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
Kasper Lindberg, Lucas Slot, Hongjie Chen, Manuel Wiedmer

25 November 2024

Algorithms & Data Structures — Exercise sheet 10 — HS 24

The solutions for this sheet are submitted on Moodle until 1 December 2024, 23:59.

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.

The solutions are intended to help you understand how to solve the exercises and are thus more detailed than what would be expected at the exam. All parts that contain explanation that you would not need to include in an exam are in grey.

Exercise 10.1 *Shortcutting closed Eulerian walks* **(1 point)**.

Let $G = (V, E)$ be the complete graph meaning we have an edge set E equal to all possible edges between distinct vertices. Furthermore, there is a weight function $w : E \to \mathbb{R}_{\geq 0}$ on the edges which satisfies a 'metric' property: for any distinct $u, v, w \in V$ we have $w(\{u, w\}) \leq w(\{u, v\}) + w(\{v, w\})$. Given a sequence of edges $C = (e_1, \ldots, e_k)$ which form a cycle in G, the weight of this cycle is defined as $w(C) = \sum_{i=1}^{k} w(e_i)$.

Now consider another graph $H = (V, F)$ which has the same vertices as G. Furthermore, we can create a weight function $w' : F \to \mathbb{R}_{\geq 0}$ compatible with w meaning for each $f \in F$, if f's endpoints are u and v then $w'(f) = w(\{u, v\})$.

Suppose that H is connected and has a closed Eulerian walk denoted by a sequence of edges $T = (e_1, \ldots, e_k)$. The weight of the walk is defined as $w'(T) = \sum_{i=1}^{k} w'(e_i)$. Describe an algorithm that takes a closed Eulerian walk T of H and produces a Hamiltonian cycle C in G with weight at most that of the closed Eulerian walk, meaning $w(C) \leq w'(T)$. Your algorithm should have runtime $O(|F|)$. Argue why your algorithm is correct and why it satisfies the runtime bound.

Solution:

Algorithm Say our closed Eulerian walk is $T = (e_1, \ldots, e_k)$, then since this is a walk on the vertices of a graph, we can write it as a sequence of the vertices it visits $W = (v_1, \ldots, v_k, v_{k+1})$ where $v_1 = v_{k+1}$ and $e_i = \{v_i, v_{i+1}\}$. Since our graph is connected, the walk must reach every vertex so W contains every vertex at least once.

To convert this walk into a Hamiltonian cycle on the vertices using edges from G, we essentially want to use W until we hit a repeat vertex and then 'shortcut' to the next new vertex. Formally, your Hamiltonian cycle C will be a subsequence of W where for each vertex $v \in V$ we remove all occurrences of v except for the first one in W. Then re-add v_1 to the end of this new subsequence, to get C. Since all edges are available to you in G, this walk C is indeed realizable in G and is a Hamiltonian cycle since we visit each vertex once (except the first one).

Runtime Since constructing W from T takes $O(|F|)$ time and constructing C from W also takes $O(|F|)$ time, the algorithm has runtime $O(|F|)$.

Correctness proof Say $C = (u_1, \dots, u_n, u_{n+1})$ with $u_1 = u_{n+1}$, what we need to prove now is that $\sum_{i=1}^{n} w(\{u_i, u_{i+1}\}) \leq w'(T)$. To do this, consider the various intervals of indexes of W which are deleted to get C. We write these as $I_1, \dots, I_\ell$ where $I_i = \{a_i, a_i + 1, \dots, b_i\}$ for $a_i < b_i$. For simplicity, fix an interval I_i and let $a = a_i$ and $b = b_i$. Then instead of the interval of edges $(\{v_{a-1}, v_a\}, \{v_a, v_{a+1}\}, \dots, \{v_b, v_{b+1}\})$ we have from W, we replace it with $(v_{a-1}, v_{b+1}) \in E$, call this a shortcut edge. Using the metric property multiple times we get

$$\begin{aligned} w(\{v_{a-1}, v_{b+1}\}) &\leq w(\{v_{a-1}, v_a\}) + w(\{v_a, v_{b+1}\}) \\ &\leq w(\{v_{a-1}, v_a\}) + w(\{v_a, v_{a+1}\}) + w(\{v_{a+1}, v_{b+1}\}) \\ &\leq \dots \\ &\leq \sum_{j=a}^{b+1} w(\{v_{j-1}, v_j\}). \end{aligned}$$

By compatibility of w and w' we can replace the w in the last line with w' since those are all edges in F.

Consider a sum over all shortcut edges, then we get the inequality

$$\sum_{i=1}^{\ell} w(\{v_{a_i-1}, v_{b_i+1}\}) \leq \sum_{i=1}^{\ell} \sum_{j=a_i}^{b_i+1} w'(\{v_{j-1}, v_j\}).$$

For edges in T not deleted by any interval, these will remain in C and compatibility of the weights means they are the same. So adding these edges to both sides of the equation we now consider the full walks on both sides meaning

$$w(C) = \sum_{i=1}^{n} w(\{u_i, u_{i+1}\}) \leq \sum_{i=1}^{k} w'(\{v_i, v_{i+1}\}) = w'(T).$$

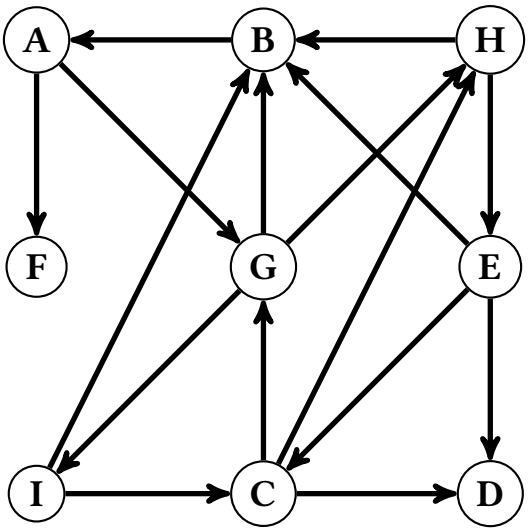
Guidelines for correction:

Award $1/2$ point for the right general idea of shortcutting but incorrect proof. Award 1 point for right idea and right proof (though there are many ways to describe the process and they don't all need to be this formal).

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

Execute a breadth-first search (BFS) on the following graph $G = (V, E)$ starting at vertex A using the algorithm you have seen in the lecture, which updates a queue Q.

You should always enqueue the neighbours of a vertex in alphabetical order.



(a) Write down all operations that are executed on the queue Q during the BFS (in order), and write down the contents of Q after each operation is executed (in order). For example, the first operation is $\text{enqueue}(A)$ and after this operation, $Q = [A]$. The second operation is $\text{dequeue}(A)$, and after this operation, $Q = [\,]$.

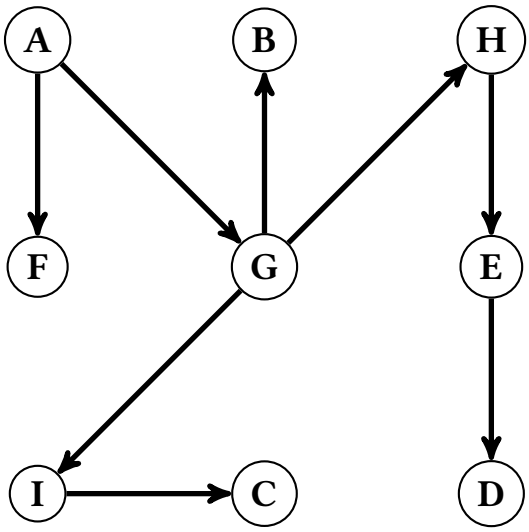
Solution:

0	$\text{enqueue}(A)$	$Q = [A]$
1	$\text{dequeue}(A)$	$Q = []$
2	$\text{enqueue}(F)$	$Q = [F]$
3	$\text{enqueue}(G)$	$Q = [F, G]$
4	$\text{dequeue}(F)$	$Q = [G]$
5	$\text{dequeue}(G)$	$Q = []$
6	$\text{enqueue}(B)$	$Q = [B]$
7	$\text{enqueue}(H)$	$Q = [B, H]$
8	$\text{enqueue}(I)$	$Q = [B, H, I]$
9	$\text{dequeue}(B)$	$Q = [H, I]$
10	$\text{dequeue}(H)$	$Q = [I]$
11	$\text{enqueue}(E)$	$Q = [I, E]$
12	$\text{dequeue}(I)$	$Q = [E]$
13	$\text{enqueue}(C)$	$Q = [E, C]$
14	$\text{dequeue}(E)$	$Q = [C]$
15	$\text{enqueue}(D)$	$Q = [C, D]$
16	$\text{dequeue}(C)$	$Q = [D]$
17	$\text{dequeue}(D)$	$Q = []$

(b) Give a *shortest path tree* for G (with root A).

Solution:

An example of a shortest path tree is:



(c) Let $S_k := \{v \in V : \mathrm{dist}(A, v) = k\}$. Write down S_k for $k = 0, 1, 2, 3, 4, 5$.

Solution:

We have: $S_0 = \{A\}, S_1 = \{F, G\}, S_2 = \{B, H, I\}, S_3 = \{C, E\}, S_4 = \{D\}, S_5 = \emptyset$.

(d) An edge $e \in E$ is called *critical* if there is a vertex $v \in V$ to which the distance from A increases after removing e. Write down all critical edges of G.

Solution:

For an edge to be critical, it must be part of the shortest path tree of part (b). But, not every edge in that tree has to be critical. We check by hand that all edges in the tree except (E, D) are critical. Note that this corresponds to the fact that we could have replaced (E, D) by (C, D) to obtain another shortest path tree. Or, to the fact that all shortest paths in G starting at A are unique, except the one from A to D.

(e) Write down the enter-number and leave-number of each vertex of the graph. (These numbers indicate the number of queue-operations that have been executed as a vertex enters (respectively, leaves) the queue. For example, $\mathrm{enter}(A) = 0$ and $\mathrm{leave}(A) = 1$.)

Solution:

vertex	enter	leave
A	0	1
B	6	9
C	13	16
D	15	17
E	11	14
F	2	4
G	3	5
H	7	10
I	8	12

(f) Let $t_k := \min_{v \in V}\{\mathrm{leave}(v) : \mathrm{dist}(A, v) \geq k\}$. Write down t_k for $k = 0, 1, 2, 3, 4, 5$. (We define $\min \emptyset := \infty$.)

Solution:

Using (c) and (e) we find that $t_0 = 1, t_1 = 4, t_2 = 9, t_3 = 14, t_4 = 17, t_5 = \infty$.

(g) Let $R_k := \{v \in V : t_k \leq \text{leave}(v) < t_{k+1}\}$. Write down R_k for $k = 0, 1, 2, 3, 4, 5$.

Solution:

We have: $R_0 = \{A\}, R_1 = \{F, G\}, R_2 = \{B, H, I\}, R_3 = \{C, E\}, R_4 = \{D\}, R_5 = \emptyset$.

Note that, as you have seen in the lecture, $S_i = R_i$ for all i.

Guidelines for correction:

Award 1 point if there is at most 1 mistake in all of the parts above. Award 1/2 point if there are at most 3 mistakes in all of the parts above.

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 $G = (V, E)$. Note that this is equivalent to checking whether the directed graph is strongly connected.

(b) Describe an algorithm that checks the correctness of the government's 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:

Algorithm description. The algorithm is the following. Let $v_0 \in V$ be an arbitrary 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 $E' = \{(v, u) : (u, v) \in E\}$. 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.

The idea of this algorithm is to check whether there is a path from v_0 to any other vertex and whether there is a path from any other vertex to v_0. If directing the edges preserves the connectivity

as claimed by the Swiss government then we can go from any city u to v_0 and from v_0 to any city u so the algorithm correctly recognizes this. On the other hand, if $V = V_0 = V_0'$ then we know that all vertices can be reached from v_0 and all vertices can reach v_0 so there is a walk (and thus also a path) between any pair of vertices. Correctness is shown formally in the following paragraph.

Correctness. 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).

Runtime. Note that the first DFS takes time $O(|V| + |E|)$, constructing the graph G' takes times $O(|V| + |E|)$ and the second DFS also takes time $O(|V| + |E'|)$. Since $|V| = n$ and $|E'| = |E| = m$ the total runtime is indeed $O(n + m)$.

Exercise 10.4 *Number of minimal paths* **(1 point)**.

Let $G = (V, E)$ be an undirected graph with n vertices and m edges. Let $v, v' \in V$ be two distinct vertices and suppose that the distance between the two is k.

Describe an algorithm which counts the number of paths from v to v' of length k. The runtime of your algorithm should be at most $O(n + m)$. You are provided with the number of vertices n, and the adjacency list Adj of G. Argue why your algorithm is correct and why it satisfies the runtime bound.

Hint: *Modify BFS.*

Solution:

Algorithm Consider a BFS procedure starting at v and for each vertex u we have two variables d_u and p_u which we want to, at the end, represent the distance from v to u and the number of paths from v to u of that distance. Initialize $d_v \leftarrow 0$ and $p_v \leftarrow 1$ and for all other vertices, $d_u \leftarrow \infty$ and $p_u \leftarrow 0$. Initialize all vertices as unexplored. We start from v and mark it as explored. Say in the BFS procedure we are at a vertex u, then we go to all adjacent vertices w. If w is unexplored then we set $d_w \leftarrow d_u + 1$ and $p_w \leftarrow p_u$ and mark w as explored and add it to the queue. Otherwise if w is explored and $d_w = d_u + 1$ then we update $p_w \leftarrow p_w + p_u$. And finally if $d_w < d_u + 1$ we do nothing.

After completing the BFS procedure we output $p_{v'}$.

Runtime Since the above algorithm only has $O(1)$ additional runtime at each step of a BFS, its runtime is (asymptotically) the same as BFS, i.e. $O(n+m)$.

Correctness proof To argue correctness, we first recall that at any point of the algorithm, the queue $Q = \{v_1, \ldots, v_k\}$ satisfies $d_{v_1} \leq \cdots \leq d_{v_k} \leq d_{v_1} + 1$. We can show this formally by induction. The base case is when we just have the starting vertex v so this vacuously holds. Now suppose at timestep t of the algorithm our queue looks like $Q = \{v_1, \ldots, v_k\}$. Then we process the vertex v_1 at time step $t+1$ and it adds all its unexplored neighbors $u_1, \ldots, u_\ell$ to the queue and sets $d_{u_1} = \cdots = d_{u_\ell} = d_{v_1} + 1$. Our new queue looks like $\{v_2, \ldots, v_k, u_1, \ldots, u_\ell\}$. Since $d_{v_1} \leq d_{v_2} \leq \cdots \leq d_{v_k} \leq d_{v_1} + 1$ by the inductive hypothesis and $d_{v_1} + 1 = d_{u_1} = \cdots = d_{u_\ell} \leq d_{v_2} + 1$, we get

$$d_{v_2} \leq \cdots \leq d_{v_k} \leq d_{u_1} \leq \cdots \leq d_{u_\ell} \leq d_{v_2} + 1.$$

Now stringing all these large inequalities together for each time step of the queue, we can conclude that $d_{v_1} \leq d_{v_2} \leq \cdots \leq d_{v_n}$ where vertex v_i is the vertex processed at timestep i, and $v_1 = v$.

Consider a minimal path from v to a vertex w, $P = (w_0, w_1, \ldots, w_k)$ where $w_0 = v$ and $w_k = w$. Then for sake of contradiction, let index i be the first index such that $d_{w_i} > i$. But then at index $i-1$ we have $d_{w_{i-1}} \leq i - 1 < d_{w_i}$ and by our previous fact, this means that w_{i-1} was processed before w_i. Since w_{i-1} and w_i are adjacent it must have been that w_i had been marked explored by a previous vertex, but this is a contradiction since any node u processed before w_{i-1} has $d_u \leq d_{w_{i-1}} \leq i-1$ and we have that $d_{w_i} = d_u + 1$. Since P was a minimal path, we can conclude that these d_w do in fact correspond to the distance between v and w.

Finally, every minimal path to w, written as $P = (w_1, \ldots, w_{k-1}, w_k)$ where $w_1 = v$ and $w_k = w$, is also a minimal path to w_{k-1} if you delete w_k. Thus the total number of minimal paths to w is the sum of the minimal paths ending at u where u is adjacent to w and of distance $k-1$. We can see that since the d_u and d_w are always equal to the correct distances once discovered, the algorithm indeed implements this sum.

Guidelines for correction:

Award $1/2$ point for a correct algorithm but incorrect proof. Award 1 point for correct algorithm and correct proof.

Proof correctness doesn't need to be too rigorous but should prove path counter = sum of path counters of vertices adjacent and one less distance.

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 Dijkstra(G, s)

Input: A starting vertex s and a weighted graph G represented via $c(\cdot, \cdot)$. Specifically, for two vertices u, v the value $c(u, v)$ represents the cost of the edge from u to v (or ∞ if no such edge exists).

Operations:

$d[s] \leftarrow 0$ ▷ 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

  for each $v \in V \setminus S$ **do**

    for each $u \in S$ **do**

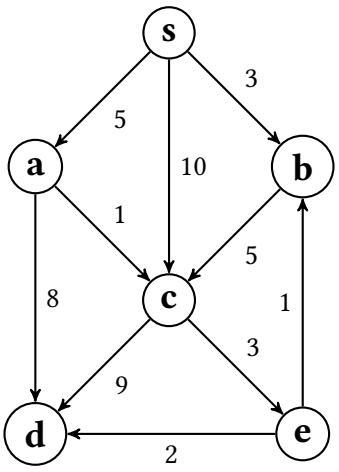
      if $c(u, v) < \infty$ **then**

        $d[v] \leftarrow \min\{d[v], d[u] + c(u, v)\}$

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 next lecture. 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 iteration of the while-loop, write down:

1) $d[u]$ for all $u \in V$ (which are upper bounds on the distances from **s** to u computed so far),

2) the set S (which contains vertices for which the distance has been correctly computed so far),

3) and the predecessors for each vertex $u \in S \setminus \{s\}$. (A predecessor of a vertex $u \in S \setminus \{s\}$ is a vertex $v \in S$ which satisfies $d[u] = d[v] + c(v, u)$.)

Solution:

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

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

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

After we choose **a**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 6$, $d[\mathbf{d}] = 13$, $d[\mathbf{e}] = \infty$, $S = \{\mathbf{s}, \mathbf{a}, \mathbf{b}\}$, $p(\mathbf{a}) = p(\mathbf{b}) = \mathbf{s}$.

After we choose **c**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 6$, $d[\mathbf{d}] = 13$, $d[\mathbf{e}] = 9$, $S = \{\mathbf{s}, \mathbf{a}, \mathbf{b}, \mathbf{c}\}$, $p(\mathbf{a}) = p(\mathbf{b}) = \mathbf{s}, p(\mathbf{c}) = \mathbf{a}$.

After we choose **e**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 6$, $d[\mathbf{d}] = 11$, $d[\mathbf{e}] = 9$, $S = \{\mathbf{s}, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{e}\}$, $p(\mathbf{a}) = p(\mathbf{b}) = \mathbf{s}, p(\mathbf{c}) = \mathbf{a}, p(\mathbf{e}) = \mathbf{c}$.

After we choose **d**: $d[\mathbf{s}] = 0$, $d[\mathbf{a}] = 5$, $d[\mathbf{b}] = 3$, $d[\mathbf{c}] = 6$, $d[\mathbf{d}] = 11$, $d[\mathbf{e}] = 9$, $S = \{\mathbf{s}, \mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}\}$, $p(\mathbf{a}) = p(\mathbf{b}) = \mathbf{s}, p(\mathbf{c}) = \mathbf{a}, p(\mathbf{d}) = \mathbf{e}, p(\mathbf{e}) = \mathbf{c}$.

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

(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 the minimum 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 $d[\mathbf{b}] = 3$.

This example shows that Dijkstra's algorithm stopps working correctly if if we allow negative edge weights, which is generally important to remember. The reason for this is that if we chose v^* with minimal $d[v^*]$, this value is no longer equal to the distance between s and v^* because there could be a path going over previously unexplored vertices that is actually shorter than the path the algorithm has found when processing v^*. This cannot happen for non-negative weights.

One might further think that adding the absolute value of the most negative edge weight w to all edges and then running Dijkstra would fix this problem, but this is also false. The reason is that we add w to all edges so the length of paths with more edges increases stronger than the length of paths with few edges. So it can happen that after adding w, the previously shortest path P between some pair of vertices is now actually longer than some other path P' because P' has fewer edges than P and is therefore less affected by adding w.