KØBENHAVNS UNIVERSITET

Diskret Matematik og Formelle Sprog: Exam April 12, 2023
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. 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 (80 p) Consider the snippet of code

```
check (A, lo, hi)
    mid := floor ((lo + hi) / 2)
    success := TRUE
    i := lo
    while (i < mid and success)
        if (A[i] > A[mid])
            success := FALSE
        i := i + 1
    i := mid + 1
    while (i <= hi and success)
        if (A[i] < A[mid])
            success := FALSE
        i := i + 1
    if (lo < mid - 1 and success)
        success := check (A, lo, mid - 1)
    if (mid + 1 < hi and success)
        success := check (A, mid + 1, hi)
    return success
```

where `A` is an array indexed from 1 to `A.size` that contains elements that can be compared, and the function `floor` rounds down to the nearest integer.

1a (30 p) Explain in plain language what the result is of an algorithm call `check (A, 1, A.size)` (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, what holds when the algorithm returns `TRUE`?

Solution: This is a recursive algorithm. It first computes an index mid so that $A[\mathsf{mid}]$ is the midpoint of the array lo and hi. Then follow two while loops. In the first while loop all elements $A[i]$ for $\mathsf{lo} \leq i < \mathsf{mid}$ are compared to $A[\mathsf{mid}]$, and if it ever holds that $A[i] > A[\mathsf{mid}]$,

then the Boolean flag success is set to `FALSE` (which will cause the algorithm to return `FALSE` at the end). The second while loop checks in the same way for all indices i such that $\mathsf{mid} < i \leq \mathsf{hi}$ that $A[i] \geq A[\mathsf{mid}]$, or else success is set to `FALSE`.

After the two while loops, it follows from the discussion above that if the Boolean flag success is still `TRUE`, then $A[\mathsf{mid}]$ is greater than or equal to all elements preceding it in the array, and smaller than or equal to all elements succeeding it. That is, although we might not know much about the rest of the array, we know that in the array between $A[\mathsf{lo}]$ and $A[\mathsf{hi}]$ the element at $A[\mathsf{mid}]$ is in the right position if we would wish to sort the array in increasing order.

If there is more than one element between indices lo and $\mathsf{mid} - 1$ (inclusive) and if success is `TRUE`, then a recursive call is made on the subarray between these two positions, which will check in the same way if the middle element in the left subarray is in the right position with respect to sortedness. If no recursive call is made but success is `TRUE`, then we know from the discussion above that all elements between indices lo and mid are sorted. In the same way, if there are at least two elements between indices $\mathsf{mid}+1$ and hi (inclusive), a recursive call is made, and otherwise we know that all elements between indices mid and hi are sorted (assuming that success is `TRUE`), After all recursive calls (if any) have been made, we have that success is `TRUE` if all recursive checks yielded a positive result, meaning that all elements in the whole array were in the correct position with respect to sortedness in increasing order. Thus, the algorithm call `check (A, 1, A.size)` returns `TRUE` if and only if the whole array is sorted.

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

Solution: To analyze the time complexity of the algorithm, we try to understand how much work is done per call and how many recursive calls are made.

Starting with the latter, we can see that every call of the algorithm generates at most two subcalls, and that these calls are made on arrays of (slightly less than) half the size. This means that if we let $n = \texttt{A.size}$, then there will be a at most $\lceil \log n \rceil$ levels of recursion.

If we can analyse the time needed for each recursive call, then we can sum up to get the total time complexity. All lines inside the `check` algorithms run in constant time except for the while loops (and the recursive calls, but that time is accounted for in the analysis of each recursive call). All statements inside the while loops are also constant time. The while loops together iterate over all elements in the array (except mid) and so the total time required for these iterations, and hence for the full call, is linear in $\mathsf{hi} - \mathsf{lo}$, scaling like $K(\mathsf{hi} - \mathsf{lo})$ for some (large enough) constant K.

Summing up, we get:

- At top level, we have one call taking time at most $K \cdot n$,
- At recursion level 1, we have at most two calls taking time at most $K \cdot n/2$ each, for a total cost of $(K \cdot n/2) + (K \cdot n/2) = K \cdot n$ for all calls at level 1.
- At recursion level 2, we have at most four calls taking time at most $K \cdot n/4$ each, summing up to a cost of at most $K \cdot n$ for all calls at level 2.
- Et cetera, all the way down to recursion level $\lceil \log n \rceil$, where we will get at most a linear number of calls, each taking constant time.

We see that we have $\mathrm{O}(\log n)$ levels of recursion, and that the total cost of the recursive calls at each level is $\mathrm{O}(n)$. Hence, the time complexity of the algorithm is $\mathrm{O}(n \log n)$ (and it is not hard to see that this is also a lower bound, so that the time complexity is in fact $\Theta(n \log n)$, but this is not needed for a full score).

1c (20 p) Can you improve the algorithm (i.e., change the pseudocode) to run asymptotically faster while retaining the same functionality? In case you can design an asymptotically faster algorithm, analyse the time complexity of your new algorithm. For the best algorithm that you have—either your new one, or the old one—can you prove that the running time of this algorithm is asymptotically optimal?

Solution: Since by analyzing the algorithm we have determined that it is simply checking whether the array is sorted in increasing order, it is not hard to write a more efficient algorithm performing this check. For instance, the following pseudocode should do the job:

```
check (A, lo, hi)
    success := TRUE
    i := lo
    while (i < hi and success)
        if (A[i] > A[i+1])
            success := FALSE
        i := i + 1
    return success
```

This check runs in linear time, which is clearly asymptotically optimal since we have to look at the full array to know for sure whether it is sorted or not.

2 (60 p) Post-pandemic life in academia has meant noticeable changes for both Jakob and his family, in ways both good and bad.

2a Last autumn Jakob attended his first non-virtual international conference in a very long time (since February 2020, to be precise), and the conference had more than 100 participants. A colleague at the conference pointed out to Jakob that this meant that either there were participants from more than 10 different countries, or else more than 10 people from some particular country attended the conference. Jakob, who had not studied the list of participants that closely, was amazed at this claim. Can you explain to Jakob, without even having looked at the list of participants, why the claim has to be true?

Solution: If there are participants from more than 10 countries, then there is nothing to prove. Suppose therefore that the strictly more than 100 participants are coming from at most 10 different countries. Then the pigeonhole principle (with participants as pigeons and countries as pigeonholes) says that at least 11 particpants are coming from the same country.

2b While Jakob was away, he suggested to his children that they should entertain themselves at home with the following game: Write down the numbers 1 to 20 on a sheet of paper. Erase any two distinct numbers a and b and replace them by the number $a + b - 1$. Now do the same again with all numbers currently on the sheet, i.e., pick any two members a' and b' of the multi-set $\big(\{1, 2, \ldots, 20\} \setminus \{a, b\}\big) \cup \{a+b-1\}$ of all numbers between 1 and 20 except a and b plus the new number $a+b-1$, and replace the two chosen numbers by their sum minus one $a' + b' - 1$. Repeat this procedure until there is only a single number left on the sheet of paper. The children found this game a little bit repetitive, however. Can you describe the range of possible outcomes for this game? For a full score, provide proofs of any claims you make.

Solution: The sum of all numbers on the paper before the game starts is $\sum_{i=1}^{20} i = \frac{20 \cdot 21}{2} = 210$. After the first two numbers a and b have been erased and replaced by $a + b - 1$, the sum of

all numbers is $210 - 1 = 209$, and this holds regardless of which numbers a and b were chosen. When in the second step two new numbers a' and b' are erased and replaced by $a' + b' - 1$, the sum changes to $(210 - 1) - 1 = 210 - 2 = 208$, and again this fact does not depend on the concrete choice of a' and b'. Continuing this reasoning, we see that it is an invariant that after i numbers have been erased, the sum of the numbers currently written on the sheet of paper must be $210 - i$. The game ends after 19 numbers have been erased, at which point the final remaining number (which is also the sum of all numbers currently written on the sheet of paper) is $210 - 19 = 191$.

If we would want to set up a formal induction proof, then we could pick as our induction hypothesis that "after i numbers have been erased, the sum of the numbers currently written on the sheet of paper is $210 - i$", and then prove the base case and induction step as per the reasoning above.

Note that it was clearly stated in the problem that we considered *multi-sets*, and that the process should continue until there was a single number left on the sheet of paper. Therefore, solutions that consider the process to have terminated when there are several numbers of the sheet of paper which are all the same did not parse the problem statement correctly (but have still been awarded a fairly generous number of points if the problem was otherwise solved correctly under this erroneous interpretation).

3 (40 p) Let $a_1, a_2, a_3, \ldots$ be a sequence of numbers defined by

$$a_n = \begin{cases} 5 & \text{if } n = 1, \\ 13 & \text{if } n = 2, \\ 5a_{n-1} - 6a_{n-2} & \text{if } n > 2. \end{cases}$$

Prove that for all positive integers n it holds that $a_n = 2^n + 3^n$.

Solution: We prove the claim by induction. Note that in the induction step we will need to talk about both a_{n-1} and a_{n-2}, and therefore our induction hypothesis will need to assume that both a_{n-1} and a_{n-2} are on the right form. For the same reason, we need two base cases.

Base case ***(*** $n = 1$ ***and*** $n = 2$***):*** We have $a_1 = 5 = 2^1 + 3^1$ and $a_2 = 13 = 2^2 + 3^2$.

Induction step: Suppose that $a_{n-2} = 2^{n-2} + 3^{n-2}$ and $a_{n-1} = 2^{n-1} + 3^{n-1}$, and consider $a_n = 5a_{n-1} - 6a_{n-2}$. Calculating, we get

$$\begin{aligned} a_n &= 5a_{n-1} - 6a_{n-2} && \text{[by definition]} \\ &= 5\big(2^{n-1} + 3^{n-1}\big) - 6\big(2^{n-2} + 3^{n-2}\big) && \text{[by the induction hypothesis]} \\ &= 2 \cdot 2^{n-1} + 3 \cdot 2^{n-1} + 3 \cdot 3^{n-1} + 2 \cdot 3^{n-1} \\ &\qquad - 3 \cdot 2 \cdot 2^{n-2} - 2 \cdot 3 \cdot 3^{n-2} && \text{[regrouping]} \\ &= 2^n + 3^n + 3 \cdot 2^{n-1} + 2 \cdot 3^{n-1} - 3 \cdot 2^{n-1} - 2 \cdot 3^{n-1} \\ &= 2^n + 3^n \end{aligned}$$

as desired, and so the induction step goes through. The claimed equality now follows by the induction principle.

One remark is that instead of having an induction hypothesis with claims about both a_{n-1} and a_{n-2}, we could simply use strong induction. We would still need to establish two base cases for the strong induction to go through, though.

Another comment is that it is also possible to solve this problem by using material from Section 3.5 on recurrence relations in the KBR textbook (more specifically, Theorem 1 on page 115).

Section 3.5 in KBR was not part of the required reading for the course, but a fully correct solution using this approach yielded a full score.

4 (100 p) Recall that a graph $G = (V, E)$ consists of a set of vertices V connected by edges E, where every edge is a pair of vertices. If there is an edge (u, v) between two vertices u and v, then the two vertices are said to be *neighbours* and are both *incident* to the edge. We say that a sequence of edges (v_1, v_2), (v_2, v_3), (v_3, v_4), $\ldots$, (v_{k-1}, v_k), in E is a *path* from v_1 to v_k.

In this problem, we wish to express properties of graphs in both natural language and predicate logic, and to translate between the two forms. We do this as follows:

- The universe is the set of vertices V of G.
- The binary predicate $E(u, v)$ holds if and only if there is an edge from u to v in G.
- The unary predicate $S(v)$ is used to identify a subset of vertices $S = \{v \mid v \in V, S(v) \text{ is true}\}$ for which some property might or might not hold.

Below you find five graph properties written as predicate logic formulas and six graph properties defined in natural language. Most of the predicate logic formulas have corresponding natural language definitions, but not all.

Your task is to determine which of the predicate logic formulas (a), $\ldots$, (e) match which—if any of the natural language definitions (1), $\ldots$, (6). Please make sure to motivate your answers clearly.

Predicate Logic Formulas:

(a) $\forall u \forall v \big(E(u, v) \rightarrow E(v, u)\big)$

(b) $\forall u \forall v \big(E(u, v) \rightarrow S(u) \vee S(v)\big)$

(c) $\forall u \forall v \exists w \big(S(w) \wedge (E(u, w) \vee E(v, w))\big)$

(d) $\forall u \forall v \big(E(u, v) \rightarrow ((S(u) \wedge \neg S(v)) \vee (\neg S(u) \wedge S(v)))\big)$

(e) $\forall u \forall v \big((u \neq v \wedge S(u) \wedge S(v)) \rightarrow E(u, v)\big)$

Natural Language Definitions:

(1) S is a *dominating set* in G, i.e., every vertex v in the graph either is in S or is a neighbour of a vertex in S.

(2) S is a *clique* in G, i.e., a set of vertices that are all neighbours with each other.

(3) The graph G is *undirected*.

(4) The graph G is *connected*.

(5) S is a *vertex cover* in G, i.e., for every edge at least one of the vertices incident to it is in S.

(6) The graph G is *bipartite* with bipartition $(S, V \setminus S)$, i.e., all edges go between S and $V \setminus S$.

Solution: Formula (a) matches the description (3) of the graph being undirected. The formulas says that whenever $E(u, v)$ holds, then $E(v, u)$ also holds. This is to say that the ordering of the vertices in an edge does not matter, but that "the edge exists in both direction"—i.e., that the graph is undirected.

Formula (b) matches the description (5) of vertex cover. The formula says that if there is an edge between two vertices u and v, then at least one of these vertices is in S (i.e., the set defined by the unary predicate $S(\cdot)$). This is the definition of S being a vertex cover.

Formula (c) says that for any pair of vertices u and v, there is a vertex w that is in the set S and that is a neighbour of either u or v. This does not match any of the natural language descriptions.

Formula (d) matches the description (6) of bipartiteness. It says that if there is an edge between u and v, then either u is in S and v is not, or the other way round. That is, exactly one of the vertices is any edge is in S, which shows that $(S, V \setminus S)$ is indeed a bipartition of the graph.

Formula (e) matches the description (2) of clique. It says that if two distinct vertices v and w are both in S, then there has to be an edge between them. This is exactly the condition for vertices in a clique.

5 (60 p) When Jakob is socializing in the evenings after conferences in the United States, he tries to convince his colleagues that poker should really be played with a *republican deck of cards*, i.e., with cards of all ranks 2–10 plus the aces, but without kings, queens, or jacks.

5a Suppose that you are dealt 5 cards from a perfectly shuffled republican deck of cards. What is the probability of getting a *full house*, i.e., three of a kind (three cards of the same rank) with a pair (two other cards of the same rank)? Explain clearly how you obtain the expression in your answer.

Solution: In a "republican deck of cards" we have 10 ranks and 4 cards in each rank, for a total of 40 cards in the deck. The total number of hands of 5 cards is therefore $\binom{40}{5}$. All of these hands are equally likely (assuming a perfectly shuffled deck of cards).

In order to determine the probability of getting a full house, we therefore only need to count all (distinct) ways of getting a full house, and then divide by the total number of hands, to get the probability. In order to get a full house, we first choose the first rank in one of 10 ways, and then we have to draw 3 cards of this rank, which can be done in $\binom{4}{3}$ ways. The second rank can be chosen in 9 ways, and two cards of that rank can be drawn in $\binom{4}{2}$ ways. Hence, the total number of full houses is $10 \cdot \binom{4}{3} \cdot 9 \cdot \binom{4}{2}$, and the probability of getting a full house when 5 cards are drawn randomly is

$$\frac{10 \cdot \binom{4}{3} \cdot 9 \cdot \binom{4}{2}}{\binom{40}{5}} .$$

5b It is a sad fact that so far Jakob has had quite limited success in convincing colleagues that a republican deck of cards is the true American way of playing poker. Jakob believes this is because the colleagues have all been brainwashed by the big casinos in Las Vegas, and that this in turn is because with a republican deck of cards the probability for the players of getting a strong hand (like a full house) would be higher than with a standard deck of cards, making it more likely that casinos might lose money.

Ignoring the wider ramifications of the worldwide conspiracy that Jakob alleges to have discovered, is the factual claim true that the probability of getting, e.g., a full house is

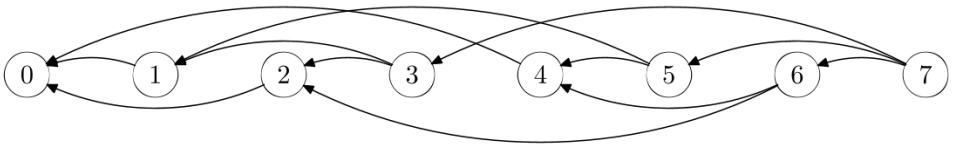


Figure 1: Directed graph D_S representing relation S in Problem 6.

higher with a republican deck of cards than with a standard deck of cards when being dealt 5 cards from a shuffled deck? (Note that you should not have to do very precise calculations to be able to determine whether this claim is true or not, and a clear intuitive explanation of which answer should be the correct one can give generous partial credits.)

Solution: Jakob actually happens to be right this time (not necessarily about the worldwide conspiracy, but about the probability). In a standard deck of cards, the probability of getting a full house is

$$\frac{13 \cdot \binom{4}{3} \cdot 12 \cdot \binom{4}{2}}{\binom{52}{5}} .$$

If we compare this with the probability above for a "republican deck of cards", then we see that the denominator grows much more when going from a republican deck to a full deck than does the numerator. Phrased differently, the number of ways of getting full houses is certainly larger with a full, standard deck of cards, but the number of getting all kinds of crazy 5-card hands increases even more with a standard deck of cards. (This can of course be verified also by performing the calculations comparing the two probabilities.)

6 (60 p) Consider the relation S described by the directed graph D_S in Figure 1.

6a (10 p) Write down the matrix representation M_S of the relation S and describe briefly but clearly how you constructed this matrix.

Solution: The matrix representation of the relation S is an 8×8 matrix (with indexing starting from row and column 0, just to be consistent with the names of the elements in the relation) 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

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

(where we note, in particular, that rom 0 is an all-zero row in the matrix, since there are no outgoing edges from the vertex labelled 0 in D_S).

6b (10 p) Let us write I to denote the inverse of the relation S. What is the matrix representation of I? Write it down and explain how you constructed it.

Solution: For the inverse relation I of S, we have that $(i, j) \in I$ if and only if $(j, i) \in S$. This means that we could generate the matrix for I by first reversing all edges in Figure 1 and then proceeding as in our solution to Problem 6a. However, a faster way of getting the same result is to use that the matrix representation of the inverse of a relation is the transpose of the matrix representing the original relation. Therefore, we have that

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

is the matrix representation of the inverse relation I.

6c (10 p) Now let T be the transitive closure of the relation I. 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 I, we can proceed as follows:

1. Start by setting $T' = T = I$.
2. Go over all triples (i, j, k) such that $(i, j) \in I$ and $(j, k) \in I$, 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.

Looking at the directed graph representation D_S in Figure 1, but reversing the edges so that we get the graph for I, a pair (i, k) should be in the transitive closure precisely when there is a path from i to k in the graph. Starting from each element $i = 0, 1, \ldots, 7$ and writing down which other vertices are reachable from i yields the matrix

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

for the transitive closure T of I.

6d (20 p) Finally, let R be the reflexive closure of the relation T. 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? Be as specific in your reply as you can.)

Solution: Although the problem statement does not ask for it, let us write down the matrix representation

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

of the reflexive closure R of T. This is the subset relation on the universe of three elements. If we write all numbers in binary, each number can be viewed as an indicator vector for a subset, so that, e.g.:

- $0 = (000)_2$ corresponds to $\emptyset$,
- $1 = (001)_2$ corresponds to $\{1\}$,
- $2 = (010)_2$ corresponds to $\{2\}$,
- $3 = (011)_2$ corresponds to $\{1, 2\}$,
- $4 = (100)_2$ corresponds to $\{3\}$,
- et cetera all the way up to $7 = (111)_2$ corresponding to $\{1, 2, 3\}$.

With this interpretation, we can verify that the relation T in Problem 6c is the strict subset relation, and taking the reflexive closure gives the relation where every subset is also related to itself.

7 (80 p) In this problem we wish to understand algorithms for minimum spanning trees.

7a (50 p) Consider the graph G in Figure 2, which is given to us in adjacency list representation, with the neighbour lists sorted so that vertices are encountered in lexicographic order. (For instance, the neighbour list of c is (a, b, d, e) sorted in that order.)

Run Prim's algorithm on the graph G, starting with the vertex a. Show the minimum spanning 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: We illustrate in Figure 3 how the heap used for the priority queue changes during the execution of Prim's algorithm. We 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.

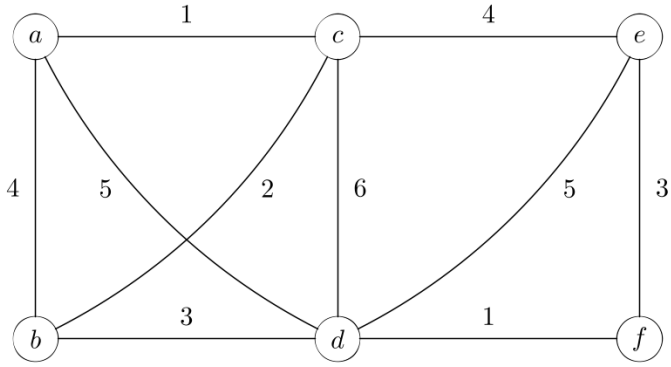


Figure 2: Graph G on which to run Prim's algorithm in Problem 7a.

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) gives value 4 to b (i.e., , the weight of the edge), and so shifts b to the top and pushes f down below b, yielding Figure 3c. Relaxing (a, c) decreases the key of c to 1 and so swaps b and c, yielding Figure 3d. Finally, relaxing (a, d) decreases the key of d to 5, which causes d to bubble up and f to bubble down, resulting in the heap in Figure 3e. There are no further outgoing edges from a to relax.

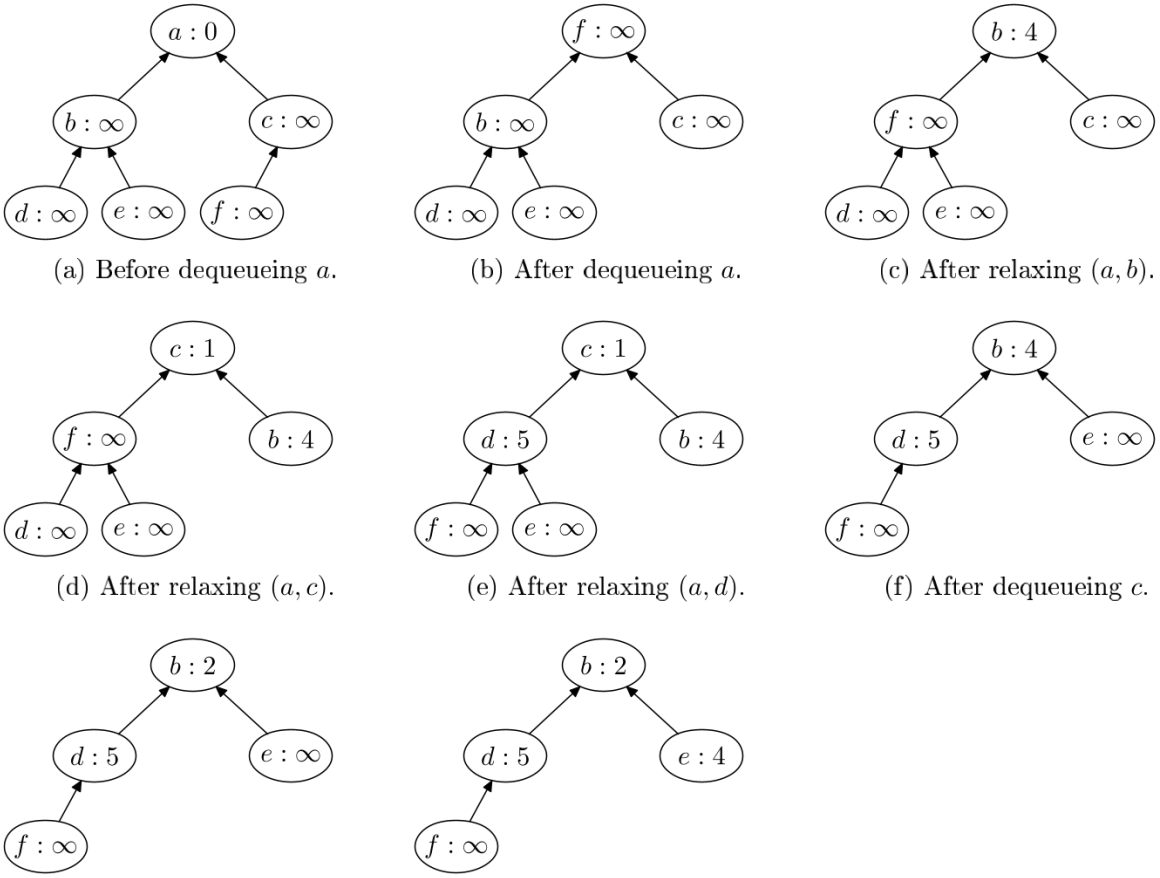
2. Since c is now at the top of the heap, it is the vertex dequeued next. This will add the edge (a, c) to the spanning tree, which we indicate in Figure 4. When c is removed, e is moved to the top. This violates the min-heap property, since the key of e is not smaller than or equal to those of its children. Since b has smaller key than d, 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 3f.

   Relaxing the edge (c, b) decreases the key value of b to 2, which is the qeight of the edge. Since b is already the root, the heap does not change except for this key update and now looks like in Figure 3g. Relaxing (c, d) has no effect, since the key value is smaller than the edge weight. Relaxing (c, e) updates the key of e to 4 but does not change the structure of the heap, since the parent b of e has a smaller key (see Figure 3h). There are no further edges incident to c to relax. 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 spanning tree (since the latest update of the key value of b was when relaxing the edge (c, b)). The heap now has e at the root with children d and f. When we relax (b, d), the key of d becomes smaller than that of e, and so these two vertices swap places. Other than that, all neighbours of b have already been dequeued and there are no edges to relax.

4. Vertex d now has the smallest key 3 and is dequeued next. This adds the edge (b, d) to the spanning paths tree, since the key of d was last updated when (b, d) was relaxed. Relaxing (d, e) does not lead to any key decrease, but relaxing (d, f) updates the key of f to 1, meaning that e and f trade places in the heap.

5. Now vertex f has the smallest key 1 and so is dequeued, adding (d, f) to the spanning tree. When we relax (f, e) the key of e decreases to 3.



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

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

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

Figure 3: Heap configurations for priority queue in Prim's algorithm in Problem 7a.

6. Finally, vertex e is dequeued, adding the just relaxed edge (f, e) to the spanning tree. The queue is now empty, and there is nothing to relax.

The minimum spanning tree computed as described above is indicated by the bold edges in Figure 4.

7b (30 p) Jakob has designed a fairly nifty MST algorithm that works as follows for $G = (V, E)$:

1. Initialize $T = \emptyset$ and $S = \{v\}$, where v is a randomly chosen vertex of the graph.
2. Iterate $|V| - 1$ times:
   (a) Let w be the vertex most recently added to S.
   (b) Among all edges from w to vertices x not yet added to S, pick the edge (w, x) with lowest cost and add to T, and add x to S with predecessor w.
   (c) If w does not have any neighbours not already in S, go back to the predecessor of w, and then to the predecessor of the predecessor, et cetera, until you find a vertex z that has at least one neighbor not already in S. Use that vertex z instead of w in this iteration.

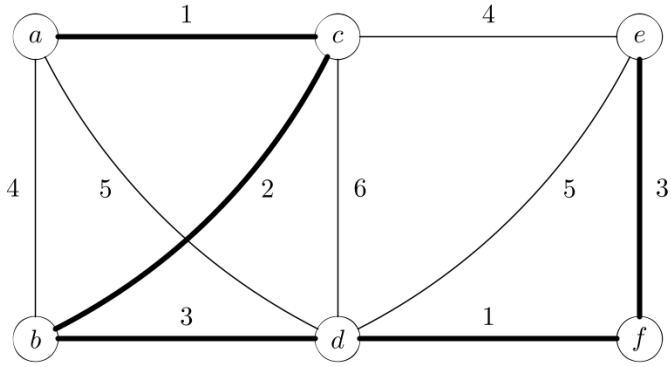


Figure 4: Graph G in Problem 7a with MST generated by Prim's algorithm.

Jakob claims that T as computed by this algorithm is a minimum spanning tree, and that the algorithm is also faster than Prim's algorithm since it runs in time $O(|V| + |E|)$. Determine whether Jakob's algorithm is valid by giving a proof of correctness or a counter-example. Regardless of whether the algorithm is correct or not, did Jakob get the time complexity analysis right? Motivate your answer clearly.

Solution: No, unfortunately, for this problem Jakob is wrong about both correctness and time complexity.

To see that the algorithm is incorrect, consider a graph on vertices $\{1, 2, 3\}$ with edges $(1, 2)$ of weight 12, $(1, 3)$ of weight 13, and $(2, 3)$ of weight 23. Suppose the algorithm starts in vertex 1, i.e., it initializes $S = \{1\}$ (which is one valid random choice for which the algorithm would have to work). Then the algorithm will pick the edge $(1, 2)$, add vertex 2 to S, and move to this new vertex. From there, it will pick edge $(2, 3)$ and add it to the set, resulting in a spanning tree. But clearly this spanning tree does not have minimal weight, since the edge $(1, 3)$ should instead be chosen together with the edge $(1, 2)$ to obtain a minimum spanning tree.

Regarding the time complexity, in order to achieve time $O(|V| + |E|)$ we would need to argue that the processing cost of each edge is constant. But this is not clear at all—in the backtracking step (c), an edge that has already been considered before could be considered again, and it is not at all clear that edges would be reconsidered only a constant number of times. On the contrary, it is not too hard to cook up examples where at least some edges are revisited a linear number of times. Therefore, without carefully chosen data structures, the time complexity looks more like $O(|V| \cdot |E|)$. With the right data structures the running time could probably be improved quite a bit, but the complexity of the algorithm as described in the problem statement is definitely not $O(|V| + |E|)$.

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:

$$S \to TU \tag{1a}$$
$$T \to aTb \tag{1b}$$
$$T \to \tag{1c}$$
$$U \to bUc \tag{1d}$$
$$U \to \tag{1e}$$

Grammar 2:

$$S \to TU \tag{2a}$$
$$T \to aT \tag{2b}$$
$$T \to bT \tag{2c}$$
$$T \to \tag{2d}$$
$$U \to bU \tag{2e}$$
$$U \to cU \tag{2f}$$
$$U \to \tag{2g}$$

Grammar 3:

$$S \to TU \tag{3a}$$
$$T \to abT \tag{3b}$$
$$T \to \tag{3c}$$
$$U \to bcU \tag{3d}$$
$$U \to \tag{3e}$$

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.

Grammar 1 does not generate a regular language. By letting U produce the empty string and focusing on T, we see that all strings $a^n b^n$ are in the language generated by the grammar but that strings $a^m b^n$ for $m \neq n$ are not. Distinguishing between such strings requires counting, which we know that regular expressions cannot do.

(Note, however, that it is *not* sufficient to say only that the production $T \to aTb$ makes the grammar produce a non-regular language, because this is not true. If we consider, for instance, the (stupid) grammar

$$T \to aTb \tag{4a}$$
$$T \to aT \tag{4b}$$
$$T \to Tb \tag{4c}$$
$$T \to \tag{4d}$$

then this grammar generates the language $\{a^m b^n \mid m, n \in \mathbb{N}\}$, which is regular.)

In **Grammar 2**, the non-terminal T generates any (possibly empty) alternation of a and b, which corresponds to the regular expression $(a|b)^*$. In the same way, U generates strings matching the regular expression $(b|c)^*$. The start symbol S of Grammar 2 generates first a string from T and then a string from U, meaning that we can concatenate to obtain the regular expression $(a|b)^*(b|c)^*$ for the language generated by Grammar 2.

Grammar 3 is quite similar in spirit to Grammar 2. The non-terminal T generates any (possibly empty) repetition of the string ab, which corresponds to the regular expression $(ab)^*$. In the same way, U generates strings matching the regular expression $(bc)^*$, and so the start symbol of Grammar 3 generates strings matching the regular expression $(ab)^*(bc)^*$.