Eidgenössische
Technische Hochschule
Zürich

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

Departement of Computer Science
Markus Püschel, David Steurer
Johannes Lengler, Gleb Novikov, Chris Wendler, Ulysse Schaller

30. November 2020

Algorithms & Data Structures — Exercise sheet 11 — HS 20

Exercise Class (Room & TA): _____

Submitted by: _____

Peer Feedback by: _____

Points: _____

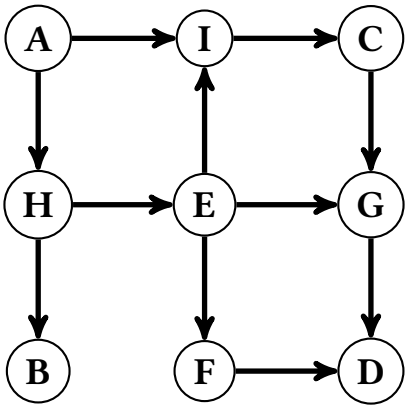
Submission: On Monday, 07. December 2020, hand in your solution to your TA *before* the exercise class starts. Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

Exercise 11.1 *Breadth-First Search.*

Execute a breadth-first search (Breitensuche) on the following graph starting from vertex A. Use the algorithm presented in the lecture.

Give the order in which the vertices are enqueued by a breadth-first search starting in A. You should list vertices several times if they are enqueued more than once. When processing the neighbors of a vertex, process them in alphabetical order.

The BFS-order is the order in which the vertices are enqueued for the first time in a breadth-first search. What is the BFS-order of the above execution? Does it give a topological sorting?



Solution: The order in which the vertices are enqueued by a breadth-first search is

A, H, I, B, E, C, F, G, G, D, D.

The BFS-order is A, H, I, B, E, C, F, G, D. It does not give a topological ordering, since there is an edge (E, I) in the graph.

Exercise 11.2 *Shortest paths by hand* **(1 Point)**.

At the end of the lecture on November 26, we discussed an algorithm for finding shortests paths when all edge costs are nonnegative. Here is the pseudo-code for that algorithm, which is typically called Dijkstra's algorithm:

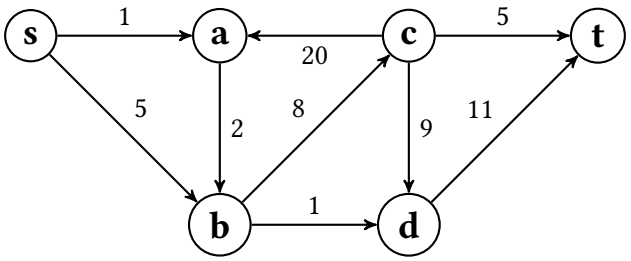
---

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 ∞)

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 exercises from this sheet.

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 $v \in V \setminus S$, which will be discussed during the lecture on December 3. In the next sheets and the exam you should use the efficient version of the algorithm (not 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 node in the graph. After each step (i.e. after each choice of v^*), write down:

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

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

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

   Solution: When we choose s: $d[s] = 0$, $d[a] = d[b] = d[c] = d[d] = d[t] = \infty$, $M = \{s\}$, there is no $p(s)$.

   When we choose a: $d[s] = 0$, $d[a] = 1$, $d[b] = 5$ $d[c] = d[d] = d[t] = \infty$, $M = \{s, a\}$, there is no $p(s)$, $p(a) = p(b) = s$.

   When we choose b: $d[s] = 0$, $d[a] = 1$, $d[b] = 3$, $d[c] = d[d] = d[t] = \infty$, $M = \{s, a, b\}$, there is no $p(s)$, $p(a) = s, p(b) = a$.

When we choose d: $d[s] = 0, d[a] = 1, d[b] = 3, d[c] = 11, d[d] = 4, d[t] = \infty, M = \{s, a, b, c, d\}$, there is no $p(s), p(a) = s, p(b) = a, p(c) = b, p(d) = b$.

When we choose c: $d[s] = 0, d[a] = 1, d[b] = 3, d[c] = 11, d[d] = 4, d[t] = 15, M = \{s, a, b, d, c, t\}$, there is no $p(s), p(a) = s, p(b) = a, p(c) = b, p(d) = b, p(t) = d$.

When we choose t: $d[s] = 0, d[a] = 1, d[b] = 3, d[c] = 11, d[d] = 4, d[t] = 15, M = \{s, a, b, d, c, t\}$, there is no $p(s), p(a) = s, p(b) = a, p(c) = b, p(d) = b, p(t) = d$.

b) Change the weight of the edge $(\mathbf{b}, \mathbf{d})$ 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.

When we choose s: $d[s] = 0, d[a] = d[b] = d[c] = d[d] = d[t] = \infty$.

When we choose a: $d[s] = 0, d[a] = 1, d[b] = 5, d[c] = d[d] = d[t] = \infty$.

When we choose b: $d[s] = 0, d[a] = 1, d[b] = 3, d[c] = \infty, d[d] = d[t] = \infty$.

When we choose d: $d[s] = 0, d[a] = 1, d[b] = 3, d[c] = 11, d[d] = 2, d[t] = \infty$.

When we choose c: $d[s] = 0, d[a] = 1, d[b] = 3, d[c] = 11, d[d] = 2, d[t] = 13$.

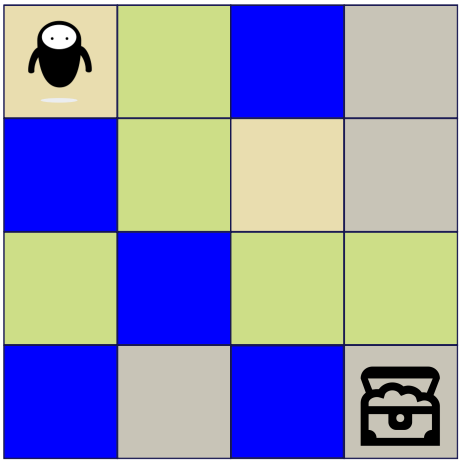
When we choose t: $d[s] = 0, d[a] = 1, d[b] = 3, d[c] = 11, d[d] = 2, d[t] = 13$.

c) Now, additionally change the weight of the edge $(\mathbf{c}, \mathbf{d})$ from 9 to -10. 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 d is 1, but the algorithm computes exactly the same values of $d[\cdot]$ as in part b), so $d[d] = 2$.

Exercise 11.3 *Robot.*

Consider the following game:



The game is played on a $d \times d$ board with four different types of fields (grassland, water, desert and mountain). You start at top left field and have to move to the bottom right field. At each turn you may move to an adjacent field (a field that shares a border with your current field). Moving through a

grassland field requires 3 minutes, moving through a desert field requires 5 minutes, moving through a mountain field requires 7 minutes and you cannot swim (which makes water fields impassable). You spend zero time in the starting and target field. You may assume that there is always at least one possible way from top left to bottom right. The goal of this exercise is to find the fastest way from the top left field to the bottom right field.

a) Model the problem as a graph problem:

1) Describe your graph. What are the vertices, what are the edges and the weights of the edges?

Solution: The graph $G = (V, E, w)$ is as follows: V is a set of non-water fields, and there are two directed edges (u, v) and (v, u) if and only if fields $u \in V$ and $v \in V$ share a border. The weight $w((u, v))$ of an edge $(u, v) \in E$ is the time that is required to move through the field $v \in V$ (so for a target field t and for each edge $(u, t) \in E$, $w((u, t)) = 0$).

2) What is the graph problem that we are trying to solve?

Solution: Finding the shortest path between the starting field $s \in V$ and the target field $t \in V$ in the graph $G = (V, E, w)$.

3) Solve the problem using an algorithm discussed in the lecture (without modification).

Solution: We can apply Dijkstra's algorithm to (G, s) to find the shortest path between s and t.

b)* Now, we modify the game a little bit: You learned how to swim and, thus, moving through a water field requires 11 minutes. However, swimming is very exhausting for you and you cannot swim through more than w water fields. Again, you may assume that the game boards are generated in a way that they are solvable using w of the water fields. Model the modified problem as a graph problem. Find a description as graph problem such that you can directly apply one of the algorithms in the lecture, without modifications to the algorithm.

1) Describe your graph. What are the vertices, what are the edges and what are the weights on the edges?

Solution: The graph G is as follows: for each non-water non-target field a, V contains $w + 1$ vertices $a_0, \dots a_w$, and for each water field b, V contains w vertices $b_1, \dots, b_w$. For a target field V contains one vertex t.

If non-water fields a and a' share a border, then E contains directed edges (a_i, a'_i) and (a_i, a'_i) for all $i = 0, 1, \dots, w$.

If the target field t shares a border with some non-water field a, then E contains directed edges (a_i, t) for all $i = 0, 1, \dots, w$. If the target field t shares a border with some water field b, then E contains directed edges (b_i, t) for all $i = 1, \dots, w$.

If a non-water non-target field a and a water field b share a border, then E contains directed edges (a_i, b_{i+1}) and (b_i, a_i) for all $i = 0, 1, \dots, w$. If two water fields b and b' share a border, then E contains directed edges (b_j, b'_{j+1}) and (b'_j, b_{j+1}) for all $j = 1, \dots, w - 1$.

The weight of an edge $(u, v) \in E$ is the time that is required to move through the field $v \in V$ (so for a target field t and for each edge $(u, t) \in E$ its weight is 0).

You can understand the graph G as $w + 1$ copies (levels) of the graph from part a), connected in such a way that matches the described problem. Each copy has the following crucial property: if one stands at a vertex belonging to the i-th level of G, then we are still allowed to use $w - i$ water fields up to the target vertex t.

Indeed, we start the run on the 0-th level, (which makes sense since at the beginning we have w water fields left at our disposal). From that point on, we can always move within a given level i exactly like in the graph from part a), but we can additionally make water-field moves. Only, if we do such a move, we will need to keep track that we can use one water-field less than before. Therefore, we direct each edge corresponding to a water-field move to the corresponding end-vertex in the $(i+1)$-st level. By construction, the w-th level of the graph does not allow us to use any additional water-field moves, as desired.

Finally, it's important to mention that we do not need to use all of our possible water-field moves to reach the target. Hence, we merge all vertices corresponding to t together, making the target vertex reachable from any level.

2) What is the graph problem that we are trying to solve?

Solution: Finding the shortest path between the vertex $s_0 \in V$ that corresponds to the starting field s and the target vertex $t \in V$ in the graph G.

3) Solve the problem using an algorithm discussed in the lecture (without modification).

Solution: We can apply Dijkstra's algorithm to (G, s_0) to find the shortest path between s_0 and t.

Exercise 11.4 *Arbitrage.*

When trading currencies, *arbitrage* means to exploit price differences in order to profit by exchanging currencies multiple times. For example, on June 2nd, 2009, 1 US Dollar could be exchanged for 95.729 Yen, 1 Yen for 0.00638 Pound sterling, and 1 Pound sterling for 1.65133 US Dollars. If a trader exchanged 1 US Dollar for Yen, exchanged the obtained amount for Pound sterling and finally exchanged this amount back to US Dollars, he would have obtained $95.729 \cdot 0.00638 \cdot 1.65133 \approx 1.0086$ US Dollars, corresponding to a gain of 0.86%.

a) You are given n currencies $\{1, \ldots, n\}$ and an $(n \times n)$ exchange rate matrix R with positive rational number entries. For two currencies $i, j \in \{1, \ldots, n\}$ one unit of currency i can be exchanged for $R(i, j) > 0$ units of currency j. The goal is to decide whether an arbitrage opportunity exists, i.e., if there exists a sequence of k different currencies $W_1, \ldots, W_k \in \{1, \ldots, n\}$ such that $R(W_1, W_2) \cdot R(W_2, W_3) \cdots R(W_{k-1}, W_k) \cdot R(W_k, W_1) > 1$ holds.

Model the above problem as a graph problem. Show how the input can be transformed into a directed, weighted graph $G = (V, E, w)$ that contains a cycle with negative weight *if and only if* an arbitrage activity is possible. Justify why G contains a negative cycle if and only if an arbitrage opportunity exists.

Hint: *Using logarithms might be beneficial because of the property* $\ln(a \cdot b) = \ln(a) + \ln(b)$.

Solution. We create a complete graph $G = (V, E)$ with the vertices $V = \{1, \ldots, n\}$. An edge $(i, j) \in E$ gets the weight $w(i, j) = -\log R(i, j)$. Then suppose an arbitrage opportunity with the sequence of currencies $W_1, \ldots, W_k$ is possible. Then it must be the case that

$$
\begin{aligned}
& R(W_1, W_2) \cdot R(W_2, W_3) \cdot \ldots \cdot R(W_{k-1}, W_k) R(W_k, W_1) > 1 \\
\Leftrightarrow\ & \log\left(R(W_1, W_2) \cdot R(W_2, W_3) \cdot \ldots \cdot R(W_{k-1}, W_k) R(W_k, W_1)\right) > 0 \\
\Leftrightarrow\ & \log R(W_1, W_2) + \log R(W_2, W_3) + \ldots + \log R(W_{k-1}, W_k) + \log R(W_k, W_1) > 0 \\
\Leftrightarrow\ & -\log R(W_1, W_2) - \log R(W_2, W_3) - \ldots - \log R(W_{k-1}, W_k) - \log R(W_k, W_1) < 0 \\
\Leftrightarrow\ & w(W_1, W_2) + w(W_2, W_3) + \ldots + w(W_{k-1}, W_k) + w(W_k, W_1) < 0
\end{aligned}
$$

consequently G contains a cycle of negative weight. Because we only used equivalence transformations, the argument applies in both directions.

b) The Bellman-Ford algorithm can be used to find out whether a graph contains negative cycles. After ℓ iterations of the Bellman-Ford loop $d[v]$ is equal to $d(s,v)^{\leq \ell}$, i.e., the minimum weight of a s-v walk with at most ℓ edges. A graph contains a negative cycle that can be reached from s if and only if there exists a vertex v such that $d(s,v)^{\leq n-1} \neq d(s,v)^{\leq n}$, where n is the number of vertices.

Use the previous part of this exercise to design an algorithm that decides if an arbitrage opportunity exists. What is the best running time you can get (in terms of n)?

Solution. Since the graph is complete, any negatively-weighted cycle can be reached from any vertex, we can run Bellman-Ford to detect a cycle starting from any vertex in G. There are $|V| = n$ vertices and $|E| = n(n-1) \in \Theta(n^2)$ edges. The algorithm therefore has a running time of $\Theta(|V||E|) = \Theta(n^3)$.

Exercise 11.5 *Finding cheap train connections* **(2 points)**.

An efficient vaccine has been found and you can finally plan your next holiday. You want to visit a city in Europe and since you care about the environment, you will use only the train. Suppose that there are n cities in Europe that have a train station, including Zürich, which will be your starting point. You are wondering about which one you should visit, so you first decide to analyze the travel costs.

There are some direct train connections between these n cities (not necessarily in both ways), and for each of these connections you know its cost, which is a positive integer.

For simplicity you can assume that every city has a unique number from $\{1, \ldots, n\}$ and that each city is represented by its number.

a) Model these cities, direct train connections and their costs as a directed graph: give a precise description of the vertices, the edges and the weights of the edges of the graph $G = (V, E, w)$ involved.

Solution: Each city is a vertex in the directed graph. Two vertices $u, v \in V$ are connected by a directed edge $e \in E$, if there exists a direct train connection from city u to city v. The weight $w(e)$ of the edge $e = (u, v)$, is the cost of the direct train connection from u to v.

In b) and c) you can assume that the directed graph is represented by a data structure that allows you to traverse the direct predecessors and direct successors of a vertex u in time $\mathcal{O}(\deg_-(u))$ and $\mathcal{O}(\deg_+(u))$ respectively, where $\deg_-(u)$ is the in-degree of vertex u and $\deg_+(u)$ is the out-degree of vertex u.

b) You start from Zürich and you want to fill the array d of minimal traveling costs to each city. That is, for each city C, $d[C]$ is the minimal cost that you must pay to travel from Zürich to C by train.

Assume that you are given the graph from a). What is an efficient algorithm that fills the array d?

Solution: Dijkstra's Algorithm can solve this problem efficiently.

c) It can happen that there are different traveling options from Zürich to some city C that have minimal cost $d[C]$. In this case, you would like to take the option that requires the least stopovers (i.e. that uses the fewest direct connections), among those that have minimal cost.

Assume that you are given the graph from a) and the correctly filled array d. Find an efficient algorithm that fills the array ℓ such that for each city C, $\ell[C]$ is the minimal number of direct connections that you need to use in order to go from Zürich to C with minimal cost (i.e. with cost $d[C]$). What is the running time of your algorithm?

Solution: The following DP will compute the minimal number of direct connections needed in order to go from Zürich to any other city using minimal cost.

For simplicity, we identify the n cities with $\{1, \dots, n\}$. We first sort the cities by increasing minimal cost, i.e. 1 will correspond to Zurich (we need cost 0 to go to Zurich), and for each $1 \leq i \leq j \leq n$ it holds that $d[i] \leq d[j]$.

Dimensions of the DP table: The DP table is an array of length n.

Definition of the DP table: $DP[i]$ contains the minimal number of direct connections that are needed in order to go from Zurich to city i with minimal cost.

Computation of an entry: Initialize $DP[1]$ to 0: since all costs are positive, the cheapest way to go to Zurich is by not taking any train (i.e. using 0 direct connections).

Let $i > 1$. Again, since all costs are positive, any cheapest route to city i will only go through cities j that can be reached with a strictly cheapest cost. In other words, it only uses cities j with $d[j] < d[i]$, and since we have sorted the cities by increasing cost, $j < i$. Any city $j < i$ that has a direct connection to i can be used to get to i with minimal cost if and only if $d[i] = d[j] + w((j, i))$. Among those cities, we want to choose one that can be reached with as few direct connections as possible (while still achieving minimal cost $d[j]$). Therefore, the entries of DP can be computed as

$$DP[i] = 1 + \min\{DP[j] : j < i, (j, i) \in E, d[i] = d[j] + w((j, i))\},$$

where the minimum equals infinity if it is empty (note that the minimum is empty if and only if we cannot reach i from Zurich, so this is also correct).

Calculation order: We can compute the entries of DP from smallest to largest, since we have seen that for computing an entry $DP[i]$ we only need the values $DP[j]$ for $j < i$.

Running time: First, it takes time $\mathcal{O}(n \log n)$ to sort the cities by increasing minimal cost. Then, we need time $\mathcal{O}(1 + \deg_-(i))$ to compute $DP[i]$, so the running time of the DP itself is

$$\mathcal{O}\left(\sum_{i=1}^{n}(1 + \deg_-(i))\right) = \mathcal{O}(n + m),$$

where m denotes the number of edges in the graph. Thus, the total running time is $\mathcal{O}(n \log n + m)$.