Eidgenössische Technische Hochschule Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Johannes Lengler, David Steurer
Lucas Slot, Manuel Wiedmer, Hongjie Chen, Ding Jingqiu

27 November 2023

Algorithms & Data Structures — Exercise sheet 10 — HS 23

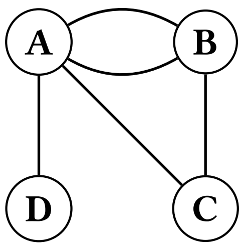
The solutions for this sheet are submitted at the beginning of the exercise class on 4 December 2023.

Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

You can use results from previous parts without solving those parts.

Exercise 10.1 *Eulerian tours in multigraphs* **(1 point)**.

A *multigraph* $G = (V, E)$ is a graph which is permitted to have multiple copies of the same edge. That is, the edges E form a *multiset* (a set in which elements are allowed to occur multiple times). For example, the multigraph with $V = \{1, 2, 3, 4\}$ and $E = \{\{A, B\}, \{A, B\}, \{A, D\}, \{B, C\}, \{A, C\}\}$ is depicted below. To avoid confusion, the term *simple graph* is sometimes used to indicate that duplicate edges are not allowed.

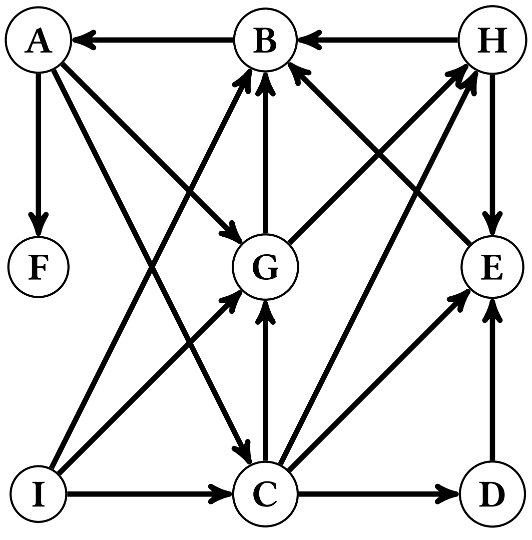


(a) An Eulerian tour in a multigraph is a tour which visits every edge exactly once. If multiple copies of an edge exist, the tour should visit each of them exactly once. Given a multigraph $G = (V, E)$, describe an algorithm which constructs a *simple* graph $G' = (V', E')$ such that G has a Eulerian tour if and only if G' has a Eulerian tour. The new graph should satisfy $|V'| \leq |V| + |E|$, and $|E'| \leq 2 \cdot |E|$. The runtime of your algorithm should be at most $O(n + m)$. You are provided with the number of vertices n and an adjacency list of G (if there are multiple edges between $v, w \in V$, then w appears that many times in the list of neighbours of v).

(b)* Let $G = (V, E)$ be a *simple* graph, and let $f : E \to \mathbb{N} \cup \{0\}$ be a function. A Eulerian f-tour of G is a tour which visits each edge $e \in E$ exactly $f(e)$ times. Describe an algorithm which constructs a simple graph $G' = (V', E')$ such that G has a Eulerian f-tour if and only if G' has a Eulerian tour. The new graph should satisfy $|V'| \leq |V| + \sum_{e \in E} f(e)$, and $|E'| \leq 2 \sum_{e \in E} f(e)$. The runtime of your algorithm should be at most $O(n + m + \sum_{e \in E} f(e))$.

Exercise 10.2 *Depth-first search* **(1 point)**.

Execute a depth-first search (*Tiefensuche*) on the following graph. Use the algorithm presented in the lecture. Always do the calls to the function "visit" in alphabetical order, i.e. start the depth-first search from A and once “visit(A)” is finished, process the next unmarked vertex in alphabetical order. When processing the neighbors of a vertex, also process them in alphabetical order.



(a) Mark the edges that belong to the depth-first forest (*Tiefensuchwald*) with a “T” (for tree edge).

(b) For each vertex in the depth-first forest, give its *pre-* and *post*-number.

(c) Give the vertex ordering that results from sorting the vertices by pre-number. Give the vertex ordering that results from sorting the vertices by post-number.

(d) Mark every forward edge (*Vorwärtskante*) with an “F”, every backward edge (*Rückwärtskante*) with a “B”, and every cross edge (*Querkante*) with a “C”.

(e) Does the above graph have a topological ordering? If yes, write down the topological ordering we get from the above execution of depth-first search; if no, argue how we can use the above execution of depth-first search to find a directed cycle.

(f) Draw a scale from 1 to 18, and mark for every vertex v the interval I_v from pre-number to post-number of v. What does it mean if $I_u \subset I_v$ for two different vertices u and v?

(g) Consider the graph above where the edge from B to A is removed and an edge from F to I is added. How does the execution of depth-first search change? Does the graph have a topological ordering? If yes, write down the topological ordering we get from the execution of depth-first search; if no, argue how we can use the execution of depth-first search to find a directed cycle. If you sort the vertices by *pre-number*, does this give a topological sorting?

Exercise 10.3 *Driving on highways.*

In order to encourage the use of train for long-distance traveling, the Swiss government has decided to make all the m highways between the n major cities of Switzerland one-way only. In other words, for any two of these major cities C_1 and C_2, if there is a highway connecting them it is either from C_1 to C_2 or from C_2 to C_1, but not both. The government claims that it is however still possible to drive from any major city to any other major city using highways only, despite these one-way restrictions.

(a) Model the problem as a graph problem. Describe the set of vertices V and the set of edges E in words. Reformulate the problem description as a graph problem on the resulting graph.

(b) Describe an algorithm that verifies the correctness of the claim in time $O(n+m)$. Argue why your algorithm is correct and why it satisfies the runtime bound.

__Hint:__ You can make use of an algorithm from the lecture. However, you might need to modify the graph described in part (a) and run the algorithm on some modified graph.

Exercise 10.4 *Strongly connected components* **(1 point)**.

Let $G = (V, E)$ be a directed graph with n vertices and m edges. Recall from Exercise 9.5 that two distinct vertices $v, w \in V$ are *strongly connected* if there exist both a directed path from v to w, and from w to v.

The vertices of G can be partitioned into disjoint subsets $V_1, V_2, \ldots, V_k \subseteq V$ with $V = V_1 \cup V_2 \cup \ldots \cup V_k$, such that any two distinct vertices $v, w \in V$ are strongly connected if and only if they are in the same subset V_ℓ, for some $1 \leq \ell \leq k$. The subsets V_ℓ are called the *strongly connected components* of G.

As in Exercise 9.5, you are provided with the number of vertices n, and the adjacency list Adj of G.

(a) Describe an algorithm that outputs the strongly connected components of G in time $O(n \cdot (n + m))$.

__Hint:__ Apply the algorithm of Exercise 9.5 several times. After each application, remove a vertex from G.

It turns out that we can find the strongly connected components of G in time $O(n + m)$. In the rest of the exercise we construct an algorithm to do so.

(b)* Let $L = [v_1, v_2, \ldots, v_n]$ be a list containing the vertices of G in the *reversed* post-order of a DFS. Show that L has the following property:

'For any distinct $v, w \in V$, if there is a directed path from v to w, then

(1) v and w are strongly connected; and/or

(2) there exists a $u \in V$ which is in the same strongly connected component as v, and which appears before w in L.'

Remark. You are allowed to use this part in the rest of the exercise, even if you do not solve it.

(c) Let $\overleftarrow{G} = (V, \overleftarrow{E})$ be the directed graph obtained by inverting all edges in G. Let v_1 be the first element of L. Let $W \subseteq V$ be the set of vertices w for which there is a directed path from v_1 to w in $\overleftarrow{G}$. Show that W is a strongly connected component of G.

(d) Describe an algorithm that outputs all strongly connected components of G. The runtime of your algorithm should be at most $O(n + m)$. Prove that your algorithm is correct, and achieves the desired runtime.

__Hint:__ Use DFS on the inverted graph $\overleftarrow{G}$. Make visit*-calls based on the list L.*

Exercise 10.5 *Shortest paths by hand.*

Dijkstra's algorithm allows to find shortest paths in a directed graph when all edge costs are nonnegative. Here is a pseudo-code for that algorithm:

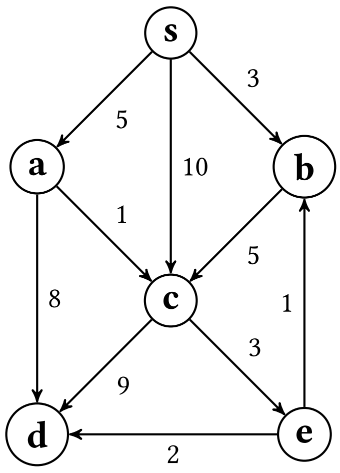
Algorithm 1

```
Input: a weighted graph, represented via $c(\cdot,\cdot)$. Specifically, for two vertices $u, v$ the value $c(u,v)$
represents the cost of an edge from $u$ to $v$ (or $\infty$ if no such edge exists).
function DIJKSTRA($G, s$)
    $d[s] \leftarrow 0$                                  ▷ upper bounds on distances from $s$
    $d[v] \leftarrow \infty$ for all $v \neq s$
    $S \leftarrow \emptyset$                                  ▷ set of vertices with known distances
    while $S \neq V$ do
        choose $v^* \in V \setminus S$ with minimum upper bound $d[v^*]$
        add $v^*$ to $S$
        update upper bounds for all $v \in V \setminus S$:
            $d[v] \leftarrow \min_{\text{predecessor } u \in S \text{ of } v} d[u] + c(u,v)$
            (if $v$ has no predecessors in $S$, this minimum is $\infty$)
```

We remark that this version of Dijkstra's algorithm focuses on illustrating how the algorithm explores the graph and why it correctly computes all distances from s. You can use this version of Dijkstra's algorithm to solve this exercise.

In order to achieve the best possible running time, it is important to use an appropriate data structure for efficiently maintaining the upper bounds $d[v]$ with $v \in V \setminus S$ as you will see in the lecture on November 30. In the other exercises/sheets and in the exam you should use the running time of the efficient version of the algorithm (and not the running time of the pseudocode described above).

Consider the following weighted directed graph:



(a) Execute the Dijkstra's algorithm described above by hand to find a shortest path from **s** to each vertex in the graph. After each step (i.e. after each iteration of the while-loop), write down:

1) the upper bounds $d[u]$, for $u \in V$, between **s** and each vertex u computed so far,

2) the set M of all vertices for which the minimal distance has been correctly computed so far,

3) and the predecessor $p(u)$ for each vertex in M.

(b) Change the weight of the edge $(\mathbf{a}, \mathbf{c})$ from 1 to -1 and execute Dijkstra's algorithm on the new graph. Does the algorithm work correctly (are all distances computed correctly) ? In case it breaks, where does it break?

(c) Now, additionally change the weight of the edge $(\mathbf{e}, \mathbf{b})$ from 1 to -6 (so edges $(\mathbf{a}, \mathbf{c})$ and $(\mathbf{e}, \mathbf{b})$ now have negative weights). Show that in this case the algorithm doesn't work correctly, i.e. there exists some $u \in V$ such that $d[u]$ is not equal to a minimal distance from **s** to u after the execution of the algorithm.