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
Gleb Novikov, Tommaso d'Orsi, Ulysse Schaller, Rajai Nasser

20. December 2021

Algorithms & Data Structures Homework 13 HS 21

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.

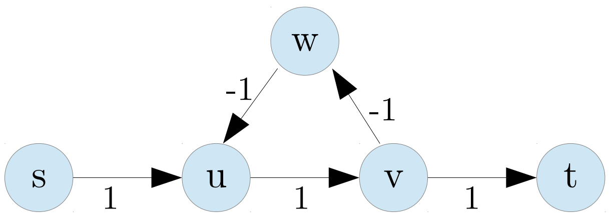
Exercise 13.1 *Shortest path with negative edge weights (part I).*

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

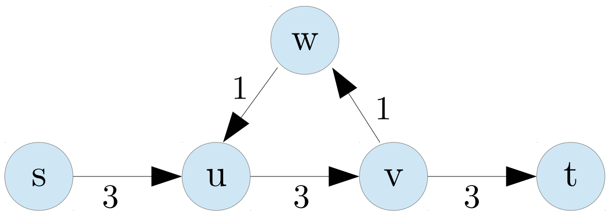
Since Dijkstra's algorithm must not be used whenever some edge weights are negative (i.e., $w_{\min} < 0$), one could come up with the idea of applying a transformation to the edge weight of every edge $e \in E$, namely $w'(e) = w(e) - w_{\min} + 1$, such that all weights become positive, and then find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

Show that this is not a good idea by providing an example graph G with a weight function w, such that the above approach finds a path P that is not a shortest path in G (this path P can start from the vertex of your choice). The example graph should have exactly 5 nodes and not all weights should be negative.

Solution: Consider for example the following graph:



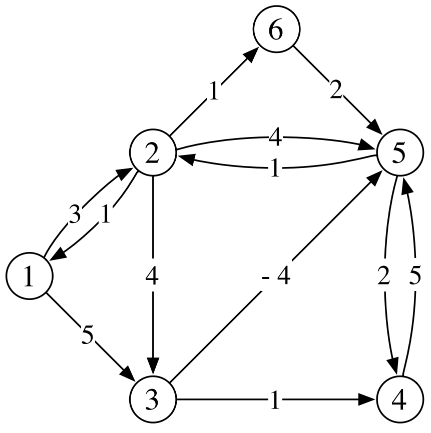
We have that $w_{\min} = \min_{e\in E} w(e) = -1$, thus we add the value $1 - (-1) = 2$ to every edge weight to obtain the following transformed graph:



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.

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 ?

   Solution: The shortest path from vertex 1 to vertex 6 is $(1, 3, 5, 2, 6)$ and has length $5-4+1+1 = 3$.

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.

   Solution: 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. 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 ?

   Solution: We can use the algorithm of Bellman and Ford which runs in $O(nm)$ time.

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: 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 $\{0, \dots, k\}$. Since all entries on the diagonal are non-negative, we can conclude that the graph does not contain any negative cycle.

from\to	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

$k = 0$

from\to	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

$k = 1$

from\to	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

$k = 2$

from\to	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

$k = 3$

from\to	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

$k = 4$

from\to	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

$k = 5$

from\to	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 *Invariant and correctness of algorithm (**This exercise is from the January 2020 exam**).*

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$[1,\ldots,n]$ 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)

```
$L[1],\ldots,L[n] \leftarrow 0,\ldots,0$
for $i = 1,\ldots,n$ do
    $v \leftarrow \text{top}[i]$
    $L[v] \leftarrow \max_{(u,v)\in E} \{L[u] + w((u,v))\}$
return $\max_{1\le i\le n} L[i]$
```

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

Show that the pseudocode above satisfies the following loop invariant INV(k) for $1 \le k \le n$: 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 $1 \le j \le k$.

Specifically, prove the following 3 assertions:

i) INV(1) holds.

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

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.

Solution:

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 $1 \le j \le k$, the maximum $\max_{(u,v)\in E} \{L[u] + w((u,v))\}$ 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 $\max_{1\le i\le n} L[i]$ corresponds to computing the length of the longest path in G.

Running time:

The running time is in $\Theta(|E| + |V|)$. The loop takes time $\Theta(|E| + |V|)$ since $\sum_{v \in V} \deg_-(v) = |E|$, and taking the maximum at the end takes time $\Theta(|V|)$.

Exercise 13.4 *Underground world (**This exercise is from the January 2021 exam**).*

a) Consider the following problem. The Swiss government is negotiating a deal with Elon Musk to build a tunnel system between all major Swiss cities. They put their faith into you and consult you. They present you with a map of Switzerland. For each pair of cities it depicts the cost of building a bidirectional tunnel between them. The Swiss government asks you to determine the cheapest possible tunnel system such that every city is reachable from every other city using the tunnel network (possibly by a tour that visits other cities on the way).

   i) Model the problem as a graph problem. Describe the set of vertices, the set of edges and the weights in words. What is the corresponding graph problem ?

      Solution: The map of Switzerland defines a undirected graph. There is a vertex $v \in V$ for each city. For each pair of cities $u, v \in V$ we create an edge $\{u, v\} \in E$. The edge $\{u, v\}$ gets a weight $w(\{u, v\})$ equal to the cost of building a tunnel between the cities u and v.

      The graph problem corresponding to the computation of the cheapest possible tunnel system is the computation of the minimum spanning tree in this graph.

   ii) Use an algorithm from the lecture to solve the graph problem. State the name of the algorithm and its running time in terms of $|V|$ and $|E|$ in Θ-notation.

      Solution: We can compute the minimum spanning tree using Kruskal's algorithm, which has a running time of $\Theta(|E| \log |V|)$.

b) Now, the Swiss tunneling society contacts the government and proposes to build the tunnel between Basel and Geneva for half of Musk's cost. Thus, the government contacts you again. They want you to solve the following problem: Given the solution of the old problem in a) and an edge for which the cost is divided by two, design an algorithm that updates the solution such that the new edge cost is taken into account. *In order to achieve full points, your algorithm must run in time $O(|V|)$.*

   Hint: You are only allowed to use the *solution* from a), i.e. the set of tunnels in the chosen tunnel system. You are not allowed to use any intermediate computation results from your algorithm in a).

   i) Describe your algorithm (for example, via pseudocode). A high-level description is enough.

      Solution: Let E_1 be the edge set of the MST T_1 from the previous part of the exercise.

      We have to distinguish the following two cases:

      A) *The updated edge is part of E_1:* If the updated edge is already part of E_1, E_1 is also a MST for the updated graph.

      B) *The updated edge is not part of E_1:* If the updated edge $e = \{u, v\} \in E$ is not part of E_1, we observe that adding e to E_1 creates a cycle C as E_1 is a tree. In order to fix that and to obtain the new MST E_2, we remove the edge e' with the largest cost from the cycle C in $(V, E_1 \cup \{e\})$ by performing the following steps:

         1) We compute the path P from u to v in T_1 by performing BFS.

         2) We extend the path P to a cycle C by adding the edge $\{u, v\}$ to it.

3) We search the edge $e' \in C$ with the largest cost in C by iterating over all the edges of C.

4) We return $T_2 = (V, E_2)$ with $E_2 = (E_1 \cup \{e\}) \setminus \{e'\}$.

ii) Prove the correctness of your algorithm and show that it runs in time $O(|V|)$.

Solution:

Running time: Computing a path between u and v using BFS in 1) and searching the edge e' also requires only $O(|V|)$ in 3) both only requires $O(|V|)$ because $|E_1| = |V| - 1$. The other steps are possible in constant time.

Correctness: In the case A) we just reduce the weight of an edge of the MST, thus, it trivially stays the MST and our algorithm is correct.

The case B) works as follows:

Case $e \notin$ MST of G': If any MST of G' (the new graph) does not contain e, then any MST of G' is an MST of G and vice versa, and the statement is true (in this case e is always the unique maximal edge in the cycle C).

Case $e \in$ MST of G': We compare which edge Kruskal's algorithm adds/rejects in G and G'. Let the resulting trees be $T = T_1$ and T' respectively. We prove (by induction over the steps of the algorithm) that the algorithms make the same decisions except that e is accepted and e' is rejected for G' (unless $e = e'$, then the decisions are completely identical). In other words, we prove that $T' = T_2 \cap \{\text{processed edges}\}$.

The only difference between the two algorithms is that e is processed earlier for G'. Before e is processed, it is clear that both algorithms make the same decisions. Similarly, after both algorithms have processed e', the accepted edge set has the same connected components (in either case all vertices on C are connected with each other), so the algorithms make the same decisions after e' is processed.

When e is processed for G', there are two cases. Either $e = e'$. In this case all other edges of C have already been accepted, so e is rejected. In this case it is clear that the algorithms continue to make the same decisions. Or $e \neq e'$. In this case, the accepted edge set is a subset of $T_2 \setminus \{e\}$, so the endpoints of e are in different connected components, and e is accepted.

In the remainder, we may assume $e \neq e'$. When e' is processed for G', all other edges of C are already added, so e' is rejected.

It remains to consider the steps of G' between processing e and e'. In this stage, the accepted edges for G form a subset of the accepted edges for G'. Namely, the latter set also contains e. Therefore, it is clear that any edge rejected for G is also rejected for G'. On the other hand, consider an edge $\tilde{e}$ that is accepted for G in this phase. When $\tilde{e}$ is processed for G', the set of accepted edges is a subset of $T_2 \setminus \{\tilde{e}\}$, so the endpoints of $\tilde{e}$ are in different connected components, and $\tilde{e}$ is accepted for G' as well.

Therefore, we have shown that the algorithms make the same decisions except that e is accepted and e' is rejected (or exactly the same decisions if $e = e'$). Hence, at termination the algorithm outputs $T' = T \cup \{e\} \setminus \{e'\} = T_2$, as desired.

Exercise 13.5 *Cheap flights (**This exercise is from the January 2020 exam**).*

Suppose that there are n airports in the country Examistan. Between some of them there are direct flights. For each airport there exists at least one direct flight from this airport to some other airport. Totally there are m different direct flights between the airports of Examistan.

For each direct flight you know its cost. The cost of each flight is a strictly positive integer.

You can assume that each airport is represented by its number, i.e. the set of airports is $\{1, \dots, n\}$.

a) Model these airports, direct flights 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 (if possible, in words and not formal).

   Solution: Each airport 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 flight starting from airport u to airport v. The weight $w(e)$ of the edge $e = (u, v)$, is the cost of the direct flight from u to v.

   Notice that the graph might not be connected, but $|E| \geq |V|$, since "For each airport there exists at least one direct flight from this airport to some other airport."

In points 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 $O(\deg_{-}(u))$ and $O(\deg_{+}(u))$ respectively, where $\deg_{-}(u)$ is the in-degree of vertex u and $\deg_{+}(u)$ is the out-degree of vertex u.

b) Suppose that you are at the airport S and you want to fill the array d of minimal traveling costs to each airport. That is, for each airport A, $d[A]$ is a minimal cost that you must pay to travel from S to A.

   Name the most efficient algorithm that was discussed in lectures which solves the corresponding graph problem. If several such algorithms were described in lectures (with the same running time), it is enough to name one of them. State the running time of this algorithm in Θ-notation in terms of n and m.

   Solution: Name of the algorithm used to solve this problem: Dijkstra's Algorithm

   Runtime: $O(m + n \log n)$ if implemented with Fibonnachy heap, $O\big((m + n) \cdot \log n\big)$ if implemented with binary heap.

c) Now you want to know *how many* optimal routes there are to airport T. In other words, if $c_{\min}$ is the minimal cost from S to T then you want to compute *the number of routes from S to T of cost $c_{\min}$*.

   Assume that the array d from b) is already filled. Provide an as efficient as possible *dynamic programming* algorithm that takes as input the graph G from task a), the array d from point b) and the airports S and T, and outputs the number of routes from S to T of minimal cost.

   Address the following aspects in your solution and state the running time of your algorithm:

   1) *Definition of the DP table*: What are the dimensions of the table $DP[\dots]$? What is the meaning of each entry ?

   2) *Computation of an entry*: How can an entry be computed from the values of other entries ? Specify the base cases, i.e., the entries that do not depend on others.

3) *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps ?
4) *Extracting the solution*: How can the final solution be extracted once the table has been filled ?
5) *Running time*: What is the running time of your algorithm ? Provide it in Θ-notation in terms of n and m, and justify your answer.

Solution:

Size of the DP table / Number of entries: We use a 1-dimensional DP table consisting of n entries.

Meaning of a table entry:

$DP[i]$ is the number of optimal routes from S to the airport i.

Computation of an entry (initialization and recursion):

$DP[S] = 1$. If $d[v] = \infty$, $DP[v] = 0$. If $v \neq S$ and $d[v] < \infty$, then

$$DP[v] = \sum_{\substack{u:(u,v)\in E \\ d[u]+w((u,v))=d[v]}} DP[u]\,.$$

Order of computation: The order of the array d. That is, if $d[i] < d[j]$, then i is before j in this order.

Computing the result: The result is contained in $DP[T]$.

Running time in concise Θ-notation in terms of n and m. Justify your answer.

Hint: *Note that the array d is a part of the input, so you don't need to include the time that is required to fill this array to the running time here.*

We need $\Theta(n \log n)$ time to sort the array d. To fill the DP table we need $\Theta(n+m)$, since the time required to compute $DP[v]$ is $\Theta(\deg_-(v)+1)$, and $\sum_{v\in V} \Theta(\deg_-(v)+1) = \Theta(n+m)$. Hence the running time of the algorithm described above is $\Theta(n \log n + m)$.