Eidgenössische
Technische Hochschule
Zürich

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

Department Informatik
Markus Püschel
Gleb Novikov
Ulysse Schaller

David Steurer
Tommaso d'Orsi
Rajai Nasser

Exam

Algorithmen und Datenstrukturen

January 31, 2022

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 (23P)	T2 (15P)	T3 (9P)	T4 (13P)	Prog. (40P)	**Σ** (100P)
Score						
Corrected by						

Theory Task T1. / 23 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 $n \geq 4$.

Solution:

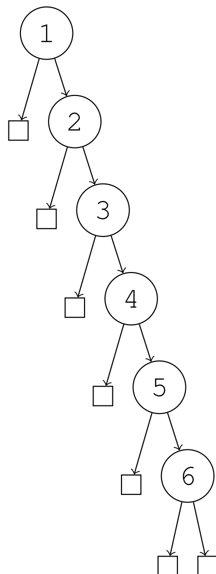
Claim	true	false
$n + n^2 = \Theta(n^2)$	☒	☐
$n^{100} \leq O(2^n)$	☒	☐
$\sum_{i=1}^{\lceil\sqrt{n}\rceil} i^3 \geq \Omega(n^2)$	☒	☐
$\frac{1}{2}n^2 - \sum_{i=1}^{n-1} i \leq O(n)$	☒	☐
Suppose $a_1 = 1$ and $a_{i+1} \leq O(a_i)$ for all i. It follows that $a_n \leq O(n)$.	☐	☒

/ 3 P

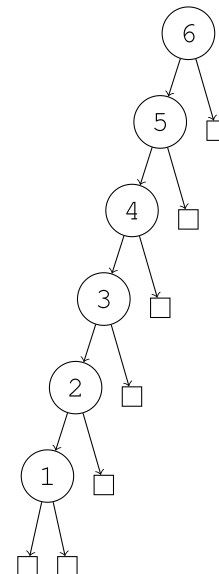
b) *Search trees:*

i) Draw a **binary search tree** of maximum possible **depth** that contains exactly the numbers $\{1, 2, 3, 4, 5, 6\}$. The depth of the tree is the longest path from the root to any of its leaves. (If there are multiple possible trees, draw any valid solution.)

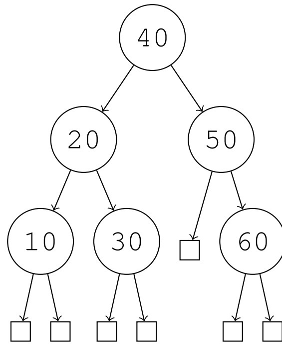
Solution:



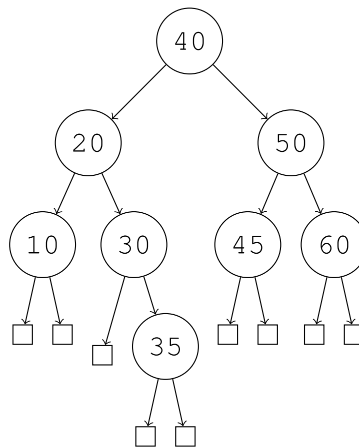
OR:



ii) Draw the **binary search tree** obtained from the following tree by performing the two operations `INSERT`(45) and `INSERT`(35), in that order.

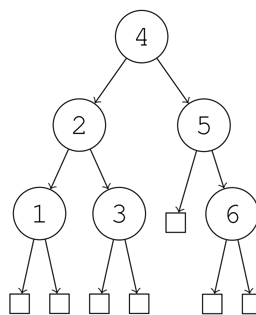


Solution:



iii) Draw the **AVL tree** that is obtained when inserting the keys $4, 2, 1, 3, 5, 6$ in this order into an empty tree (it suffices to draw only the final tree).

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.

As a reminder, here are a few definitions:

A *walk* is a sequence of vertices $v_1, \ldots, v_k$ such that for every two consecutive vertices v_i, v_{i+1} there exists an edge from v_i to v_{i+1}.

A *simple walk* $v_1, \ldots, v_k$ is a walk with the additional property that $v_i \neq v_j$ whenever $i \neq j$ (i.e., all vertices are distinct).

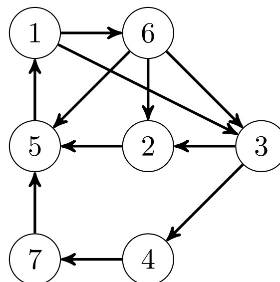
A *simple cycle* $v_1, \ldots, v_k$ is a walk where additionally $v_1 = v_k$, $v_i \neq v_j$ whenever $i < j < k$ (i.e., all vertices except the endpoints are distinct), and $k \geq 4$ (so $v_1 \to v_2 \to v_1$ is not allowed).

Solution:

Claim	**true**	**false**
Every graph that is connected and Eulerian is bipartite.	☐	☒
In a directed graph suppose there exists a walk with vertices s and t as endpoints. Then there exists a simple walk with vertices s and t as endpoints.	☒	☐
In any tree $T = (V, E)$ with $\|V\| \geq 10$, we can always add at least one additional edge $e \notin E$ to T such that the resulting graph is bipartite (the set of vertices must remain the same).	☐	☒
In every undirected graph $G = (V, E)$ with $\|V\| = \|E\|$ there exists a simple cycle as a subgraph.	☒	☐
Given an undirected graph G with all degrees even, there always exists a way to direct the edges of G (i.e., convert each edge $\{a, b\}$ into either $a \to b$ or $b \to a$) such that in the resulting graph it holds that at every vertex v, the in-degree and out-degree are equal (but this number can differ between different vertices).	☒	☐

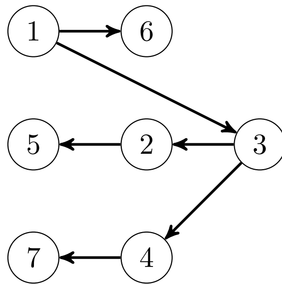
/ 2 P

d) *Depth-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:

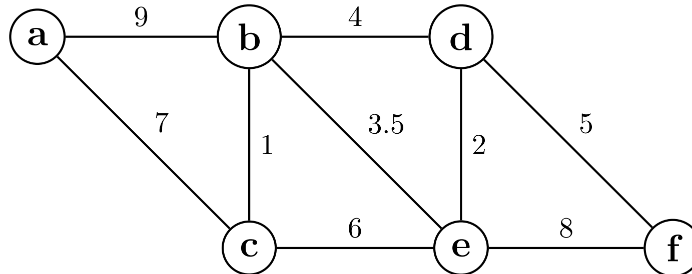


ii) Write out all the cross edges and all the back edges (specify which ones are which).

Solution:
Back edges: $5 \to 1$.
Cross edges: $6 \to 5$, $6 \to 2$, $6 \to 3$, $7 \to 5$.

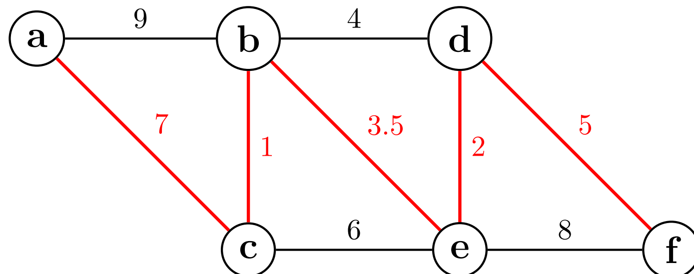
/ 2 P

e) *Minimum Spanning Tree:* Consider the following graph:



i) Highlight the edges that are part of the minimum spanning tree. (Either in the picture above, or you can recreate the graph below).

Solution:

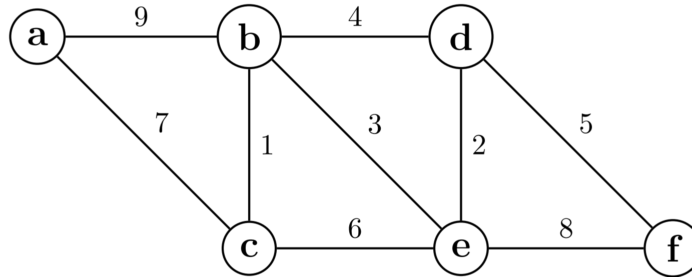


ii) Write out all positive integers x such that we could replace the weight 3.5 of the edge $\{b, e\}$ in the above graph with x, such that the edge would be in at least one minimum spanning tree of the resulting graph.

Solution: 1, 2, 3, 4.

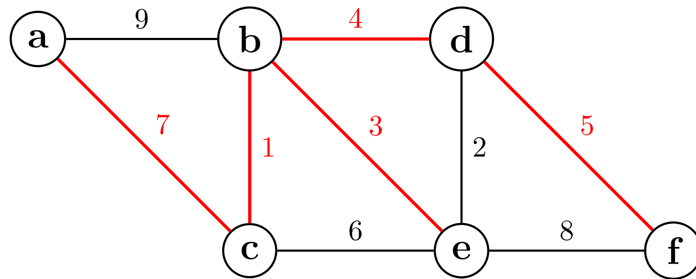
/ 2 P

f) *Shortest Path Tree:* Consider the following graph:



i) Highlight the edges that are part of the shortest-path tree rooted at vertex a (i.e., the output of Dijkstra's algorithm if we were to start from vertex a).

Solution:



ii) Write out all positive integers x such that we could replace the weight 8 of the edge $\{e, f\}$ in the above graph by x, such that the edge would be in at least one shortest-path tree rooted at a of the resulting graph.

Solution: 1, 2, 3, 4, 5, 6.

/ 4 P

g) *Sorting algorithms 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.

Solution:

Claim	**true**	**false**
There exists an array of length n for which the runtime of `InsertionSort` is $\Theta(n^{1.5})$.	☒	☐
There exists an array of length n for which the runtime of `MergeSort` is $\Theta(n)$.	☐	☒
There exists an array of length n for which the runtime of `HeapSort` is $\Theta(n^2)$.	☐	☒
Suppose a sequence of n bits (every element is either a zero or one) is given as input. There exists an algorithm with runtime $O(n)$ which sorts any such sequence.	☒	☐

/ 15 P

Theory Task T2.

In this part, you should **justify your answers briefly**.

/ 4 P

a) *Counting iterations:* For the following code snippets, derive an expression for the number of times f is called. Simplify the expression as much as possible and state it in Θ-notation as concisely as possible.

i) Snippet 1:

Algorithm 1

```
for j = 1, ..., n^2 do
    for k = 1, ..., j do
        f()
    f()
```

Solution: The number of calls to f is

$$\sum_{j=1}^{n^2}\left(\left(\sum_{k=1}^{j}1\right)+1\right)=\sum_{j=1}^{n^2}(j+1)=\left(\sum_{j=1}^{n^2}j\right)+n^2=\frac{n^2\left(n^2+1\right)}{2}+n^2=\frac{n^4}{2}+\frac{3}{2}n^2$$

which is $\Theta(n^4)$.

ii) Snippet 2:

Algorithm 2

```
for j = 1, ..., n do
    k ← 1
    while k ≤ j^2 − 1 do
        ℓ ← 1
        while ℓ ≤ n do
            f()
            ℓ ← 2ℓ
        k ← k + 1
```

Solution: The number of calls to f is

$$\sum_{j=1}^{n}\sum_{k=1}^{j^2-1}\lfloor\log_2 n\rfloor=\lfloor\log_2 n\rfloor\left(\sum_{j=1}^{n}\sum_{k=1}^{j^2-1}1\right)=\lfloor\log_2 n\rfloor\left(\sum_{j=1}^{n}(j^2-1)\right)$$

$$=\lfloor\log_2 n\rfloor\left(\left(\sum_{j=1}^{n}j^2\right)-n\right)=\lfloor\log_2 n\rfloor\left(\frac{n(n+1)(2n+1)}{6}-n\right)$$

which is $\Theta(n^3 \log n)$.

/ 3 P b) *Induction:* In this question, the *depth* of a binary tree is the length of the longest path between its root and its leaves. (In particular, a tree with a single node has depth 0.) Note that

A binary tree T is called *leafy* if for every node v of T that is not a leaf, the left subtree or the right subtree of v is a leaf.

Show that for all $i \in \mathbb{N} \setminus \{0\}$, there are exactly $3 \cdot 2^{i-1}$ leafy binary trees of depth i.

Note that there are exactly three binary trees of depth 1, namely , and .

Solution:

Base case: For $i = 1$, there are three leafy binary trees, namely , and .

Induction hypothesis: Let $k \in \mathbb{N} \setminus \{0\}$. Assume that there are exactly $3 \cdot 2^{k-1}$ leafy binary trees of depth k.

Induction step: Consider a leafy binary tree T of depth $k+1$. Since $k+1 > 0$, T is not a leaf. Let T_l and T_r be its left and right subtrees. Since T is leafy, either T_l or T_r is a leaf. First, we observe that both T_l and T_r must be leafy: otherwise, there would exist a non-leaf node in T_l or T_r which would have no leaf subtree, and T, which contains both T_l and T_r, would not be leafy. Assume that T_l is a leaf. Then the depth of T_r is exactly the depth of T minus one, i.e., k. Hence, as T_r is leafy, there are exactly 2^k choices for T_r by our induction hypothesis. Similarly, assuming that T_r is a leaf, there are exactly 2^k choices for T_l. Since the trees formed in the two cases are all pairwise distinct, we conclude that there are exactly $3 \cdot 2^{k-1} + 3 \cdot 2^{k-1} = 3 \cdot 2^k$ leafy trees of depth $k+1$.

/ 4 P c) *Cross edges*

Let $G = (V, E)$ be a directed graph and $v \in V$. Perform a DFS on G from v. Show that the subgraph consisting of all *cross edges* of G resulting from this DFS does not contain a cycle.

Hint: *First show that for any cross edge (v_1, v_2), v_2 must have been reached by the DFS before v_1.*

Solution:

<u>Lemma:</u> For any cross edge $(v_1, v_2) \in E$, v_2 must have been reached by the DFS before v_1.

Proof of the lemma: Assume that v_1 is reached before v_2. Let $W = [w_1, \ldots, w_k]$ be the list of out-neighbors of v_1, sorted in the ordered in which they are checked by the DFS. Since vertex v_2 must be in this list, let j such that $v_2 = w_j$. If w_j has not been explored by the DFS while processing $w_1, \ldots, w_{j-1}$, then w_j is entered after processing w_{j-1} using the edge $(v_1, w_j) = (v_1, v_2)$, and (v_1, v_2) is a tree edge. If w_j has been explored earlier, say during the processing of some w_k with $k < j$, then there is a path from w_k to w_j in the DFS tree. Hence, there is a path from v_1 to w_j through $w_k = v_2$ in the tree, and (v_1, v_2) is a forward edges. In all cases, if v_1 is reached before v_2, (v_1, v_2) cannot be a back edge. Hence, if $(v_1, v_2) \in E$ is a cross edge, v_2 must have been reached by the DFS before v_1.

Now, assume there is a cycle $v_1, v_2, \ldots, v_k, v_1$ consisting of cross edges $(v_1, v_2), \ldots, (v_k, v_1)$ of G. For all $v \in V$, denote by $\mathsf{pre}(v)$ the pre-number of vertex v. By our lemma, we have $\mathsf{pre}(v_1) > \mathsf{pre}(v_2)$, $\mathsf{pre}(v_2) > \mathsf{pre}(v_3)$, …, $\mathsf{pre}(v_k) > \mathsf{pre}(v_1)$. This implies $\mathsf{pre}(v_1) > \mathsf{pre}(v_1)$, a contradiction. Hence, such a cycle cannot exist.

/ 4 P

d) *Well-colored graphs*

Let C be a finite set of *colors* that contains the color `Blue` (i.e., $\texttt{Blue} \in C$) and $G = (V, E)$ a directed graph that has a topological ordering. Since G has a topological ordering, we can assume without loss of generality that $V = \{1, \ldots, k\}$ and $1, \ldots, k$ is a topological ordering of V. (The vertices can always be relabeled so that this holds.)

Further, consider a coloring $\mu : V \to C$ of G, i.e., a mapping from vertices of G to colors in C. For each vertex $v \in V$, the color of v is given by $\mu(v)$.

The graph G is called *well-colored* by μ if and only if the following property holds:

For any path $v_1, \ldots, v_k$, such that $\mu(v_1) = \texttt{Blue}$, we have $\mu(v_k) \neq \texttt{Blue}$.

Describe an algorithm that returns `True` if G is well-colored, and `False` otherwise. You can assume that G is provided as adjacency lists.

Give the runtime complexity of your algorithm in tight big-O notation.

Solution:

Algorithm 3 Checking well-colored graphs

```
Input: vertices V = {1, ..., k}, adjacency lists L = [ℓ_1, ..., ℓ_k], coloring μ
reachable_blue ← bool[k]                          ▷ Initialized with False
for i = 1 to k do
    [w_1, ..., w_p] ← ℓ_i                          ▷ Get all successors of vertex i
    for j = 1 to p do
        if reachable_blue[w_j] then
            reachable_blue[i] = True
            if μ(i) == Blue then
                return False
    if μ(i) == Blue then
        reachable_blue[i] = True
return True
```

This algorithm has complexity $O(|V| + |E|)$: each vertex and each edge is checked exactly once.

/ 9 P

Theory Task T3.

An array of non-negative integers $A = [a_1, \dots, a_n]$ is called *summy* if and only if, for all $i \in \{2, \dots, n\}$, there exists a (possibly empty) set $I \subseteq \{1, \dots, i-1\}$ such that $a_i = \sum_{j \in I} a_j$. In other terms, every integer in the array except the first one must be the sum of (distinct) integers that precede it in the array.

For example,

- The array $[2, 2, 4, 6, 0, 12]$ is summy, because $2 = 2$, $4 = 2 + 2$, $6 = 2 + 4$, $12 = 2 + 4 + 6$.
- The array $[2, 2, 4, 6, 0, 13]$ is not summy, since 13 can not be written as a sum of integers from $\{2, 2, 4, 6, 0\}$.

Provide a *dynamic programming* algorithm that, given an array A of length n, returns `True` if the array is summy, and `False` otherwise. In order to obtain full points, your algorithm should have an $O(n \cdot \max A)$ runtime. Address the following aspects in your solution:

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 $\max A$, and justify your answer.

Size of the DP table / Number of entries: DP is two-dimensional with indices $\{0, \dots, n\} \times \{0, \dots, \max A\}$. The total number of entries is $(n + 1)(\max A + 1)$.

Meaning of a table entry: $DP[i][k]$ is `True` iff there exists $I \subseteq \{1, \dots, i\}$ such that $k = \sum_{j \in I} a_j$.

Computation of an entry (initialization and recursion):

$$
\begin{aligned}
DP[0][k] &= (k == 0) \\
DP[i][0] &= \texttt{True} \\
DP[i][k] &= DP[i-1][k] \wedge DP[i-1][k - a_i] \qquad\qquad i > 0, k > a_i
\end{aligned}
$$

Order of computation: In increasing order of i and k, e.g.

for $i = 0$ to n **do**
  for $k = 0$ to $\max A$ **do**
    $DP[i][k] = \dots$

Extracting the result: The result is $\bigwedge_{i=2}^{n} DP[i-1][a_i]$.

Running time: Each table update happens in constant time, and there are $O(n \max A)$ elements in the table. Extracting the result takes $O(n)$ time. The overall runtime is therefore $O(n \max A + n) = O(n \max A)$.

Theory Task T4. / 16 P

Tina wants to travel around Iceland by car in December. There are n towns in Iceland labeled by $1, 2, \ldots, n$. For some pairs $i, j \in \{1, 2, \ldots, n\}$, towns i and j are connected by a two-way road, represented by $\{i, j\}$. The set of all roads is denoted with $R = \{\{i_1, j_1\}, \{i_2, j_2\}, \ldots, \{i_m, j_m\}\}$, with road $\{i_k, j_k\}$ having length L_k. If no road is blocked, the road network is such that Tina can go from town 1 to every other town in Iceland.

However, in winter, the weather conditions in Iceland can get very bad. Unfortunately, a snowstorm takes place before her journey. As a result, some roads are blocked at the beginning of her journey. Let $S \subseteq R$ be the set of roads being blocked.

As format of input, you are given the set $E = \{(i_1, j_1, b_1, L_1), (i_2, j_2, b_2, L_2), \ldots, (i_m, j_m, b_m, L_m)\}$, where, for any $k \in \{1, 2, \ldots, m\}$, $b_k = 1$ if road $\{i_k, j_k\}$ is in S, and $b_k = 0$ otherwise.

/ 2 P

a) Due to the snowstorm, all roads in S are blocked. Write down the pseudocode of an algorithm, which checks whether Tina can reach every town in Iceland starting from town 1. The algorithm should run in time $O(m)$. You don't need to describe in detail how to convert a set of edges to an adjacency list.

Solution:

We convert the list of edges to an adjacency list, and then run a DFS from node 1 that marks all reachable vertices. We output whether all of them can be reached.

Algorithm 4

```
edgeList ← list of edges, as given in input
visited[1...n] ← [false, false, ..., false]
adj[1...n] ← empty adjacency list of tuples

function DFS(u)
    visited[u] ← true
    for each (u, v, b, L) ∈ adj[u] do
        if b = 0 and not visited[v] then
            DFS(v)

convert edgeList to adj such that adj[u] has all tuples adjacent to u
DFS(1)
all ← true if visited = [true, ..., true]; false otherwise
Print("Can Tina reach every town? Answer: ", all)
```

Grading:

- 1 P for using the DFS algorithm.

- 1 P for writting pseudocode in correct form.
- We deduct 1 P for not dealing with blocked roads.
- For (a),(b),(c),(d), we deduct 0.5 if the solution contains minor mistake. We don't deduct points for typos.

/ 4 P

b) The local government wants to clear some of the blocked roads (i.e., make them passable) such that Tina can reach (via passable roads only) every other town starting from town 1. Describe an algorithm which finds the minimum total length of roads in S to be cleared, such that Tina can arrive at every other town starting from town 1. The algorithm should run in time $O(m \cdot \log(n))$.

You need to address the following aspects in your algorithm description:

- the graph algorithm used to solve this problem;
- the construction of the graph that you run this algorithm on;
- the correctness of the algorithm, i.e why any solution to the original problem gives you a solution to the graph problem and why any solution to the graph problem gives you a solution to the original problem
- the total running time of your algorithm.

You can directly use the algorithms covered in lecture material, and you can directly use their running time bounds without proof.

Solution:

- We use the MST algorithm to solve this problem.
- We use the same vertices and edges as given in the original problem. Furthermore, the weight of an edge i that is in S is its lenght L_i, while the weight of edges not in S is set to 0. We run the MST algorithm on this weighted graph.
- Let ALG be the value returned by our algorithm, and let OPT be optimal (correct) value.

  For one direction, let T be the MST. By definition, the total length of all its blocked edges is ALG. Since the graph is connected, T spans all vertices. Hence, by clearing up all blocked edges in T, Tina can reach all vertices (starting anywhere, even from vertex 1). Hence, $OPT \leq ALG$.

  For the other direction, let T^* be the set of edges that are cleared by the optimal solution. By definition, their total length is OPT. Let $E_{passable}$ be the set of non-blocked edges in the original graph. The problem guarantees Tina can reach all vertices via $T^* \cup E_{passable}$. Hence, there exists a spanning tree T' that is a subgraph of $T^* \cup E_{passable}$, hence the length of all blocked edges in T' is at most OPT. The MST will, by its minimality, have a smaller cost. Hence, $ALG \leq OPT$.
- Our algorithm (1) transforms the graph, and then (2) runs the MST algorithm. Step (1) is easily implemented in $O(m)$ time, while step (2) takes $O(m \log n)$ time. The total runtime is $O(m \log n)$.

Grading:

- 1 pt for using MST algorithm;
- 1 pt for running algorithm on the correct graph; We don't give points if the edge weights are set to be 0 or 1.
- 1 pt for arguing the correctness of algorithm;
- 1 pt for arguing why the algorithm provided runs in time $O(m \log n)$.
- For (b),(c),(d), if your algorithm is wrong, then you don't get the point for running time. Also if you just state the graph algorithm running time, then 0.5 points will be deducted.
- For (b),(c) proof of correctness, you get 0.5 point if the solution contains correct idea and 1 point if it's convincing.

/ 4 P

c) Due to the effort of the road authority, the roads in S are eventually not blocked anymore, but they are still in unpleasant condition. Now Tina wants to begin her journey and starts from town 1. Suppose Tina's car travels with speed 1 on every road (i.e., it will take L_k units of time to travel through $\{i_k, j_k\}$). However, whenever Tina travels through two roads in S consecutively, she needs to make a stop and spends D amount of time for repairing her car.

For an example, suppose Tina's car travels through roads $e_1 = \{1,2\}, e_2 = \{2,4\}, e_3 = \{4,7\}, e_4 = \{7,10\}, e_5 = \{10,n\}$. If $e_2, e_3, e_4, e_5 \in S$ and $e_1 \notin S$, then Tina's car needs to stop in towns $7, 10$, and it takes her $2D + L(e_1) + L(e_2) + L(e_3) + L(e_4) + L(e_5)$ time to reach town n, where $L(e_1), L(e_2), \dots, L(e_5)$ represent the lengths of roads $e_1, e_2, \dots, e_5$.

Describe an algorithm which finds the shortest time for Tina to reach town n. The algorithm should run in time $O(m \cdot \log(n))$.

You need to address the following aspects in your algorithm description:

- the graph algorithm used to solve this problem;
- the construction of the graph that you run this algorithm on;
- the correctness of the algorithm, i.e why any solution to the original problem gives you a solution to the graph problem and why any solution to the graph problem gives you a solution to the original problem
- the total running time of your algorithm.

Hint: You want to consider a new graph with vertex set $\{1, 2, \dots, n\} \times \{0, 1\}$, where for each town i, $\{0, 1\}$ records whether town i is reached immediately after going through a road in S.

Solution

- We use Dijkstra's algorithm to solve this problem.
- Let $G = (V, E)$ be the original graph given in problem. We will construct a directed graph $G' = (V', E')$ where $V' = V \times \{0, 1\}$. The vertex $(k, 1)$ represents that the last road we used to reach k was blocked, and $(k, 0)$ means it was not blocked.

For each non-blocked (two-way) edge $(u, v, 0, L) \in E$, we add to E' directed edges $((u, i), (v, 0), L)$ and $((v, i), (u, 0), L)$ for both $i \in \{0, 1\}$ (a total of 4 directed edges).

For each blocked (two-way) edge $(u, v, 1, L) \in E$, we add to E' directed edges $((u, 0), (v, 1), L)$, $((u, 1), (v, 1), L + D)$ and $((v, 0), (u, 1), L)$, $((v, 1), (u, 1), L + D)$ (a total of 4 directed edges). Further we add edge $((u, 1), (v, 1), L)$ if $v = n$ and edge $((u, 1), (v, 1), L)$ if $v \neq n$.

We start the Dijkstra from $(1, 0)$ and return the shorter of the two paths to either $(n, 0)$ or $(n, 1)$.

- Let ALG be the value returned by our algorithm, and let OPT be optimal (correct) value.

  For one direction, let $P' = ((u_0, b_0), (u_1, b_1), \ldots, (u_k, b_k))$ be the path in G' found by Dijkstra. We construct $P'' = (u_0, u_1, \ldots, u_k)$. By construction, P'' is a valid path between 1 and n in G. Furthermore, by construction, the length of P' is exactly the length of P'' plus we add D for each time we cross two consecutive blocked edges. Therefore, the length of P' is a valid solution to the problem. Therefore, $OPT \leq ALG$.

  For the other direction, let $P^* = (1 = u_0, u_1, \ldots, u_k = n)$ be the optimal path. We construct a path $P' = ((u_0, 0))$. For $i \geq 1$, if the edge $\{u_i, u_{i+1})$ is blocked, we append $(u_i, 1)$ to P' and $(u_i, 0)$ otherwise. By construction, the path P' exists and its total length is OPT. By minimality of the path returned by Dijkstra, we have $ALG \leq OPT$.

- Our algorithm first (1) constructs the graph, and then (2) runs the MST algorithm. Step (1) is easily implemented in $O(m)$ time, while step (2) takes $O(m \log n)$ time. The total runtime is $O(m \log n)$.

Grading

- 1 pt for using Dijkstra's algorithm;(0.5 pt for shortest path algorithm)
- 1.5 pts for constructing the correct graph(0.5 pts for correct set of vertices and 1 pt for correct set of edges).
- 1 pts for the correctness of your algorithm(0.5 pt for either direction of proof).
- 0.5 pt for arguing the running time of provided algorithm.

/ 4 P

d) Christina, a friend of Tina, gets a snow plow and plans to go for a road cleaning tour[1]. Every road has two sides. During the cleaning tour, when Christina traverses a road $e = \{u, v\}$ in the direction from u to v, the side of e for this direction is cleared. As a result, she receives $p(u, v)$ francs (from the inhabitants living on that cleared side of road e). At the same time, traveling along (and clearing) road e incurs a cost of $c(e)$ francs. So she makes money by clearing a road e from u to v if $p(u, v) > c(e)$ and she loses money if $p(u, v) < c(e)$.

Last year she noticed that even if all roads were blocked by snow, there was no profitable clearing tour starting from her hometown. Christina wants to find a different town, from which there is a profitable clearing tour if all the roads are blocked by snow.

To obtain full points for this task, you can choose to fulfill **one** of the following:

[1] A tour is a traveling path which starts and ends in the same town.

- Design an algorithm running in time $O(n \cdot m)$, which decides whether there is such a town (i.e., a town starting from which there is a profitable clearing tour if all the roads are blocked by snow);
- Design an algorithm running in time $O(n^3)$, which finds all such towns.

You need to address the following aspects in your algorithm description:

- the graph algorithm used to solve this problem;
- the construction of the graph that you run this algorithm on;
- the extraction of final solution;
- the total running time of your algorithm.

Solution 1: Design an algorithm running in time $O(n \cdot m)$, which decides whether there is such a town (i.e., a town starting from which there is a profitable clearing tour if all the roads are blocked by snow).

- We will use Bellman-Ford's algorithm that detects if there is a negative cycle.
- We use the same set of vertices and edges as in the problem, except we set the weight of an edge $e = (u, v)$ to be $c(e) - p(u, v)$.

  We will report there is a profitable path iff there exists a negative cycle in our constructed graph.
- The graph transformation takes $O(m)$ time and running the Bellman-Ford cycle detection takes $O(m \cdot n)$ time. The total runtime is $O(m \cdot n)$.

Solution 2: Design an algorithm running in time $O(n^3)$, which finds all such towns.

- We will use Floyd-Warshall's algorithm to compute all-pairs shortest paths in the constructed graph.
- We use the same set of vertices and edges as in the problem, except we set the weight of an edge $e = (u, v)$ to be $c(e) - p(u, v)$.

  For each edge $e = (u, v)$ (in the constructed graph), we will check if the weight $c(e) - p(u, v)$ of the edge e plus the shortest path from v to u is negative. If yes, we mark both u and v as towns from which we can start a profitable path.
- The graph transformation takes $O(m)$ time and running Floyd-Warshall takes $O(n^3)$ time. The total runtime is $O(n^3)$.

Grading

- 1 pt for using Bellman-Ford's algorithm for task 1 or Floyd-Warshall algorithm for task 2;
- 1 pts for constructing the correct graph.
- 1 pt for extracting the solution.
- 1 pt for arguing the running time of provided algorithm.