KØBENHAVNS UNIVERSITET

# Introduktion til diskret matematik og algoritmer: Problem Set 4

**Due:** Wednesday March 26 at 17:15 CET.
**Submission:** Please submit your solutions via *Absalon* as a PDF file. State your name and e-mail address close to the top of the first page. Solutions should be written in LaTeX or some other math-aware typesetting system with reasonable margins on all sides (at least 2.5 cm). Please try to be precise and to the point in your solutions and refrain from vague statements. Never, ever just state the answer, but always make sure to explain your reasoning. *Write so that a fellow student of yours can read, understand, and verify your solutions.* In addition to what is stated below, the general rules for problem sets stated on *Absalon* always apply.
**Collaboration:** Discussions of ideas in groups of two to three people are allowed—and indeed, encouraged—but you should always write up your solutions completely on your own, from start to finish, and you should understand all aspects of them fully. It is not allowed to compose draft solutions together and then continue editing individually, or to share any text, formulas, or pseudocode. Also, no such material may be downloaded from or generated via the internet to be used in draft or final solutions. Submitted solutions will be checked for plagiarism.
**Grading:** A score of 120 points is guaranteed to be enough to pass this problem set.
**Questions:** Please do not hesitate to ask the instructor or TAs if any problem statement is unclear, but please make sure to send private messages—sometimes specific enough questions could give away the solution to your fellow students, and we want all of you to benefit from working on, and learning from, the problems. Good luck!

**1** (60 p) The purpose of this problem is to gain an understanding of strongly connected components in directed graphs.

**1a** (30 p) Compute the strongly connected components of the graph in Figure 1 by making a dry-run of the algorithm in CLRS. Make sure to explain carefully the different steps in the algorithm execution, including in which order vertices are dealt with during graph traversal and why, and also what the final output is.

We assume that 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 $(b, d, e)$ sorted in this order.)

**Solution:** The algorithm for computing the strongly connected components of a graph $G$ in CLRS and the lecture notes is as follows:

1. Run a depth-first search for $G$ in Figure 1, and note the finishing times.
2. Run a depth-first search for the inverted graph $G^T$ Figure 2, but iterate over the vertices in decreasing order of finishing times in step 1.
3. Output the subset of vertices in each tree in the forest constructed in step 2 as the strongly connected components of $G$.

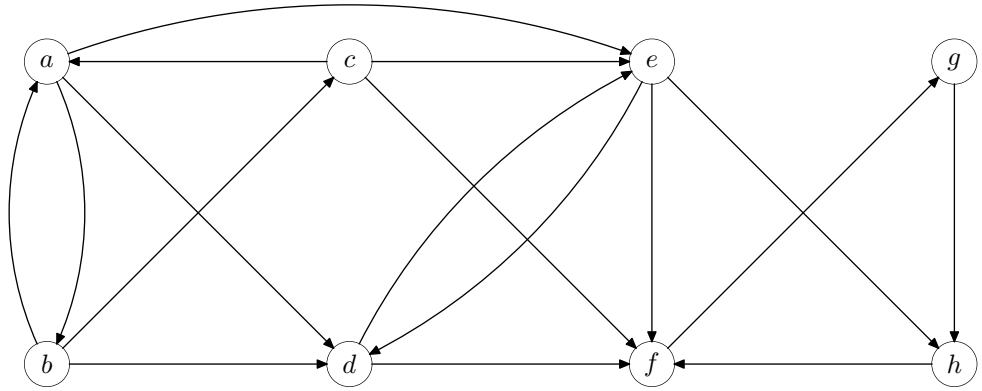



Figure 1: Directed graph $G$ for which to compute strongly connected components in Problem 1a.

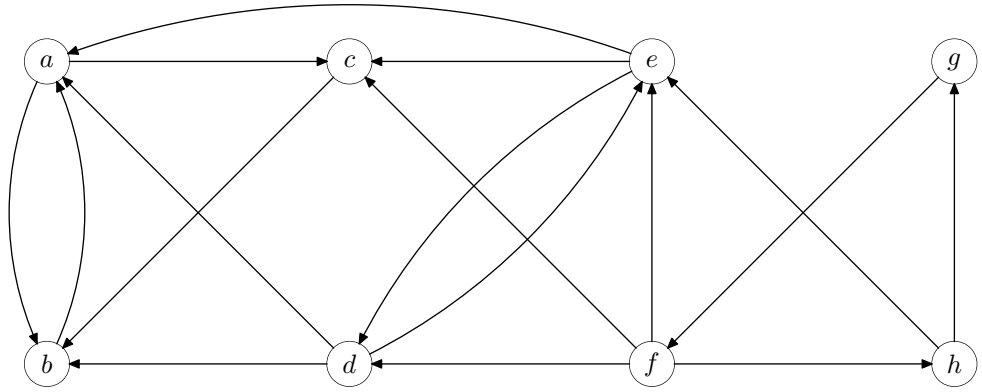



Figure 2: Inverted graph $G^T$ for computation of strongly connected components in Problem 1a.

Note that we are not asked here to argue why this algorithm is correct, so it is sufficient to make an accurate dry-run including a clear but concise explanation of the different steps in the computations.

The result of the depth-first search for $G$ is illustrated in Figure 3, with discovery and finish times marked as $\langle$*discovery time*$\rangle/\langle$*finish time*$\rangle$ above or below each vertex. Note that when we write "neighbour" below we mean "out-neighbour" unless specified otherwise. The chain of recursive calls are as follows (starting at time 1):

- At time 1 we visit the vertex $a$, the first neighbour of which is $b$, which is undiscovered and so we recursively visit this vertex.
- At time 2 we visit $b$. The first neighbour $a$ of $b$ has already been discovered, but the second neighbour $c$ has not yet been visited and so we recursively visit it.
- At time 3 we visit $c$. The first neighbour $a$ of $c$ has already been discovered, but the second neighbour $e$ is now visited.
- At time 4 we visit $e$, and since the first neighbour $d$ of this vertex has not yet been discovered we visit it.
- At time 5 we visit $d$. The first neighbour $e$ is marked as discovered, but the second neighbour $f$ gets a visit.

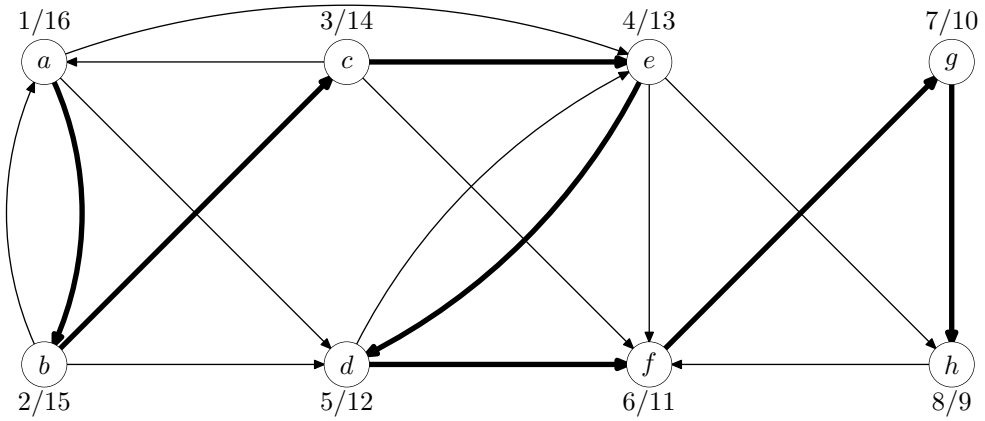



Figure 3: Directed graph $G$ with depth-first search tree and start and finishing times.

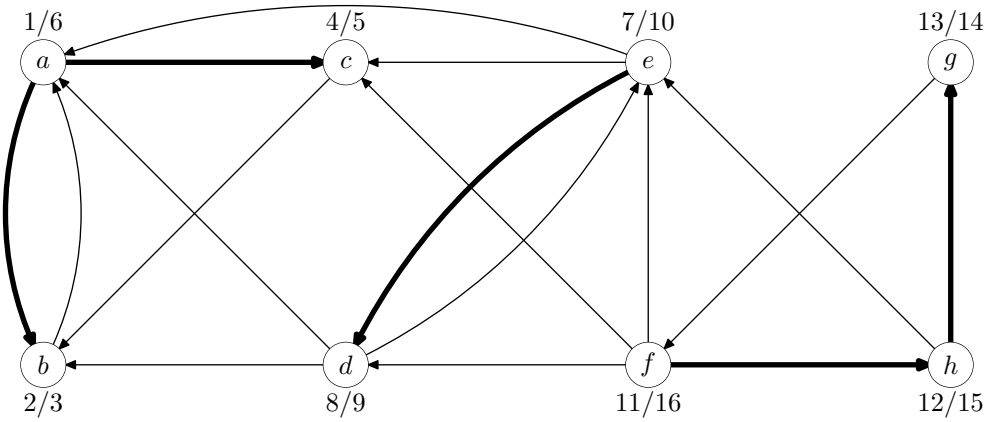



Figure 4: Directed graph $G^T$ with DFS forest identifying strongly connected components.

- At time 6 we visit $f$, whose only neighbour $g$ has not yet been discovered and therefore gets a visit.
- At time 7 we visit $g$ and immediately continue to its only neighbour $h$, since this vertex has not yet been discovered.
- At time 8, we visit $h$.
- At this point, all vertices in $G$ have been discovered, and so we will just finish the processing of all vertices in the reverse order of the calls above and stamp them with finishing times accordingly (so that $h$ gets finishing time 9, $g$ gets finishing time 10, $f$ gets finishing time 11, et cetera).

We now invert all vertices in $G$ to obtain $G^T$ as in Figure 2 and run a DFS for $G^T$ where we iterate over vertices in decreasing order with respect to their finishing times above. The result of this second DFS is shown in Figure 4 (where we include also the discovery and finishing times for completeness, although they are not needed for the discussion below).

- We start with $a$, since this vertex has finishing time 16. From $a$ we visit $b$ and immediately finish processing of this latter vertex, since the only out-neighbour of $b$ in $G^T$ is the already

discovered vertex $a$. Returning to $a$, we visit the second neighbour $c$, which is also immediately finished since its only out-neighbour $b$ in $G^T$ has already been processed. There are no more neighbours of $a$, and so the first DFS-visit call at top level terminates, leading to the conclusion that $\{a, b, c\}$ is a strongly connected component in $G$.

- The currently non-discovered vertex with the highest finishing time from step 1 is $e$, and so we make a DFS-visit call for this vertex. The neighbours $a$ and $c$ of $e$ have already been processed, but $d$ has not yet been discovered and so gets a visit. When we visit $d$, all out-neighbours $a$, $b$, and $e$ in $G^T$ have already been discovered, and so $d$ is immediately finished. When we return to $e$ there are no more out-neighbours in $G^T$ to process, so the second DFS-visit call at top level terminates, resulting in the strongly connected component $\{d, e\}$.
- Among the remaining non-discovered vertices the one with the highest finishing time is $f$. From $f$ we visit the only out-neighbour $h$ in $G^T$, and from $h$ we visit the only out-neighbour $g$. Now all vertices in $G^T$ have been discovered, and so we just return from all recursive visit calls. The final connected component reported by the algorithm is $\{f, g, h\}$.

Summing up, at the end of execution the algorithm has determined that the strongly connected components of the graph $G$ in Figure 1 are $\{a, b, c\}$, $\{d, e\}$, and $\{f, g, h\}$.

**1b** (30 p) In order to get a deeper understanding of operations on Boolean matrices, Jakob has performed some fairly extensive experiments on adjacency matrices $A_G$ for small directed graphs $G$. He now claims to have made the discovery that if he computes

$$\bigvee_{i=1}^{\infty} (A_G)_{\odot}^{i} = A_G \vee (A_G \odot A_G) \vee (A_G \odot A_G \odot A_G) \vee (A_G \odot A_G \odot A_G \odot A_G) \vee \dots \ ,$$

then it holds (possibly after reordering the vertices in $G$, corresponding to swapping rows and columns in $A_G$) that this matrix can be written on the form

$$\bigvee_{i=1}^{\infty} (A_G)_{\odot}^{i} = \begin{pmatrix} M_{1,1} & M_{1,2} & \cdots & M_{1,s} \\ M_{2,1} & M_{2,2} & \cdots & M_{2,s} \\ \vdots & \vdots & \ddots & \vdots \\ M_{s,1} & M_{s,2} & \cdots & M_{s,s} \end{pmatrix}$$

where the submatrices $M_{i,j}$ provide information about the strongly connected components of $G$ in the following sense. If $G$ has $s$ strongly connnected components $C_1$, $C_2$, $\dots$, $C_s$ of sizes $n_1$, $n_2$, $\dots$, $n_s$, respectively, then:

- Each matrix $M_{i,j}$ has dimensions $n_i \times n_j$.
- If there is a path from some $u \in C_i$ to some $v \in C_j$ in $G$, then $M_{i,j}$ contains 1s everywhere. (In particular, all matrices $M_{i,i}$ on the diagonal contain only 1s.)
- If there is no path from any $u \in C_i$ to any $v \in C_j$ in $G$, then $M_{i,j}$ contains 0s everywhere.

Sadly, Jakob is completely unable to explain this amazing fact, and he also cannot determine any upper bound on the time complexity of computing $\bigvee_{i=1}^{\infty} (A_G)_{\odot}^{i}$.

Is Jakob right about his claim? If so, present a concise and clear explanation to help Jakob see why this is so. If he is wrong, give a simple, concrete counter-example. Also, regardless of whether Jakob is correct or not, can you provide an efficient algorithm for computing $\bigvee_{i=1}^{\infty} (A_G)_{\odot}^{i}$ (for the plain adjacency matrix $A_G$ without any row or column swaps) together with as tight an upper bound as possible for the time complexity?

**Solution:** Yes, Jakob is actually right this time. In what follows, let us identify the integer indices of the rows and columns in $A_G$ with the vertices, so that we think of the vertices in $G$ as numbered from 1 to $n$.

First we note that for any positive integer $t$ it holds that the $(i,j)$-entry of the Boolean matrix $\big(A_G\big)^t_\odot$ is 1 precisely when there is a path from vertex $i$ to vertex $j$ of length exactly $t$. This can be shown by induction. The base case is that $A_G$ contains all length-1 paths, also known as edges. Suppose the claim is true for $t$. Consider any path from $i$ to $j$ of length exactly $t+1$ and let $\ell$ be the last vertex before $j$ on this path. Then by the induction hypothesis it holds that entry $(i,\ell)$ in $\big(A_G\big)^t_\odot$ is 1, and since $(\ell,j)$ is an edge we have that the entry $(\ell,j)$ in $A_G$ is also 1. By the definition of Boolean matrix multiplication, it follows that entry $(i,j)$ in $\big(A_G\big)^{t+1}_\odot = \big(A_G\big)^t_\odot \odot A_G$ is 1. Our claim now follows by the induction principle. Furthermore, we can conclude that the $(i,j)$-entry of $\bigvee_{t=1}^{\infty}\big(A_G\big)^t_\odot$ is 1 precisely when there is some path from $i$ to $j$ in $G$ of *some length* $t$, regardless of what this exact length is.

Rearrange the vertices of $G$ so that the $n_1$ first vertices are in the strongly connected component $C_1$, the $n_2$ next vertices are in the strongly connected component $C_2$, et cetera. This renumbering of vertices just corresponds to a sequence of pairwise swaps of rows and the same pair of columns of the matrix $A_G$. Let us next write

$$\bigvee_{i=1}^{\infty}\big(A_G\big)^i_\odot = \begin{pmatrix} M_{1,1} & M_{1,2} & \cdots & M_{1,s} \\ M_{2,1} & M_{2,2} & \cdots & M_{2,s} \\ \vdots & \vdots & \ddots & \vdots \\ M_{s,1} & M_{s,2} & \cdots & M_{s,s} \end{pmatrix} \tag{1}$$

where we simply define the submatrices $M_{i,j}$ to have dimensions $n_i \times n_j$. Let us analyse what we know about these submatrices.

- Consider first any two vertices $k,\ell \in C_i$. Since these two vertices are in the same strongly connected component, there are paths from $k$ to $\ell$ and from $\ell$ to $k$, and hence the $(k,\ell)$- and $(\ell,k)$-entries in (1) are both 1 by the reasoning above. Hence, the submatrix $M_{i,i}$ consists of all 1s.

- Consider next any pair of vertices $k \in C_i$ and $\ell \in C_j$ for $i \neq j$ and suppose that there is a path from $k$ to $\ell$. Then the $(k,\ell)$-entry in (1) is 1, again by the reasoning above. Furthermore, since there is a path from any vertex in $C_i$ to $k$ and from $\ell$ to any vertex in $C_j$, we conclude that the submatrix $M_{i,j}$ must consist of all 1s in this case.

- Consider finally any pair of vertices $k \in C_i$ and $\ell \in C_j$ for $i \neq j$ such that there is no path from $k$ to $\ell$. Then it follows from the previous case in our case analysis that there cannot exist any path from any vertex in $C_i$ to any vertex in $C_j$, and so $M_{i,j}$ must contain 0s everywhere.

This establishes all the properties that Jakob claimed.

Let us finally consider the time complexity of computing $\bigvee_{i=1}^{\infty}\big(A_G\big)^i_\odot$. We make three key observations:

1. If there is a path between any two vertices in an $n$-vertex graph, then a shortest such path cannot have length larger than $n$. Hence, it holds that $\bigvee_{i=1}^{\infty}\big(A_G\big)^i_\odot = \bigvee_{i=1}^{n}\big(A_G\big)^i_\odot$.

2. If we have already computed $\big(A_G\big)^t_\odot$, then we can compute $\big(A_G\big)^{t+1}_\odot = \big(A_G\big)^t_\odot \odot A_G$ in time $\mathrm{O}\big(n^3\big)$ just by coding up the definition of Boolean matrix multiplication in the most straightforward way possible.

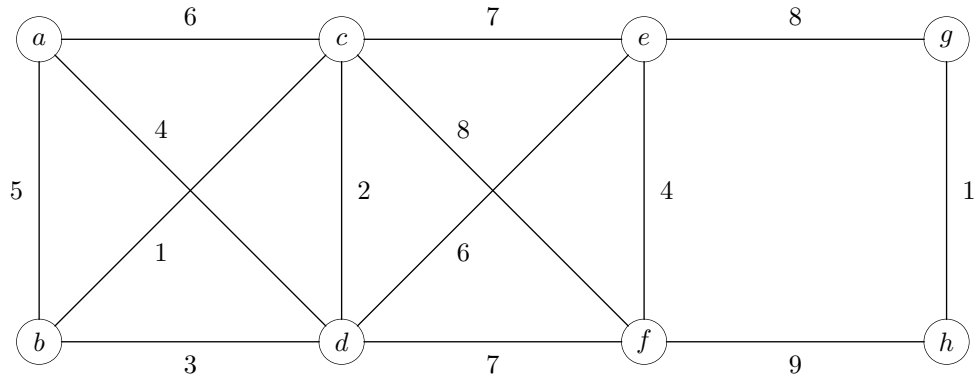



Figure 5: Undirected graph for which to compute minimum spanning tree in Problem 2a.

3. The Boolean disjunction $A \vee B$ of two $n \times n$ matrices $A$ and $B$ can be computed in time $\mathrm{O}(n^2)$, again by just writing down the definition in code. Hence, if we have already computed the two matrices $\bigvee_{i=1}^{t} (A_G)_{\odot}^{i}$ and $(A_G)_{\odot}^{t+1}$, then we can compute $\bigvee_{i=1}^{t+1} (A_G)_{\odot}^{i} = \bigvee_{i=1}^{t} (A_G)_{\odot}^{i} \vee (A_G)_{\odot}^{t+1}$ in time $\mathrm{O}(n^2)$.

Putting all of this together, we can compute $\bigvee_{i=1}^{t+1} (A_G)_{\odot}^{i}$ from $\bigvee_{i=1}^{t} (A_G)_{\odot}^{i}$ in time $\mathrm{O}(n^3)$, and doing this for $t = 1, 2, \ldots, n$ yields a total time complexity of $\mathrm{O}(n^4)$. Note that we do not claim that this is a tight bound, but this certainly enough for a full score given what we have learned during this course.

**2** (80 p) The purpose of this problem is to deepen our understanding of minimum spanning trees in undirected graphs.

**2a** (30 p) Generate a minimum spanning tree by running Kruskal's algorithm by hand on the graph in Figure 5. Assume that edges of the same weight are sorted in lexicographic order (so that for three hypothetical edges $(u, v)$, $(u, w)$, and $(v, w)$ of the same weight, we would have $(u, v)$ coming before $(u, w)$, which would in turn come before $(v, w)$).

Describe how the forest changes at each step (but you do not need to describe in detail how the set operations are implemented). Also show the final tree produced by the algorithm.

**Solution:** We illustrate the result of running Kruskal's algorithm in Figure 6. A detailed description of the execution follows. After sorting the edges in increasing order of weight and splitting ties as specified in the problem statement, we will have the edges listed in the following order, which is also the order in which they will be processed:

***Weight 1:*** $(b, c)$, $(g, h)$.

***Weight 2:*** $(c, d)$.

***Weight 3:*** $(b, d)$.

***Weight 4:*** $(a, d)$, $(e, f)$.

***Weight 5:*** $(a, b)$.

***Weight 6:*** $(a, c)$, $(d, e)$.

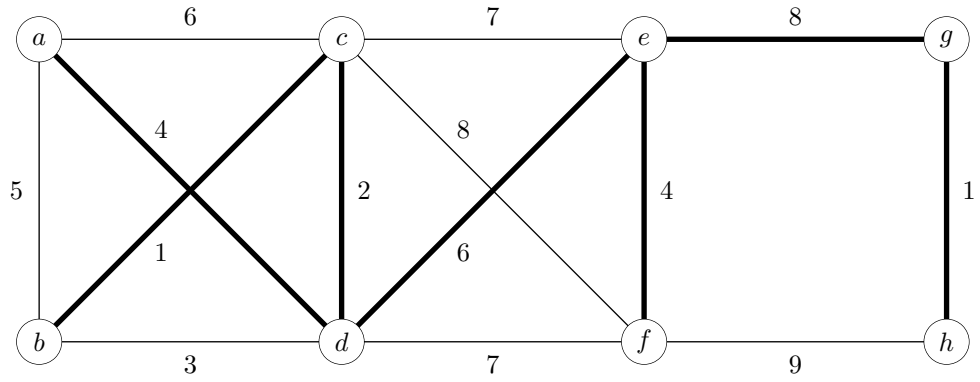



Figure 6: Graph in Figure 5 with minimum spanning tree as computed by Kruskal's algorithm.

***Weight 7:*** $(c, e)$, $(d, f)$.

***Weight 8:*** $(c, f)$, $(e, g)$.

***Weight 9:*** $(f, h)$.

The algorithm now does the following, starting with a set of singleton subtrees $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$, $\{e\}$, $\{f\}$, $\{g\}$, and $\{h\}$:

1. Looking first at edges of weight 1, according to the sorting order specified in the problem statement the edge $(b, c)$ is considered first. This edge merges $\{b\}$ and $\{c\}$ into a subtree $\{b, c\}$ and is added.
2. Next we consider the edge $(g, h)$, which merges $\{g\}$ and $\{h\}$ into another subtree $\{g, h\}$ and is added.
3. Moving on to edges of weight 2, the edge $(c, d)$ grows the subtree $\{b, c\}$ further to $\{b, c, d\}$ and is added.
4. The edge $(b, d)$ of weight 3 creates a cycle and so is discarded.
5. For the edges of weight 4, according to our sorting order the edge $(a, d)$ is considered first. This edge grows the subtree $\{b, c, d\}$ to $\{a, b, c, d\}$ and is added.
6. The edge $(e, f)$ creates a subtree $\{e, f\}$ and is added.
7. The edge $(a, b)$ of weight 5 creates a cycle and is thrown away.
8. Considering next edges of weight 6, $(a, c)$ also creates a cycle and is discarded.
9. The edge $(d, e)$ joins the two trees $\{a, b, c, d\}$ and $\{e, f\}$ into a larger tree $\{a, b, c, d, e, f\}$, and so is added.
10. For edges of weight 7, both edges $(c, e)$ and $(d, f)$ create cycles and are discarded.
11. For weight 8, we first consider the edge $(c, f)$, which also creates a cycle and so is thrown away.
12. The edge $(e, g)$ creates a spanning tree for the full graph, and so is added. Since we know we now have the the right number of edges, and adding any more edge must create a cycle somewhere, we can terminate without considering the final remaining edge $(f, h)$.

**2b** (10 p) Suppose that some vertex $v$ with several neighbours in a graph $G$ has a unique neighbour $u$ such that the edge $(u, v)$ has strictly smaller weight than any other edge incident to $v$. Is it true that the edge $(u, v)$ must be included in any minimum spanning tree? Prove this or give a simple counter-example.

**Solution:** Yes, any minimum spanning tree has to include the edge $(u, v)$, since this is the unique light edge in a cut with vertex $v$ on one side and the rest of the graph on the other side. If there were an MST without the edge $(u, v)$, then adding that edge to the MST would create a cycle, from which we could remove a heavier edge and get another spanning tree for the graph. This contradicts that the tree we started with was an MST.

**2c** (20 p) Suppose that some vertex $v$ with several neighbours in a graph $G$ has a unique neighbour $u$ such that the edge $(u, v)$ has strictly larger weight than any other edge incident to $v$. Is it true that the edge $(u, v)$ can never be included in any minimum spanning tree? Prove this or give a simple counter-example.

**Solution:** No, this is not true. Consider the case when $u$ has only one neighbour and this neighbour is $v$. Then the edge $(u, v)$ has to be included in any MST.

**2d** (20 p) Suppose that $T$ is a minimum spanning tree for a weighted, undirected graph $G$. Modify $G$ by adding some constant $c \in \mathbb{R}^+$ to all edge weights. Is $T$ still a minimum spanning tree? Prove this or give a simple counter-example.

**Solution:** If the spanning tree has $M$ edges, then after the edge weight increase the new tree has a total weight that increased by an amount $M \cdot c$. But any spanning tree for a graph $G$ will have the same number of edges, and hence the weight increase will be exactly the same for all spanning trees. This means, in particular, that all MSTs before the weight increase remain MSTs also after the weight increase.

**3** (100 p) Assume that we are given the directed graph in Figure 7. 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 this order.)

**3a** (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 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 8. At the outset, the vertex $a$ has key 0 and all other vertices have key $\infty$. We will use the notation $v : k$ in the heap when vertex $v$ has key value $k$.

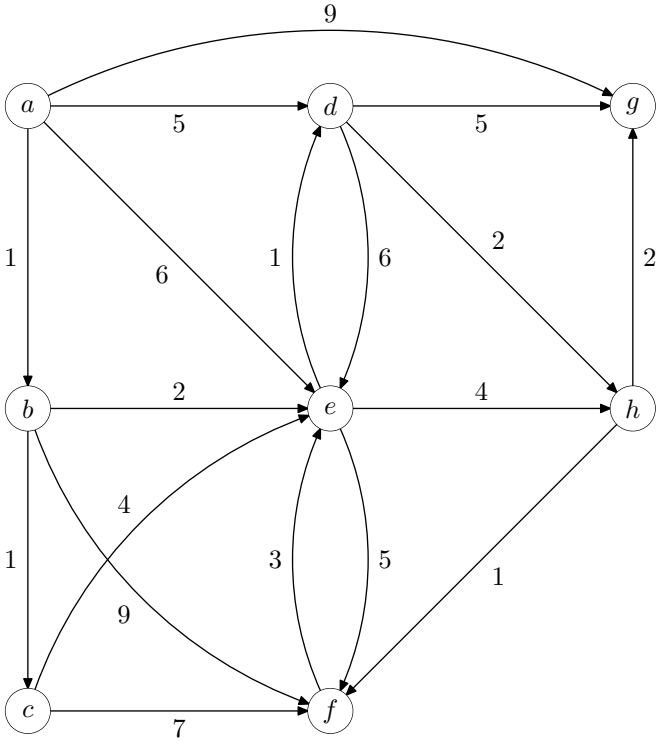


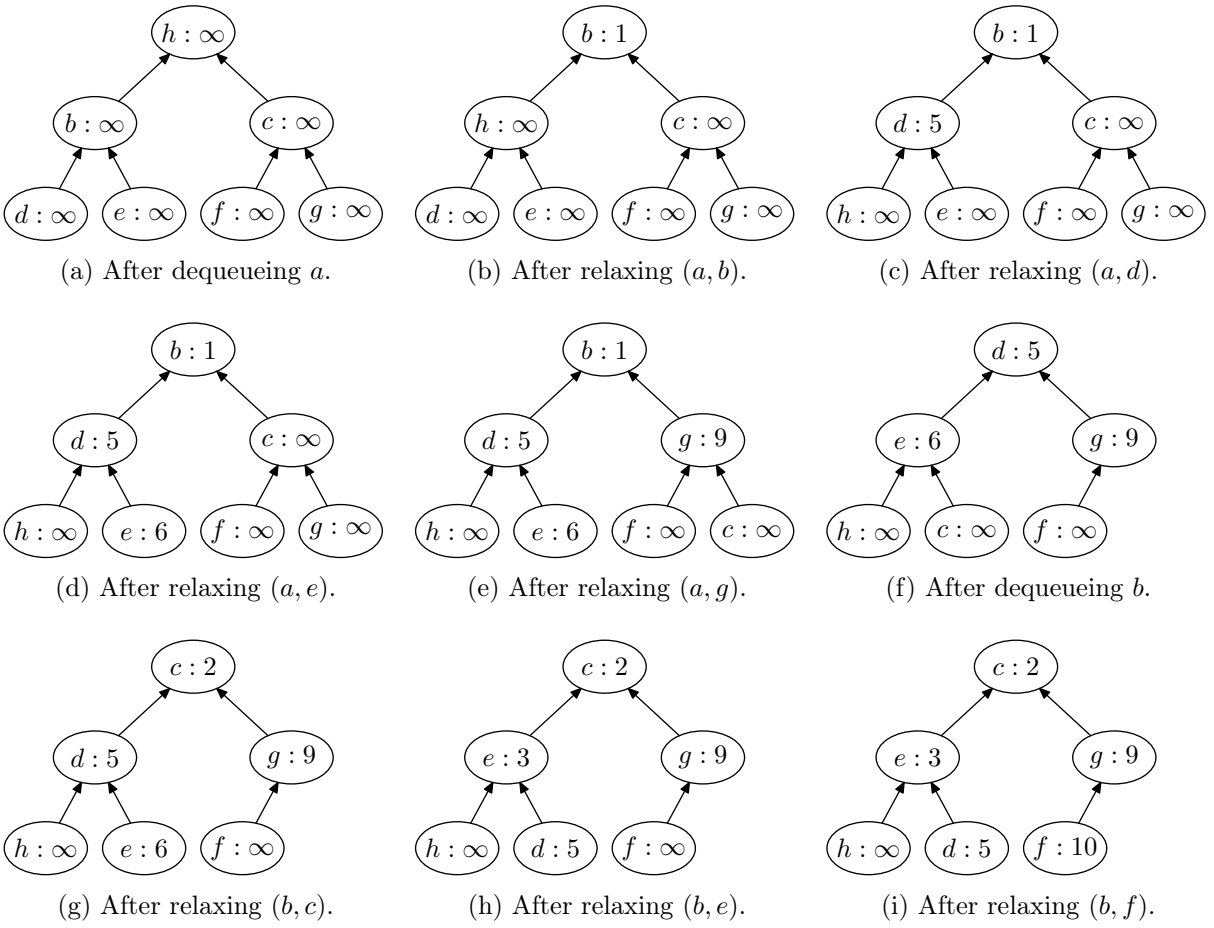

Figure 7: Directed graph for Dijkstra's algorithm in Problem 3a.

1. After $a$ has been dequeued, vertex $h$ is moved to the top of the heap and we have the configuration in Figure 8a. Relaxing the edge $(a, b)$ shifts $b$ to the top and pushes $h$ down, yielding Figure 8b. Relaxing $(a, d)$ swaps $d$ and $h$, yielding Figure 8c. 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 8d). Finally, relaxing $(a, g)$ makes $g$ bubble up and $c$ bubble down, so that we have the configuration in Figure 8e 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 9. 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 8f.

   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 8g. Relaxing $(b, e)$ swaps $d$ and $e$, resulting in Figure 8h. 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 8i).

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




(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 8: Heap configurations for priority queue in Problem 3a.

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 $(d, h)$ 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 9.

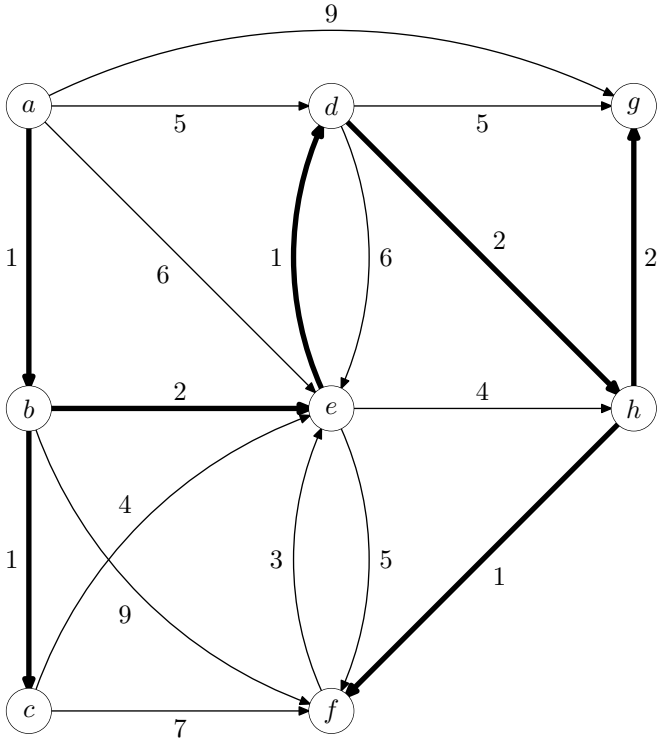



Figure 9: Shortest paths computed by Dijkstra's algorithm for graph in Figure 7.

**3b** (10 p) Consider the directed tree of shortest paths $T$ produced in Problem 3a. 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 3a. (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 $a \to b \to f$.)

**3c** (10 p) Consider the directed tree of shortest paths $T$ produced in Problem 3a. 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.

**3d** (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., $O(|E| \log |V|)$.