KØBENHAVNS UNIVERSITET

Diskret Matematik og Formelle Sprog: Problem Set 5

Due: Thursday March 30 at 23:59 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. 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 (70 p) In this problem we wish to understand Dijkstra's algorithm.

1a (40 p) Assume that we are given the directed graph D in Figure 1. 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 done in Jakob's notes and video lectures, 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.

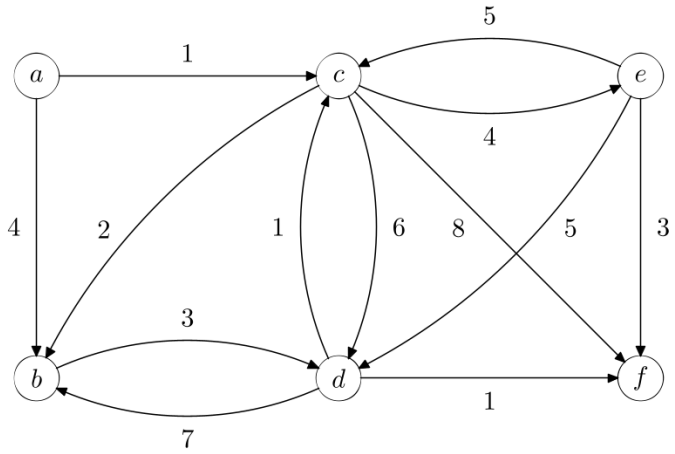


Figure 1: Directed graph D on which to run Dijkstra's algorithm in Problem 1a.

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

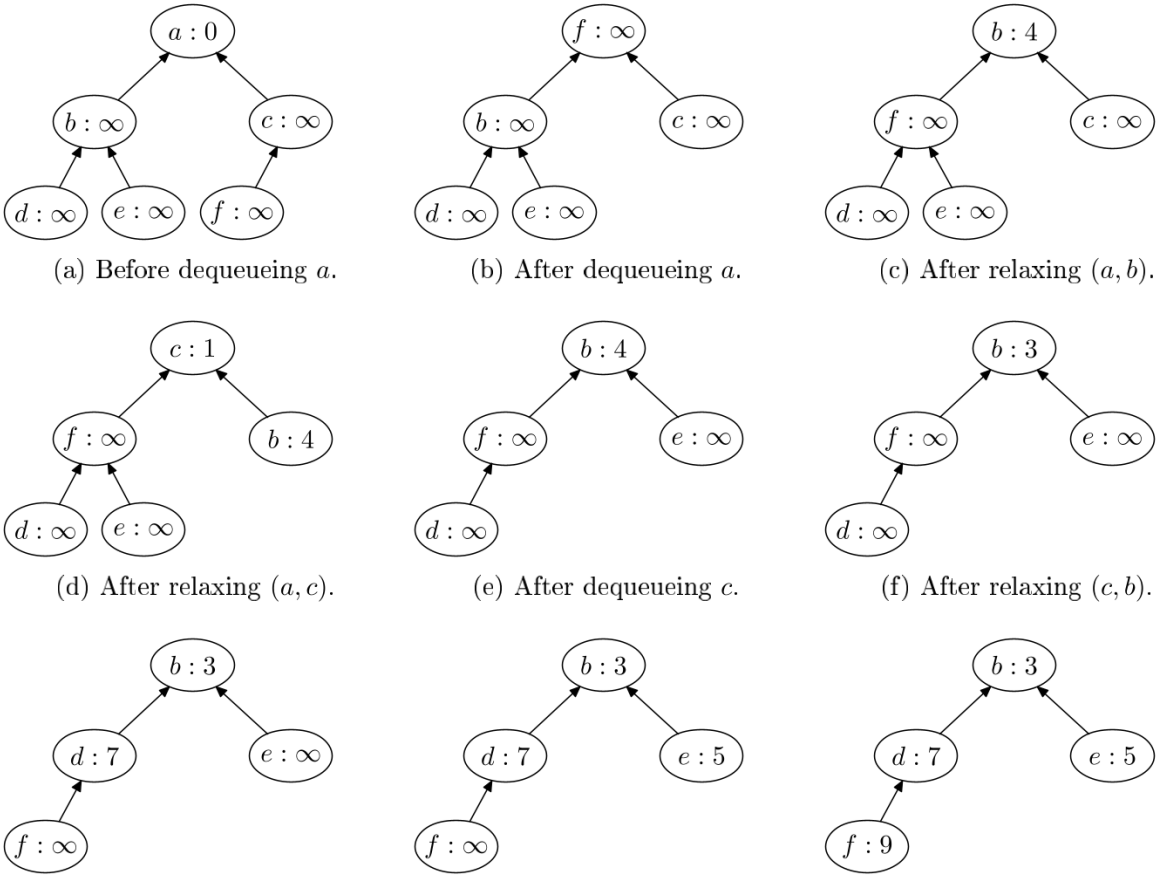
Dequeued vertex		a	c	b	e	d	f
Estimate for a	0	0					
Estimate for b	∞	**4**	**3**	3			
Estimate for c	∞	**1**	1				
Estimate for d	∞	∞	**7**	**6**	6	6	
Estimate for e	∞	∞	**5**	5	5		
Estimate for f	∞	∞	**9**	9	**8**	**7**	7

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 2 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 2a.

1. After a has been dequeued, vertex f is moved to the top of the heap and we have the configuration in Figure 2b. Relaxing the edge (a, b) shifts b to the top and pushes f down below b, yielding Figure 2c. Relaxing (a, c) swaps b and c, yielding Figure 2d. 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 3. 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 2e.

   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 2f. Relaxing (c, d) decreases the key value of d to $1 + 6 = 7$ and makes d bubble up above f, resulting in Figure 2g. 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 2h). 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



(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 2: Heap configurations for priority queue in Problem 1a.

been relaxed, and the resulting heap is as in Figure 2i. 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.

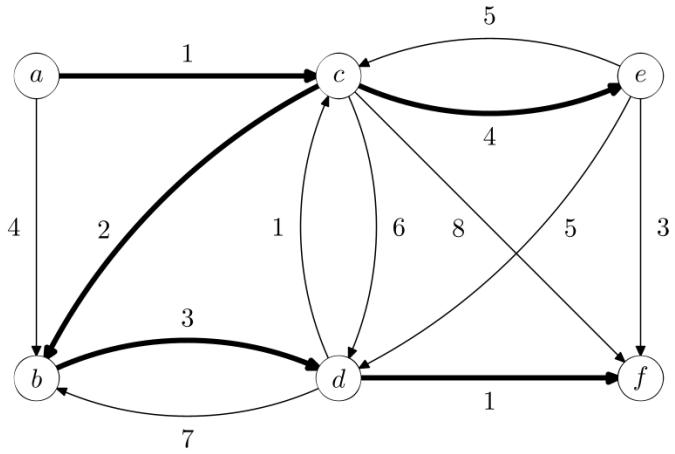


Figure 3: Shortest paths tree computed by Dijkstra's algorithm for graph in Figure 1.

The computed shortest paths tree is indicated by the bold edges in Figure 3.

1b (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

- 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 $O\big(|V(G)|+|E(G)|\big)$. and Dijkstra's algorithm will run in the same time complexity $O\big(|E(G)|\log|V(G)|\big)$ as we are used to (assuming an implementation with a min-heap for the priority queue).

2 (90 p) In this problem we wish to understand how to compute topological orderings and/or strongly connected components of directed graphs.

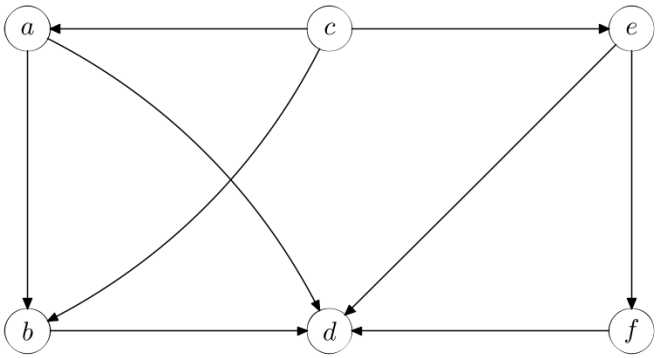


Figure 4: Directed graph H for which to compute a topological ordering in Problem 2a.

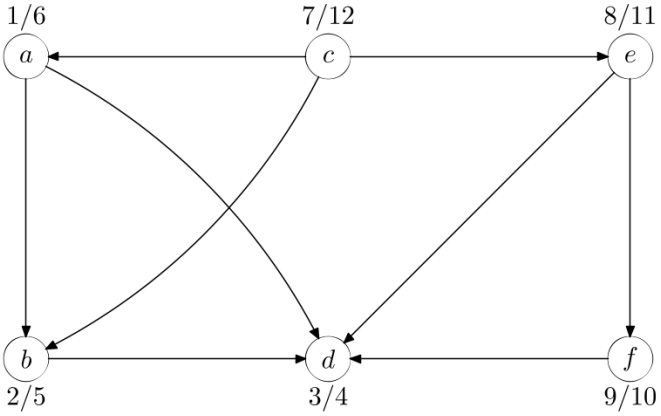


Figure 5: Directed graph H in Problem 2a. with DFS tree and start and finishing times.

2a (30 p) Consider the graph H in Figure 4. 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 this is done in Jakob's lecture notes (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 5 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.
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 5) 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).

2b (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 6 on page 7 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 6 and $U_2 = \{e, f\}$ and explain how you constructed this graph.

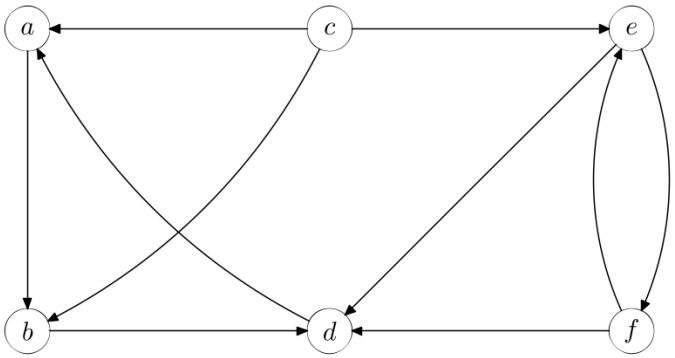


Figure 6: Directed graph G for which to compute contractions in Problem 2b.

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.)

To get a full score for Problem 2b 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 6 and $U_1 = \{a, b, d\}$ is depicted in Figure 7. 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 8.

2c (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.

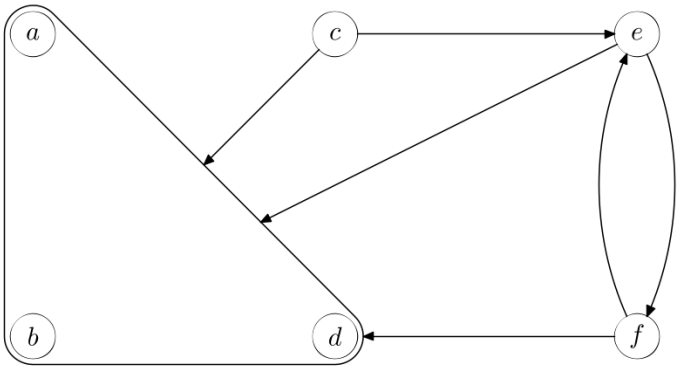


Figure 7: Contracted graph $\mathsf{contract}(G, U_1)$ for G in Figure 6 and $U_1 = \{a, b, d\}$.

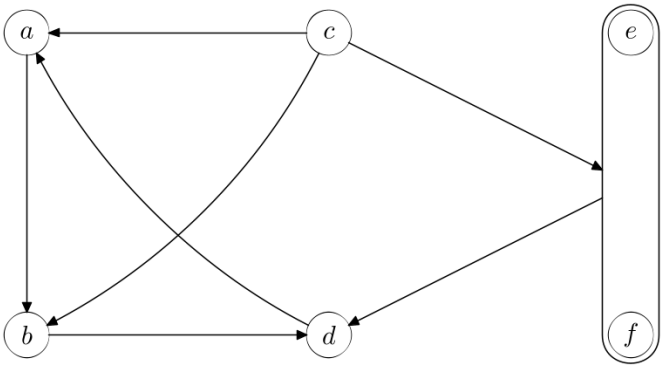


Figure 8: Contracted graph $\mathsf{contract}(G, U_2)$ for G in Figure 6 and $U_2 = \{e, f\}$.

3. Otherwise, fix the back edge that made the topological sort fail and use this edge to find a cycle C in G'.
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 6 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 6, 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 7. The label of the new vertex v_{U_1} is the vertex set U_1.

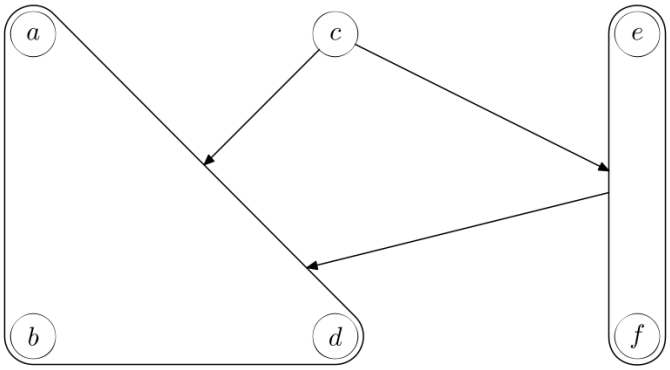


Figure 9: Fully contracted graph obtained from G in Figure 6 at end of algorithm in Problem 2c.

- 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 9. 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 $\big\{\mathsf{vlabels}\big(v'\big) \,\big|\, v' \in V(G')\big\}$ 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 $\mathsf{vlabels}\big(v'\big)$ 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 $\mathsf{vlabels}\big(v'\big)$ 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 $u \in \mathsf{vlabels}\big(u'\big)$ and $v \in \mathsf{vlabels}\big(v'\big)$, 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 $u \in \mathsf{vlabels}\big(u'\big)$ and $v \in \mathsf{vlabels}\big(v'\big)$, 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 $\mathrm{O}(|V| + |E|)$.

3 (60 p) On the final problem set, Jakob wanted to add a couple of examples of the power of inductive proofs. This did not go so well, however, since Jakob's proofs turned out to be a little bit too strong. Spceifically, Jakob was able to use mathematical induction to show

1. that all swans have the same colour (presumably white, so that there are no black swans after all), and
2. that all positive integers are in fact equal.

Both of these claims are fairly disturbing from a mathematical point of view. Please help Jakob by pointing out clearly what goes wrong in his induction proofs below.

3a **Theorem.** All swans have the same colour.

Proof. We prove by induction over n that any set of n swans have the same colour.

For the base case, if we have a set of $n = 1$ swan, then this swan vacuously has the same colour as itself.

For the induction step, assume as our induction hypothesis that all sets of n swans have the same colour, and consider a set of $n + 1$ swans. Fix some swan S_1 in this set. If we remove S_1, then we have n swans left, and by the induction hypothesis they all have the same colour. Let C be this colour.

Now consider another swan S_2 in the set. If we remove S_2 from our set of $n + 1$ swans instead of S_1, then we again have n swans left, and they all have the same colour by the induction hypothesis. Since S_1 is one of the swans in this set, it must have the same colour C as all the others. Hence, all $n + 1$ swans have the same colour.

It follows by the principle of mathematical induction that any set of n swans must always have the same colour.

Solution: The base case is correct.

The induction step is also correct for all $n \geq 3$. It is true that if I have a set of $n \geq 3$ objects with the property that any subset of $n - 1$ objects must all have the same colour, then the only way this can happen is that all n objects have the same colour.

However, for $n = 2$ there is a gap in the argument, and this is why we can "prove" a false theorem. If we remove swan S_2 from the set, then there is only swan S_1 left. This swan S_1 could have a different colour, contrary to what the induction step argument claims.

3b **Theorem.** All positive integers are equal.

Proof. By induction over n.

For the base case, the positive integer 1 is equal to itself.

For the induction step, assume as our induction hypothesis that $n - 1 = n$. Adding 1 to both sides of this equality, we derive that $n = n + 1$.

It follows from this by the principle of mathematical induction that for all integers n the equality $n = n + 1$ holds. But then by transitivity we obtain that all integers are equal, and the claim in the theorem statement has thus been established.

Solution: The induction step in this proof is just fine, but the base case is wrong.

Since the induction hypothesis is that $n - 1 = n$, for the base case we would need to prove that $0 = 1$ or $1 = 2$ or something. But in the base case in the "proof" above, we are not proving a base case of the form $n - 1 = n$, but rather $n = n$. This is of course true, but this does not match what we need when we want to use the base case as induction hypothesis for our first induction step.