem statement the children only collected 77 mushrooms together. This contradicts the lower bound on the number of mushrooms that we just obtained. Hence, at least two of the children must have collected the same number of mushrooms.

2b (60 p) When all mushrooms had been cleaned, Mikkel taught the children what it means for two positive integers to be relatively prime. He then wrote the numbers 1, 2, 3, ..., 22 on 22 sheets of paper and performed the following experiment:

- First, all the 22 sheets of papers were randomly shuffled.
- Then each of the 12 children randomly picked one sheet of paper.
- Finally, the children tried to identify a pair amongst themselves who held sheets of paper with relatively prime numbers.

This experiment was repeated several times, and every single time some pair of children found that they had drawn relatively prime numbers. Together with the children, Mikkel discussed whether this was just a weird coincidence or whether this always has to happen. What was the conclusion of this discussion? Please make sure to provide formal proofs backing up any claims you make.

Solution: This is not a coincidence, but always has to happen. This can be argued by using the pigeonhole principle. We consider our 12 "pigeons" to be the 12 numbers between 1 and 22 that are chosen. We create 11 "pigeonholes" by grouping consecutive numbers together into pairs $\{1,2\}$, $\{3,4\}$, ..., $\{19,20\}$, $\{21,22\}$. We now make two observations:

1. Two integers a and $a+1$ are always relatively prime. Perhaps the easiest way to see this is that any divisor of two numbers a and b must also divide $b-a$, which, in particular, means the greatest common divisor of a and $b=a+1$ must also divide $b-a=1$. In other words, the greatest common divisor of a and $a+1$ is 1, which is the definition of the two numbers being relatively prime.

2. Since we have 12 numbers but only 11 pairs, for at least some pair $\{a, a+1\}$ both of the numbers in the pair are chosen.

This shows that at least one pair of children have to choose relatively prime numbers.

3 (50 p) In this problem we consider formulas in propositional logic. Decide for each of the formulas below whether it is tautological or not and then do the following:

- If the formula is a tautology, prove this by either (i) presenting a full truth table for all subformulas analogously to how we did it in class, or (ii) providing an explanation based on the rules and equivalences we have learned. You only need to to one of (i) or (ii), but you are free to do both if you like, and crisp and clear explanations can compensate for minor slips in the truth table.

- If the formula is *not* a tautology, present a falsifying assignment. Also, explain how you can change a single connective in the formula to turn it into a tautology, and try to provide a natural language description of what the tautology you obtain in this way encodes (i.e., not just mechanically replacing each connective by a word, but explaining what the underlying logical principle is).

3a (20 p) $(p \rightarrow (q \wedge r)) \rightarrow ((q \vee \neg p) \wedge (r \vee \neg p))$

Solution: This formula is a tautology. To show this, let us work on the second half of the formula, i.e., the conclusion of the outermost implication. Using commutativity and distributivity, we know that

$$(q \vee \neg p) \wedge (r \vee \neg p) \;\equiv\; (\neg p \vee q) \wedge (\neg p \vee r) \;\equiv\; \neg p \vee (q \wedge r)\,. \tag{3a}$$

For the implication connective we know that $x \to y \equiv \neg x \vee y$, and applying this to (3a) we get

$$\neg p \vee (q \wedge r) \;\equiv\; p \to (q \wedge r)\,. \tag{3b}$$

But this last formula in (3b) is the premise of the outermost implication, and any formula that is logically equivalent to a formula on the form $x \to x$ is certainly a tautology.

We can also construct a truth table for $(p \to (q \wedge r)) \to ((q \vee \neg p) \wedge (r \vee \neg p))$ as follows:

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

We see that in the middle column corresponding to the outermost implication we only have $\top$, so the formula always evaluates to true and hence is a tautology.

3b (30 p) $((p \wedge q) \to r) \to ((r \vee \neg p) \wedge (r \vee \neg q))$

Solution: This formula is *not* a tautology—if we set $p = \top$ and $q = r = \bot$, then the premise $((p \wedge q) \to r)$ of the outermost implication is true (since $p \wedge q$ is false), but the conclusion $(r \vee \neg p) \wedge (r \vee \neg q)$ of the outermost implication is false (since $r \vee \neg p$ is false).

To analyse the structure of this formula, let us work on the conclusion of the outermost implication. Using the distributivity, commutativity, and De Morgan rules we can rewrite the conclusion as

$$(r \vee \neg p) \wedge (r \vee \neg q) \;\equiv\; r \vee (\neg p \wedge \neg q) \;\equiv\; (\neg p \wedge \neg q) \vee r \;\equiv\; \neg(p \vee q) \vee r\,, \tag{4a}$$

and by applying the equivalence $x \to y \equiv \neg x \vee y$ for the implication connective we can rewrite (4a) further as

$$\neg(p \vee q) \vee r \;\equiv\; (p \vee q) \to r\,. \tag{4b}$$

From this we see that if we we change the AND in the premise of the formula $((p \wedge q) \to r) \to ((r \vee \neg p) \wedge (r \vee \neg q))$ in Problem 3b to an OR, we get the formula

$$((p \vee q) \to r) \to ((r \vee \neg p) \wedge (r \vee \neg q)) \tag{5}$$

which is clearly a tautology, since in view of the rewriting in (4a) and (4b) this is equivalent to a formula on the form $x \to x$. We can also argue directly that the formula (5) is a tautology by observing that what the formula says is that if the implication $(p \vee q) \to r$ holds and r is false, it has to hold that both p and q are false. This is certainly a true statement, since an implication is not true if the premise is true and the conclusion is false.

Another possible solution is to change the AND in the conclusion of the formula in Problem 3b to an OR to get

$$((p \wedge q) \to r) \to ((r \vee \neg p) \vee (r \vee \neg q))\,, \tag{6}$$

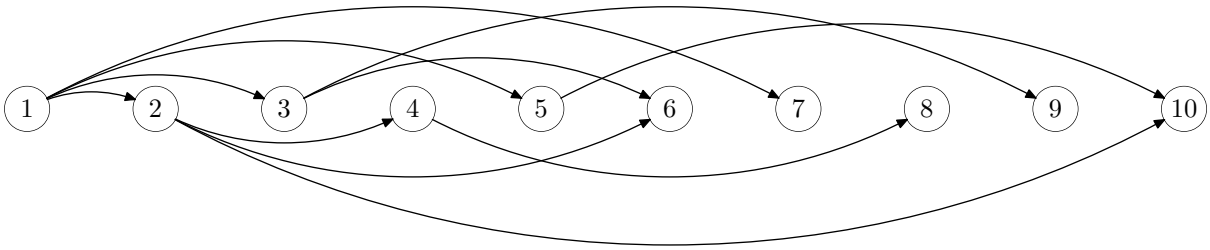


Figure 1: Directed graph D_S representing relation S in Problem 4.

and this formula can also be argued to be a tautology in a similar way.

Continuation of statement of Problem 3:

(Note that $\rightarrow$ denotes logical implication. and $\neg$ denotes logical negation. Negation is assumed to bind harder than the binary connectives, but other than that all formulas are fully parenthesized for clarity.)

4 (50 p) Consider the relation S described by the directed graph D_S in Figure 1.

4a (10 p) Write down the matrix representation M_S of the relation S and describe briefly but clearly how you construct this matrix.

Solution: The matrix representation of the relation S is a 10×10 matrix where there is a 1 in position (i, j) if $(i, j) \in S$ and a 0 in this position otherwise. Looking at the directed graph representation D_S in Figure 1, this means that there should be a 1 in position (i, j) if and only if there is a directed edge from i to j in D_S. This yields the matrix

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

(where we note, in particular, that from row 6 onwards there are only zeros in the matrix, since there are no outgoing edges from the vertices labelled $6, 7, \ldots, 10$ in D_S).

4b (10 p) Let us write T to denote the transitive closure of the relation S. What is the matrix representation of T? Write it down and explain how you constructed it.

Solution: To obtain the transitive closure T of the relation S, we can proceed as follows:

1. Start by setting $T' = T = S$.
2. Go over all triples (i, j, k) such that $(i, j) \in T$ and $(j, k) \in T$, and add (i, k) to T' since the elements i and k should also be related by transitivity.

3. If new pairs were added in step 2, so that $T' \neq T$, then set $T = T'$ and go to step 2 again. Otherwise T is the transitive closure.

Referring to the directed graph representation D_S in Figure 1, a pair (i, k) should be in the transitive closure precisely when there is a path from i to k in D_S. Starting from each vertex $i = 1, 2, \ldots, 10$ and writing down which other vertices are reachable from i yields the matrix

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

for the transitive closure T of S.

4c (10 p) Now let R be the reflexive closure of the relation T. What is the matrix representation of R? Write it down and explain how you constructed it.

Solution: To get the reflexive closure R of the relation T, we just need to add all pairs (i, i) so that the resulting relation is reflexive. For the matrix representation this corresponds to adding 1s on the diagonal, which yields the the matrix

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

for the reflexive closure R of T.

4d (20 p) Can you explain in words what the relation R is by describing how it can be interpreted? (In particular, is it similar to anything we have discussed during the course?)

Solution: The relation R is the divisibility relation restricted to the integers $\{1, 2, \ldots, 10\}$. In other words, we have that $(i, j) \in R$ if and only if $i \mid j$. This can be verified by going over all rows i in the matrix M_R and checking in each row that there is a 1 in column j if and only if $i \mid j$. The first row is all-1s, corresponding to that 1 divides all integers. In the second row, every other position is 1, corresponding to that every other number is even. In the third row, every third position is 1, since every third number is divisible by 3, et cetera.

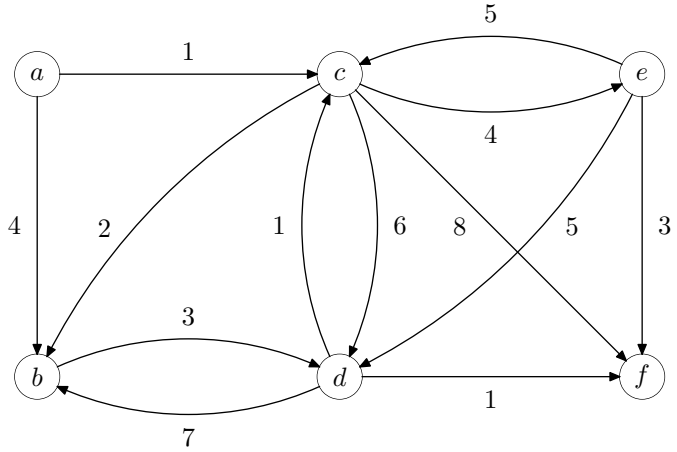


Figure 2: Directed graph D on which to run Dijkstra's algorithm in Problem 5a.

5 (70 p) In this problem we wish to understand Dijkstra's algorithm.

5a (40 p) Assume that we are given the directed graph D 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 vertices are encountered in this order when going through neighbour lists. (For instance, the out-neighbour list of d is (b, c, f) sorted in that order.)

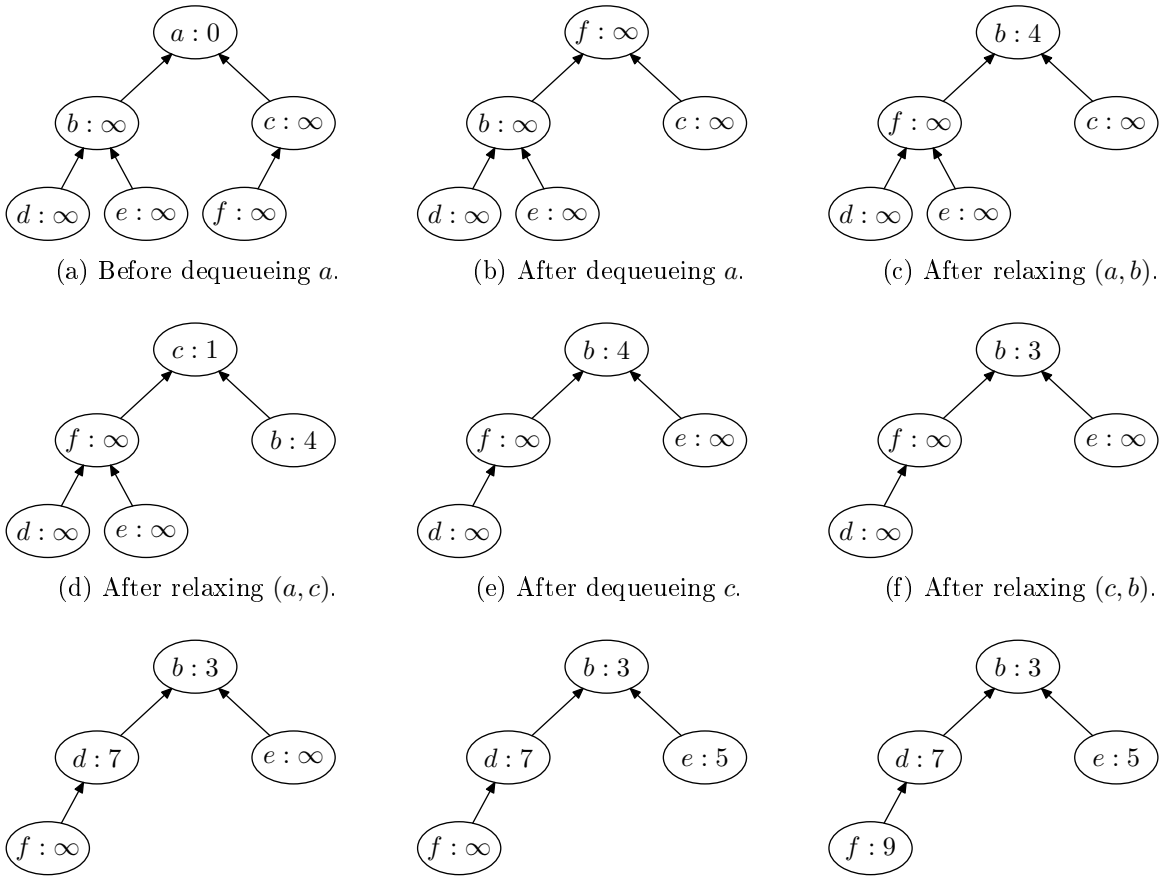
Run Dijkstra's algorithm by hand on the graph D as we have learned in class, starting in the vertex a. Show the directed tree T produced at the end of the algorithm. Use a heap for the priority queue implementation. Assume that in the array representing the heap, the vertices are initially listed in 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 priority queue. For the rest of the vertices, you should describe how the algorithm considers all neighbours of the dequeued vertices and how key values are updated, but you do not need to draw any heaps.

Solution: Let us first note that the solution presented below is quite detailed in order to help students understand all details of the execution of Dijkstra's algorithm. To get a full score on this problem, it is sufficient to provide just the details requested in the problem statement.

As we did in class, let us present a table of how the distance estimates change for different vertices as the algorithm execution proceeds:

Dequeued vertex		a	c	b	e	d	f
Estimate for a	0	0					
Estimate for b	∞	**4**	**3**	3			
Estimate for c	∞	**1**	1				
Estimate for d	∞	∞	**7**	**6**	6	6	
Estimate for e	∞	∞	**5**	5	5		
Estimate for f	∞	∞	**9**	9	**8**	**7**	7

We will explain this table in what follows, where we note right away that updated key values due to relaxations are highlighted in bold font, and we also show in Figure 3 how the heap used for the priority queue changes. We will use the notation $v : k$ in the heap when vertex v has key value k. At the outset, the vertex a has key 0 and all other vertices have key ∞ as in Figure 3a.



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

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

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

Figure 3: Heap configurations for priority queue in Problem 5a.

1. After a has been dequeued, vertex f is moved to the top of the heap and we have the configuration in Figure 3b. Relaxing the edge (a, b) shifts b to the top and pushes f down below b, yielding Figure 3c. Relaxing (a, c) swaps b and c, yielding Figure 3d. There are no further outgoing edges from a to relax.

2. Since c is now at the top of the heap, it is the vertex dequeued next. This will add the edge (a, c) to the shortest path tree, which we indicate in Figure 4. When c is removed, e is moved to the top. This violates the min-heap property, since the key of e is not smaller than or equal to those of its children. Since b has smaller key than f, we swap e and b. This restores the heap property (since the left subtree of the root was not changed, and e is now the root of a singleton subheap), and so the heap after removal of c looks as in Figure 3e.

   Relaxing the edge (c, b) decreases the key value of b to $1 + 2 = 3$. Since b is already the root, the heap does not change except for this key update and now looks like in Figure 3f. Relaxing (c, d) decreases the key value of d to $1 + 6 = 7$ and makes d bubble up above f, resulting in Figure 3g. Relaxing (c, e) updates the key of e to $1+4 = 5$ but does not change the structure of the heap, since the parent b of e has a smaller key (see Figure 3h). Finally, relaxing (c, f) updates the key of f to $1 + 8 = 9$ but again does not change the structure of the heap, since the parent d of f has a smaller key. Now all outgoing edges from c have

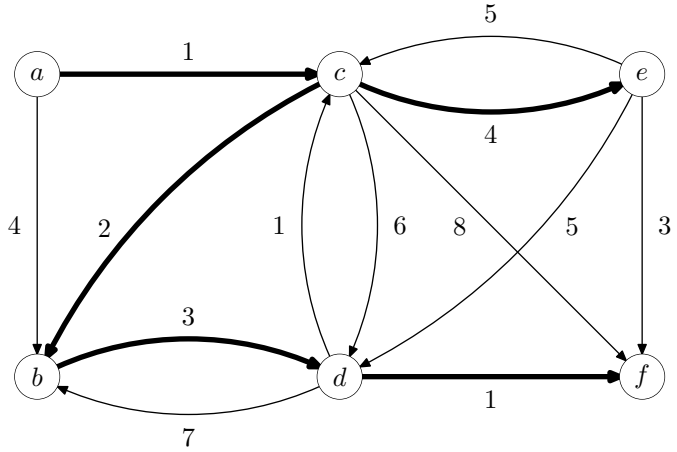


Figure 4: Shortest paths tree computed by Dijkstra's algorithm for graph in Figure 2.

been relaxed, and the resulting heap is as in Figure 3i. According to the instructions in the problem statement, we do not need to provide any further heap illustrations from this point on.

3. Since b is now the vertex with the smallest key value it is dequeued next, and the edge (c, b) is added to the shortest paths tree (since the latest update of the key value of b was when relaxing the edge (c, b)). When we relax (b, d), we see that the distance 3 to b plus the edge weight 3 sum to 6, which is smaller than the current key 7 of d, so the key value of d is updated. There are no other outgoing edges from b to relax.

4. Vertex e now has the smallest key 5 and is dequeued next. This adds the edge (c, e) to the shortest paths tree, since the key of e was last updated when (c, e) was relaxed. Relaxing (e, d) does not lead to any decrease in the estimate, but relaxing (e, f) updates the key of f to 8.

5. Now vertex d has the smallest key 6 and so is dequeued, adding (b, d) to the shortest paths tree. When we relax (d, f) the key of f to $6 + 1 = 7$.

6. Finally, vertex f is dequeued, adding the just relaxed edge (d, f) to the shortest paths tree. There are no outgoing edges from f to relax.

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

5b (30 p) Suppose that we have another type of weighted directed graph $G = (V, E, w)$ where the weight function $w : V \to \mathbb{R}_{\geq 0}$ is defined not on the edges but on the vertices. The cost of a path $P = v_0 \to v_1 \to \ldots \to v_{n-1} \to v_n$ is $\sum_{i=0}^{n-1} w(v_i)$. As before, we are interested in finding the cheapest paths from some start vertex s to all other vertices in the graph.

Design an algorithm that solves this problem as efficiently as possible (for any directed graph G with non-negative vertex weights). Prove that your algorithm is correct, and analyze its time complexity.

Solution: This problem can be solved in different ways, but here we present what is perhaps the most obvious solution.

Given a graph G as in the problem statement with weights on the vertices, construct a new graph G' with the same vertices and edges, except that

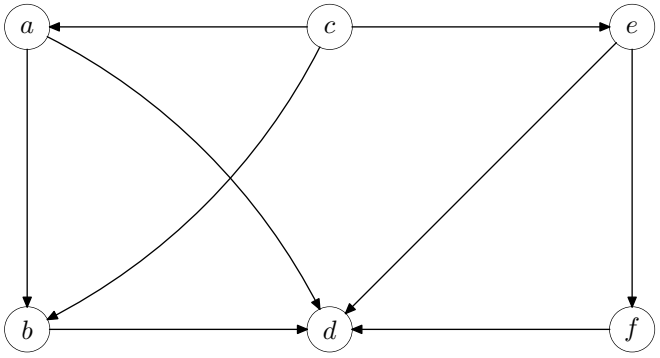


Figure 5: Directed graph H for which to compute a topological ordering in Problem 6a.

- there are no weights on the vertices;
- for every vertex u, all outgoing edges (u, v) in G' get weight $w(u)$, i.e., the weight of the vertex u in the original graph G.

Now we run Dijkstra's algorithm on the graph G' with the start vertex s and report the distances and shortest path tree for G' as our answer.

To see that this is correct, note that all paths P in G and G' are the same, and by construction the "vertex cost" of P in G is equal to the standard "edge cost" of P in G'. Also, we are guaranteed that all vertex weights in G are non-negative, so the edge weights in G' are non-negative as required by Dijkstra's algorithm.

As for the time complexity of our algorithm, it is clear that the modified graph G' can be build in time $\mathrm{O}\big(|V(G)|+|E(G)|\big)$. and Dijkstra's algorithm will run in the same time complexity $\mathrm{O}\big(|E(G)|\log|V(G)|\big)$ as we are used to (assuming an implementation with a min-heap for the priority queue).

6 (90 p) In this problem we wish to understand how to compute topological orderings and/or strongly connected components of directed graphs.

6a (30 p) Consider the graph H in Figure 5. The graph is again given to us with out-neighbour adjacency lists sorted in lexicographic order, so that this is the order in which vertices are encountered when going through neighbour lists. Run the topological sort algorithm on H and explain the details of the execution analogously to how we did this in class (including which recursive calls are made). Report what the resulting topological ordering is.

Solution: In order to sort the graph H topologically, we should run a depth-first search and output the vertices in *decreasing* order with respect to finishing times. Also, if we ever from some vertex u visit a neighbour v that has been discovered but has not been finished, then (u, v) is a back edge, meaning that the graph contains a cycle and cannot be topologically sorted.

In Figure 6 we report discovery and finishing times from our depth-first search on H, where the outer loop iterates over all vertices in H in alphabetical order starting in vertex a. The execution proceeds as follows (where we note that in exam solutions there is absolutely no need to write exquisitely phrased complete sentences as below—you can be much briefer as long as it is perfectly clear what you mean):

1. We visit a at time 1 and start processing its out-neighbour list.

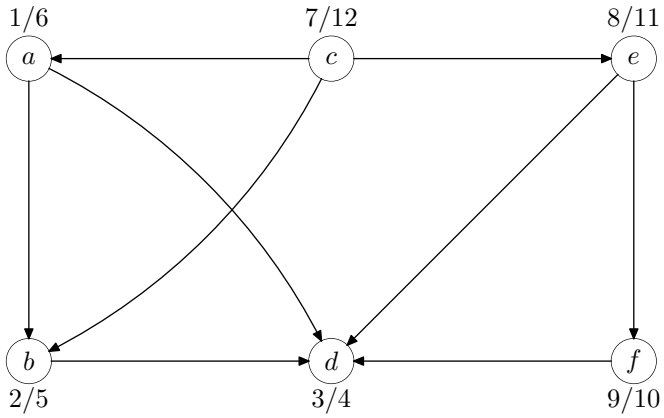


Figure 6: Directed graph H in Problem 6a. with DFS tree and start and finishing times.

2. The first out-neighbour b of a has not been discovered, so we visit b recursively at time 2 and start processing its neighbour list.

3. The first out-neighbour d of b has not been discovered, so we visit d recursively at time 3. Since d has no out-neighbours, we finish processing d at time 4 and return to the visit call for b.

4. Since b has no out-neighbours except for d, we finish processing b at time 5 and return to the visit call for a.

5. The second out-neighbour d of a has already been visited and finished (so (a, d) is not a back edge). Since a has no further out-neighbours, we finish processing a at time 6 and return to the loop iterating over all (non-visited) vertices in the graph.

6. The first non-visited vertex in alphabetical order is c, so we visit c at time 7 and start processing its neighbour list.

7. The first two out-neighbours a and b of c have already been visited and finished (and so do not yield back edges), but the third out-neighbour e has not been discovered, so we visit it recursively at time 8 and start processing its neighbour list.

8. The first out-neighbour d of e have already been visited and finished, but the second neighbour f has not been discovered, so we visit it recursively at time 9. Since the only out-neighbour d of f has already been discovered and finished, we finish processing f at time 10 and return to the visit call for e.

9. Since all out-neighbours of e have now been processed, we finish processing e at time 11 and return to the visit call for c.

10. Since all out-neighbours of c have now been processed, we finish processing c at time 12. Since all vertices in H have now been visited, this terminates the depth-first traversal of the graph.

Listing the vertices of the graph in decreasing order of finishing times (as shown in Figure 6) now yields the topological ordering c, e, f, a, b, d, (which can easily be verified to be a correct ordering, in case we want to double-check that we did not make any mistakes along the way).

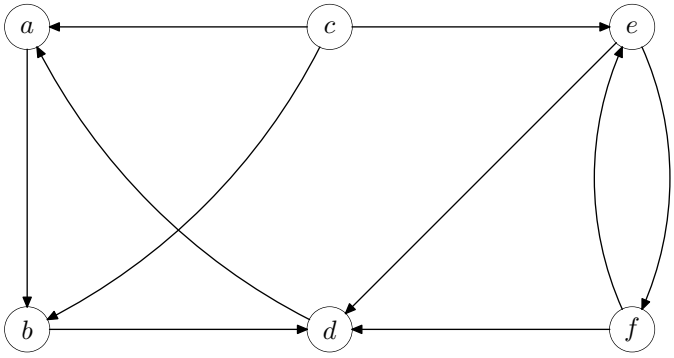


Figure 7: Directed graph G for which to compute contractions in Problem 6b.

6b (10 p) In this subproblem, we want develop our ability to parse new definitions of operations on graphs and apply these operations.

Suppose that $G = (V, E)$ is any directed graph and let $U \subseteq V$ be a subset of the vertices. Let the *contraction of G on U* be the graph $\mathsf{contract}(G, U) = (V', E')$ defined on the vertex set

$$V' = (V \setminus U) \cup \{v_U\},$$

where v_U is a new vertex, and with edges

$$\begin{aligned} E' = \quad & \big(E \cap ((V \setminus U) \times (V \setminus U))\big) \\ & \cup \big\{(v_U, w) \,\big|\, w \in V \setminus U \text{ and } \exists u \in U \text{ such that } (u, w) \in E\big\} \\ & \cup \big\{(w, v_U) \,\big|\, w \in V \setminus U \text{ and } \exists u \in U \text{ such that } (w, u) \in E\big\}. \end{aligned}$$

Demonstrate that you understand this definition by drawing $\mathsf{contract}(G, U_1)$ for the graph G in Figure 7 on page 13 and for $U_1 = \{a, b, d\}$ and also explaining how you constructed the graph $\mathsf{contract}(G, U_1)$. Also, draw $\mathsf{contract}(G, U_2)$ for the same graph G in Figure 7 and $U_2 = \{e, f\}$ and explain how you constructed this graph.

Solution: Just by parsing the definitions in the problem statement, we see that the "contracted graph" $\mathsf{contract}(G, U)$ should be constructed as follows:

- All vertices in U should be replaced by a new "supervertex" v_U, but the rest of the vertices in $V \setminus U$ are unchanged.
- Any edges outside of U, i.e., edges in $E \cap ((V \setminus U) \times (V \setminus U))$, are left unchanged.
- Any edges between two vertices in U disappear.
- Any outgoing edge from some $u \in U$ to some $w \in V \setminus U$ is replaced by an outgoing edge (v_U, w) from the vertex v_U. (If there are such edges for several different $u \in U$, only one edge is generated from v_U in the contracted graph.)
- Any incoming edge to some $u \in U$ from some $w \in V \setminus U$ is replaced by an incoming edge (w, v_U) to the vertex v_U. (If there are such edges for several different $u \in U$, only one edge is generated to v_U in the contracted graph.)

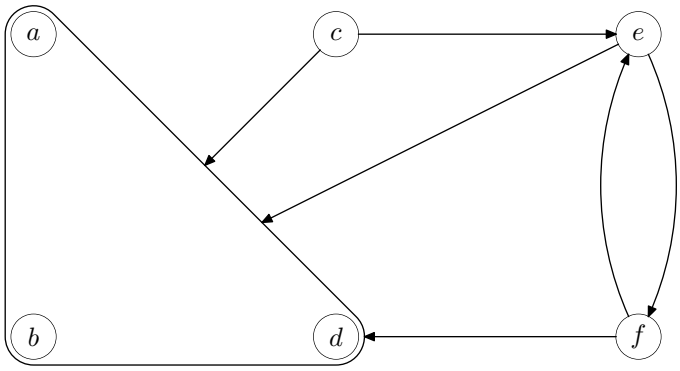


Figure 8: Contracted graph $\mathsf{contract}(G, U_1)$ for G in Figure 7 and $U_1 = \{a, b, d\}$.

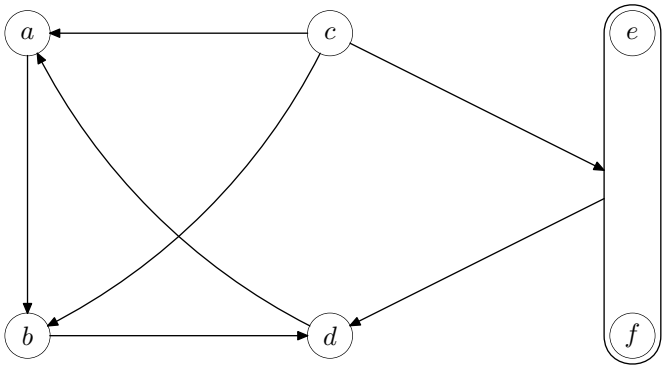


Figure 9: Contracted graph $\mathsf{contract}(G, U_2)$ for G in Figure 7 and $U_2 = \{e, f\}$.

To get a full score for Problem 6b we also need to show that we can apply this definition for two concrete examples as requested in the problem statement, so let us do this.

The graph $\mathsf{contract}(G, U_1)$ obtained from G in Figure 7 and $U_1 = \{a, b, d\}$ is depicted in Figure 8. There are no outgoing edges from $U_1 = \{a, b, d\}$ to any other vertices in the graph, but there are incoming edges to U_1 from c (to both a and b, but these edges are collapsed to a single edge) as well as from e and f (in both cases to d). Hence, we obtain edges $\big(c, v_{U_1}\big)$, $\big(e, v_{U_1}\big)$, and $\big(f, v_{U_1}\big)$.

For the graph $\mathsf{contract}(G, U_2)$ contracted on $U_2 = \{e, f\}$ we have that there is an incoming edge to U_2 from c (to e) and an outgoing edge to d (since there are edges from both e and f, but these are collapsed to a single edge in the contracted graph). This yields the contracted graph shown in Figure 9.

6c (50 p) Consider the following idea for an algorithm for computing the strongly connected components of a directed graph $G = (V, E)$:

1. Set $G' = G$ and label every vertex $v \in V$ by the singleton vertex set $\mathsf{vlabels}(v) = \{v\}$. Set s to be the first vertex of G (sorted in alphabetical order, say).
2. Run topological sort on G' starting with the vertex s. If this call succeeds, output $\{\mathsf{vlabels}(v) \mid v \in V(G')\}$ as the strongly connected components of G and terminate.
3. Otherwise, fix the back edge that made the topological sort fail and use this edge to find a cycle C in G'.

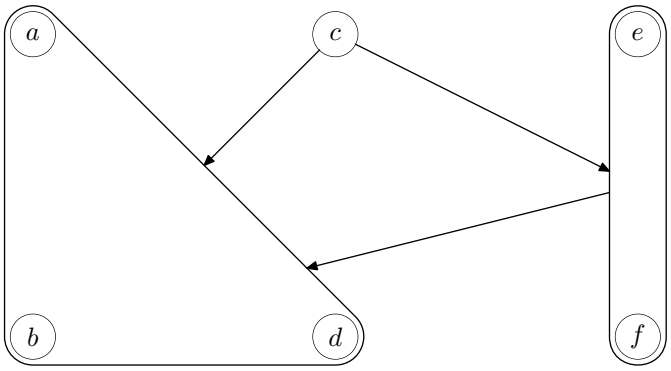


Figure 10: Fully contracted graph obtained from G in Figure 7 at end of algorithm in Problem 6c.

4. Update G' to $G' = \mathsf{contract}(G', C)$ and label the new vertex v_C by $\mathsf{vlabels}(v_C) = \bigcup_{u \in C} \mathsf{vlabels}(u)$.
5. Set s to be the new vertex v_C and go to 2.

Does this algorithm compute strongly connected components correctly? Argue why it is correct or provide a simple counter-example. (For partial credit, you can instead run the algorithm on the graph in Figure 7 and report what it does for that particular graph.)

Regardless of what the algorithm does, is it guaranteed to terminate, and if so what is the time complexity? Will the algorithm run faster or slower than the algorithm for strongly connected components that is covered in CLRS and the lecture notes?

Solution: Let us start by running the algorithm on the graph G in Figure 7, as suggested in the problem statement, to develop some intuition. Here is what happens:

- We first make a depth-first search starting in vertex a. From a we visit vertex b, from where we recursively visit vertex d, where we discover a back edge to a. We have found a cycle $a \to b \to d \to a$, and therefore contract the graph on $U_1 = \{a, b, d\}$ to obtain the new graph $\mathsf{contract}(G, U_1)$, which very conveniently is already illustrated in Figure 8. The label of the new vertex v_{U_1} is the vertex set U_1.
- The next DFS call starts by visiting the vertex v_{U_1}, which has no out-neighbours. We next visit c, which recursively visits e, which recursively visits f, where a back edge is discovered. We have found another cycle $e \to f \to e$, and therefore contract the graph $\mathsf{contract}(G, U_1)$ further on $U_2 = \{e, f\}$ to obtain the graph $\mathsf{contract}\big(\mathsf{contract}(G, U_1), U_2\big)$ in Figure 10. Again, the label of the new vertex v_{U_2} is the set U_2.
- The final DFS call discovers no cycles, so the proposed algorithm outputs that the strongly connected components of the original graph are $\{a, b, d\}$, $\{c\}$, and $\{e, f\}$.

By visual inspection we can see that the algorithm works correctly for our particular example graph. We now want to argue that this is in fact the case also in general, after which we will analyse the time complexity of our new algorithm (which—spoiler alert—will be significantly slower than the one we learned during the course).

It follows straight from the definition that two vertices u and v in a graph are in the same strongly connected components if and only if there are paths from u to v and from v to u. Based on this, we can make the following observations about the proposed algorithm (which can be

formally established by induction over the number of contraction operations on the graph where required):

1. For any graph G' encountered during algorithm execution, it holds that the collection of sets $\{\mathsf{vlabels}(v') \mid v' \in V(G')\}$ forms a partition of the vertices in the original graph G.

2. For any vertex v' in any graph G' encountered during algorithm execution, it holds that all vertices in $\mathsf{vlabels}(v')$ should be in the same strongly connected component of the original graph G. This is so since there are paths in both directions between any pair of vertices in $\mathsf{vlabels}(v')$ by construction.

3. For any graph G' encountered during algorithm execution, it follows that for any pair of distinct vertices u' and v' in G' and any original vertices $u \in \mathsf{vlabels}(u')$ and $v \in \mathsf{vlabels}(v')$, if there is a path from from u to v in the original graph G, then there is a path from from u' to v' in the current contracted graph G'. This is so since graph contraction operations maintain such connectivity.

4. When the algorithm terminates, there are no cycles in the current contracted graph G'. This is so since the topological sorting in step 2 has been successful.

5. It follows from items 3 and 4 that for any pair of distinct vertices u' and v' in the final graph G' and any original vertices $u \in \mathsf{vlabels}(u')$ and $v \in \mathsf{vlabels}(v')$, either there is no path from u to v or there is no path from v to u, and so any two such vertices should not be in the same strongly connected component of G.

We can conclude from items 1, 2 and 5 that when the algorithm terminates it holds that the collection of vertex sets $\{\mathsf{vlabels}(v') \mid v' \in V(G')\}$ does indeed contain the strongly connected components of the original graph G. This concludes our argument that the proposed algorithm is correct.

Regarding the time complexity of the proposed algorithm—which we can determine without reasoning about correctness—we can first note that the algorithm must terminate, since the number of vertices in the graph decreases in between calls to the topological sort in step 2.

The initialization in step 1 takes time $\mathrm{O}(|V|)$. If n' is the number of vertices and m' is the number of edges in the current graph G', then step 2 runs in time $\mathrm{O}(n' + m')$, and it is not hard to argue that steps 3 and 4 can also be implemented in this time, whereas step 5 takes constant time. (We do not require any detailed argument for this.) In a worst-case scenario, each graph contract operation could eliminate only a constant number of vertices and edges. In this case steps 2–5 will be executed $\Theta(|V|)$ times on graphs with $\Theta(|V|)$ vertices and $\Theta(|E|)$ edges, in which case the overall running time will be $\mathrm{O}(|V|(|V|+|E|))$. This is worse by a linear factor $|V|$ compared to the algorithm that we learned during the course, which runs in time $\mathrm{O}(|V| + |E|)$.

7 (60 p) Consider the regular expression $(a(bb|c^*)d^*)^*$ and determine which of the strings 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. *bbacccd*

2. *abbdaccc*

3. *aaaccc*

4. *abbcccd*

5. *abbddcccddd*

6. *abbabba*

Solution: We give 10 p per fully correct and satisfactorily motivated answer. It stands to reason that any such answer will have to involve a discussion of how how to interpret the regular expression (in order to argue why accepted strings are accepted), so let us start by explaining this.

Because of the outermost star *, the regular expression $(a(bb|c^*)d^*)^*$ accepts a concatenation of zero or more strings on the following form:

- First comes exactly one character a.
- Then comes either the string bb or zero or more repetitions of the character c.
- Finally, we have zero or more repetitions of the character d.

With this in mind, we get the following answers:

1. *bbacccd* ✗ (No string accepted by the regular expression can start with the character b.)
2. $abbdaccc = (abbd)\big(ac^3d^0\big)$ ✓
3. $aaaccc = \big(ac^0d^0\big)\big(ac^0d^0\big)\big(ac^3d^0\big)$ ✓
4. *abbcccd* ✗ (No string accepted by the regular expression can have a character b directly followed by a character c—there has to be a character a in between.)
5. *abbddcccddd* ✗ (No string accepted by the regular expression can have a d directly followed by a c—there has to be an a in between.)
6. $abbabba = \big(abbd^0\big)\big(abbd^0\big)\big(ac^0d^0\big)$ ✓

8 (90 p) Consider the following context-free grammars, where a, b, c are terminals, S, T, U are non-terminals, and S is the starting symbol.

Grammar 1:

$$S \to TaT \tag{7a}$$
$$T \to bT \tag{7b}$$
$$T \to ccT \tag{7c}$$
$$T \to \tag{7d}$$

Grammar 2:

$$S \to TaT \tag{8a}$$
$$T \to bTb \tag{8b}$$
$$T \to U \tag{8c}$$
$$U \to cU \tag{8d}$$
$$U \to \tag{8e}$$

Grammar 3:

$$S \to aTa \tag{9a}$$
$$T \to TbT \tag{9b}$$
$$T \to U \tag{9c}$$
$$U \to cUc \tag{9d}$$
$$U \to \tag{9e}$$

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 fully correct and satisfactorily motivated answer.

In **Grammar 1**, the non-terminal T generates any (possibly empty) alternation of b and cc, which corresponds to the regular expression $(b|cc)^*$. The first production $S \to TaT$ just means that somewhere in the string produced there has to appear a single a. It follows that the language generated by Grammar 1 is captured by the regular expression $(b|cc)^*a(b|cc)^*$.

Grammar 2 can generate strings on the form $ab^nc^mb^n$ for $m, n > 0$ by first applying $S \to TaT$ and then getting rid of the first T by using $T \to U$ and $U \to$. The second T will now have to generated a balanced number of characters b since the only way of generating this character is by using $T \to bTb$, and the productions $T \to U$ and $U \to cU$ can then produce a non-zero number of characters c in between the two sequences of bs. It follows by the same line of reasoning that the grammar *cannot* generate, for example, strings $ab^nc^mb^{n+1}$. In order to distinguish between strings $ab^nc^mb^n$ and $ab^nc^mb^{n+1}$ we need to be able to count, and this is nothing that finite automata can do. Hence, the language generated by Grammar 2 is *not* regular.

Note that although you do not need to prove that finite automata cannot count, but can refer to Mogesen's notes for this, some care is nevertheless needed to get a correct counting argument. To see one example of a faulty argument, note that it is also true that Grammar 2 can generate strings $(bb)^na(bb)^n$, but it is hard to get a counting argument out of this since also strings $(bb)^ma(bb)^n$ for $m \neq n$ can be generated. Hence, for this type of strings we cannot claim that counting is required, other than for keeping track of odd or even numbers of b, which a regular expression can certainly do.

In **Grammar 3**, the first production $S \to aTa$ makes sure that the string will start and end with a, and that everything in between these two characters a will be generated by the non-terminals T and U. Looking at the non-terminal U, the productions $U \to cUc$ and $U \to$ just mean that U produces an even number of characters c (possibly zero). In view of this, the productions $T \to TbT$ and $T \to U$ will just generate any (possibly empty) alternation of b and cc, which corresponds to the regular expression $(b|cc)^*$. We can conclude that the whole language generated by Grammar 3 is captured by the regular expression $a(b|cc)^*a$.