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

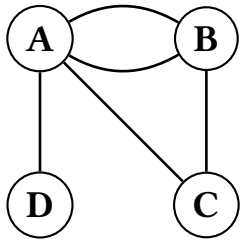
The solutions for this sheet are submitted at the beginning of the exercise class on 4 December 2023.

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 $E = \big\{\{A, B\}, \{A, B\}, \{A, D\}, \{B, C\}, \{A, C\}\big\}$ is depicted below. To avoid confusion, the term *simple graph* is sometimes used to indicate that duplicate edges are not allowed.



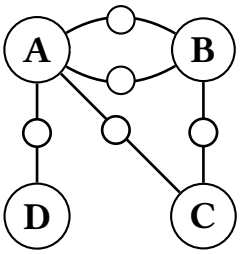
(a) An Eulerian tour in a multigraph is a tour which visits every edge exactly once. If multiple copies of an edge exist, the tour should visit each of them exactly once. Given a multigraph $G = (V, E)$, describe an algorithm which constructs a *simple* graph $G' = (V', E')$ such that G has a Eulerian 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).

Solution:

The idea is to *subdivide* each edge of G. For the construction, it is convenient to first number the edges $E = \{e_1, e_2, \ldots, e_{|E|}\}$. For each k, we write v_k, w_k for the endpoints of e_k (i.e., $e_k = \{v_k, w_k\}$). Now, to construct V', we start with all vertices in V, and then add a vertex v'_k for each edge $e_k \in E$, $1 \leq k \leq |E|$. Then, the edges E' are given by:

$$E' = \bigcup_{k=1}^{|E|} \{e_k^1, e_k^2\}, \quad \text{where we set } e_k^1 = \{v_k, v'_k\},\ e_k^2 = \{w_k, v'_k\}.$$

See the picture below for an illustration of this procedure for the multigraph example above.



Note that $G' = (V', E')$ satisfies the required bounds on the number of vertices and edge, and that our algorithm runs in time $O(n + m)$. It remains to see that G has an Eulerian tour if and only G' does. First, if G has an Eulerian tour $T = (e_{j_1}, e_{j_2}, \ldots, e_{j_{|E|}})$, then we can construct an Eulerian tour T' in G' by replacing each e_{j_k} by the two edges $e^1_{j_k}, e^2_{j_k}$ (in the correct order). On the other hand, if T' is an Eulerian tour in G', then we note that for any k, the edges e^1_k, e^2_k must appear directly adjacent in T' (in either order), since they are the only edges that have v'_k as an endpoint. But then we may obtain an Eulerian tour in G by simply replacing all these pairs by the edge e_k.

(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 $O(n + m + \sum_{e \in E} f(e))$.

Solution:

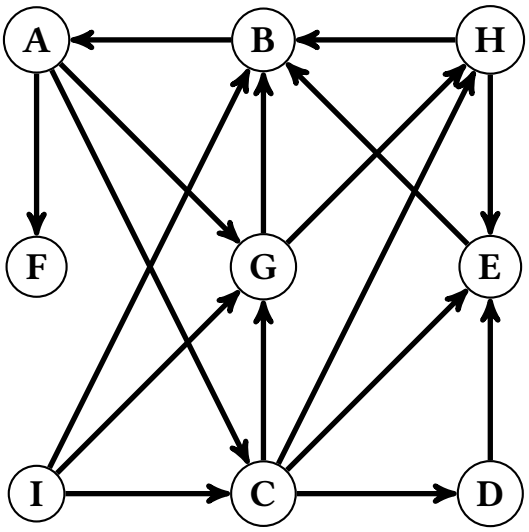
To construct G', first, we remove all edges e from G with $f(e) = 0$. Then, we construct a multigraph $H = (V, F)$, where F contains exactly $f(e)$ copies of each edge in G. Note that $|F| = \sum_{e \in E} f(e)$. Note also that, by definition, an Eulerian tour exists in H if and only if a Eulerian f-tour exists in G. Finally, we use part (a) to convert H into a simple graph $G' = (V', E')$, where we know that $|V'| \leq |V| + |F| = |V| + \sum_{e \in E} f(e)$ and $|E'| \leq 2 \cdot |F| = 2 \cdot \sum_{e \in E} f(e)$.

Guidelines for correction:

Award 1/2 points for a correct construction, and 1/2 points for a correct proof of the iff-statement.

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

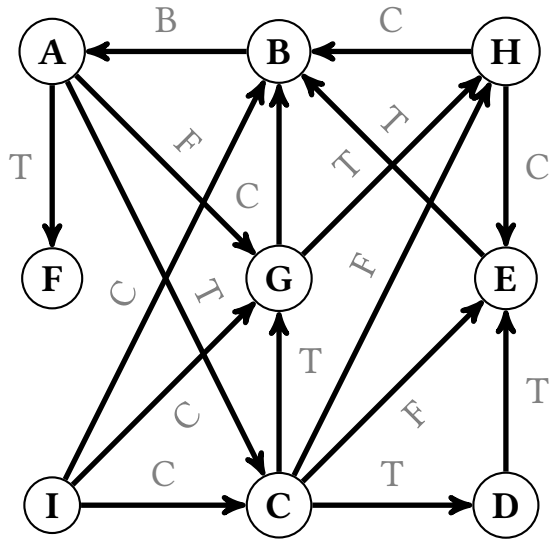
Execute a depth-first search (*Tiefensuche*) on the following graph. Use the algorithm presented in the lecture. Always do the calls to the function "visit" in alphabetical order, i.e. start the depth-first search from A and once "visit(A)" is finished, process the next unmarked vertex in alphabetical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

Solution:

In the following, both the solution to subtask (a) and the solution to subtask (d) are showed.



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

Solution:

A(1,16) B(5,6) C(2,13) D(3,8) E(4,7) F(14,15) G(9,12) H(10,11) I(17,18).

(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.

Solution:

Pre-ordering: A, C, D, E, B, G, H, F, I.

Post-ordering: B, E, D, H, G, C, F, A, I.

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

Solution:

See above in the solution to part (a).

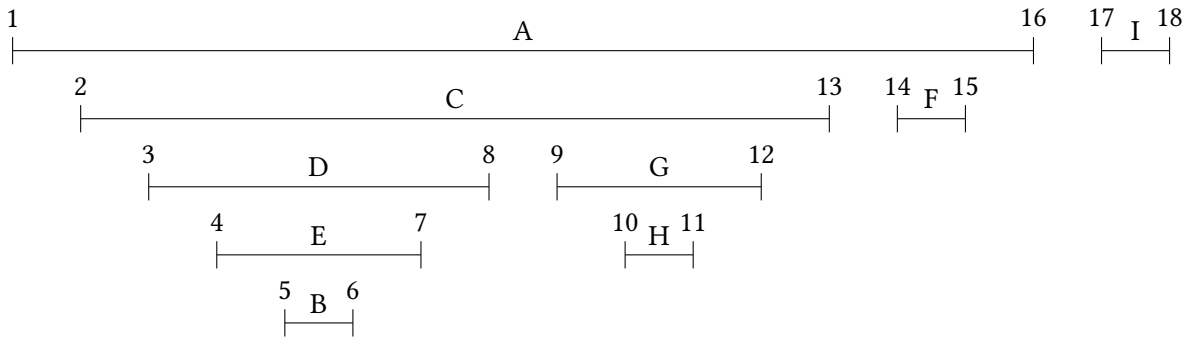
(e) Does the above graph have a topological ordering? If yes, write down the topological ordering we get from the above execution of depth-first search; if no, argue how we can use the above execution of depth-first search to find a directed cycle.

Solution:

The decreasing order of the post-numbers gives a topological ordering whenever the graph is acyclic. This is the case if and only if there are no back edges. If there is a back edge, then together with the tree edges between its end points it forms a directed cycle. In our graph, the only back edge is B $\rightarrow$ A, and the tree edges from A to B are A $\rightarrow$ C, C $\rightarrow$ D, D $\rightarrow$ E and E $\rightarrow$ B. Together they form the directed cycle (A $\rightarrow$ C $\rightarrow$ D $\rightarrow$ E $\rightarrow$ B $\rightarrow$ A).

(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?

Solution:



If $I_u \subset I_v$ for two different vertices u and v, then u is visited during the call of visit(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?

Solution:

The execution of the depth-first search only changes in the last step, where I is visited from F instead of starting the call of "visit(I)" after completing "visit(A)".

This gives the following post-ordering: B, E, D, H, G, C, I, F, A. Since the graph has no back edges anymore, it has a topological ordering. The topological ordering we get from the execution of the depth-first search (reversed post-ordering) is: A, F, I, C, G, H, D, E, B.

The pre-ordering is A, C, D, E, B, G, H, F, I; it does not give a topological ordering, since there is for example the edge (G, B) in the graph.

Guidelines for correction:

The following 5 elements are important in this exercise. If all of them are solved correctly, award 1 point. If at least 3 are solved correctly, award $1/2$ point.

- Labeling the graph in parts (a) and (d).
- Determining the pre- and post numbers as well as the pre- and post orderings in parts (b) and (c).
- Finding a directed cycle in part (e) using the execution of the DFS.
- Mentioning what it means if $I_u \subset I_v$ in part (f).
- Answering the three question in part (g).

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.

Solution:

V is the set of major cities in Switzerland (which is of size $|V| = n$), and there is a directed edge from $u \in V$ to $v \in V$ if and only if there is a highway going from city u to city v. The corresponding graph problem is to determine whether for any two vertices $u, v \in V$, there is a (directed) path from u to v in $G = (V, E)$.

(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.*

Solution:

The algorithm is the following. Let $v_0 \in V$ be any vertex in the graph. We first run DFS starting from vertex v_0 on the graph G described in part (a), and denote by V_0 the set of vertices that were visited by the DFS during "visit(v_0)" (including v_0 itself). Then, we define a new graph $G' = (V, E')$ with the same vertices, and whose edges are given by the reversed edges of G, i.e. we have $E' = \{(v, u) \in V^2 : (u, v) \in E\}$. We run DFS again starting from vertex v_0 on the graph G', and denote by V_0' the set of vertices that were visited by the DFS (again only during "visit(v_0)" and including v_0). The algorithm outputs that the claim is correct if $V_0 = V = V_0'$, and that the claim is false otherwise. Note that the first DFS takes time $O(|V| + |E|)$, constructing the graph G' takes times $O(|V| + |E|)$ and the second DFS takes also time $O(|V| + |E'|)$. Since $|V| = n$ and $|E'| = |E| = m$ the total runtime is indeed $O(n + m)$.

For correctness, we have to show the equivalence of the following two statements:

(1) For any two vertices $u, v \in V$, there is a (directed) path from u to v in $G = (V, E)$.

(2) $V_0 = V$ and $V_0' = V$.

(1) $\Longrightarrow$ (2):

Note that V_0 is the set of vertices v for which there is a directed path from v_0 to v in G. By (1), there is a path from v_0 to v for all vertices $v \in V$, and thus $V_0 = V$. On the other hand, V_0' is the set of all vertices v for which there is a directed path from v to v_0 in G. Indeed, if $v \in V_0'$, this means that in the graph G' with reversed edges there is a path from v_0 to v, which corresponds to a path from v to v_0 in the original graph G. Again by (1), we conclude that $V_0' = V$. Together, these show (2).

(2) $\Longrightarrow$ (1):

Let $u, v \in V$. Since $V_0' = V$ and we have seen that V_0' is the set of vertices from which we can reach v_0 in G, we know there is a directed path P_u from u to v_0 in G. Since $V_0 = V$ and V_0 is the set of vertices that we can reach from v_0 in G, we know there is a directed path P_v from v_0 to v in G. Concatenating the paths P_u and P_v, we obtain a directed walk from u to v in G. Since the existence of a walk from u to v is equivalent to the existence of a path from u to v, this shows (1).

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.*

Solution:

For each $v \in V$, create a list $L_v = [v]$. We iteratively apply the following procedure:

(i) Apply the algorithm of Exercise 9.5 to find two strongly connected vertices, say v, w in G. If no such vertices exist, stop and output L_v for each vertex v that is still in G.

(ii) Set $L_v \leftarrow L_v \cup L_w$.

(iii) For every in-neighbor x of w (except possibly v) add an edge (x, v) to G. For every out-neighbor y of w (except possibly v) add an edge (v, y) to G. Then remove w from G.

For the runtime of the algorithm, note that in each iteration, one vertex is removed from G, and so there can be at most n iterations. Each iteration can be executed in time $O(n + m)$, leading to total runtime $O(n \cdot (n + m))$.

For correctness, note first that, at any point during the algorithm, for any vertex v still in G, all elements of the list L_v are strongly connected to v. Second, note that after removing w in step, and adding the new edges in step (iii), we have not changed strong connectivity of any of the remaining vertices. We conclude that, when the algorithm terminates, the lists L_v of each remaining vertex v contains the strongly connected component of G to which v belongs.

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.

Solution:

Let V_{v_1} be the strongly connected component to which v_1 belongs. For any $w \in V_{v_1}$, there must be a directed path from w to v_1 in G, by definition. But then there is a directed path from v_1 to w in $\overleftarrow{G}$, meaning $w \in W$. So, $V_{v_1} \subseteq W$.

Now let $w \in W$. By definition, there is a directed path from w to v_1 in G. By part (b), we know this means that either v and w are strongly connected, or there exists a $u \in V$ which appears before v_1 in L, and is strongly connected to w. But v_1 is the first element of L, and so we must have that v and w are strongly connected, i.e., $w \in V_{v_1}$. So, $V_{v_1} \supseteq W$.

(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*.*

Solution:

We first compute the list L as in the above, and then we apply DFS on $\overleftarrow{G}$, with two modifications:

(i) We always call visit on the left-most unmarked vertex $v \in V$ in L.

(ii) We count the number k of (outer) calls made to visit. Whenever we reach an unmarked vertex in the k-th call to visit, we mark that vertex with this number.

The total runtime of our algorithm is $O(n+m)$ (as we traverse G only twice using DFS). We claim that when the algorithm terminates, the strongly connected components of G are given by the sets $V_k := \{v \in V : \text{mark}(v) = k\}$. That is, they are exactly the unmarked vertices reached by each of the (outer) calls to visit.

To prove the claim, we use induction on k. We have shown the base case in part (c).

Now let $k > 1$, and let W be the set of unmarked vertices reached by the k-th (outer) call $\text{visit}(v_k)$ during our algorithm.

We show first $W \subseteq V_k$, the strongly connected component containing v_k. Let $w \in W$. Then, by definition, there is a directed path from w to v_1 in G. By part (b), we know this means that either v and w are strongly connected, or there exists a $u \in V$ which appears before v_1 in L, and is strongly connected to w. In the former case we are done. In the latter, note that u must have already been reached in an earlier visit call (as it appears before v_1, and we make outer visit calls in order of L).

But that is not possible, as, by induction[1], this means w should have already been marked (since it is in the same strongly connected component as u). So we cannot be in the latter case.

To show that $W \supseteq V_k$, let $u \in V_k$. Then by definition, there is a directed path P from v_k to u in $\overleftarrow{G}$. If no vertex on this path was already marked (before the current call to visit), then $u \in W$, and we are done. So suppose for a contradiction that there is a marked vertex x in P. Note that, since $u \in V_k$, there is a directed cycle containing u and v_k in G. But also, there is directed path from u to x and from x to v_k in G. That is, v_k is strongly connected to x. But then, using induction, v_k should have already been marked in an earlier call to visit (the one that marked x).

Guidelines for correction:

The following points are important in this exercise:

- In part (a), the correct construction.
- In part (a), a proof of correctness.
- In part (c), a correct proof of $\subseteq$.
- In part (c), a correct proof of $\supseteq$.
- In part (d), a correct algorithm.
- In part (d), a correct proof of $\subseteq$.
- In part (d), a correct proof of $\supseteq$.

If at least 2 of these bullets are present, award $1/2$ points. If at least 5 are present, award 1 point.

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

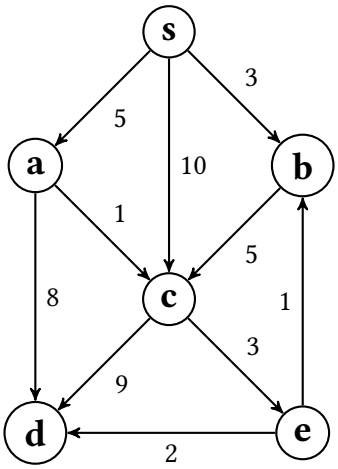
Input: a weighted graph, represented via $c(\cdot, \cdot)$. Specifically, for two vertices u, v the value $c(u, v)$ represents the cost of an edge from u to v (or ∞ if no such edge exists).
function DIJKSTRA(G, s)
  $d[s] \leftarrow 0$ ▷ upper bounds on distances from s
  $d[v] \leftarrow \infty$ for 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 ∞)

[1]It is not necessarily true that u was marked in the previous call to visit (it could have been any earlier call). However, we use here that the induction hypothesis holds for *all* earlier calls. This principle is called *strong* induction.

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.

Solution:

In the beginning: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = d[\mathbf{b}] = d[\mathbf{c}] = d[\mathbf{d}] = d[\mathbf{e}] = \infty$, $M = \emptyset$.

After we choose **s**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 10$, $d[\mathbf{d}] = d[\mathbf{e}] = \infty$, $M = \{\mathbf{s}\}$, there is no $p(\mathbf{s})$.

After we choose **b**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 8$, $d[\mathbf{d}] = d[\mathbf{e}] = \infty$, $M = \{\mathbf{s}, \mathbf{b}\}$, there is no $p(\mathbf{s})$, $p(\mathbf{b}) = \mathbf{s}$.

After we choose **a**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 6$, $d[\mathbf{d}] = 13$, $d[\mathbf{e}] = \infty$, $M = \{\mathbf{s}, \mathbf{a}, \mathbf{b}\}$, there is no $p(\mathbf{s})$, $p(\mathbf{a}) = p(\mathbf{b}) = \mathbf{s}$.

After we choose **c**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 6$, $d[\mathbf{d}] = 13$, $d[\mathbf{e}] = 9$, $M = \{\mathbf{s}, \mathbf{a}, \mathbf{b}, \mathbf{c}\}$, there is no $p(\mathbf{s})$, $p(\mathbf{a}) = p(\mathbf{b}) = \mathbf{s}, p(\mathbf{c}) = \mathbf{a}$.

After we choose **e**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 6$, $d[\mathbf{d}] = 11$, $d[\mathbf{e}] = 9$, $M = \{\mathbf{s}, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{e}\}$, there is no $p(\mathbf{s})$, $p(\mathbf{a}) = p(\mathbf{b}) = \mathbf{s}, p(\mathbf{c}) = \mathbf{a}, p(\mathbf{e}) = \mathbf{c}$.

After we choose **d**: $d[\mathbf{s}] = 0, d[\mathbf{a}] = 5, d[\mathbf{b}] = 3, d[\mathbf{c}] = 6, d[\mathbf{d}] = 11, d[\mathbf{e}] = 9, M = \{\mathbf{s}, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}\}$, there is no $p(\mathbf{s})$, $p(\mathbf{a}) = p(\mathbf{b}) = \mathbf{s}, p(\mathbf{c}) = \mathbf{a}, p(\mathbf{d}) = \mathbf{e}, p(\mathbf{e}) = \mathbf{c}$.

We find the shortest path by backtracking them using the predecessors. In this example, they are given by **s**, **s** $\to$ **a**, **s** $\to$ **b**, **s** $\to$ **a** $\to$ **c**, **s** $\to$ **a** $\to$ **c** $\to$ **e** $\to$ **d** and **s** $\to$ **a** $\to$ **c** $\to$ **e**.

(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?

Solution:

The algorithm works correctly.

In the beginning: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = d[\mathbf{b}] = d[\mathbf{c}] = d[\mathbf{d}] = d[\mathbf{e}] = \infty$.

After we choose **s**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 10$, $d[\mathbf{d}] = d[\mathbf{e}] = \infty$.

After we choose **b**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 8$, $d[\mathbf{d}] = d[\mathbf{e}] = \infty$.

After we choose **a**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 4$, $d[\mathbf{d}] = 13$, $d[\mathbf{e}] = \infty$.

After we choose **c**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 4$, $d[\mathbf{d}] = 13$, $d[\mathbf{e}] = 7$.

After we choose **e**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 4$, $d[\mathbf{d}] = 9$, $d[\mathbf{e}] = 7$.

After we choose **d**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 4$, $d[\mathbf{d}] = 9$, $d[\mathbf{e}] = 7$.

(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.

Solution:

The algorithm doesn't work correctly, for example, the distance from **s** to **b** is 1 (via the path **s**-**a**-**c**-**e**-**b**), but the algorithm computes exactly the same values of $d[\cdot]$ as in part (b), so $d[\mathbf{b}] = 3$.