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

14. December 2020

Algorithms & Data Structures — Homework 13 — HS 20

Exercise Class (Room & TA): _____

Submitted by: _____

Peer Feedback by: _____

Points: ____

Submission: This exercise sheet is not to be turned in. The solutions will be published at the end of the week, before Christmas.

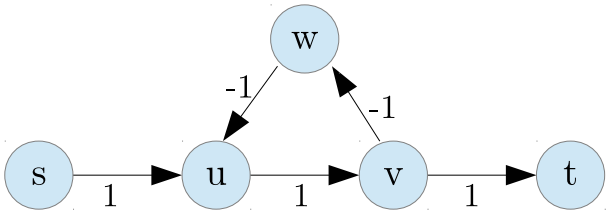
Exercise 13.1 *Shortest path with negative edge weights (part I).*

Let $G = (V, E, c)$ be a graph with edge weights $c : E \to \mathbb{Z} \setminus \{0\}$ and $c_{\min} = \min_{e \in E} c(e)$.

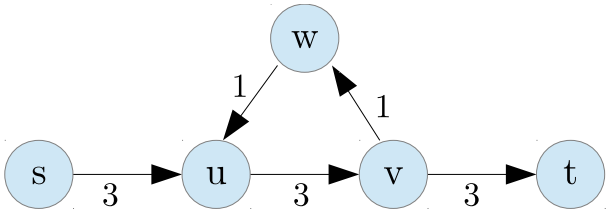
a) Since Dijkstra's algorithm must not be used whenever some edge weights are negative (i.e., $c_{\min} < 0$), one could come up with the idea of applying a transformation to the edge weight of every edge $e \in E$, namely $c'(e) = c(e) - c_{\min} + 1$, such that all weights become positive, and then find a shortest path P in G by running Dijkstra with these new edge weights c'.

Show that this is not a good idea by providing an example graph G with a weight function c, such that the above approach finds a path P that is not a shortest path in G. The example graph should have exactly 5 nodes and not all weights should be negative.

Solution.
Consider for example the following graph:



We have that $c_{\min} = \min_{e \in E} c(e) = -1$, thus we add the value $1 - (-1) = 2$ to every edge weight to obtain the following transformed graph:



A shortest s-t-path in the trasformed graph is $\langle s, u, v, t\rangle$. However, there is a shorter path in the original graph since the vertices $\langle u, v, w, u\rangle$ form a cycle with negative weight. Hence, for an *arbitrary* s-t-path in the original graph, we can always find a path with smaller weight by following this cycle once more.

b) Now consider the problem of finding a minimum spanning tree of a connected undirected graph $G = (V, E, c)$. Show that if we add any number b to all weights of the edges, then (the edge sets of) minimum spanning trees of G do not change.

So if we have an algorithm that finds a minimum spanning tree in any connected undirected graph with positive edge weights, we can also use it to find a minimum spanning tree in arbitrary connected undirected graph (by using new weights $c'(e) = c(e) - c_{\min} + 1$).

Solution.
Let T be any spanning tree of G. Before changing the edge weights its weight is

$$W(T) = \sum_{e \in T} c(e),$$

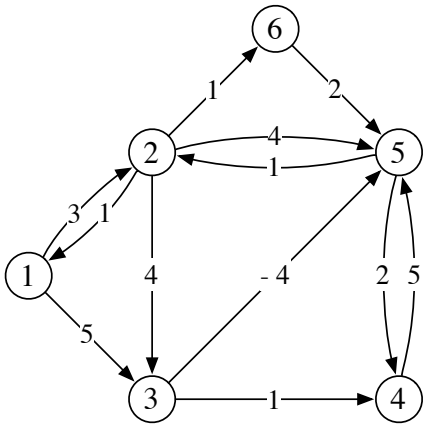
and after we add b to all edge weights, the weight of T becomes

$$W'(T) = \sum_{e \in T} (c(e) + b) = \sum_{e \in T} c(e) + \sum_{e \in T} b = W(T) + (|V| - 1)b.$$

Hence for any spanning tree its weight changes by the same number, so if a tree T had minimal weight before adding b, it will also have minimal weight after adding b.

Exercise 13.2 *Shortest path with negative edge weights (part II).*

We consider the following graph:



1. What is the length of the shortest path from vertex 1 to vertex 6?
2. Consider Dijkstra's algorithm (that fails here, because the graph has negative edge weights). Which path length from vertex 1 to vertex 6 is Dijkstra computing? State the sets $S, V \setminus S$ immediately before Dijkstra is making its first error and explain in words what goes wrong.
3. Which efficient algorithm can be used to compute a shortest path from vertex 1 to vertex 6 in the given graph? What is the running time of this algorithm in general, expressed in n, the number of vertices, and m, the number of edges?

4. On the given graph, execute the algorithm by Floyd and Warshall to find *all* shortest paths. Express all entries of the $(6 \times 6 \times 7)$-table as 7 tables of size 6×6. (It is enough to state the path length in the entry without the predecessor vertex.) Mark the entries in the table in which one can see that the graph does not contain a negative cycle.

Solution.

1. The shortest path from vertex 1 to vertex 6 is $(1, 3, 5, 2, 6)$ and has length $5 - 4 + 1 + 1 = 3$.
2. With Dijkstra's algorithm we find the path $(1, 2, 6)$ having length 4. The first mistake happens already after having processed vertex 1. The sets at that point in time are $S = \{1\}$ and $V \setminus S = \{2, 3, 4, 5, 6\}$. To vertex 2, we know a path of length 3, to vertex 3 a path of length 5. To the other vertices, we do not know a path so far. Hence, Dijkstra's algorithm choses vertex 2 to continue, i.e., includes 2 into S, which corresponds to the assumption, that we already know the shortest path to this vertex. This is clearly a mistake, since the path $(1, 3, 5, 2)$ has only length 2.
3. We can use the algorithm of Bellman and Ford which runs in $\mathcal{O}(nm)$ time.
4. Each of the following tables corresponds to a fixed value $k \in \{0, 1, 2, 3, 4, 5, 6\}$ and contains the lengths of all shortest paths that use only vertices in $\{0, \ldots, k\}$. Since all entries on the diagonal are non-negative, we can conclude that the graph does not contain any negative cycle.

von \ nach	1	2	3	4	5	6
1	0	3	5	∞	∞	∞
2	1	0	4	∞	4	1
3	∞	∞	0	1	-4	∞
4	∞	∞	∞	0	5	∞
5	∞	1	∞	2	0	∞
6	∞	∞	∞	∞	2	0

$k = 0$

von \ nach	1	2	3	4	5	6
1	0	3	5	∞	∞	∞
2	1	0	4	∞	4	1
3	∞	∞	0	1	-4	∞
4	∞	∞	∞	0	5	∞
5	∞	1	∞	2	0	∞
6	∞	∞	∞	∞	2	0

$k = 1$

von \ nach	1	2	3	4	5	6
1	0	3	5	∞	*7*	*4*
2	1	0	4	∞	4	1
3	∞	∞	0	1	-4	∞
4	∞	∞	∞	0	5	∞
5	*2*	1	*5*	2	0	*2*
6	∞	∞	∞	∞	2	0

$k = 2$

von \ nach	1	2	3	4	5	6
1	0	3	5	*6*	*1*	4
2	1	0	4	*5*	*0*	1
3	∞	∞	0	1	-4	∞
4	∞	∞	∞	0	5	∞
5	2	1	5	2	0	2
6	∞	∞	∞	∞	2	0

$k = 3$

von \ nach	1	2	3	4	5	6
1	0	3	5	6	1	4
2	1	0	4	5	0	1
3	∞	∞	0	1	-4	∞
4	∞	∞	∞	0	5	∞
5	2	1	5	2	0	2
6	∞	∞	∞	∞	2	0
$k = 4$						

von \ nach	1	2	3	4	5	6
1	0	*2*	5	*3*	1	*3*
2	1	0	4	*2*	0	1
3	*-2*	*-3*	0	*-2*	-4	*-2*
4	*7*	*6*	*10*	0	5	*7*
5	2	1	5	2	0	2
6	*4*	*3*	*7*	*4*	2	0
$k = 5$						

von \ nach	1	2	3	4	5	6
1	**0**	2	5	3	1	3
2	1	**0**	4	2	0	1
3	-2	-3	**0**	-2	-4	-2
4	7	6	10	**0**	5	7
5	2	1	5	2	**0**	2
6	4	3	7	4	2	**0**
$k = 6$						

Exercise 13.3 *Variants of shortest-path-problems.*

Let $G = (V, E)$ be a directed weighted graph with positive edge weights and let $s, t \in V$ be two vertices.

Design an efficient algorithm for each of the following variants of the shortest-path problem and state the running time of your algorithm.

1. We would like to find the shortest path from s to t that passes through two additionally given vertices u and v.
2. Let $k \in \mathbb{N}$. Among all paths from s to t with at most k vertices in between (i.e., with at most $k+1$ edges) we would like to find the shortest such path.

Solution.

1. There are two possible orders in which we can visit u and v: either we first go on a path from s to u, then from u to v, and finally from v to t, or we first go on a path from s to v, then from v to u, and finally from u to t. To make sure that it is a shortest path, each of these subpaths must also be a shortest path. Therefore, we can see the shortest s-t-path as the concatenation of three shortest paths as specified above.

   Hence, we compute 6 shortest paths: 1. from s to u, 2. from s to v, 3. from u to v, 4. from v to u, 5. from v to t, 6. from u to t. Then, we concatenate 1, 3, and 5 to on path and 2,4, and 6 to the other path and choose the shorter one.

   Cost: We compute 6 (a constant number of) shortest paths. The cost is thus asymptotically the same as the computation of one shortest path. Running Dijkstra's algorithm with Fibonacci Heaps, the running time is thus $\mathcal{O}(|E| + |V| \cdot \log(|V|))$, where $|E|$ is the cardinality of the edges and $|V|$ the cardinality of the vertices. If we use a binary heap, the running time is $\mathcal{O}((|E|+|V|)\cdot\log(|V|))$.

2. We modify the algorithm of Bellman-Ford. For each vertex $v \in V$ and each $i \in \mathbb{N}$ we remember the length $d_{i,v}$ of a shortest path from s to v that contains at most i edges. Furthermore, let $p_{i,v}$ be the predecessor of v in a shortest s-v-path with at most i edges. We get the following pseudecode:

   BELLMAN-FORD(V, E, w, s)

   Input: directed, weighted Graph (V, E, w), start vertex $s \in V$

   Output: Predecessor $\pi_{i,v}$ for all $v \in V$ and all $i \in \{0, \ldots, k\}$

   ```
   1 for each v ∈ V do d_{0,v} ← ∞; π_{0,v} ← null
   2 d_{0,s} ← 0
   3 for i ← 1, ..., k + 1 do
   4     for each v ∈ V do d_{i,v} ← d_{i−1,v}; π_{i,v} ← π_{i−1,v}
   5     for each (u, v) ∈ E do
   6         if d_{i−1,u} + w((u, v)) < d_{i,v} then
   7             d_{i,v} ← d_{i−1,u} + w((u, v))
   8             π_{i,v} ← u
   ```

   After i iterations of the loop in step 3, the algorithm considered all paths from start vertex that contain at most i edges. After $k + 1$ iterations, the algorithm has thus considered all paths that contain at most k intermediate vertices. A shortest s-t-path with at most $k + 1$ edges, can then be reconstructed by going backwards from t: we initially set $v \leftarrow t$ and follow the predecessor vertices until we reach s. See also the pseudocode below.

RECONSTRUCT-SHORTEST-PATH($\pi_{i,v}$, s, t)

Input: predecessor $\pi_{i,v}$ for all $v \in V$ ans all $i \in \{0, \dots, k\}$, vertices s and t

Output: A shortest (s,t)-path $P = \langle v_0, v_1, \dots, v_l \rangle$ with $v_0 = s$, $v_l = t$ and $l \leq k+1$

1 **if** $d_{k+1,t} = \infty$ **then** report that no (s,t)-path with at most $k+1$ edges exists.
2 $v \leftarrow t$; $P \leftarrow \langle\rangle$; $i \leftarrow k+1$
3 **while** $v \neq$ **null do** $P \leftarrow v \oplus P$; $v \leftarrow \pi_{i,v}$; $i \leftarrow i-1$
4 **return** P

Hereby let P be stored as sorted list. The operator $\oplus$ defines the concatenation of two lists, and $v \oplus P$ defines the list that is created when v is put to the beginning of P.

Every iteration in the first algorithm requires $\mathcal{O}(|V|+|E|)$ time. Additionally, we can reconstruct a shortest path with the second algorithm in $\mathcal{O}(|V|)$ time. The total time is thus $\mathcal{O}(k \cdot (|V|+|E|))$, i.e. $\mathcal{O}(k \cdot |E|)$ if the graph is connected.

Exercise 13.4 *Invariant and correctness of algortihm (**Exam exercise from January 2020**).*

Given is a weighted directed acyclic graph $G = (V, E, w)$, where $V = \{1, \dots, n\}$. The goal is to find the length of the longest path in G.

Let's fix some topological ordering of G and consider the array top$[1, \dots, n]$ such that top$[i]$ is a vertex that is on the i-th position in the topological ordering.

Consider the following pseudocode

Algorithm 1 Find-length-of-longest-path(G, top)

$L[1], \dots, L[n] \leftarrow 0, \dots, 0$
for $i = 1, \dots, n$ **do**
  $v \leftarrow$ top$[i]$
  $L[v] \leftarrow \max\limits_{(u,v) \in E} \big\{ L[u] + w\big((u,v)\big) \big\}$
return $\max\limits_{1 \leq i \leq n} L[i]$

Here we assume that maximum over the empty set is 0.

Show that the pseudocode above satisfies the following loop invariant INV(k) for $1 \leq k \leq n$: After k iterations of the for-loop, $L[\text{top}[j]]$ contains the length of the longest path that ends with top$[j]$ for all $1 \leq j \leq k$.

Specifically, prove the following 3 assertions:

i) INV(1) holds.

ii) If INV(k) holds, then INV($k+1$) holds (for all $1 \leq k < n$).

iii) INV(n) implies that the algorithm correctly computes the length of the longest path.

State the running time of the algorithm described above in Θ-notation in terms of $|V|$ and $|E|$. Justify your answer.

Proof of i).

In the first iteration we have $v = \text{top}[1]$. By the definition the first vertex in topological order has no incoming edges. Thus, $L[\text{top}[1]]$ gets assigned the maximum over the empty set, which we assume to be 0. As a consequence, INV(1) holds as there is no longest path that ends at top$[1]$ and $L[\text{top}[1]] = 0$.

Proof of ii).

In the $(k+1)$-th iteration we have $v = \text{top}[k+1]$. By the definition of topological ordering we have that all $u \in V$ with $(u, \text{top}[k+1]) \in E$ are in $\{\text{top}[1], \ldots, \text{top}[k]\}$. The length of the longest path via u ending at v can be decomposed into the length of the longest path ending at u plus the weight of the edge (u, v). Therefore, given INV(k), i.e., $L[\text{top}[j]]$ contains the length of the longest path for all $1 \leq j \leq k$, the maximum $\max\limits_{(u,v)\in E} \big\{L[u] + w\big((u,v)\big)\big\}$ computes the length of the longest path ending at v. Consequently, INV$(k+1)$ holds given INV(k) holds.

Proof of iii).

$INV(n)$ implies that each entry $L[v]$ contains the length of the longest path ending at v. Thus, computing the maximum $\max\limits_{1\leq i\leq n} L[i]$ corresponds to computing the length of the longest path in G.

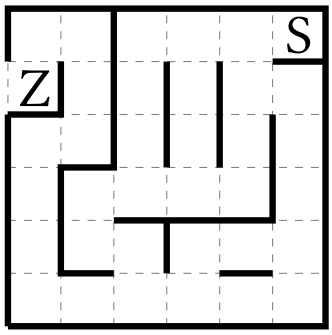
Running time:

The running time is in $\Theta(|E| + |V|)$. $|E|$ for the for loop, i.e., $\sum_{v\in V} \deg_-(v) = |E|$, and $|V|$ for $\max\limits_{1\leq i\leq n} L[i]$.

Exercise 13.5 *Minotaurus einsperren (**Klausuraufgabe vom Februar 2017**).*

King Minos instructs Daedalus to construct a maze to imprison the Minotaur. Daedalus presents his maze with n fields and a given starting field as a drawing on gridded paper. In the following figure you can see an example maze with $n = 36$ fields. The starting field is indicated by an S, and the target field at the exit with a Z. We want to determine how fast the Minotaur can escape from the given maze.

1. Model this problem as a shortest path problem:
   - Describe how the maze can be represented as a graph such that the following is true: The number of vertices on a shortest path between two vertices representing the starting and target field corresponds exactly to the smallest number of fields that have to be visited for reaching the target field Z.
   - Indicate how many vertices and edges your graph has in dependency of n.
   - Name an algorithm of the lecture that solves the shortest path problem for this graph as efficiently as possible. Also, provide the running time as concisely as possible in Θ notation.

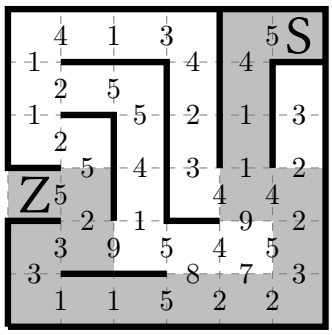
   

   Example: In the example on the right the Minotaur has to visit at least 21 fields (including the starting and the target field) to escape.

2. Various obstacles exist to complicate the escape. This means the time required to move from one field to another changes from obstacle to obstacle. For two adjacent fields that are not separated by a wall you are given the time that it takes to move from one field to another.

How can the modeling from task 1. be adapted to compute, under consideration of the given times, a fastest route to escape from the maze?

- Describe how the maze can be represented as a graph such that the following is true: The length of a shortest path between two vertices representing the starting and target field corresponds exactly to the minimum time necessary for reaching the target field Z.
- Indicate how many vertices and edges your graph has in dependency of n.
- Name an algorithm of the lecture that solves the shortest path problem for this graph as efficiently as possible. Also, provide the running time as concisely as possible in Θ notation.

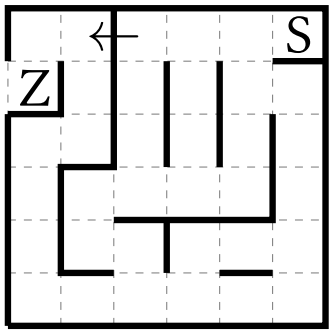


Example: In the example on the right one needs at least 44 time units to escape. The fastest route is indicated in gray.

3. As another variant of task 1., the Minotaur has the force to destroy exactly one inner wall of the maze (i.e., one wall between two fields for which both these fields are inside the maze).

   Compute how many fields the Minotaur has to visit on his escape at least, by modeling the problem as a shortest path problem. Name an algorithm that solves this problem as efficiently as possible. Also, provide the running time as concisely as possible in Θ notation.

   Example: In the example on the right the Minotaur can destroy the wall that is marked with an arrow, and has to visit only 7 fields to escape (and not 21 as in task a)).



Solution.

1. We create the undirected graph that contains a vertex for every field in the labyrinth. Two vertices are connected, if and only if the corresponding fields are neighbours and not separated by a wall.

   The labyrinth has n fields, thus the graph has $\Theta(n)$ vertices. Every vertex is incident to at most 4 edges, thus the graph has at most $\Theta(n)$ edges. To compute a shortest path from S to Z we can apply BFS that runs in time $\Theta(n)$.

2. We again create an undirected graph that contains a vertex for each field in the labyrinth. Two vertices are connected, if and only if the corresponding fields are neighbours and not separated by a wall. The cost of an edge corresponds to the cost to go from one field to the other.

   The labyrinth has n fields, thus the graph has $\Theta(n)$ vertices. Every vertex is incident to at most 4 edges, thus the graph has at most $\Theta(n)$ edges. Since now the edges are weighted, we look for the lightest path from S to Z. We use Dijkstra's algorithm that runs in $\mathcal{O}((|E| + |V|) \cdot \log(|V|)) = \mathcal{O}(n \log n)$ time.

3. We create to independent identical copies K_1 and K_2 of the graph created in the first subtask. In both graphs, we replace the undirected edge by two directed edges, one in each direction. We assume that K_1 corresponds to the situation before destroying a wall, and K_2 to the situation afterwards. Now, we go through all pairs of neighbouring fields that are separated by an inner

wall. Let u_1 and v_1 be the corresponding vertices of such a pair in K_1, and u_2 and v_2 in K_2. We add a directed edge from u_1 to v_2 and another one from v_1 to u_2.

Now, we look for the shortest path from S_1 to Z_1 or Z_2. We can do so again by BFS and stop as soon as we have found one of the two targets. The running time is again $\Theta(n)$.