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

16 December 2024

Algorithms & Data Structures Exercise sheet 13 HS 24

The solutions for this sheet are not submitted.

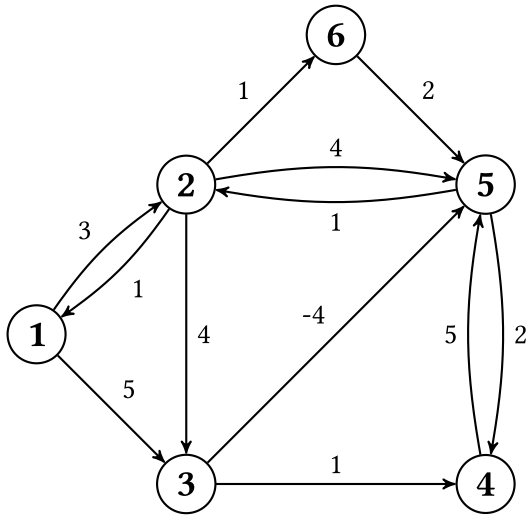
Exercises that are marked by * are challenge exercises.

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 13.1 *Shortest path with negative edge weights.*

We consider the following graph:



(a) 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 $5 - 4 + 1 + 1 = 3$.

(b) 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 $(1, 2, 6)$ that has length 4. The first mistake happens 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 chooses 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.

(c) 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.

(d) 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 interior vertices in $\{1, \ldots, k\}$. Since all entries on the diagonal are non-negative, we can conclude that the graph does not contain any negative cycle.

Changes are marked by italic font.

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 13.2 *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 $V = \{1, \ldots, n\}$. 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 $\text{top}[1, \ldots, n]$ such that $\text{top}[i]$ 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**
  $v \leftarrow \text{top}[i]$
  $L[v] \leftarrow \max_{(u,v) \in E} \left\{ L[u] + w\big((u, v)\big) \right\}$
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 $\text{top}[j]$ for all $1 \le j \le k$.

Specifically, prove the following 3 assertions:

i) $\text{INV}(1)$ holds.

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

iii) $\text{INV}(n)$ 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 assumed to be 0. As a consequence, $\text{INV}(1)$ holds as there is no longest path that ends at $\text{top}[1]$ and $L[\text{top}[1]] = 0$.

Proof of ii).

We assume $\text{INV}(k)$ holds. 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 $\text{INV}(k)$, 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, $\text{INV}(k+1)$ 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 $\Theta(|E| + |V|)$. 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 13.3 *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 $\{1, \ldots, n\}$.

(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.

This graphs fulfills the condition $|E| \geq |V|$ (since "For each airport there exists at least one direct flight from this airport to some other airport."). However, note that this does not imply that the graph is connected.

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

"$O(m + n \log n)$ if implemented with Fibonnachy heap" would also be correct.

(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.

In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps? Describe the calculation order in pseudocode.
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

***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.*

Solution:

1. *Dimensions of the DP table*: DP[1 ...n]
2. *Subproblems*: $DP[i]$ is the number of optimal routes from S to the airport i.

3. *Recursion*: If $d[v] = \infty$, $DP[v] = 0$. If $v \neq S$ and $d[v] < \infty$, then

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

Base case: $DP[S] = 1$.

First, notice that if $d[v] = \infty$, then there is no way to reach v from S. Hence, there are no routes of minimal cost between them.

For the recursive case, we want to calculate the entry $DP[v]$ for some vertex v. Consider an optimal paths between S and v, $(S, \ldots, w, v)$. As all weights are positive, w's distance has to be smaller than that of v. By our calculation order, all vertices with a smaller distance have already been calculated and, by our assumption, for every vertex w, $DP[w]$ contains the number of optimal paths between S and w. Now we extend the intermediate paths by appending v. Then, the number of possible pathways is defined by $\sum_{\substack{w:(w,v)\in E \\ d[w]+w((w,v))=d[v]}} DP[w]$. Therefore, our recursion is correct.

4. *Calculation order:* Let $i_1, \ldots, i_n$ be a order of vertices, such that for all $i_j \neq i_k$, if $d[i_j] < d[i_k]$, then i_j is before j_k. This order corresponds to order defined by the array d. It can be determined by sorting d and remembering the original indexes.

   for $i = i_1 \ldots i_n$ **do**
   Compute $DP[i]$

5. *Extracting the solution:* The result is contained in $DP[T]$.

6. *Running time:* 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 $\Theta(n \log n + m)$.

Exercise 13.4 *Elevator.*

Consider the following definitions for a **directed** graph $G = (V, E)$:

1. The *out-degree* of a vertex $v \in V$, denoted by $\deg_{\text{out}}(v)$, is the number of edges of E that start at v, i.e., $\deg_{\text{out}}(v) = |\{(v, w) \in E \mid w \in V\}|$.
2. The *in-degree* of a vertex $v \in V$, denoted by $\deg_{\text{in}}(v)$, is the number of edges that end at v, i.e., $\deg_{\text{in}}(v) = |\{(u, v) \in E \mid u \in V\}|$.
3. A *Eulerian walk* is a sequence $v_1, \ldots, v_k \in V$ such that $k = |E| + 1$ and $\{(v_i, v_{i+1}) \mid 1 \leq i < k\} = E$. Note that this definition implies (v_i, v_{i+1}) being different edges for $1 \leq i < k$.

In this exercise, you can use without proof the following result:

Lemma 1. A **directed** graph $G = (V, E)$ admits a Eulerian walk if, and only if, all of the following conditions holds:

1. At most one vertex $v \in V$ is such that $\deg_{\text{out}}(v) = \deg_{\text{in}}(v) + 1$;
2. At most one vertex $v \in V$ is such that $\deg_{\text{in}}(v) = \deg_{\text{out}}(v) + 1$;
3. Every vertex that satisfies neither (i) nor (ii) is such that $\deg_{\text{out}}(v) = \deg_{\text{in}}(v)$;

4. The undirected graph G' obtained by ignoring the direction of edges in G is connected.

(a) Write down the pseudocode of an $O(|V| + |E|)$ time algorithm that takes as input a **directed** graph G, and returns `true` if G has a Eulerian walk, and `false` otherwise. Justify its correctness and complexity.

Solution:

See Algorithm 2.

The algorithm works by checking if all conditions from the previous lemma are fulfilled.

For conditions 1-3, it is sufficient to compute the set of in- and out-neighbors (or simply the degrees) of all nodes and check the equations. This can be done straightforwardly in time $O(|V| + |E|)$.

For conditions 4, we perform a DFS from any vertex (here, vertex 0) on the *undirected* graph and check whether all vertices are marked (i.e., reached) by the algorithm. This can be done in $O(|V| + |E|)$ (DFS) and $O(|V|)$ (reachability check) respectively.

In total, the complexity of our algorithm in $O(|V| + |E|)$.

(b) Alice is launching iFahrstuhl™, a start-up developing the next generation of elevators. Assume a building with n floors indexed from 1 to n and an elevator which has room for a single person. The elevator receives requests in the form of pairs $(i, j) \in \{1, \ldots, n\}^2$ of distinct floors between which a single person is willing to travel.

Consider the scenario where m people want to use the elevator. For $1 \leq t \leq m$, the t-th people want to go from floor i_t to floor j_t. These requests are given as a finite set $S = \{(i_1, j_1), \ldots, (i_m, j_m)\}$.

A finite set $S = \{(i_1, j_1), \ldots, (i_m, j_m)\}$ of requests is called *optimal* if the pairs can be ordered such that all requests can be processed and the elevator is never empty when moving between two floors (except maybe on its way to fetching the first person).

For example, for $n = 5$, the set $S_1 = \{(2, 3), (4, 1), (3, 4)\}$ is optimal, since it can ordered as $\{(2, 3), (3, 4), (4, 1)\}$, which means that the elevator can start on floor 2 to fetch person 1, go to floor 3, drop person 1 and fetch person 3, go to floor 4, drop person 3 and fetch person 2, go to floor 1, drop person 2, and terminate there. However, the set $S_2 = \{(2, 3), (4, 1)\}$ is not optimal, since there is no way a single elevator can satisfy both requests without moving empty from floor 3 to floor 4 or floor 1 to floor 2.

Given a set of requests S, Alice's elevators should be able to decide whether it's optimal. Model the problem of detecting optimal sets of requests as a graph problem and provide an algorithm to solve it. Describe the vertex and edge set, edge weights (if needed), the graph problem you solve, the algorithm you use, and its complexity. To obtain full points, your algorithm should run in time $O(n + |S|)$.

Solution:

The problem is equivalent to the existence of an Euler path in the unweighted directed graph $G_1 = (V_1, E_1)$ defined by

$$\begin{aligned} V_1 &= \{1, \ldots, n\} \\ E_1 &= S. \end{aligned}$$

We can use the algorithm from question (a) to find this Euler path. Its complexity is $O(|V_1| + |E_1|) = O(n + |S|)$.

Algorithm 2 Check if a directed graph has a Eulerian path

```
function DFS(in_neighbors, out_neighbors, v, marked)
    if marked[v] then
        continue
    else
        marked[v] ← True
    for w ∈ in_neighbors[v] ∪ out_neighbors[v] do
        DFS(in_neighbors, out_neighbors, w, marked)
function CheckEulerian(V,E)
    if |V| = 0 then
        return True
    out_neighbors ← array[|V|]                          ▷ Initialized to ∅
    in_neighbors ← array[|V|]                           ▷ Initialized to ∅
    for (v, w) ∈ E do                                   ▷ Compute neighbors
        out_neighbors[v] ← out_neighbors[v] ∪ {w}
        in_neighbors[w] ← in_neighbors[w] ∪ {v}
    has_plusone, has_minusone = False, False            ▷ Check conditions 1-3
    for v ∈ V do
        if |out_neighbors[v]| = |in_neighbors[v]| then
            continue
        else if |out_neighbors[v]| = |in_neighbors[v]| + 1 then
            if has_plusone then
                return False
            else
                has_plusone = True
        else if |out_neighbors[v]| = |in_neighbors[v]| − 1 then
            if has_minusone then
                return False
            else
                has_minusone = True
        else
            return False
    marked ← bool[|V|]                                  ▷ Initialized to False
    DFS(in_neighbors, out_neighbors, 0, marked)
    for v ∈ V do                                        ▷ Check condition 4
        if ¬marked[v] then
            return False
    return True
```

(c) Alice's startup has installed k single-person elevators in your n-floor building. Unfortunately, not all elevators can reach all floors. Hence, for each elevator $j \in \{1, \ldots, k\}$, you are given a set $F_j \subseteq \{1, \ldots, n\}$ of floors it can reach. When you arrive in front of an elevator j, say on floor $f \in F_j$, you can immediately call it, after which you have to wait until it reaches your floor from its current position, moving at the constant speed of 1 time unit per floor. When the elevator arrives, you choose the destination floor $f' \in F_j$, and the elevator brings you to this floor at the constant speed of 0.5 time units per floor (for security reasons, the elevator is slower when it is not empty). The time spent moving between elevators on the same floor, calling the elevator or choosing the destination floor is negligible, since you are very fast at interacting with elevators.

You are alone in the building at floor 1, with each elevator j being initally located on floor f_j. You would like to go to floor n. What is the minimal amount of time that you have to travel using Alice's elevators? If you cannot reach floor n, then output ∞.

Model the problem as a graph problem and provide an algorithm to solve it. Describe the vertex and edge set, edge weights (if needed), the graph problem you solve, the algorithm you use, and its complexity. To obtain full points, your algorithm should run in time $O((n + K) \log n)$, where $K = \sum_{j=1}^{k} |F_j|^2$.

Solution:

The cost of your journey between s and d using a sequence of elevators $j_1, \ldots, j_p$ and the sequence of floors $k'_0 = s, \ldots, k'_{p-1}, k'_p = d$ will be the sum of the time spent in the various elevators, i.e. $\sum_{\ell=1}^{p}(2 \cdot |k'_\ell - k'_{\ell-1}|)$, and the time spent waiting for each elevator when calling them from your starting point, i.e. $\sum_{\ell=1}^{p} |f_{j_\ell} - k'_{\ell-1}|$. The total waiting time is $\sum_{\ell=1}^{p}(|f_{j_\ell} - k'_{\ell-1}| + 2 \cdot |k'_\ell - k'_{\ell'-1}|)$. We note (i) that as all speeds are positive, you will never need to go twice through the same floor, (ii) that using the same elevator twice is useless (better than using it twice $a \to b$ and $a' \to b'$, you could have used it $a \to b'$), and (iii) that to move from a to b, you will always pick the nearest available elevator, which will be at its initial position (since by (ii) you did not yet use it).

In the end, our problem is equivalent to finding the shortest-path between vertices s and d in the following weighted graph $G_2 = (V_2, E_2, w_2)$:

$$V_2 = \{1, \ldots, n\}$$
$$E_2 = \{(a, b) \mid j \in \{1, \ldots, k\}, a \in F_j, b \in F_j, a \neq b\}$$

$$w_2((a, b)) = \min\{|f_j - a| + 2 \cdot |b - a| \mid j \in \{1, \ldots, k\} \land a, b \in F_j\}$$

As all weights are positive, we can use Dijkstra's algorithm to find the shortest path. Its runtime is $O((|V_2| + |E_2|) \log |V_2|)$. The number of vertices is $|V_2| = n$ and the number of edges is $|E_2| \leq \sum_{j=1}^{k}(|F_j|(|F_j| - 1)) = O(K)$. Hence, the overall complexity is $O((n + K) \log n)$.

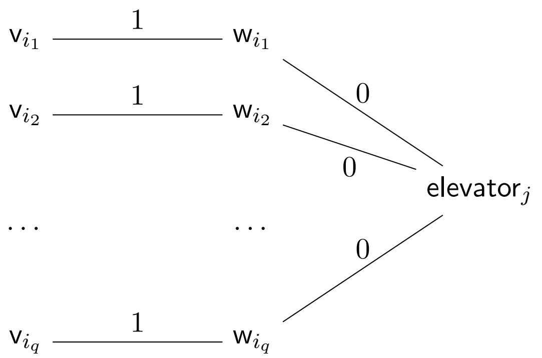
(d) Continue the setting of (c). Elevator doors in your building need maintenance, but the people in your building also need elevators. In your building, there is exactly one elevator door per elevator and floor, which needs to be functional in order for the elevator to be used from or to this floor. Even if a door is not functional, the elevator can still be used between all other floors where a functional door is present. Alice wants to select as many elevator doors as possible to be maintained during the next working day such that all floors can be reached from each other using the elevators and the remaining functional doors (those not in maintenance).

Model the problem as a graph problem and provide an algorithm to solve it. Describe the vertex and edge set, edge weights (if needed), the graph problem you solve, the algorithm you use, and its complexity. To obtain full points, your algorithm should run in time $O((n + K') \log(n + K'))$, where $K' = \sum_{j=1}^{p} |F_j|$.

Hint: *Consider the set of vertices*

$$V = \{\mathsf{v}_1, \ldots, \mathsf{v}_n\} \cup \{\mathsf{w}_1, \ldots, \mathsf{w}_n\} \cup \{\mathsf{elevator}_1, \ldots, \mathsf{elevator}_j\}$$

and use subgraphs (“gadgets”) of the form



where $F_j = \{i_1, \ldots, i_q\}$.

Solution:

Call the gadget above G_j. This gadget represents the inside of elevator j ($\mathsf{elevator}_j$), its doors (w_{i_ℓ}), and the floors it serves (v_{i_ℓ}). To move from, say, floor s to floor d, you start at v_s, use the door w_s, enter $\mathsf{elevator}_j$ that brings you to floor d, then use the door w_d to leave the elevator, and end at v_d. Putting door w_{i_ℓ} in maintenance removes the edge $(\mathsf{v}_{i_\ell}, \mathsf{w}_{i_\ell})$ from the graph, as this door is no longer usable.

Consider $G_3 = \bigcup_{j=1}^{k} G_j$. Finding the maximal number M of doors that you can that you can maintain to ensure that all floors remain reachable from each other is equivalent to finding the minimal number of doors $m = K' - M$ that you *must keep in function* to ensure the same property. Now, the value m is exactly the value of the *minimum spanning tree* of G_3. We can use, e.g., Kruskal’s algorithm to compute it, resulting in a $O(|E_3| \log |E_3|)$ runtime, where E_3 denotes the edge set of G_3. Since $|E_3| = \sum_{j=1}^{k}(2|F_j|) = 2K' = O(K')$, we get an overall complexity of $O(K' \log K')$.