Eidgenössische
Technische Hochschule
Zürich

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

Department Informatik
Markus Püschel
Johannes Lengler
Chris Wendler

David Steurer
Gleb Novikov
Ulysse Schaller

Exam

Algorithmen und Datenstrukturen

January 26, 2021

DO NOT OPEN!

Last name, first name: _____

Student number: _____

With my signature I confirm that I can participate in the exam under regular conditions. I will act honestly during the exam, and I will not use any forbidden means.

Signature: _____

Good luck!

	T1 (20P)	T2 (19P)	T3 (9P)	T4 (12P)	P1 (??P)	P2 (??P)	**Σ** (100P)
Score							
Corrected by							

Theory Task T1. **/ 20 P**

In this problem, you have to provide **solutions only**. You do not need to justify your answer.

/ 5 P a) *Asymptotic notation quiz:* For each of the following claims, state whether it is true or false. You get 1P for a correct answer, -1P for a wrong answer, 0P for a missing answer. You get at least 0 points in total.

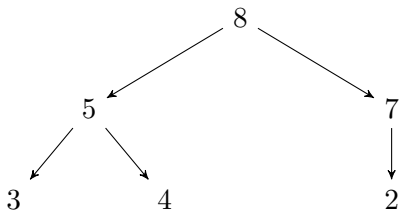
Assume $n \geq 2$.

Claim	true	false
$n^2 + 10n - 23 \geq \Omega(n^{2.5})$	☐	☒
$\sqrt[3]{n} \leq \mathcal{O}(\frac{\sqrt{n}}{\log n})$	☒	☐
$\log_3(n^4) = \Theta(\log_6(n^2))$	☒	☐
$\sum_{i=1}^{n} \sqrt{i} = \Theta(n^{1.5})$	☒	☐
$\sum_{i=1}^{n} i! \geq \Omega(n \cdot n!)$	☐	☒

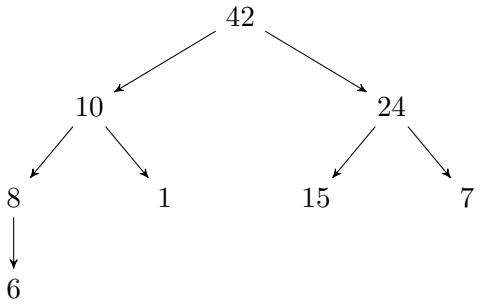
/ 2 P b) *Max-Heaps:*

i) Draw a Max-Heap that contains the keys $8, 4, 2, 3, 5, 7$ (note that several solutions are possible here).

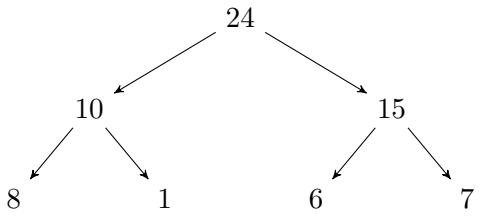
Solution:



ii) Draw the Max-Heap obtained from the following Max-Heap by performing the operation `DELETE-MAX` once.



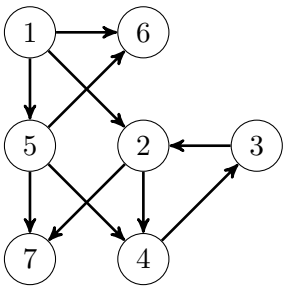
Solution:



/ 5 P c) *Graph quiz:* For each of the following claims, state whether it is true or false. You get 1P for a correct answer, -1P for a wrong answer, 0P for a missing answer. You get at least 0 points in total.

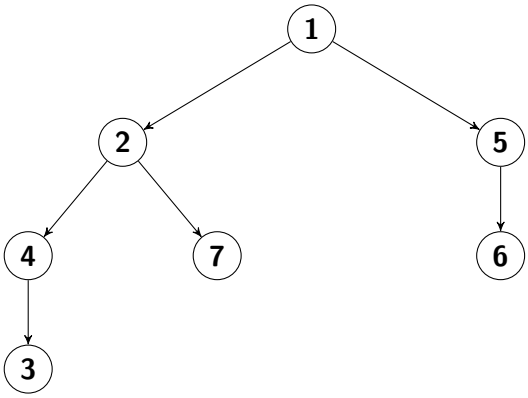
Claim	**true**	**false**
The topological ordering of a directed acyclic graph is unique.	☐	☒
For all $n \in \mathbb{N}$, there exists a directed acyclic graph on n vertices with $\binom{n}{2}$ edges.	☒	☐
Let $v \in V$ be a vertex of an undirected graph $G = (V, E)$ with adjacency matrix A. It takes time $\Theta(1 + \deg(v))$ to compute $\deg(v)$ from A.	☐	☒
If every vertex of an undirected graph G has even degree, then G has an Eulerian walk.	☐	☒
In order to run Dijkstra's algorithm on a directed graph G, you first need to have a topological ordering of G.	☐	☒

/ 2 P d) *Depth-first search / Breadth-first search:* Consider the following directed graph:



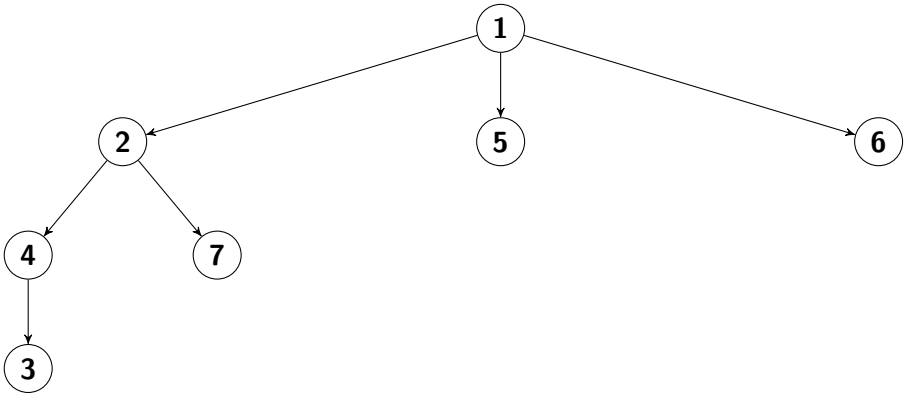
i) Draw the depth-first tree resulting from a depth-first search starting from vertex 1. Process the neighbors of a vertex in increasing order.

Solution:

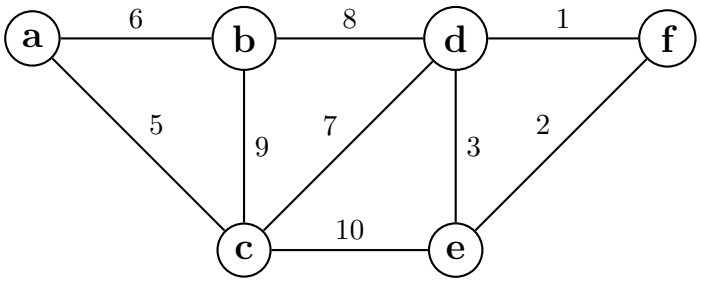


ii) Draw the breadth-first tree resulting from a breadth-first search starting from vertex 1. Process the neighbors of a vertex in increasing order.

Solution:

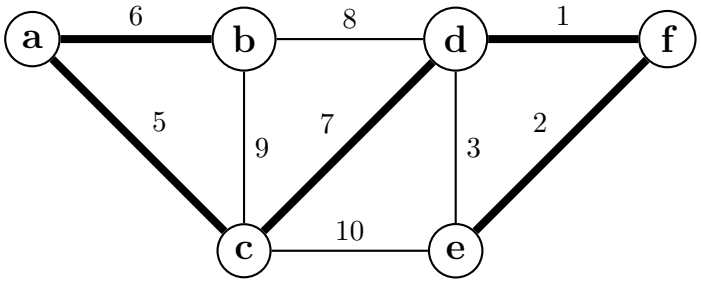


/ 1 P e) *Minimum Spanning Tree:* Consider the following graph:



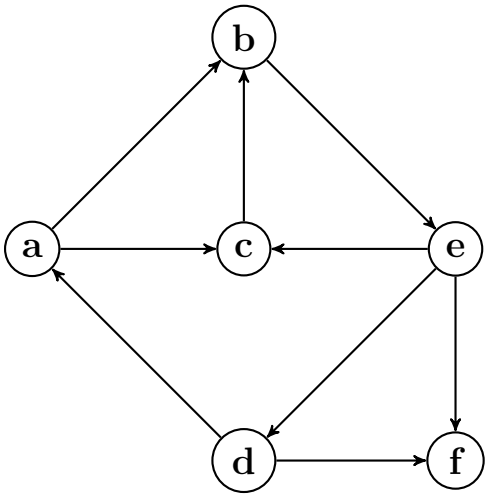
Highlight the edges that are part of the minimum spanning tree.

Solution:



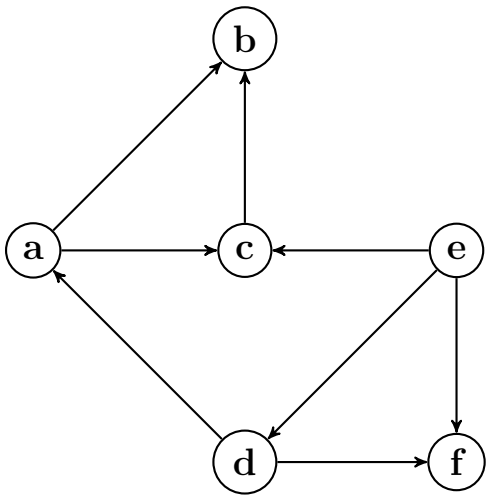
/ 2 P

f) *Topological sorting:* Consider the following directed graph G:



i) Remove the smallest possible number of edges from G such that a topological ordering of its vertices exists.

Solution: We remove the edge (b, e):



ii) Compute a topological ordering of the vertices of your modified graph.

Solution: e, d, a, c, b, f

/ 3 P g) *Sorting algorithms:*

i) Consider the sequence $6, 5, 4, 1, 2, 3$. How many swaps does `Bubble Sort` perform to sort this sequence? *Give the exact number of swaps required.*

Solution: 12

ii) Consider the sequence $6, 5, 4, 1, 2, 3$. How many swaps does `Selection Sort` perform to sort this sequence? *Give the exact number of swaps required.*

Solution: 4

iii) Let $n \in \mathbb{N}$ be an even number and consider the sequence with the following structure:

$$2, 1, 4, 3, 6, 5, \ldots, n, n-1.$$

How many swaps does `Insertion Sort` perform to sort this sequence? *Give the exact number, not just the asymptotics.*

Solution: $n/2$

Theory Task T2. **/ 19 P**

In this part, you should **justify your answers briefly**, e.g. by sketching the derivation.

/ 3 P a) *Induction:* For the following task you may use the identity

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1} \quad \text{for all } 1 \leq k \leq n$$

without proof. Show by mathematical induction that for any integer $n \geq 1$,

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n.$$

Solution:

Base case $n = 1$:

$$\sum_{k=0}^{1} \binom{1}{k} = \binom{1}{0} + \binom{1}{1} = 1 + 1 = 2^1.$$

Induction hypothesis: For some $n \geq 1$,

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n.$$

Induction step $n \to n+1$:

$$\begin{aligned}
\sum_{k=0}^{n+1} \binom{n+1}{k} &= \binom{n+1}{0} + \binom{n+1}{n+1} + \sum_{k=1}^{n} \binom{n+1}{k} \\
&= 2 + \sum_{k=1}^{n} \left[\binom{n}{k} + \binom{n}{k-1} \right] \\
&= \left(1 + \sum_{k=1}^{n} \binom{n}{k} \right) + \left(1 + \sum_{k=0}^{n-1} \binom{n}{k} \right) \\
&= \sum_{k=0}^{n} \binom{n}{k} + \sum_{k=0}^{n} \binom{n}{k} \\
&= 2^n + 2^n = 2^{n+1}.
\end{aligned}$$

/ 2 P b) *Recurrence relations:*

For this exercise, you may use the following master theorem from exercise sheet 4:

Theorem 1 (Master theorem) *Let $a, C > 0$ and $b \geq 0$ be constants and $T : \mathbb{N} \to \mathbb{R}^+$ a non-decreasing function such that for all $k \in \mathbb{N}$ and $n = 2^k$,*

$$T(n) \leq aT(n/2) + Cn^b.$$

Then

- *If $b > \log_2 a$, $T(n) \leq \mathcal{O}(n^b)$.*
- *If $b = \log_2 a$, $T(n) \leq \mathcal{O}(n^{\log_2 a} \cdot \log n)$.*
- *If $b < \log_2 a$, $T(n) \leq \mathcal{O}(n^{\log_2 a})$.*

Consider the following recursive function that takes as an input a positive integer m that is a power of two (that is, $m = 2^k$ for some integer $k \geq 0$).

Algorithm 1 $g(m)$

```
if m > 1 then
    g(m/2)
    g(m/2)
    for i = 1, ..., 6⌊√m⌋ do
        f()
else
    f()
```

Let $T(m)$ be the number of calls of the function f in $g(m)$.

i) Give a recursive formula for $T(m)$. Don't forget to provide the base case as well.

Solution:

If $m > 1$, we have $T(m) = 2T(m/2) + 6\lfloor\sqrt{m}\rfloor$. Moreover, the base case is $T(1) = 1$.

ii) Determine $T(m)$ in $\mathcal{O}$-notation. Your answer should be as tight as possible.

Solution:

By part i), we have $T(m) \leq 2T(m/2) + 6m^{1/2}$. So we can use the Master theorem with $a = 2$, $b = 1/2$ and $C = 6$ (which corresponds to case 3) to deduce that $T(m) \leq \mathcal{O}(m)$.

/ 4 P c) *Graph connectivity*

Recall the following two definitions from the exercises.

Definition 1 *A vertex v in a connected graph is called a* cut vertex *if the subgraph obtained by removing v (and all its incident edges) is disconnected.*

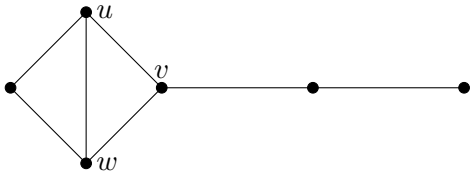
Definition 2 *An edge e in a connected graph is called a* cut edge *if the subgraph obtained by removing e (but keeping all the vertices) is disconnected.*

In the following, we always assume that the original graph is connected. Prove or find a counterexample to the following statements:

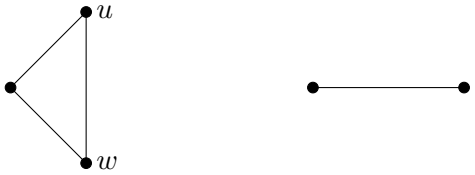
i) If a vertex v is part of a cycle, then it is not a cut vertex.

Solution:

The following graph is a counterexample:



Indeed, v is clearly part of a cycle (the triangle uvw for example), but removing v yields the following graph:

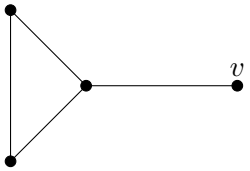


The above graph is disconnected. Hence, v is also a cut vertex.

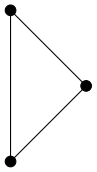
ii) If a vertex v is not a cut vertex, then v must be part of a cycle.

Solution:

The following graph is a counterexample:



Indeed, v is not part of a cycle (remember that the vertices forming a cycle must be disjoint). However, removing v yields the following connected graph:



Hence, v is also not a cut vertex.

iii) If an edge e is part of a cycle (that is, e connects two consecutive vertices in a cycle), then it is not a cut edge.

Solution:

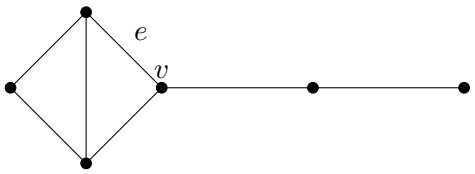
This statement is correct, and we can prove it as follows. Let G be a conncted graph and let $e = \{v_1, v_2\}$ be an edge of G that is part of a cycle $v_1 \dots v_k$ for some $k \geq 3$. To

show that e is not a cut edge, we will show that any two vertices u and w of G can be joined by a walk that does not use the edge e. So consider any two vertices u and w. Since G is connected, there is a walk $uu_1 \dots u_n w$ from u to w in G. If the walk doesn't use the edge e, then we are already done. If the walk does use edge e, this means that the vertices v_1 and v_2 must appear consecutively (at least once) in $uu_1 \dots u_n w$. We replace every appearance of $v_1 v_2$ in the walk by the path $v_1 v_k v_{k-1} \dots v_2$, and every apperance of $v_2 v_1$ by the same path in the other direction $v_2 v_3 \dots v_k v_1$. This yields a walk from u to w that does not use the edge e and concludes the proof.

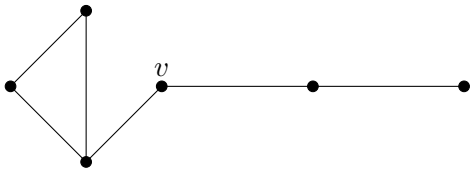
iv) If v is a cut vertex and e is an edge incident to v, then e is a cut edge.

Solution:

The following graph is a counterexample:



Indeed, we have seen in part a) that v is a cut vertex of this graph. However, removing e yields the following connected graph:



Hence e is an edge incident to v that is not a cut edge.

/ 5 P d) *Insertion sort invariant*

Let $A[0, \dots, n-1]$ be an integer array of size n. Consider the following implementation of insertion sort:

Algorithm 2 `InsertionSort`(A)

for $i = 1 \dots n-1$ **do**
Find the smallest index $j \in \{0, \dots, i\}$ such that $A[i] \leq A[j]$.
Shift the subarray $A[j, \dots, i-1]$ by one to the right, and move the element $A[i]$ to position j.

i) Formulate an invariant $INV(i)$ that holds after the ith iteration of the `for`-loop (the iteration with $i = 1$ is the first iteration).

Solution: $INV(i)$: After the ith iteration, the $A[0, \dots, i]$ is sorted.

ii) Use this invariant to prove correctness of the algorithm `InsertionSort`.

1. Show that the invariant holds at the beginning (base case).

Solution: $INV(0)$ trivially holds as $A[0, \dots, 0]$ contains only one element.

2. Let $1 \leq i \leq n-2$. Show that if $INV(i)$ holds after the ith iteration of the `for`-loop, then $INV(i+1)$ holds after the $(i+1)$st iteration (induction step).

   Solution: At the start of the $(i+1)$th loop iteration $A[0,\ldots,i]$ is sorted because $INV(i)$ holds according to IH. Thus, shifting $A[j,\ldots,i-1]$ one to the right corresponds to moving all elements that are larger than $A[i]$ one to the right. Now, moving $A[i]$ puts it into the correct position, i.e., $A[0,\ldots,i-1]$ contains only elements that are smaller than $A[i]$ and $A[i+1,\ldots,n-1]$ only elements that are larger or equal to $A[i]$.

3. Show that if $INV(n-1)$ holds at the end of the algorithm, then the array A is sorted.

   Solution: $INV(n-1)$ per definition implies that $A[0,\ldots,n-1] = A$ is sorted.

/ 5 P e) *Finding a cheap cycle*

Let $G = (V, E)$ be a weighted undirected graph, where all edge weights are positive. Provide an efficient algorithm that, given an edge $e \in E$, outputs the weight of the cheapest cycle (that is, the cycle of smallest total weight) that contains e, and outputs ∞ if e is not contained in any cycle. Give the running time of your algorithm in terms of $|V|$ and $|E|$. In order to get full points, your algorithm should run in time $\mathcal{O}((|V| + |E|) \log |V|)$

You do not need to write a proof of correctness or a runtime analysis. If you use algorithms known from the lecture as sub-routines, you do not need to re-discuss how they work.

Solution:

The following algorithm takes as input a graph $G = (V, E)$ and an edge $uv \in E$, and returns the desired quantity. Here, $w(uv)$ denotes the weight of the edge uv.

Algorithm 3

Let $E' := E \setminus \{uv\}$ and consider the graph $G' := (V, E')$.
Run Dijkstra's algorithm on the graph G' with starting vertex u to get the distance d between u and v in G'.
return $d + w(uv)$

Indeed, in order to find the cycle containing uv of smallest weight, we need to find the path between u and v of smallest weight among all such paths that do not use the edge uv. The weight of the cycle will then be the sum of the weight of this path and the weight of uv. This is exactly what the above algorithm does. Note that if uv is not contained in any cycle in G, then u and v will not be in the same connected component of G'. Thus, Dijkstra's algorithm will return $d = \infty$, and hence our algorithm will also return ∞, as desired.

The running time of the algorithm is simply the time that Dijkstra's algorithm need to run on G', which is

$$\mathcal{O}((|V| + |E'|) \log |V|) = \mathcal{O}((|V| + (|E| - 1)) \log |V|) = \mathcal{O}((|V| + |E|) \log |V|).$$

Theory Task T3. **/ 9 P**

Consider the following problem. You are given an array of n integers $a_1, \ldots, a_n \in \mathbb{N}$ summing to $A := \sum_{i=1}^{n} a_i$, which is a multiple of 3. You want to determine whether it is possible to partition $\{1, \ldots, n\}$ into three disjoint subsets I, J, K such that their corresponding elements yield the same sum, i.e.

$$\sum_{i \in I} a_i = \sum_{j \in J} a_j = \sum_{k \in K} a_k = \frac{A}{3}.$$

Note that I, J, K form a partition of $\{1, \ldots, n\}$ if and only if $I \cap J = I \cap K = J \cap K = \emptyset$ and $I \cup J \cup K = \{1, \ldots, n\}$.

For example, the answer for the input $[2, 4, 8, 1, 4, 5, 3]$ is *yes*, because there is the partition $\{3, 4\}$, $\{2, 6\}$, $\{1, 5, 7\}$ (corresponding to the subarrays $[8, 1]$, $[4, 5]$, $[2, 4, 3]$, which are all summing to 9). On the other hand, the answer for the input $[3, 2, 5, 2]$ is *no*.

Provide a *dynamic programming* algorithm that determines whether such a partition exists. Your algorithm should have an $\mathcal{O}(nA^2)$ runtime to get full points. Address the following aspects in your solution:

1) *Definition of the DP table*: What are the dimensions of the table $DP[\ldots]$? 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. Justify your answer.
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 A, and justify your answer.

Size of the DP table / Number of entries: $(n+1) \times (A+1) \times (A+1)$.

Meaning of a table entry: For $0 \leq m \leq n$ and $0 \leq B, C \leq A$, the corresponding entry in the DP table is defined as

$$DP[m, B, C] = \begin{cases} 1 & \text{if there are two disjoint sets } I, J \subseteq \{1, \ldots, m\} \\ & \text{such that } \sum_{i \in I} a_i = B \text{ and } \sum_{j \in J} a_j = C, \\ 0 & \text{otherwise.} \end{cases}$$

Computation of an entry (initialization and recursion):

We initialize the values for $m = 0$ as

$$DP[0, B, C] = \begin{cases} 1 & \text{if } B = C = 0, \\ 0 & \text{otherwise.} \end{cases}$$

The other entries are then computed as

$$DP[m+1, B, C] = \max\{DP[m, B, C], DP[m, B - a_{m+1}, C], DP[m, B, C - a_{m+1}]\}.$$

Indeed, it is possible to get two disjoint subsets of $\{a_1, \ldots, a_{m+1}\}$ summing to B and C if and only if there are two disjoint subsets of $\{a_1, \ldots, a_m\}$ that are summing to either B and C (so we don't need to use a_{m+1}), $B - a_{m+1}$ and C (so we add a_{m+1} to the first subset), or B and $C - a_{m+1}$ (so we add a_{m+1} to the second subset).

Order of computation: We can compute the values $DP[m, B, C]$ by increasing order in m. The order for B and C doesn't matter.

Extracting the result: The answer to the problem is *yes* if $DP[n, A/3, A/3] = 1$ and *no* if $DP[n, A/3, A/3] = 0$.

Running time: We need to fill $(n+1)(A+1)^2$ entries, and each of them can be computed in constant time $\Theta(1)$. Therefore, the running time is $\Theta(nA^2)$.

Theory Task T4. **/ 12 P**

1. 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 $\Theta(|E| \log |V|)$.

2. 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 $\mathcal{O}(|V|)$.*

   Hint: You are only allowed to use the *solution* from 1., 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 1.

   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 $\mathcal{O}(|V|)$.

Solution:

Running time: Computing a path between u and v using BFS in 1) and searching the edge e' also requires only $\mathcal{O}(|V|)$ in 3) both only requires $\mathcal{O}(|V|)$ 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 been accepted before, 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.

Alternative proof for the Case $e \in$ MST of G':

Some MST of G' contains e. Let T' be such an MST in G'. Consider the trees T_u and T_v that remain after removing e from T'. If we add e^* — the smallest edge in G that connects T_u and T_v — we get a spanning tree T^* of G. Note that $w(T^*) = w(T'') + w(e^*) - w(e)$.

Now consider the MST T_1 in G from part a) and replace by e the edge $e_1 = \{u_1, v_1\}$ from the path (in T_1) from u to v such that u_1 is a vertex in T_u and v_1 is a vertex in T_v. Denote the resulting spanning tree in G' by T''. Note that by definition of e^*, $w(e_1) \geq w(e^*)$. Let's show that T'' is an MST in G'. Indeed, since T_1 is an MST in G,

$$w(T'') = w(T_1) + w(e) - w(e_1) \leq w(T^*) + w(e) - w(e_1) \leq w(T^*) + w(e) - w(e^*) = w(T')\,.$$

Hence for any MST T_1 in G there exists an edge e_1 in the path from u to v such that replacing this edge by e gives MST T_2 in G'. It is easy to see that such e_1 has maximal weight in the cycle C (otherwise we could get a smaller tree than T_2 by removing maximal edge instead of e_1).