Assignment Solutions Complete Collection for AI2615

Prof. Chihao Zhang, Prof. Yuhao Zhang, Prof. Biaoshuai Tao

T.A.

Xiaolin Bu, Qing Chen, Zhidan Li, Lejun Min,
Jiaxin Song, Wenqian Wang, Yulin Wang,
Zhenrong Xue, Ruofeng Yang, Zonghan Yang,
*Zichen Zhu**

Spring 2022

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Contents

*Please feel free to contact me (JamesZhutheThird@vip.qq.com) if there is any mistake.
[1]The problem labeled †, ‡ and § applies to **AI2615-1**, **AI2615-2** and **AI2615-3** respectively.
[2]This document is modified from LaTeX template [Github] [OverLeaf] [SJTU-ShareLatex].

1 Assignment I - Basic Algorithms

1.1 *Problem 1 - Master Theorem (20 Points)*†‡§

Prove the following generalization of the master theorem. Given constants $a \geq 1, b > 1, d \geq 0$, and $w \geq 0$, if $T(n) = 1$ for $n < b$ and $T(n) = aT(n/b) + n^d \log^w n$, we have

$$T(n) = \begin{cases} O(n^d \log^w n) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \\ O(n^d \log^{w+1} n) & \text{if } a = b^d \end{cases}.$$

Solution. By the recurrence relation, we have

$$\begin{aligned} T(n) &= aT(n/b) + n^d \log^w n \\ &= a^2 T(n/b^2) + n^d \log^w n + a(n/b)^d \log^w (n/b) \\ &= a^3 T(n/b^3) + n^d \log^w n + a(n/b)^d \log^w (n/b) + a^2 (n/b^2)^d \log^w (n/b^2) \\ &\quad \vdots \\ &= n^d \log^w n + a(n/b)^d \log^w (n/b) + a^2 (n/b^2)^d \log^w (n/b^2) + \\ &\quad \cdots + a^{\log_b n} (n/b^{\log_b n})^d \log^w (n/b^{\log_b n}) \\ &< n^d \log^w n (1 + (a/b^d) + (a/b^d)^2 + \cdots + (a/b^d)^{\log_b n}) \end{aligned}$$

If $a < b^d$, then we have $a/b^d < 1$. Thus,

$$T(n) < n^d \log^w n \frac{1 - (a/b^d)^{\log_b n}}{1 - a/b^d} = O(n^d \log^w n).$$

If $a > b^d$, choose $\varepsilon > 0$ such that $\log_b a > d + \varepsilon$. Since $n^d \log^w n = O(n^{d+\varepsilon})$, applying the original master theorem on the following recurrence will yield $T(n) = O(n^{\log_b a})$:

$$U(n) = aU(n/b) + O(n^{d+\varepsilon}).$$

If $a = b^d$, we have $e/b^d = 1$. Therefore,

$$T(n) < n^d \log^w n \cdot \log_b n = O(n^d \log^{w+1} n).$$

*This solution is provided by **Prof. Biaoshuai Tao**.* □

1.2 *Problem 2 - Merge Sort (20 points)*[†‡§]

Recall that Merge Sort divides the sequence into two subsequences with nearly the same size and sort them recursively. What if we want the two subsequences to have different sizes? [3]

Please use the *one third dividing approach* (dividing the sequence $a_1, a_2, \ldots, a_n$ into $a_1, a_2, \ldots, a_{\lceil n/3 \rceil}$ and $a_{\lceil n/3 \rceil + 1}, a_{\lceil n/3 \rceil + 2}, \ldots, a_n$) to complete the *One Third Merge Sort Algorithm*, and analyze its time complexity.

Solution. The algorithm is given in Algorithm 1.

Algorithm 1 One-Third-Merge-Sort

ONETHIRDMERGESORT($a[1, \ldots, n]$)

1: **if** $n \leq 2$: sort a directly with at most one comparison
2: $b \leftarrow$ ONETHIRDMERGESORT($a[1, \ldots, \lceil \frac{n}{3} \rceil]$)
3: $c \leftarrow$ ONETHIRDMERGESORT($a[\lceil \frac{n}{3} \rceil + 1, \ldots, n]$)
4: **return** MERGE(b, c)

Let $T(n)$ be the time complexity for sorting an array of size n. It is easy to obtain the following recurrence relation for $T(n)$:

$$T(n) = T\left(\frac{1}{3}n\right) + T\left(\frac{2}{3}n\right) + O(n).$$

Therefore, there exists a constant $c > 0$ such that

$$T(n) \leq T\left(\frac{1}{3}n\right) + T\left(\frac{2}{3}n\right) + cn.$$

We conjecture that $T(n) = O(n \log n)$, and we will prove by induction that $T(n) \leq bn \log n$ for some constant $b > 0$.

For the base step, we consider $n = 2$. We have $T(2) = O(1)$, which is clearly at most $2b \log 2$ for a sufficiently large constant b.

For the inductive step, suppose $T(i) \leq bi \log i$ for every $i \leq n - 1$. We aim to prove that $T(n) \leq bn \log n$ for some constant b (that does not depend on n). By the recurrence relation, we have

$$\begin{aligned} T(n) &\leq T\left(\frac{1}{3}n\right) + T\left(\frac{2}{3}n\right) + cn \\ &\leq b \cdot \frac{1}{3}n \log\left(\frac{1}{3}n\right) + b \cdot \frac{2}{3}n \log\left(\frac{2}{3}n\right) + cn && \text{(induction hypothesis)} \\ &= bn \log n - \left(\frac{1}{3}b \log 3 + \frac{2}{3}b \log \frac{3}{2} - c\right) \cdot n, \end{aligned}$$

which is less than $bn \log n$ for a sufficiently large constant b that does not depend on n. This completes the inductive step. Thus, $T(n) = O(n \log n)$.

*This solution is provided by **Prof. Biaoshuai Tao**.* □

[3] Think: How the complexity depneds on $\alpha \in (0, 1)$ if one uses α *dividing approach*?

1.3 *Problem 3 - Median Trick (30 points + 5 points)*[†‡§]

For two vectors $\mathbf{a} = (a_1, \ldots, a_d), \mathbf{b} = (b_1, \ldots, b_d) \in \mathbb{R}^d$, we say $\mathbf{a}$ is *greater than* $\mathbf{b}$ if $a_k > b_k$ for each $k = 1, \ldots, d$. You are given two collections of vectors $A, B \subseteq \mathbb{R}^d$. The objective is to count the number of pairs $(\mathbf{a}, \mathbf{b}) \in (A, B)$ such that $\mathbf{a}$ is greater than $\mathbf{b}$. You can assume all the entries in all the vectors are distinct. Let $n = |A| + |B|$ and d be the dimension of the vectors.

1. (10 points) Design an $O(n \log n)$ time algorithm for this problem with $d = 1$.
2. (20 points) Design an algorithm for this problem with $d = 2$. Your algorithm must run in $o(n^{1.1})$ time.[4]
3. (5 points, bonus) Generalize the algorithm in the last question so that it works for general d. Analyze its running time. The running time must be in terms of n and d.

Solution. We will only solve **(1)** and **(3)**, as a solution for **(3)** can be directly used for **(2)**.

(1) When $d = 1$, A and B are sets of numbers. We first sort $A \cup B$ in descending order, while we label each number by either A or B indicating the membership and keep such labels during the sorting. Now we scan the sorted array, and maintain a set $S \subset A$ during the scanning which is initially set to be empty. Whenever a number with label A is scanned, we include it to S, and whenever a number with label B is scanned, we output all (a, b) such that $a \in S$ and b is this scanned number.

The correctness of this algorithm is self-evident. To analyze its time complexity, the sorting part requires $O(n \log n)$ time, while the scanning part only requires $O(n)$ times if we do not include the time for output. Therefore, the total time complexity is $O(n \log n)$.

(3) Firstly, we can assume without loss of generality that all vectors in $A \cup B$ have different values in each of the d coordinates: if some vectors have a same value at some coordinate i, we can add some disturbances ϵ to this value if the vectors are in B, and subtract some disturbances ϵ to this value if the vectors are in A. By choosing different small enough ϵ, it is easy to see that the output result does not change.

The main trick here is to find the median on one coordinate for $A \cup B$. In details, we find the median x of the d-th coordinates of the vectors in $A \cup B$. Every pair $(\mathbf{a}, \mathbf{b})$ such that $\mathbf{a}$ is greater than $\mathbf{b}$ should be of one of the three types:

- both a_d and b_d are less than x,
- both a_d and b_d are at least x, or
- $a_d \geq x$ and $b_d < x$.

[4] The are many possible algorithms for question 2. I encourage you to use *divide and conquer* and reduce to the question 1.

This naturally yields a divide and conquer algorithm.

Based on the "median trick", we can design the following algorithm OUTPUT((A, B, d)) which outputs all $\mathbf{a} \in A, \mathbf{b} \in B$ such that **a** is "bigger than" **b** *when restricted to the first* d *coordinates*.

Algorithm 2 Median Trick

1: **function** OUTPUT((A, B, d))
2: If $d = 1$, solve the problem according to Part (a), and terminate
3: Find the median x of the dth coordinates of $A \cup B$
4: Obtain $A_1 \subset A$ and $B_1 \subset B$ such that the dth coordinates of all vectors in $A_1 \cup B_1$ are less than x
5: Obtain $A_2 \subset A$ and $B_2 \subset B$ such that the dth coordinates of all vectors in $A_2 \cup B_2$ are at least x
6: OUTPUT((A_1, B_1, d))
7: OUTPUT((A_2, B_2, d))
8: OUTPUT($(A_2, B_1, d-1)$)

The correctness of the algorithm above is based on the said "median trick". To calculate the time complexity, denote by $T(n, d)$ the time complexity for $n = |A \cup B|$ and d being the restricted number of coordinates considered. Step 2 requires $O(n)$ time by the median-of-the-medians algorithm learned in the class. It is easy to see that step 3 and 4 require $O(n)$ time. Moreover, since A_1, A_2, B_1, B_2 are defined by the median, we have $|A_1 \cup B_1| = |A_2 \cup B_2| = \frac{1}{2}|A \cup B|$. Therefore, we have the recurrence relation

$$T(n, d) = 2T(n/2, d) + T(n, d-1) + cn.$$

We prove that $T(n, d) \le cn \log^d n$ by induction.

For the base step $d = 1$, $T(n, 1) \le cn \log n$ holds for all $n \ge 2$, which is true based on Part **(1)**. [5] For the other base step with $n = 2$ and $n = 3$, we have $T(n, d) = O(d)$, and $T(n, d) \le cn \log^d n$ clearly holds for sufficiently large c.

For the inductive step, assume the inequality holds for $d = 1, \dots, i-1$ and all $n \ge 2$, and the inequality also holds for $d = i$ and $n = 2, \dots, j-1$. We aim to show that it also holds for $d = i$ and $n = j$: $T(j, i) \le cj \log^i j$. By the induction hypothesis, we have

$$T(j, i) = 2T(j/2, i) + T(j, i-1) + cj \le cj \log^i \frac{j}{2} + cj \log^{i-1} j + cj \le cj \log^i j,$$

where the last inequality is due to

$$\begin{aligned} \log^i j - \log^i \frac{j}{2} &= \log^i j - (\log j - 1)^i \\ &= \log^{i-1} j + \log^{i-2} j (\log j - 1) + \cdots + (\log j - 1)^{i-1} \\ &\ge \log^{i-1} j + 1. \end{aligned}$$

Consequently, we have $T(n, d) = O(n \log^d n)$.

This solution is provided by ***Prof. Biaoshuai Tao****.* □

[5] More rigorously, Part **(1)** says $T(n, 1) = O(n \log n)$, which implies there exists some n_0 such that $T(n, 1) \le cn \log n$ holds for all $n > n_0$. However, we can refine our choice of c to make it also holds for $n = 2, \dots, n_0$ as well.

1.4 *Problem 4 - Median of Medians (30 points)*†

Recall that we have learned how to find the k-th element in a list with a randomized algorithm (randomly choose a pivot), can we do it deterministically? In this exercise, we will develop one called the *Median of Medians* algorithm, invented by Blum, Floyd, Pratt, Rivest, and Tarjan.

Why we need to pick the *pivot* x randomly in our randomized algorithm? This is to guarantee, at least in expectation, that the numbers less than x and the numbers greater than x are in close proportion. In fact, this task is quite similar to the task of "finding the k-th largest number" itself, and therefore we can bootstrap and solve it recursively!

Assume we have an array A of n distinct numbers and would like to find its k-th largest number.

1. Consider that we line up elements in groups of three and find the median of each group. Let x be the median of these $n/3$ medians. Show that x is close to the median of A, in the sense that a constant fraction of numbers in a is less than x and a constant fraction of numbers is greater than x as well. [6]
2. Design a recursive algorithm using the above idea and analyze the running time.

Solution. **(1)** Since x is the median of $\frac{n}{3}$ numbers and the numbers are distinctive to each other, there are exactly $\frac{n}{6}$ numbers smaller than x and exactly $\frac{n}{6}$ numbers larger than x for the same reason. Because those numbers are also medians of each group, exactly another number is smaller(larger) than each of them in their groups respectively. So totally there are $\frac{n}{6} * 2 = \frac{n}{3}$ numbers in a smaller than x and $\frac{n}{3}$ numbers in a larger than x as well.

(2)[7] We use *Median of Medians* to find a proper *pivot* x to recursively find the $k-th$ element. Also, we need to use this algorithm itself to find the median of medians, which implies another recursion.

By previous analysis, this reduce the size to at least $1/3$. Here are two recursions, a $T(n/3)$ for computing median of the $n/3$ medians, and $T(\gamma n)$ for computing the remaining list where γ is between $1/3$ and $1-1/3$. We have

$$2T(\frac{n}{3}) + cn \leq T(n) \leq T(\frac{n}{3}) + T(\frac{2n}{3}) + cn$$

for some constant c.

The worst case will be $T(n) = O(n \log n)$. [8]

[6] Can we improve the running time by increasing the number of elements (e.g. 4,5,6?) in each group? What is the best choice?

[7] Wikipedia contributors, "Median of medians," Wikipedia, The Free Encyclopedia, https://en.wikipedia.org/w/index.php?title=Median_of_medians (accessed March 23, 2022).

[8] *Hint:* It is not proper to use *Median of Medians* recursively to get the *pivot*. The *pivot* may be far from the true median after several recursions, and you cannot use the $1/3$ constant because the *pivot* you get this way is the median of medians of ...of medians rather than the median of medians.

Algorithm 3 QuickSelect

Input: List A, nubmer k
Output: $result$

function QUICKSELECT((A, k))
    $B \leftarrow$ MEDIANS$(A, 3)$
    $X \leftarrow$ QUICKSELECT$(B, \lceil |B|/2 \rceil)$ ▷ Recursively call QuickSelect to get the median of medians beacuse find median is a special case of find the k-th smallest element.
                                                ▷ Choose X as *pivot*
    $i \leftarrow$ PARTITION(A, X) ▷ Partition A by *pivot* X, i is the index of X in A after partition

    if $i > k$ **then**
        $result \leftarrow$ QUICKSELECT$(A[: i-1], k)$
    else
        if $i < k$ **then**
            $result \leftarrow$ QUICKSELECT$(A[i+1 :], k-i)$
        else
            $result \leftarrow X$
    return $result$

*This solution is provided by **T.A. Qiaoyu He** and **Stu. Yilin Sun**.* □

1.5 *Problem 5 - k-Smallest (30 points)*[‡§]

Given an array of n integers $x_1, x_2, ..., x_n$, there are queries of the following form: given an integer $1 \leq k \leq n$, you need to return the smallest k integers in the array (in no particular order). Design an $O(n)$ time preprocessing algorithm so that you can answer each query in $O(k)$ time.

Notice that a trivial $O(n \log n)$ time preprocessing algorithm is to sort the array, and each query is answered by just returning the first k integers, which requires $O(k)$ time.

Solution.

1. Apply the algorithm with the time cost Cn to find the $n/2$-th smallest number (where C is a constant). By the way, we have put all the numbers smaller than it in front, and the larger ones behind it.
2. Apply the same algorithm with the first half of the sequence with time cost $Cn/2$ to find the $n/4$-th smallest number.
3. Repeat to apply the algorithm to put the $n/8$-th, $n/16$-th, $\ldots$, 1-st smallest numbers in the right places.

   The above is the preprocessing, which has time complexity $C \cdot (n + n/2 + n/4 + \cdots) = 2C \times n = O(n)$. We get an array with the $1, 2, 4, \ldots, n/2$-th numbers in the right places. Besides, every segment $[2^k + 1, 2^{k+1} - 1]$ contains a random-shuffle for the numbers which should lie in it if the array were completely sorted.
4. For every query k, find the largest t with $2^t \leq k$. Print all the numbers ahead of 2^t. The remaining $k - 2^t < k/2$ numbers lie in the segment $[2^t + 1, 2^{t+1} - 1]$.
5. Find the $(k-2^t)$-th number in the segment of $[2^t+1, 2^{t+1}-1]$ with the time cost $C \times 2^t \leq C \times K = O(k)$.
6. Print out the first $k - 2^t$ numbers in the segment.

   Step 4 to 6 are designed for the queries with the time complexity of $O(k)$.

*This solution is provided by **Stu. Wentao Dong**.* □

1.6 *Problem 6 - The Second Largest Integer (30 points)*[‡§]

Given an array of n integers $x_1, x_2, \ldots, x_n$, design an algorithm that outputs *the second largest integer*. Your algorithm is required to make at most $n+\log n$ comparisons. (Notice that $n + \log n$ is the exact number, and there is no asymptotic notation here.) You can assume $n = 2^k$ for some $k \in \mathbb{Z}^+$.

Solution. Firstly, we provide a subroutine.

Algorithm 4 Subroutine: finding max element

FINDMAX($x[1, \ldots, n]$)

1: **if** $n \leq 2$: sort x directly with at most one comparison
2: $max_1 \leftarrow$ FINDMAX($x[1, \ldots, \frac{n}{2}]$)
3: $max_2 \leftarrow$ FINDMAX($x[\frac{n}{2} + 1, \ldots, n]$)
4: compare max_1 and max_2 and $max \leftarrow$ the bigger one.
5: AddtoEachCompareSet(max_1,max_2)
6: **return** max

With the $FindMax$ subroutine, we can get the largest element. We also maintain a $CompareSet$ for each element during the algorithm, which memorizes all the elements it has been compared to.

After obtaining the largest element x_j in $x_1, \ldots, x_n$ by $FindMax$, we do $FindMax$ again on x_j's $CompareSet$. The largest element in x_j's $CompareSet$ is what we want.

The correctness of $FindMax$ can be proved by induction. The reason why we can find the second largest integer in the largest element's $CompareSet$ can be proved by contradiction.

Then we analyze the number of comparisons it takes. The recurrence relation of $FindMax$ is $T(n) = 2T(\frac{n}{2}) + 1$ and we have $T(2) = 1$. Thus, $T(n) = 2^0 + 2^1 + \ldots + 2^{\log n - 1} = n - 1$. The largest integer has been compared to $\log n$ elements, which means there are $\log n$ elements in its $CompareSet$ and it takes another $T(\log n) = \log n - 1$ comparisons to find the second largest one. The total number of comparisons is $n - 1 + \log n - 1 = n + \log n - 2 < n + \log n$.

You may understand the algorithm better if you draw the corresponding comparison tree.

*This solution is provided by **T.A. Wenqian Wang**.* □

2 Assignment II - Graph Theory

2.1 *Problem 1 - Largest Profit-to-Cost Ratio (20/25 Points)*[†‡§]

Given a directed graph $G = (V, E)$ where each vertex can be viewed as a port. Consider that you are a salesman, and you plan to travel the graph. Whenever you reach a port v, it earns you a profit of p_v dollars, and it cost you c_{uv} if you travel from u to v. For any directed cycle in the graph, we can define a profit-to-cost ratio to be

$$r(C) = \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}}.$$

As a salesman, you want to design an algorithm to find the best cycle to travel with the largest profit-to-cost ratio. Let r^* be the maximum profit-to-cost ratio in the graph.

1. (10[†]/15[‡§] points) If we guess a ratio r, can we determine whether $r^* > r$ or $r^* < r$ efficiently? [9]
2. (10 points) Based on the guessing approach, given a desired accuracy $\epsilon > 0$, design an efficient algorithm to output a good-enough cycle, where $r(C) \geq r^* - \epsilon$. Justify the correctness and analyze the running time in terms of $|V|$, ϵ. In the analysis, we can assume $R = \max_{(u,v)\in E}(p_u/c_{uv})$ is a constant.

Solution. **(1)** Given a ratio r, we assign a weight $w(u, v) \triangleq rc(u, v) - p_v$ for each edge $(u, v) \in E$. Then run **Bellman-Ford** algorithm to determine whether there is a negative cycle in this graph. If it does not have a negative cycle, then $r^* \leq r$, else $r* > r$.

Correctness. First we can notice the following fact:

$$\begin{aligned} r \geq r(C) &\iff \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}} - r \leq 0 \\ &\iff \frac{\sum_{(u,v)\in C} [p_v - rc(u,v)]}{\sum_{(u,v)\in C} c_{uv}} \leq 0 \\ &\iff \sum_{(u,v)\in C} [p_v - rc(u,v)] \leq 0 \\ &\iff \sum_{(u,v)\in C} [rc(u,v) - p_v] \geq 0 \end{aligned}$$

Therefore, $r > r^*$ is equivalent to the graph $G = (V, E)$ with weight function w has no negative circles.

[9] Hint: Construct another graph with appropriate edge lengths and apply the shortest path algorithm.

Time Complexity. The time complexity of computing new weight for each edge is $O(|E|)$. The time complexity of running Bellman-Ford algorithm is $O(|V|\cdot|E|)$. Therefore, the total time complexity is $O(|V|\cdot|E|)$

(2) The top idea is using binary searching and the detailed algorithm is shown in Algorithm 5,

Algorithm 5 Find a good-enough cycle satisfying $r(C) > r^* - \epsilon$

Input: ϵ, Graph $G(V, E)$, profit $p_v, \forall v \in V$, cost $c_{uv}, \forall (u, v) \in E$
Output: a cycle C, which satisfies $r(C) > r^* - \epsilon$

use r_{max} to represent the upper bound of r^* and initial it with R
use r_{min} to represent the lower bound of r^- and initial it with -1
while $r_{max} - r_{min} > \epsilon$ **do**
    if $\frac{r_{max}+r_{min}}{2} < r^*$ **then**
        $r_{min} \leftarrow \frac{r_{max}+r_{min}}{2}$
    else
        $r_{max} \leftarrow \frac{r_{max}+r_{min}}{2}$
calculate weight $w(u, v) = r_{min}c_{u,v} - p_v$ for each edge $(u, v) \in E$
perform Bellman-Ford algorithm on G with weight function $w(u, v)$ and output a negative cycle C **return** the negative cycle C

Correctness. First, since $-1 < \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}} \leq \max_{(u,v)\in E}(\frac{p_v}{c_{uv}}), \forall C$, then r_{max} is initially no less than r^* and r_{min} is initially smaller than r^*.

Second, according to the correctness of binary searching, the property $r_{min} < r^* \leq r_{max}$ is always kept during iterations and $r_{max} - r_{min} \leq \epsilon$ after the end of the iterations. Therefore, $r^* - \epsilon < r_{min} < r^*$.

Third, since $r_{min} < r^*$, there exists a negative cycle C_1 on the graph with weight function $w(u, v) = r_{min}c_{u,v} - p_v$, which means $\sum_{(u,v)\in C_1} [r_{min}c(u, v) - p_v] < 0 \Rightarrow r_{min} < r(C_1) \Rightarrow r^* - \epsilon < r(C_1)$.

Time Complexity. The number of iterations in binary searching is $O(log(\frac{R}{\epsilon}))$ and the time complexity of each iteration is $O(|V||E|)$. The time complexity for calculating negative cycle is also $O(|V||E|)$. Therefore, the total time complexity is $O(|V||E| \cdot log(\frac{R}{\epsilon}))$

*This solution is provided by **T.A. Jiaxin Song**.* □

2.2 *Problem 2 - Eulerian circuit (20 points)*†

An Eulerian path in a directed graph is a path containing each edge exactly once. An Eulerian circuit in a directed graph is a Eulerian path that is a cycle. In the following, let $G = (V, E)$ be a *strongly connected* directed graph. [10]

1. (10 points) Prove that G contains an Eulerian circuits if and only if the in-degree and out-degree of each vertex $v \in V$ are the same. What is the sufficient and necessary condition for the existence of an Eulerian path?
2. (10 points) Give an $O(|E|)$ time algorithm to find an Eulerian circuit in case it exists. You need to clearly specify how your algorithm is implemented using appropriate data structure and prove its time complexity.

Solution. **(1).**

Sufficiency: Given that in-degree and out-degree of each vertex are the same, here we present an algorithm of finding a Eulerian circuit. Since G is strongly connected, there exists a cycle C_0. Starting from $i = 0$,

1. Delete all the edges in cycle C_i, then clean the vertices with zero degree.
   After that, the remaining graph consists of several strongly connected sub-graphs, within which the given property still holds because every vertex in C_i has both in-degree and out-degree decreased by 1.
2. Starting from one of the common vertices between C_i and the remaining graph, find a cycle C_{i+1} within that sub-graph and repeat step 1.
3. Once all the vertices are pruned, the path combination of all $C_i, (i = 0, 1, \dots)$, denoted as C_e, is an Eulerian circuit because every edge is visited only once.

Necessity: If G has an Eulerian circuit C_e starting from u, we traverse G in the order of it. Each time of visiting a vertex (except u) indicates one edge in and one edge out. And the final visit of u counteract its initial visit. So every vertex has equal in-degree and out-degree.

Sufficient and Necessary Condition for the existence of an Eulerian path is that exactly one vertex has $\deg_{out} - \deg_{in} = 1$, one vertex has $\deg_{in} - \deg_{out} = 1$, and every other vertex has equal in-degree and out-degree. This is proved by the former statements, only with the first vertex u not connected to the last vertex.

(2). According to the "Sufficiency" part in (1), we design the algorithm coded below.

[10] The in-degree of a vertex in G is the number of incoming edges: $\deg_{\text{in}}(v) = |\{u \in V : (u, v) \in E\}|$. The out-degree is the number of outgoing edges: $\deg_{\text{out}}(v) = |\{u \in V : (v, u) \in E\}|$

Algorithm 6 Find an Eulerian circuit (Hierholzer's Algorithm)

function FINDCYCLE(u)
    $u_0 = u$
    $V \leftarrow \{\}$, $E \leftarrow []$
    while $u_0 \notin u$.next_vertices **do**
        $V \leftarrow V \cup \{u\}$
        $v \leftarrow$ one of u's next vertices
        $E \leftarrow E +$ edge(u, v)
        $u \leftarrow v$
    $E \leftarrow E +$ edge(u, u_0)
    return (V, E)
function FINDEULERIANCIRCUIT($G(V, E)$)
    $C \leftarrow []$
    Pick a random vertex u
    while $G(V, E)$ is not empty **do**
        $C'(V', E') \leftarrow$ FINDCYCLE(u)
        $C \leftarrow C + C'$
        $E \leftarrow E \backslash E'$
        $V \leftarrow V \backslash \{v \in V'$ with zero degree$\}$
        Pick a vertex $u \in V'$ with $\deg_{out} > 0$ (if not exists, break)
    return C

Here in function FINDCYCLE we just need to follow a trail of edges from u until returning to u. It's not possible to get stuck at any vertex other than u because when the trail enters a vertex there must be an unused edge leaving it. Since every edge is visited only once, the complexity is $O(|E|)$.

*This solution is provided by **T.A. Lejun Min**.* □

2.3 *Problem 3 - Undirected Connected Graph (30/25 points)*[†‡§]

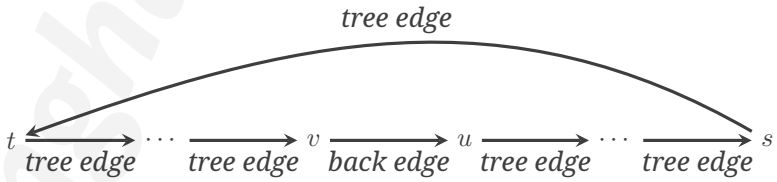
Let $G = (V, E)$ be an undirected connected graph. Let T be a depth-first search tree of G. Suppose that we orient the edges of G as follows: For each tree edge, the direction is from the parent to the child; for every non-tree (back) edge, the direction is from the descendant to the ancestor. Let G' denote the resulting directed graph.

1. (5 points) Give an example to show that G' is not strongly connected.
2. (5 points) Prove that if G' is strongly connected, then G satisfies the property that removing any single edge from G will still give a connected graph.
3. (5 points) Prove that if G satisfies the property that removing any single edge from G will still give a connected graph, then G' must be strongly connected.
4. (15[†‡]/10[§] points) Give an $O(|V|+|E|)$ algorithm to find all edges in a given undirected graph such that removing any one of them will make the graph no longer connected.

Solution. **(1)** When the graph G is $a - b$, the resulting directed graph G' is $a \to b$, which is not strongly connected.

(2) We prove this proposition by contradiction. Suppose there exists a edge in G such that removing it from G will make G disconnected. Hence G can be divided into two sub-graphs G_1 and G_2 and there is only one edge e between the two sub-graphs G_1 and G_2. We can notice that G' is not strongly connected whatever the direction of e is.

(3) First, we show that each node is strongly connected with its father node. For each node t in T, if it isn't the root node, suppose its father node is s. Due to the property of G, graph G is still connected after removing edge (s, t). Therefore, there must exist a back edge from one of the vertices v in the sub-tree t, to one of the ancestors u of s. Hence there exists a directed path from t to s in G': $t \to \ldots u \to v \to u \to \ldots \to s$.



Second, since each node is strongly connected with its parent, then it is also strongly connected with the root node. Therefore, each two nodes are strongly connected with each other in $G' \Rightarrow G'$ must be strongly connected.

(4) First of all, we define a edge e in the connected graph G is a bridge if removing it will make the graph disconnected.

The algorithm is as follows:

Algorithm 7 Find all the bridges of a connected graph

Input: Graph $G(V, E)$

Output: all the bridges

1: Do DFS Search in graph G, construct DFS search tree T, record the tree edges and the finish time of each node
2: Orient the edges of G and construct graph G' according to the question
3: Reverse the graph G' and do the second DFS according to the descending order of previous finish time and record each node's SCC index.
4: $sum \leftarrow 0$
5: **for** $e = (u, v) \in E$ **do**
6: **if** u and v have different SCC indexes **then**
7: $sum \leftarrow sum + 1$
return sum

Correctness. To prove the correctness, we need to show that:

- there is at most one path between two SCC in graph G';
- each undirected edge that corresponds to a directed edge in a SCC is not a bridge;
- each undirected edge that corresponds to a directed edge between two SCCs is a bridge.

For the first proposition, we can prove it by induction on the number of back edges: When the number of back edges is 0, all SCCs are all single nodes and there is at most one edge between two SCCs. Suppose the property holds when we have added $k(k \geq 0)$ back edges. When we add the $k+1$th back edge, it will make the two vertices that this edge connects connected. Hence it will result in merging the SCCs in a chain or just adding an edge inside a SCC. For the first case, it will just decrease the length of a path rather than making a disconnected path connected. For the second case, it will not create any edges between SCCs. Therefore, the property still holds.

For the second proposition, it can be proved by (b) and (c).

For the third proposition, it is obvious since we have proved there is at most one path between two SCCs.

Finally, the number of the edges that satisfy requested property is exactly equal to the number of directed edges between SCCs.

Time Complexity. The time complexity for two DFS is $O(|V| + |E|)$, the time complexity for constructing G' is $O(|V| + |E|)$ and the time complexity of computing the number of cut edges is $O(|E|)$. Therefore, the total time complexity is $O(|V| + |E|)$.

*This solution is provided by **T.A. Jiaxin Song**.* □

2.4 *Problem 4 - Dijkstra Implementation (30/25 points)*†§

We have seen in the class that Dijkstra algorithm fails if the graph contains negatively-weighted edges. Consider the following variant of Dijkstra algorithm. Given a directed weighted graph $G = (V, E, w)$ where $w(u, v)$ may be negative, find an integer W such that $w(u, v) + W > 0$ for each edge $(u, v) \in E$, and define the new weight of (u, v) as $w'(u, v) = w(u, v) + W$. Now, $G' = (V, E, w')$ is a positively weighted graph where the weight of each edge is increased by W. Implement Dijkstra algorithm on $G' = (V, E, w')$, and a shortest path from s to every vertex u in G' is also a shortest path in G.

1. (15†/10§ points) Does this algorithm work for directed acyclic graphs? If so, prove it; if not, provide a counterexample.

2. (15 points) A *directed grid* is a directed weighted graphs $G = (V, E, w)$ where the set of vertices is given by $V = \{v_{ij} \mid i = 1, \ldots, m; j = 1, \ldots, n\}$ ($m, n \in \mathbb{Z}^+$ are two parameters) and the set of directed edges is given by

$$E = \left(\bigcup_{i=1,\ldots,m-1;j=1,\ldots,n} \{(v_{ij}, v_{(i+1)j})\} \right) \cup \left(\bigcup_{i=1,\ldots,m;j=1,\ldots,n-1} \{(v_{ij}, v_{i(j+1)})\} \right).$$

   That is, the vertices form a $m \times n$ grid. There is a *directed* edge *from* every vertex *to* the vertex "right above" it, and there is a *directed* edge *from* every vertex *to* the vertex "to the right of" it (unless the vertex is "on the boundary"). The weight of each edge is specified as input and can be negative.

   Does the algorithm work for directed grids? If so, prove it; if not, provide a counterexample.

Alternative Problem 4 (25 points)‡

We have learned from the lecture that Dijkstra meet some problems when we have negative weights. The following algorithm attempts to find the shortest path from node s to node t in a directed graph with some negative edges:

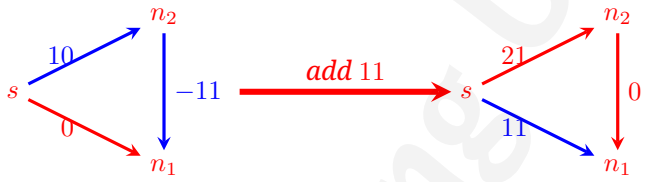
- Add a large enough number to each edge weight so that all the weights become positive, then run Dijkstra's algorithm.

Intuitively, the algorithm is not correct if there exists a negative cycle in the graph. How-

ever, does this approach work for the directed acyclic graphs?

1. (10 points) Given a DAG, either prove this algorithm correct on DAG, or give a counter-example.
2. (15 points) Consider a special directed layered graph (DLG) $G = (V, E)$. Each vertex has a layer v_ℓ chosen from $\{1, 2, 3, 4, ..., L\}$, and the layer of s is 0. We only have directed edges between adjacent layers, i.e., if $(u, v) \in E$, then $\ell_v = \ell_u + 1$. Either prove this algorithm correct on DLG, or give a counter-example.

Solution. **(a)** The algorithm is wrong on DAG. Here is a counter-example, where we need to calculate the shortest path from s to n_1.



For clarity, the left graph is the original graph, where edge $n_2 \to n_1$ has negative weight -11. The shortest path from s to n_1 is $s \to n_2 \to n_1$ in the left DAG. Unfortunately, if we add 10 on each edge's weight and get the right graph, which has no negative edges. The result of Dijkstra on the right graph is $s \to n_1$.

(b) The algorithm is correct on directed graph.

First, since there only exist edges between level i and level $i + 1$, all the paths from s to a node in level i have length i.

Second, suppose the large number is c and the original weight of edge $(u, v) \in E$ is w_{uv}. If the shortest path from s to t calculated by Dijkstra is $s \to v_1 \to ... \to v_n \to t$. Then for any other path $s \to v'_1 \to ... \to v'_n \to t$, we have

$$(w_{s,v_1} + c) + \sum_{i=1}^{n}(w_{v_i,v_j} + c) + (w_{v_n,t} + c) \le (w_{s,v'_1} + c) + \sum_{i=1}^{n}(w_{v'_i,v'_j} + c) + (w_{v_n,t} + c)$$

$$\Rightarrow w_{s,v_1} + \sum_{i=1}^{n} w_{v_i,v_j} + w_{v_n,t} \le w_{s,v'_1} + \sum_{i=1}^{n} w_{v'_i,v'_j} + w_{v_n,t},$$

which demonstrates path $s \to v_1 \to ... \to v_n \to t$ is also the shortest path in the original graph.

*This solution is provided by **T.A. Jiaxin Song**.* □

2.5 *Problem 5 - Path Containing Every Vertex (25 points)*[§§]

Given a directed graph $G = (V, E)$, a source vertex $s \in V$ and a destination vertex $t \in V$. Design an efficient algorithm to determine whether there is a path from s to t containing every vertex in V? (A vertex or an edge can appear in the path more than once.)

Solution. The algorithm starts by constructing a graph $G' = (V', E')$ where each vertex in V' corresponds to a strongly connected component in V and $(u', v') \in E'$ if there is a path in G from a vertex in the strongly connected component corresponding to u' to a vertex in the strongly connected component corresponding to v'. Let $s' \in V'$ be the vertex representing the strongly connected component containing s, and $t' \in V'$ be the vertex representing the strongly connected component containing t. Notice that G' is a directed acyclic graph.

Then the algorithm finds a topological order on G'. The algorithm outputs "yes" if and only if

1. s' is the first vertex in the topological order, and t' is the last, and
2. there is an edge $(u', v') \in E'$ between any two adjacent vertices u' and v' in the topological order.

Correctness Firstly, since a vertex and an edge can be visited multiple times, if the path enters a strongly connected component, the path can visit all the vertices in this strongly connected component before leaving it. Therefore, our problem reduces to the problem of deciding whether we can visit all the vertices in G' by a path from s' to t'. We prove that this is possible if and only if 1 and 2 above hold.

The if direction is trivial. If 1 and 2 hold, the vertices in the topological order form a s'-t' path visiting every vertex in G' exactly once.

For the only-if direction, if there is a s'-t' path visiting every vertex in G', then each vertex must be visited exactly once (if a vertex is visited twice, then there must be a cycle in G'). It is then easy to see that G' can have only one possible topological order: the one following the vertices on the path. To see this, if v' is after u' on the path, there is a path from u' to v', and v' must be after u' in the topological order. With this observation, 1 and 2 hold straightforwardly.

Time Complexity We have seen in the class that building G' and finding a topological order can both be done in $O(|V| + |E|)$ time by DFS. Therefore, the overall time complexity is $O(|V| + |E|)$.

*This solution is provided by **Prof. Biaoshuai Tao**.* □

2.6 *Problem 6 - Path with Length of k(25 points)*[§]

Given a undirected unweighted graph $G = (V, E)$, a source vertex s, a destination vertex t, and a positive integer k, we would like to determine if there is a path from s to t with length exactly k. In this question, a path can visit vertices and edges multiple times. Design an efficient algorithm to determine if there is a path from s to t with length *exactly* k.

Solution. **(a)** Given the input graph $G = (V, E)$, we construct a graph G' as follows:

- for each vertex $u \in V$, construct two vertices u^o, u^e for G', where the superscripts o and e stand for odd and even respectively.
- for each edge (u, v) in G, construct two edges (u^o, v^e) and (u^e, v^o) in G' respectively.

The key observation here is that *an odd s-t path in G corresponds to an s^e-t^o path in G' and vice versa; an even s-t path in G corresponds to an s^e-t^e and vice versa.*

The algorithm is given below.

Algorithm 8 Find the shortest odd s-t path

Input: $G = (V, E)$, $s, t \in V$
Output: a path p from s to t with a minimum odd length

1: construct G' described above
2: perform Breadth-First-Search on G' to obtain a shortest path p' from s^e to t^o
3: let $p' = \{(s^e, u_1^o), (u_1^o, u_2^e), (u_2^e, u_3^o), \ldots, (u_{\ell-1}^e, t^o)\}$
4: **return** $p = \{(s, u_1), (u_1, u_2), (u_2, u_3), \ldots, (u_{\ell-1}, t)\}$

Notice that, if we want a shortest even s-t path, we only need to change step 2 such that the shortest path p' is from s^e to t^e. Correspondingly, the last edge in p' at Step 3 would be $(u_{\ell-1}^o, t^e)$, i.e., from a vertex with an odd superscript to the destination vertex t with an even superscript.

To prove the correctness of the algorithm, we need to show that 1) p is an odd path and 2) p is the shortest one.

To show 1), we first notice that the lengths of p' and p are the same by our algorithm. Thus, it suffices to show that p' has an even length. By our construction of G', an vertex with an odd (even, resp.) superscript can only be connected to an vertex with an even (odd, resp.) superscript. Thus, the superscripts of the vertices on p' change alternatively. Since we start from a vertex s^e with an even superscript to a vertex t^o with an odd superscript, the length of p' must be odd.

To show 2), let p' be the path on G' we have found at Step 2, and let p be the path output at Step 4. Let ℓ be the length of p and p'. Suppose we have an odd-length s-t path p_* in G that is shorter than p. Let $p_* = \{(s, u_1), (u_1, u_2), \ldots, (u_{\ell_*-1}, t)\}$, where $\ell_* < \ell$ is the length of p. We consider the corresponding path p'_* in G'. Since ℓ_* is an odd number, the superscript for u_{ℓ_*-1} on path p'_* must be e. Thus, $p'_* = \{(s^e, u_1^o), (u_1^o, u_2^e), \ldots, (u_{\ell_*-1}^e, t^o)\}$ is a path from s^e to t^o in G' with length ℓ_*. We have found an s^e-t^o path p'_* in G' that is shorter than p'. Since p' found by Breadth-First-Search is supposed to be the shortest, this is a contradiction.

The time complexity is $O(|V|+|E|)$: since G' has only twice as many vertices and edges as G, Step 1 takes $O(|V|+|E|)$ time; Step 2 takes $O(|V|+|E|)$ time from what we know about Breadth-First-Search; Step 3 takes $O(|E|)$ time.

(b) The algorithm is described as follows:

- if k is odd, find a shortest odd s-t path p using the algorithm in (a) and let ℓ be its length; output "yes" if $\ell \leq k$, and output "no" otherwise.
- if k is even, find a shortest even s-t path p using the algorithm in (a) and let ℓ be its length; output "yes" if $\ell \leq k$, and output "no" otherwise.

To prove the correctness of the algorithm, we consider two cases, $\ell > k$ and $\ell \leq k$. Notice that ℓ and k have the same parity by our algorithm, so $k - \ell$ must be an even number. In the first case, we clearly know that we cannot go from s to t with exactly k steps, for otherwise, the shortest odd (even) path from s to t would have been k or even less. In the second case, let $p = \{(s, u_1), (u_1, u_2), \ldots, (u_{\ell-1}, t)\}$ be the shortest odd (even) s-t path with length ℓ. We can append the segment $\{(t, u_{\ell-1}), (u_{\ell-1}, t)\}$ to p for $\frac{k-\ell}{2}$ times and obtain an s-t path with length exactly k. The validity of this approach is based on that $\frac{k-\ell}{2}$ is a non-negative integer, since $\ell \leq k$ and $k - \ell$ is even.

The time complexity is obviously the same as the algorithm in (a), which is $O(|V| + |E|)$.

*This solution is provided by **Prof. Biaoshuai Tao**.* □

2.7 *Problem 7 - Amortized Cost of ADD (20 points)*‡

Let us consider the following situation. An integer (initially zero) is stored in binary. We have an operation called ADD that adds one to the integer. The cost of ADD depends on how many bit operations we need to do. (one bit operation can flip 0 to 1 or flip 1 to 0.) The cost can be high when the integer becomes large. Use amortized analysis to show the amortized cost of ADD is $O(1)$. You should define the potential function in your analysis.

Solution. The potential function is defined as follows:

$$\phi(n) = \sum_{i=0}^{k} a_i, \text{ where } n = \overline{(a_k ... a_1 a_0)_2}.$$

When the lowest i bits of n are 1 and the $i+1$th -lowest bit is 0, we only need to flip the lowest $i+1$ bits, hence $C_i = i+1$. Meanwhile, $\Delta\phi = -(i-1)$ since the lowest i 1 are flipped into 0 and the $i+1$th 0 is flipped into 1 and the number of 1 decreases $i-1$. Therefore,

$$\begin{aligned}
&\hat{C}_i = C_i + \Delta\phi = i + 1 - (i-1) = 2 \\
&\Rightarrow \sum_{i=0}^{n} C_i \leq 2(n+1) + \max_{1 \leq j \leq n} \phi(j) - \phi(0) \\
&\Rightarrow \sum_{i=0}^{n} C_i \leq 2(n+1) + log(n) = O(n)
\end{aligned}$$

By amortized analysis, the amortized cost of ADD is $\frac{O(n)}{n} = O(1)$.

*This solution is provided by **T.A. Jiaxin Song**.* □

2.8 *Problem 8 - Strongly Connected Directed Graph (5 points)*‡

Let G be a directed graph. Suppose that G is not strongly connected, and you are allowed to add extra edges into G. What is the smallest number of extra edges to make G strongly connected? Write down your idea and intuition, try to prove it if you believe you are right.

1. (5 points) Write down your ideas and guesses to earn the 5 points!
2. (0 points) Write down the complete proof for happiness.

Solution. **(a)** Suppose all the SCCs of graph G are $C_1, \ldots, C_k$. We construct a new graph G' as follows: Use $\{C_i\}_{i=1}^k$ as the vertice set of G'. If there is a path from C_i to C_j in G, then we create an edge from C_i to C_j in G'.

We use $d^+(C_i)$ to represent the out-degree of C_i and $d^-(C_i)$ to represent the in-degree of C_i, and define three special sets of SCCs as follows:

$$\mathcal{A} \triangleq \{C_i \mid d^-(C_i) = 0 \ \wedge d^+(C_i) \neq 0\}$$
$$\mathcal{B} \triangleq \{C_i \mid d^+(C_i) = 0 \wedge d^-(C_i) \neq 0\}$$
$$\mathcal{C} \triangleq \{C_i \mid d^+(C_i) = 0 \wedge d^-(C_i) = 0\}$$

For clarity, the vertices in $\mathcal{A}$ are called 'source' and the vertices in $\mathcal{B}$ are called 'sink'.

My guess is: when $k = 1$, the smallest number of extra edges is 0; when $k \geq 2$, the smallest number of extra edges is $\max\{|\mathcal{A}|, |\mathcal{B}|\} + |\mathcal{C}|$.

(b) For simplify, we use OPT to represent the smallest number of extra edges to make G strongly connected.

When $k = 1$, the graph G has already been strongly connected and there is no need to add edges. Hence $OPT = 0$.

When $k \geq 2$, first, we will prove $OPT \geq \max\{|\mathcal{A}|, |\mathcal{B}|\} + |\mathcal{C}|$. Since the final graph is strongly connected, we need to add at least one out-edge for each SCC in $\mathcal{A} \cup \mathcal{C}$ and at least one in-edge for each SCC in $\mathcal{B} \cup \mathcal{C}$ in graph G'. However, we can not add out-edges (or in-edges) for two SCCs in $\mathcal{A} \cup \mathcal{C}$ (or $\mathcal{B} \cup \mathcal{C}$) at the same time by just adding one directed edge in the original graph. Therefore,

$$OPT \geq \max\{|\mathcal{A} \cup \mathcal{C}|, |\mathcal{B} \cup \mathcal{C}|\} = \max\{|\mathcal{A}|, |\mathcal{B}|\} + |\mathcal{C}|.$$

Second, we will shown how to make G' strongly connected with adding no more than $\max\{|\mathcal{A}|, |\mathcal{B}|\} + |\mathcal{C}|$ extra edges in the original graph. When G' is strongly connected, G is obviously connected.

According to the definition of $\mathcal{A}$ and $\mathcal{B}$, we can notice that: (1) for each vertice u in $\mathcal{A}$, there exists an edge from u to a vertice $v \in \mathcal{B}$; (2) for each vertice v in $\mathcal{B}$, there exists an edge from a vertice $u \in \mathcal{A}$ to v. Or formally, the following two propositions are true:

- There exists a mapping $\phi_1 : \mathcal{A}_2 \to \mathcal{B}_1$, s.t. for each vertice v in A_2, there exists an edge from v to $\phi_1(v)$;
- There exists a mapping $\phi_2 : \mathcal{B}_2 \to \mathcal{A}_1$, s.t. for each vertice v in $\mathcal{B}_2$, there exists an edge from $\phi_2(v)$ to v;

Suppose the maximum matching between $\mathcal{A}$ and $\mathcal{B}$ is $(C_{a_1}, C_{b_1}), \ldots, (C_{a_l}, C_{b_l})$, define $\mathcal{A}_1 \triangleq \{C_{a_i} \mid 1 \leq i \leq \ell\}, \mathcal{A}_2 \triangleq \mathcal{A} \backslash A_1$ and $\mathcal{B}_1 \triangleq \{C_{b_i} \mid 1 \leq i \leq \ell\}, \mathcal{B}_2 \triangleq \mathcal{B} \backslash \mathcal{B}_1$. As shown in Figure 1, the red edges and green edges are the original edges between SCCs.

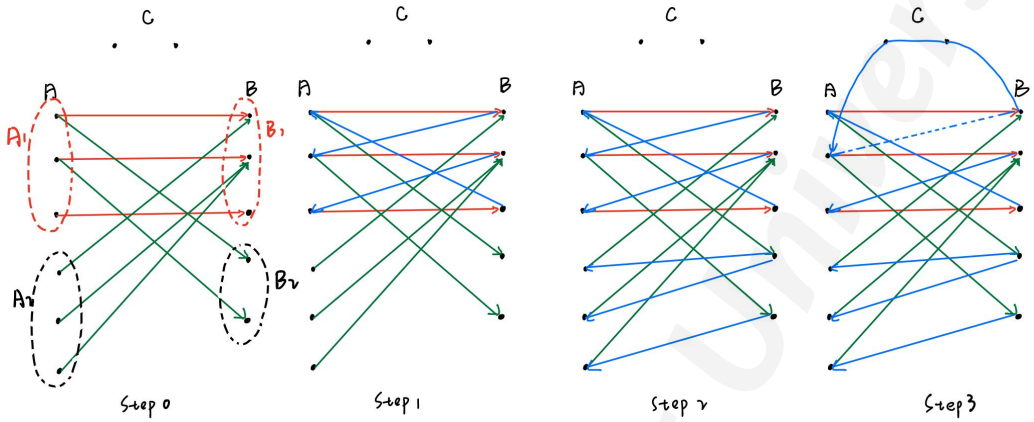


Figure 1: The process of adding edges.

As shown in Figure 1, we add the blue edges in the following three steps:

- Step1: For each $1 \leq i \leq \ell$, add an edge from C_{b_i} to $C_{a_{i+1}}$, where $C_{a_{\ell+1}} \triangleq C_{a_1}$. We have added k edges and $\mathcal{A}_1 \cup \mathcal{B}_1$ is strongly connected now.
- Step2: Use $\max\{|\mathcal{A}_2|, |\mathcal{B}_2|\}$ edges to connect the remaining vertices in $\mathcal{A}_2, \mathcal{B}_2$. For clarity, $\forall u \in \mathcal{B}_2$, add an edge from u to $v \in \mathcal{A}_2$, and $\forall v \in \mathcal{A}_2$ make sure it's connected with an edge coming from $v' \in \mathcal{B}_2$. Obviously, this can be achieved by adding $\max\{|\mathcal{A}_2|, |\mathcal{B}_2|\}$ edges. Now the following two propositions are true:
  - There exists a mapping $\phi_3 : \mathcal{A}_2 \to \mathcal{B}_2$, s.t. for each vertice v in $\mathcal{A}_2$, there exists an edge from $\phi_3(v)$ to v;
  - There exists a mapping $\phi_4 : \mathcal{B}_2 \to \mathcal{A}_2$, s.t. for each vertice v in $\mathcal{B}_2$, there exists an edge from v to $\phi_4(v)$;

  With the help of $\phi_1 \sim \phi_4$, it's not hard to verify that $\mathcal{A} \cup \mathcal{B}$ is strongly connected now.
- Step3: Randomly select an edge added in the previous steps and replace it with a path of length $|\mathcal{C}|+1$ that goes through all the vertices in $\mathcal{C}$.

Moreover, it's trivial to prove that the SCCs in $\{C_i \mid d^+(C_i) \neq 0 \wedge d^-(C_i) \neq 0\}$ are also strongly connected with $\mathcal{A} \cup \mathcal{B} \cup \mathcal{C}$.

In summary, total number of the edges added is $\ell + \max\{|\mathcal{A}_1| - \ell, |\mathcal{B}_1|\} - l + (|\mathcal{C}| + 1 - 1) = \max\{|\mathcal{A}|, |\mathcal{B}|\} + |\mathcal{C}|$.

*This solution is provided by **T.A. Jiaxin Song**.* □

3 Assignment III - Graph Theory II and Greedy Algorithm

3.1 *Problem 1 - Customer Waiting Time (20 Points)*[†]

A server has n customer waiting to be served. The service time required by each customer is known in advance: it is t_i minutes for customer i. So if, for example, the customers are served in order of increasing i, then the ith customer has to wait $\sum_{j=1}^{i} t_j$ minutes.

We wish to minimize the total waiting time

$$T = \sum_{i=1}^{n} (\text{time spent waiting by customer } i.)$$

Give an efficient algorithm for computing the optimal order in which to process the customers. Prove the correctness of your algorithm.

Solution. **Algorithm**: The optimal order is simply the ascending order of the service times. We can use QUICKSORT to sort the array $\{t_i\}$.

Proof of Correctness: The total waiting time is

$$T = \sum_{i=1}^{n} \sum_{j=1}^{i} t_j = \sum_{i=1}^{n} (n - i + 1) t_i.$$

Suppose the sequence $\{t_1, t_2, t_3, \ldots, t_n\}$ is in ascending order, i.e. $t_1 \leq t_2 \leq t_3 \leq \cdots \leq t_n$, simply by the **rearrangement inequality** we know such a T is the minimum waiting time because it is the so-called "inverse product sum".

Below is the proof of rearrangement inequality for completeness of this problem.

Denote the two sequence that have already sorted by ascending order by $\{a_i\}$ and $\{b_i\}$. Let $p_i = \sum_{j=1}^{i} b_j$, $q_i = \sum_{j=1}^{i} b_{k_j}$. It's easy to see $p_i \leq q_i$ for $i \leq n-1$ and $p_n = q_n$.

Then

$$\begin{aligned}
\sum_{i=1}^{n} a_{n-i+1} b_i &= \sum_{i=1}^{n-1} a_i (p_{n-i+1} - p_{n-i}) + a_n p_1 \\
&= \sum_{i=1}^{n-1} (a_{n-i+1} - a_{n-i}) p_i + a_1 p_n \\
&\leq \sum_{i=1}^{n-1} (a_{n-i+1} - a_{n-i}) q_i + a_1 q_n \\
&= \sum_{i=1}^{n} a_{n-i+1} b_{k_i}
\end{aligned}$$

Another side of the inequality is not used in our problem, so the proof is omitted here.

*This solution is provided by **T.A. Lejun Min** and **Stu. Yilin Sun**.* □

3.2 *Problem 2 - Kruskal's Algorithm (35/25 points)*†‡§

In the class, we learnt the Kruskal's algorithm to find a minimum spanning tree (MST). The strategy is simple and intuitive: pick the best legal edge in each step. The philosophy here is that local optimal choices will yield a global optimal. In this problem, we will try to understand to what extent this simple strategy works. To this end, we study a more abstract algorithmic problem of which MST is a special case.

Consider a pair $M = (U, \mathcal{I})$ where U is a finite set and $\mathcal{I} \subseteq 2^U$ is a collection of subsets of U. We say M is a *matroid* if it satisfies

- **(hereditary property)** $\mathcal{I}$ is nonempty and for every $A \in \mathcal{I}$ and $B \subseteq A$, it holds that $B \in \mathcal{I}$.
- **(exchange property)** For any $A, B \in \mathcal{I}$ with $|A| < |B|$, there exists some $x \in B \backslash A$ such that $A \cup \{x\} \in \mathcal{I}$.

Each set $A \in \mathcal{I}$ is called an *independent set*.[11]

1. (5 points) Let $M = (U, \mathcal{I})$ be a matroid. Prove that maximal independent sets are of the same size.
2. (5 points) Let $G = (V, E)$ be a simple undirected graph. Let $M = (E, \mathcal{S})$ where $\mathcal{S} = \{F \subseteq E | F \text{ is acyclic}\}$. Prove that M is a matroid. What are maximal sets of this matroid?[12]
3. (10†/5‡§ points) Let $M = (U, \mathcal{I})$ be a matroid. We associate each element $x \in U$ with a non-negative weight $w(x)$. For every set of elements $S \subseteq U$, the weight of S is defined as $w(S) \triangleq \sum_{x \in S} w(x)$. Now we want to find a maximal independent set with maximum weight. Consider the following greedy algorithm.

Algorithm 9 Find a maximal ind. set with maximum weight

Input: A matroid $M = (U, \mathcal{I})$ and a weight function $w : U \to \mathbb{R}_{\geq 0}$.

Output: A maximal ind. set $S \in \mathcal{I}$ with maximum $w(S)$.

```
S ← ∅
Sort U into decreasing order by weight w
for x ∈ U in decreasing order of w do
    if S ∪ {x} ∈ I then S ← S ∪ {x}
return S
```

Now we consider the first element x the algorithm added to S. [13]Prove that there must be a maximal independent set $S' \in \mathcal{I}$ with maximum weight containing x.

4. (5 points) Use above algorithm to solve the MST problem and convince yourself that it is equivalent to Kruskal's algorithm.

5. (10†/5‡§ points) Let $U \subseteq \mathbb{R}^n$ be a finite collection of n-dimensional vectors. Assume $m = |U|$ and we associate each vector $\boldsymbol{x} \in U$ a positive weight $w(\boldsymbol{x})$. For any set of vectors $S \subseteq U$, the weight of S is defined as $w(S) \triangleq \sum_{\boldsymbol{x} \in S} w(\boldsymbol{x})$. Design an efficient algorithm to find a set of vectors $S \subseteq U$ with maximum weight and all vectors in S are linearly independent.

Solution. **(1).** We prove it by contradiction. Assume that there are two maximal independent sets A_1 and A_2, $|A_1| < |A_2|$. For A_1, there is no $B \in \mathcal{I}$ such that $A_1 \subsetneq B$. According to the **exchange property**, there exists some $x \in A_2 \backslash A_1$ such that $A_1 \cup \{x\} = B \supsetneq A_1$, which leads to contradiction. Hence all maximal independent sets are of the same size.

(2).

- **Hereditary**: Firstly for any single edge $e \in E$, $\{e\} \in \mathcal{S}$. So $\mathcal{S}$ is nonempty. For every $A \in \mathcal{S}$ and $B \subseteq A$, B must be acyclic as well since no new edges are introduced.
- **Exchange**: $A, B \in \mathcal{S}$ with $|A| < |B|$. Denote the vertices of A, B as V_A, V_B. Denote the forest consisted of A's vertices and edges as F_A, and each subtree comprises a_i vertices. Consider the conditions of edges in $B \backslash A$:

  1. At least one vertex is not in F_A;
  2. Both vertices are in F_A but belong to different substrees;
  3. Both vertices are in F_A and belong to the same substree.

  1-type and 2-type edges won't make F_A cyclic, but 3-type edges will. For each subtree in F_A, there's at most $a_i - 1$ 3-type edges in B. Totally there can be at most $\sum_i (a_i - 1) = |A|$ 3-type edges in B. From the assumption we know that $|B| > |A|$, so B contains at least one 1-type or 2-type edge e. Hence, $A \cup \{e\} \in \mathcal{S}$.

Therefore M is a matroid.

The maximal sets of M is the sets of spanning trees of G. The maximal property holds because any new edge will make it cyclic.

[11]A set $A \in \mathcal{I}$ is called *maximal* if there is *no* $B \in \mathcal{I}$ such that $A \subsetneq B$.
[12]Hint: When proving the *exchange property* of two independent sets A, B, consider the forest induced by A in G.
[13]Hint: Prove by contradiction and use the exchange property.

(3). Assume that x does not belong to any maximal independent set $S' \in \mathcal{I}$ with maximum weight. According to the **exchange property**, there exists some $y_1 \in S' \backslash \{x\}$ such that $\{x, y_1\} \in \mathcal{I}$. Repeat adding new element to S until $S = \{x, y_1, \ldots, y_{n-1}\}$ and $|S| = |S'|$. Due to the algorithm, x has the maximum weight, so

$$\begin{aligned} w(S) &= \sum_{z \in S} w(z) \\ &= w(x) + \sum_{i=1}^{n-1} w(y_i) \\ &\geq \sum_{z \in S'} w(z) = w(S'), \end{aligned}$$

which leads to contradiction. Therefore there must be a maximal independent set $S' \in \mathcal{I}$ with maximum weight containing x.

(4). Simply using the settings in problem (2), we've proved that M is a matroid. Associate each edge $x \in U$ with a non-negative weight $w(x)$. The algorithm is exactly the same as Kruskal's algorithm.

To prove that the algorithm does ouput a maximal independent set with maximum weight, we can use the conclusion in (3). For every new element added to S, the property in (3) also holds (just replace $\{x\}$ with $\{x_1, \ldots, x_k\}$, $k = 1, \ldots, n$ in the proof). Hence, the final output S must be a maximal independent set with maximum weight.

(5). Let $\mathcal{I}$ be the collection of subsets of U that within each subset the vectors are linearly independent. We first prove that $M = (U, \mathcal{I})$ is a matroid.

- **Hereditary**: $\mathcal{I}$ is not empty since for every $\boldsymbol{x} \in U$, $\{\boldsymbol{x}\} \in \mathcal{I}$. For every $A \in \mathcal{I}$ and $B \subseteq A$, B must be linearly independent because no new vectors are introduced.
- **Exchange**: $A, B \in \mathcal{I}$ with $|A| < |B|$. Since A is linearly independent, it can be considered as a basis of the subspace V_A with dimension $r_A = |A|$. Likewise B is a basis of the subspace V_B with dimension $r_B = |B|$. Given that $|A| < |B|$, there must exist at least one vector $\boldsymbol{x} \in B \backslash A$ such that $\boldsymbol{x} \notin V_A$, otherwise V_B can simply be expressed by linear combinations of vectors in V_A. Hence $A \cup \{\boldsymbol{x}\}$ is also linearly independent, meaning that $A \cup \{\boldsymbol{x}\} \in \mathcal{I}$.

Then we design an algorithm with the same idea in (3).

Algorithm 10 Find a set of vectors with maximum weight

Input: A matroid $M = (U, \mathcal{I})$ and a weight function $w : U \to \mathbb{R}_{\geq 0}$.

Output: A set of vectors $S \in \mathcal{I}$ with maximum $w(S)$.

```
S ← ∅
Sort U into decreasing order by weight w
for x ∈ U in decreasing order of w do
    if S and x are linearly independent then S ← S ∪ {x}
return S
```

*This solution is provided by **T.A. Lejun Min**.* □

3.3 *Problem 3 - The Gas Cost (25 points)*[†‡§]

Suppose you are a driver, and you plan to drive from A to B through a highway with distance D. Since your car's tank capacity C is limited, you need to refuel your car at the gas station on the way. We are given n gas stations on the highway with surplus supply. Let $d_i \in (0, D)$ be the distance between the starting point A and the i-th gas station. Let p_i be the price for each unit of gas at the i-th gas station. Suppose each unit of gas exactly supports one unit of distance. The car's tank is empty at the beginning, and the 1-st gas station is at A. Design efficient algorithms for the following tasks.

1. (10[†]/5[‡§] points) Determine whether it is possible to reach B from A.
2. (15[†]/20[‡§] points) Minimized the gas cost for reaching B.

Please prove the correctness of your algorithms and analyze their running times. You are asked to implement your algorithm as efficient as possible.

Solution. **(1).** Let d_{n+1} be the distance between A and B. Simply check if $d_{i+1} - d_i < C$ for every $i = 1, 2 \ldots, n$. If so, the driver is possible to reach B. Otherwise no.

The complexity of this algorithm is $O(n)$.

(2). Take this greedy strategy: at a gas station, refuel the car just enough to reach the next cheaper gas station. If there is no cheaper gas station within reach (i.e. within distance C), fill the tank fully.

In order to get the next cheaper gas station for each station, we can use a priority queue to store the stations in descending order of $\{d_i\}$ first and then ascending order of $\{p_i\}$. Denote f_i as the "next cheaper gas station index" for station i. Let the $n+1$-th station be B, and the price of it is 0. The priiority queue contains B initially. For each station i starting from n to 1,

- Pop the stations in the priority queue until the popped station j satisfies $p_j < p_i$.
- Let $f_i = j$.
- Push station i into the queue.

Now we get the sequence $\{f_i\}$.

The main algorithm is coded below. (See Algorithm 11).

Complexity:For the computation of f_i, each station is pushed and popped only once, so the time complexity is $O(n \log n)$. The main algorithm loops every station at most once, so the complexity is $O(n)$. Therefore the overall complexity is $O(n \log n)$.

Algorithm 11 Get the minimum gas cost for reaching B

Input: $C, D, d_i \in (0, D), p_i, f_i$.

Output: The minimum gas cost for reaching B.

Complexity: $O(n)$.

```
i ← 1
gas ← 0, cost ← 0
while i ≤ n do
    j ← f_i
    if d_j − d_i ≤ gas then
        i ← j
    else if d_j − d_i ≤ C then
        i ← j
        gas ← d_j − d_i
        cost ← cost + (d_j − d_i − gas) × p_i
    else
        i ← i + 1
        gas ← C
        cost ← cost + (C − gas) × p_i
return cost
```

Proof of correctness: When reaching station i, let g_i be the gas amount according to our strategy, and g'_i be the gas amount according to the optimal strategy. We can prove by induction that for every i, $g_i = g'_i$.

- Firstly $g_0 = g'_0 = 0$.
- Suppose that for each $k < i$, $g_k = g'_k$ holds. We then prove that $g_i = g'_i$ holds as well.
- Let $j = f_i$, we consider the conditions as following:
  - j is within reach, i.e. $d_j - d_i \leq C$.

    If $g'_i > g_i$, meaning that the optimal strategy refuels more gas here, we can decrease g'_i and add the same amount to g'_j. Since $p_j < p_i$, we can still reach station j with less cost spent. It leads to contradiction.

    If $g'_i < g_i$, meaning that the optimal strategy refuels less here, we can't reach j directly from i. Therefore it requires to refuel the tank between station i and j by at least $\sum_{k=i+1}^{j-1} g_k \geq g_i$. Then we can increase g'_i and decrease the same refueling amount between i and j. Since $p_i < p_k$ for $k \in (i, j)$, we can reach station j with less cost spent. It's contradictory as well.
  - j is out of reach, i.e. $d_j - d_i > C$. Since our strategy fills the tank, g'_i can only be smaller than g_i. Similarly, Then we can increase g'_i and decrease the same refueling amount between i and j. It leads to contradiction as well.

  Hence, $g_i = g'_i$ holds. By induction, $g_i = g'_i$ for every $i \in [n]$.

Proved that $g_i = g'_i$, we know that our strategy is simply the optimal strategy.

*This solution is provided by **T.A. Lejun Min**.* □

Alternative Solution 1(Greedy Algorithm). **(1)&(2)** We solve two sub questions together, i.e., if we can reach B, we will output the minimum cost, otherwise, we will output 'impossible'. We assume that the input is given in the ascending order of the distance, otherwise, we can sort them first that additionally costs $O(nlogn)$ time.

The algorithm is built on a greedy approach. We keep driving stations by stations from A to B, Whenever we are at a station i with G unit of gas in the tank, we determine how much we need to refuel by the following steps.

- The algorithm is built on a greedy approach. We keep driving stations by stations from A to B, Whenever we are at a station i with G unit of gas in the tank, we determine how much we need to refuel by the following steps.
- If $d_j \leq d_i + C$, we refuel the tank to $d_j - d_i$ and pay $\max\{d_j - d_i - G, 0\} \cdot p_i$
- Otherwise, if $d_j > d_i + C$, we refuel the tank to C and pay $(C - G) \cdot p_i$

Proof of The Correctness

Following the greedy approach above, we prove that we can reach B with the minimized cost if it is possible. At first, if our algorithm can not reach B, then it must fail to reach the next station at a station i. Following the greedy approach, we will fill up the tank in this case, and thus $i + 1$ should be more than C apart from i. Hence, reaching the station $i + 1$ is impossible.

Then, we prove the optimality of the cost. Let $g(i)$ be the refueling unit of gas at station i in our algorithm, $G(i) = \{g(1), g(2), ..., g(i)\}$ be the set of strategy until i, we prove that $G(i)$ is a subset of one optimal strategy by induction. Then, it implies that $G(n)$ is one of the optimal strategy. We start from the base case that $G(0) = \varnothing$ is a subset of one optimal solution, then we assume that the strategy before i (i.e. $G(i-1)$) is a subset of one optimal solution, we prove that if we refuel $g(i)$ at station i (i.e. $G(i) = G(i-1) \cup g(i)$), we are still in a subset of one optimal solution. Suppose not, there is an optimal solution $G'(i) = G(i-1) \cup g'(i)$, where $g'(i) \neq g(i)$. Consider the two choices made by our algorithm at station i:

- At first, we discuss the case that the first cheaper station j (or $j = n + 1$) has distance $d_j \leq d_i + C$. If $g'(i) > g(i)$, consider the optimal strategy $G'(n) \supset G'(i)$, and let us do the following modification: decrease $g'(i)$ to $g(i)$ and increase $g'(j)$ to $g'(j) + g(i) - g'(i)$. It is easy to check that the strategy after modification is still feasible because we can still reach j with $g(i)$ refueling at i, and the total cost will decrease because p_j is cheaper than p_i, which is a contradiction. If $g'(i) < g(i)$, there should not be enough gas to reach j only by $G'(i)$, so that the optimal strategy $G'(n) \supset G(i)$ should refuel some gas after i and before j to support the car to reach j, and we have $\sum_{k=i}^{j-1} g'(k) \geq g(i)$. Then, we can increase $g'(i)$ to $g(i)$ and decrease totally $g(i) - g'(i)$ units of refueling among $g(k)$ from $i + 1$ to $j - 1$. We can easily check that it is still a feasible strategy, with at least the same cost as $G'(n)$ because all p_k can not be cheaper than i. Thus, we have that the modified strategy is still an optimal strategy, and it is a super set of $G(i) = G(i - 1) \cup g(i)$.
- Then, we discuss the case that the next cheaper station j is not reachable ($d_j > d_i + C$). In this case, we fill up the tank, so the only possible case is $g'(i) < g(i)$. Now because we need reach the location

$d_i + C < D$, similar as before, we should have $\sum_{k=i}^{j'} g'(k) \geq g(i)$ where j' is the farthest reachable station from i with $d_{j'} \leq d_i + C$. Then, consider the optimal strategy $G'(n) \supset G'(i)$, we can increase $g'(n)$ to $g(i)$ and decrease totally $g(i) - g'(i)$ unit of refueling among $g(k)$ from $i+1$ to j'. The modified strategy is still feasible with no larger cost because all p_k is not cheaper than p_i. It means the modified strategy is also optimal, and it is a super set of $G(i) = G(i-1) \cup g(i)$.

Time Complexity Analysis

Following the greedy approach, we can simply formalize the algorithm. We let the car move from the first station, determine how much gas we should refuel, and move to the next. In each round, we should at least pay $O(n)$ time to find j, so the time complexity is $O(n^2)$ in total.

Improvement by Priority Queue The slowest part in the previous analysis is that we need $O(n)$ time to find the first j that is cheaper than i. We improve this part by priority queue. We start from the end (i.e. $i = n$), and we view $n+1$ as the cheapest station of all, so we have $f[i] = n+1$. Then we calculate $f[i]$ from n to 1 with a priority queue. The priority queue stores the potential nearest cheaper stations in ascending order of the distance from A. It is easy to check that if a station j has larger distance than j', and p_j is not cheaper than $p_{j'}$, then j is impossible to play as the nearest cheaper station for those stations before j'. So the priority queue sorted by ascending order of the distance is also sorted by the descending order of the price. When we calculate $f[i]$, we should find from the first station in the queue, until the first one that is cheaper than i, and it is exactly $f[i]$. At the same time, when we pass these stations in the queue that are not cheaper than i, it means that we can pop them from the list because they all have larger distance than i. After removing, we push i into the queue as the first station. At the end, because each station can be at most pushed into the list once, be passed and popped once, we can calculate all $f[i]$ in totally $O(n)$ time. Finally, after we spend $O(n)$ time to create the array of $f[i]$, we can easily use $O(1)$ time to find j in each round i with the help of $f[i]$. The total time is bounded by $O(n)$. □

Alternative Solution 2(Linear Programming). Let g_i be the refueling unit of gas at station i, hence the original problem corresponds to the following linear programming:

$$\begin{aligned} \min_{g_1,\ldots,g_n} & \left(\sum_{i=1}^{n} g_i \times d_i\right) \\ \text{subject to} \quad & d_2 \leq g_1 \leq C \\ & d_3 \leq g_1 + g_2 \leq C + d_1 \\ & \cdots \\ & D \leq g_1 + \ldots + g_n \leq C + d_n. \end{aligned}$$

Therefore, this problem can be solved within polynomial time.

*This solution is provided by **Prof. Yuhao Zhang** and **TA. Jiaxin Song**.* □

3.4 *Problem 4 - Find A Subset Covering All Vertices (20/25 points)*[†‡§]

Given a constant $k \in \mathbb{Z}^+$, we say that a vertex u in an undirected graph *covers* a vertex v if the distance between u and v is at most k. In particular, a vertex u covers all those vertices that are within distance k from u, including u itself. Given an undirected *tree* $G = (V, E)$ and the parameter k, consider the problem of finding a minimum-size subset of vertices that covers all the vertices in G.

1. (10 points) Design an efficient algorithm for the problem above with $k = 1$.
2. (10[†]/15[‡§] points) Design an efficient algorithm for the problem above with general k.

Please prove the correctness of your algorithms and analyze their running times.

Solution. **(1)** A key observation to this question is that, for an internal vertex u at the second last level with children $u_1, \ldots, u_i$ who are leafs of the tree, we should definitely choose u: we need to choose at least one vertex in $\{u, u_1, \ldots, u_i\}$ to cover the leafs, and choosing u is at least as good as choosing any of $u_1, \ldots, u_i$. Therefore, the idea here is to iteratively find an uncovered leaf, and choose its parent. However, there are two caveats here.

Caveat 1: After choosing a vertex u, it is incorrect to remove all the vertices covered by u. Consider the example where u is an internal vertex with leafs $u_1, \ldots, u_i$ who are leafs, and u's parent, v, is also the parent of j leafs $v_1, \ldots, v_j$. That is, the leafs $v_1, \ldots, v_j$ are at a higher level than the leafs $u_1, \ldots, u_i$. In this case, when i and j are more than 1, we need to choose both u and v. However, if we choose u first and removed v (since v is covered by u), we cannot obtain the optimal solution. Therefore, we should only *label* those covered vertices instead of *removing* them.

Caveat 2: Describing the algorithm as "iteratively find an uncovered leaf and choose its parent" is still inaccurate. It is possible that, at certain stage of the algorithm, all the leafs of the trees are covered and there are still some uncovered internal vertices. In this case, the algorithm should continue by selecting the parent of an uncovered leaf *after removing covered vertices*. An accurate way of describing the algorithm is "iteratively find an uncovered *vertex at the lowest level* and choose its parent."

Algorithm 12 Find a subset covering all vertices with $k = 1$

```
sort all vertices by their levels, from the lowest to the highest; let a[1, ..., n] be the resultant array
initialize S ← ∅ and covered[a[i]] = false for each i = 1, ..., n
for each i = 1, ..., n
    if covered[a[i]] = true, continue
    include the parent of a[i] to S, or include a[i] to S if a[i] is the root
    set the value of covered for the chosen vertex and all its neighbors to true
endfor
return S
```

We can prove the correctness of the algorithm by induction. Let $u_1, u_2, \ldots, u_t$ be the order of vertices added to S, and let $S(i) = \{u_1, u_2, \ldots, u_i\}$. Firstly, by our algorithm, $S = S(t)$ covers all the vertices. It remains to show that S is optimal (i.e., no subset of smaller size can cover all vertices). Let P_i be the

statement that $S(i)$ is a subset of an optimal solution S^*. We need to prove that P_t is true. (This is sufficient: $S(t) = S$ already covers all the vertices and $S(t) \subseteq S^*$, so we have $|S| = |S(t)| \leq |S^*|$.)

For the base step, u_1 is a parent of many leafs, and the optimal solution must choose at least one vertex from u_1 or one of its children. If the optimal solution chooses one of its children, it is easy to see that, by changing this selection to u_1, all the vertices in the tree are still covered and the size of the solution is unchanged. Thus, in this case, we can modify this optimal solution to another optimal solution that contains u_1.

The inductive step is similar. Suppose P_{i-1} is true. To show P_i is true, u_i chosen at the i-th iteration must be the parent of one or many of the uncovered vertices at the lowest level. (Otherwise, u_i is the root, in which case u_i is the last vertex chosen in S and is the only uncovered vertex; the solution is clearly optimal given P_{i-1}.) Similar as before, in the optimal solution, u_i or one of its uncovered children must be chosen by the optimal solution. If u_i is not chosen by the optimal solution, changing the optimal solution's choice to u_i gives another optimal solution.

To analyze the time complexity, firstly, notice that $|E| = |V| - 1$ for trees. Step 1 requires $O(|V|)$ time, as we only need to perform a tree search by either Breadth-First-Search or Depth-First-Search. Step 2 requires $O(|V|)$ time. To analyze the time complexity for the for-loop, the difficult part is to analyze Step 6. In each iteration, a vertex and its neighbors are searched. Throughout the for-loop, an edge (u, v) is searched only when u or v is selected by S. Therefore, each edge has been searched at most twice. Thus, the time complexity for the for-loop is $O(|E|) = O(|V|)$. The overall time complexity is therefore $O(|V|)$.

(2) The idea is very similar to (1). If we have an uncovered leaf, it is always optimal to choose its k-th ancestor (or the root node if the leaf itself is at a level lower than k). There is only one subtle issue here. In (1), we can find *any* uncovered leaf and choose its parent. Here, we have to find an uncovered leaf *at the lowest level*. Consider the example where the tree has $k + 1$ levels. The root r has two children u and v, where u is a root of a subtree with k level and v is a leaf. If we first find v and choose its only ancestor r, we need to choose another vertex to cover all the vertices in the tree. However, the optimal solution only needs to choose one vertex, namely, u. Nevertheless, we will be fine if we always find an uncovered vertex *at the lowest level* and choose its k-th ancestor.

The algorithm is similar to Algorithm 3. All we need to do is to change Step 5 to "include the k-th ancestor of $a[i]$ to S, or include the root if the level of $a[i]$ is less than k", and change Step 6 to "set the value of `covered` to `true` for all the vertices covered by the chosen vertex."

A rough analysis for the time complexity gives us $O(|V|^2)$. In particular, each execution of Step 6 takes $O(|V|)$ time. In fact, there are more clever implementations of the algorithm that could give us overall time complexity $O(|V|)$ by exploiting the idea of Depth-First-Search. You will receive full credit for this problem with an $O(|V|^2)$ answer.

*This solution is provided by **Prof. Biaoshuai Tao**.* □

3.5 *Problem 5 - Submodular Function (25 points)*[‡§]

Given a ground set $U = \{1, \ldots, n\}$, a *set function on* U is a function $f : \{0,1\}^U \to \mathbb{R}$ that maps a subset of U to a real value. A set function f is *submodular* if

$$f(S \cup \{v\}) - f(S) \geq f(T \cup \{v\}) - f(T)$$

holds for any $S, T \subseteq U$ with $S \subseteq T$ and any $v \in U \setminus T$. We make the following assumptions on a submodular set function f:

- Nonnegative: $f(S) \geq 0$ for any $S \subseteq U$; you can assume $f(S)$ is always a rational number.
- Monotone: $f(S) \leq f(T)$ for any $S, T \subseteq U$ with $S \subseteq T$.
- $f(S)$ can be computed in a polynomial time with respect to $n = |U|$.

Given a positive integer $k > 0$ as an input, the goal is to find $S \subseteq U$ that maximizes $f(S)$ subject to the cardinality constraint $|S| \leq k$. Design a polynomial time $(1 - 1/e)$-approximation algorithm for this maximization problem. Prove that the algorithm you design runs in a polynomial time and provides a $(1 - 1/e)$-approximation.

Solution. This can be done by a greedy algorithm given in Algorithm 13.

Algorithm 13 Greedy algorithm for maximizing a submodular function

```
S ← ∅
for i = 1, ..., k:
    find v ∈ U \ S that maximizes f(S ∪ {v}) − f(S)  // equivalently, find v ∈ T \ S maximizing f(S ∪ {v})
    update S ← S ∪ {v}
endfor
return S
```

The algorithm requires at most $n \cdot \min\{k, n\}$ evaluations of the function $f(\cdot)$. Since $f(\cdot)$ can be computed in a polynomial time, the overall time complexity for Algorithm 13 is polynomial.

To show it provides a $(1-1/e)$-approximation, we first prove the following lemma. Let $S_\ell = \{u_1, \ldots, u_\ell\}$ be the set S after ℓ iterations of the for-loop in Algorithm 13

Lemma 1. Let $S' = \{v_1, \ldots, v_k\} \subseteq U$ be any subset of k elements in U. We have $f(S_\ell) \geq \left(1 - (1 - \frac{1}{k})^\ell\right) f(S')$.

Proof. We prove it by induction on ℓ.

For the base step with $\ell = 0$, we have $1 - (1 - \frac{1}{k})^\ell = 0$, and $f(S_\ell) \geq 0 = \left(1 - (1 - \frac{1}{k})^\ell\right) f(S')$.

For the inductive step, suppose $f(S_\ell) \geq \left(1 - (1 - \frac{1}{k})^\ell\right) f(S')$ holds for $\ell = 1, \ldots, i$. We will show that it holds for $\ell = i + 1$. By the greedy nature of the algorithm, the $(i+1)$-th element u_{i+1} must maximizes

$f(S_i \cup \{u_{i+1}\}) - f(S_i)$. Therefore, for any $v_j \in S'$, we must have

$$f(S_i \cup \{u_{i+1}\}) - f(S_i) \geq f(S_i \cup \{v_j\}) - f(S_i).$$

By averaging over $v_1, \ldots, v_k$, we have

$$\begin{aligned} f(S_i \cup \{u_{i+1}\}) - f(S_i) &\geq \frac{1}{k} \sum_{j=1}^{k} (f(S_i \cup \{v_j\}) - f(S_i)) \\ &\geq \frac{1}{k}(f(S_i \cup S') - f(S_i)). \end{aligned} \tag{†}$$

In the set cover problem or the max-k-coverage problem, we have proved (†) by an element-counting argument. Here, we will prove it by the submodularity of f. By taking $S = S_i$, $T = S_i \cup \{v_1, \ldots, v_{j-1}\}$ and $v = v_j$ in the definition $f(S \cup \{v\}) - f(S) \geq f(T \cup \{v\}) - f(T)$ of submodularity, we have

$$f(S_i \cup \{v_j\}) - f(S_i) \geq f(S_i \cup \{v_1, \ldots, v_j\}) - f(S_i \cup \{v_1, \ldots, v_{j-1}\}),$$

which implies

$$\sum_{j=1}^{k} (f(S_i \cup \{v_j\}) - f(S_i)) \geq \sum_{j=1}^{k} (f(S_i \cup \{v_1, \ldots, v_j\}) - f(S_i \cup \{v_1, \ldots, v_{j-1}\})) = f(S_i \cup S') - f(S_i),$$

which proves (†).

By rearranging (†), we have

$$\begin{aligned} f(S_{i+1}) = f(S_i \cup \{u_{i+1}\}) &\geq \frac{1}{k} f(S_i \cup S') + \left(1 - \frac{1}{k}\right) f(S_i) \\ &\geq \frac{1}{k} f(S') + \left(1 - \frac{1}{k}\right) f(S_i) && \text{(monotonicity)} \\ &\geq \frac{1}{k} f(S') + \left(1 - \frac{1}{k}\right) \left(1 - \left(1 - \frac{1}{k}\right)^{i}\right) f(S') && \text{(induction hypothesis)} \\ &= \left(1 - \left(1 - \frac{1}{k}\right)^{i+1}\right) f(S'). \end{aligned}$$

This concludes the inductive step. □

. Finally, let $S^* = \{o_1, \ldots, o_k\}$ be the optimal solution and $S_k = \{v_1, \ldots, v_k\}$ be the solution output by the algorithm. By applying Lemma 3.5 with $S' = S^*$, we have $f(S_\ell) \geq (1-(1-\frac{1}{k})^k) f(S^*) \geq (1-1/e) f(S^*)$, which implies the algorithm is a $(1-1/e)$-approximation. Notice that $(1-(1-\frac{1}{k})^k)$ is decreasing in k and it has limit $(1-1/e)$.

This solution is provided by ***Prof. Biaoshuai Tao****.* □

3.6 *Problem 6 - Spanning Tree (5 points)*[‡§]

Given an undirected unweighted connected graph $G = (V, E)$, our goal is to find a spanning tree T with the maximum number of leafs. Consider the following natural local search algorithm: for any spanning tree T, add an edge e and remove an edge f in the cycle of $T \cup \{e\}$, as long as the new tree $(T \setminus \{f\}) \cup \{e\}$ has more leafs than T. Prove that this is a $\frac{1}{9}$-approximation algorithm.

Hint: Let T be a tree output by the local search algorithm, and let $\ell(T)$ be the number of the leafs in T.

- Show that the number of vertices of degree more than two in T is less than $\ell(T)$.
- Consider any *maximal* path P in T consisting only those vertices with degrees at most 2. (A path P with this property is *maximal* if this property fails to hold for any path P' that contains P as a proper subset) Show that for *any* spanning tree S, the number of leafs of S in P is at most 4.

Extra: (not for credits, just for fun) This is actually a $\frac{1}{8}$-approximation algorithm. Can you prove it?

Solution. We will follow the hints provided.

(a) The number of vertices of degree more than two in T is less than $\ell(T)$. Let n_2 be the number of vertices with degree two, and $n_{>2}$ be the number of vertices with degree more than two. If we sum up the degrees of all vertices, we have

$$\ell(T) + 2n_2 + 3n_{>2} \leq 2(n-1),$$

where the left hand side provide a lower bound to the sum of degrees, while the right hand side is exactly this sum (there are $n-1$ edges in a tree, and each edge contribute 2 to the sum).

The left hand side can be rewritten as

$$\text{LHS} = 2(\ell(T) + n_2 + n_{>2}) + (n_{>2} - \ell(T)) = 2n + (n_{>2} - \ell(T)),$$

which further implies $n_{>2} - \ell(T) \leq -2 < 0$, and thus $n_{>2} < \ell(T)$.

(b) Consider any maximal path P in T consisting only of degree ≤ 2 vertices. Then for any spanning tree S, the number of leaves of S in P is at most 4. We prove it by contradiction, and suppose there exists a maximal path P and there are 5 leaves of S in P.

For each vertex $v \in P$, we say that v *is outgoing* if there exist $u \notin P$ such that $(u, v) \in E$. Our first observation is that there are at least three outgoing edges on P if P contains at least 5 leaves of S. This is because each outgoing edge can at most support 2 leaves, so two outgoing edges can only support 4 leaves. The figure on the left hand side of Figure 2 illustrates this. In the figure, the thin line segment represent the

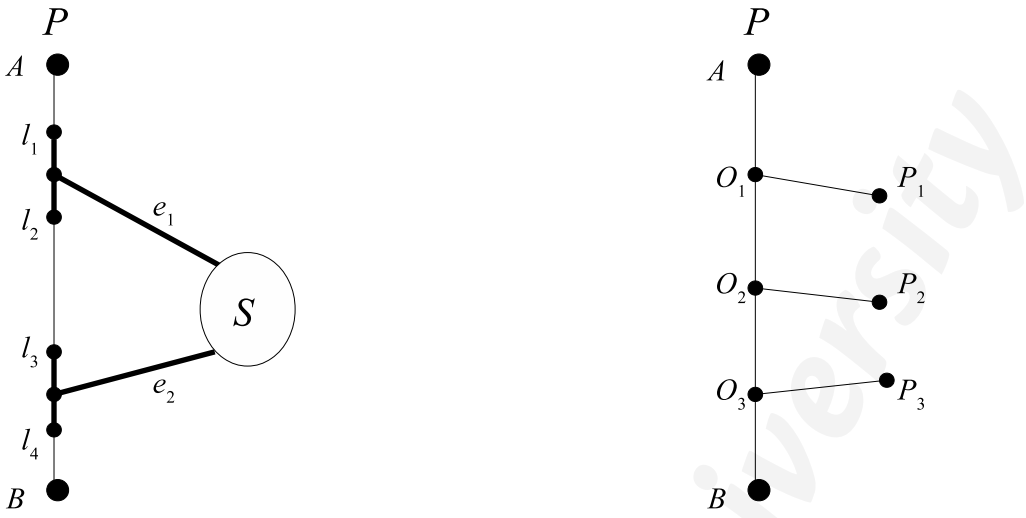


Figure 2: Problem 6 illustrations

path P with A, B being the endpoints, and the bold line segments are part of the tree S. The figure shows the two outgoing edges e_1, e_2 are supporting the four leaves $\ell_1, \ell_2, \ell_3, \ell_4$. Obviously, if there is a fifth leaf we must introduce another outgoing edge. Therefore, we conclude that there are at least three outgoing vertices. We name them O_1, O_2, O_3 in the order shown on the right hand side of Figure 2. Let P_1, P_2, P_3 be the three vertices outside P that are directed connected to them respectively.

Now consider $T \cup \{(O_2, P_2)\}$. Then $T \cup \{(O_2, P_2)\}$ contains a cycle, and either the path $O_2 \to A$ or the path $O_2 \to B$ is on this cycle. Assume without loss of generality that $O_2 \to A$ is on the cycle. Notice that we already know that the vertex O_1 between O_2 and A.

- Case (i): there exists another vertex between O_2 and A. Then we can find a middle edge (u, v) such that $u \neq A$ and $v \neq O_2$. In this case, $T' := T \cup \{(O_2, P_2)\} \setminus \{(u, v)\}$ is also a spanning tree. Moreover, T' has more leaves than T, because 1) u and v are two new leaves and 2) T' can at most lose one leaf, P_2 (which may be a leaf of T). This contradicts to that T is locally optimal.

- Case (ii): O_1 is the only vertex between O_2 and A. In this case, each of the two outgoing edges (P_1, O_1) and (P_2, O_2) has only supported one leaf. Given that there are five leaves and the third outgoing edge (P_3, O_3) can support at most two leaves, there must be another outgoing edge (O_4, P_4).

  Assume without loss of generality that O_3 is in between O_2 and O_4. Consider $T \cup \{(O_3, P_3)\}$. It has a cycle, and either $O_3 \to A$ or $O_3 \to B$ is in the cycle. If $O_3 \to A$ is in the cycle, we can set $T' = T \cup \{(O_3, P_3)\} \setminus \{(O_1, O_2)\}$ and derive the same contradiction.

  If $O_3 \to B$ is in the cycle, we can similarly consider two cases regarding whether O_4 is the only vertex between O_3 and B. If it is not, then we can follow case (i) and get contradiction. If it is, we can conclude that each of (P_3, O_3) and (P_4, O_4) has supported only one leaf, and there must be a fifth outgoing edge $(O_{2.5}, P_{2.5})$ such that $O_{2.5}$ is between O_2 and O_3. By adding $(O_{2.5}, P_{2.5})$ into T and deleting either (O_2, O_1) or (O_3, O_4) (depending on which of $O_{2.5} \to A$ and $O_{2.5} \to B$ is in the cycle), the number of leaves increases, which again contradicts that T is locally optimal.

We have seen contradictions in all cases, so the assumption that there are at least five leaves of S in P is false.

To conclude that the algorithm is a $\frac{1}{9}$-approximation, we consider two different kinds of maximal paths P.

1. If one of A, B is a leaf of T, then we say that P is a *leaf path*.
2. Otherwise, we say that P is an *inner path*.

An important observation is that *the number of inner paths is at most the number of leaf paths*. For one way to see this, we contract every path P to an single edge (A, B). Then each inner node in the tree after contraction has more than one child. It is well-known that the number of inner nodes is at most the number of leaves for such kind of trees, which implies our observation, as we can build an one-to-one correspondence between the paths and the non-root nodes: 1) each leaf node corresponds to a leaf path, and 2) each inner node corresponds to the inner path connecting to its parent.

Let n_i be the number of inner paths in T, and n_ℓ be the number of leaf paths. We obviously have $n_\ell = \ell(T)$, and we have seen $n_i \le n_\ell$.

For the optimal solution T^* that has most leaves, by (b), it can have at most 4 leaves on each of those n_i inner paths, and at most 4 leaves on each of those n_ℓ leaf paths. Even if all those $n_{>2}$ nodes that are not contained in any path are leaves of T^*, the total number of leaves in T^* is at most

$$4n_i + 4n_\ell + n_{>2} \le 4\ell(T) + 4\ell(T) + \ell(T) = 9\ell(T),$$

where we have applied the result in (a) to have $n_{>2} \le \ell(T)$. Therefore, the algorithm is a $\frac{1}{9}$-approximation.

Extra For statement (b), if the path P is a leaf path, we can actually show that there can be at most three leaves of S in P.

Suppose there are at least four leaves of S in P. Since P is a leaf path, either A or B is a leaf of T. Assume A is the leaf without loss of generality.

By similar analysis, we can show that there are at least two outgoing vertices O_1, O_2 if the number of leaves on P is at least four. Let the order of O_1, O_2 be that O_1 is the one closer to A. Since A is a leaf, the cycle in $T \cup \{(O_1, P_1)\}$ contains the path $O_1 \to B$ (it can never be $O_1 \to A$).

We can get contradictions in a similar way:

- Case (i): if O_2 is not the vertex between O_1 and B, we can add (O_1, P_1) to T and delete a middle edge between O_1 and B to get the contradiction.
- Case (ii): if O_2 is the only vertex between O_1 and B, we can argue similarly that there must be a third outgoing vertex O_3 between O_1 and P. The remaining arguments are the same, and we can finally arrive at a contradiction.

With this updated observation of (b), the total number of leaves in T^* is at most

$$4n_i + 3n_\ell + n_{>2} \le 4\ell(T) + 3\ell(T) + \ell(T) = 8\ell(T),$$

which shows that the same algorithm is an $\frac{1}{8}$-approximation.

*This solution is provided by **Prof. Biaoshuai Tao**.* □

4 Assignment IV - Dynamic Programming

4.1 *Problem 1 - Maximum Revenue (30 Points)*†‡§

Given a sequence of integers $a_1, a_2, ..., a_n$, a lower bound and and upper bound $1 \leq L \leq R \leq n$. An (L, R)-step subsequence is a subsequence $a_{i_1}, a_{i_2}, ..., a_{i_\ell}$, such that $\forall 1 \leq j \leq \ell - 1$, $L \leq i_{j+1} - i_j \leq R$. The revenue of the subsequence is $\sum_{j=1}^{\ell} a_{i_j}$. Design a DP algorithm to output the maximum revenue we can get from a (L, R)-step subsequence.

1. (10‡ points) Suppose $L = R = 1$. Design a DP algorithm in $O(n)$ to find the maximum $(1, 1)$-step subsequence.
2. (10‡ points) Design a DP algorithm in $O(n^2)$ to find the maximum (L, R)-step subsequence for any L and R.
3. (10‡ points) Design a DP algorithm in $O(n)$ to find the maximum (L, R)-step subsequence for any L and R.[14]

Solution. **(1) Algorithm**:

Subproblem: Define $mr[i]$ as the maximum (1,1)-step revenue we can get from $a_1, \ldots, a_i$ ended at a_i. The initial state is $mr[1] = a_1$ since subsequence can't be empty. The maximum revenue is exactly $\max_{1 \leq i \leq n} mr[i]$.

State transition: Since L=R=1, the subsequence should be continuous. The maximum revenue ended at a_i has only two options, either give up the subsequence ended at a_{i-1} or take them as a part of the new subsequence. It depends on which one would be larger. The recurrence relation is as follows:

$$mr[i] = max\{mr[i-1] + a_i, a_i\}, \qquad 1 < i \leq n$$

Topological Order: We can solve this problem from $mr[i]$, 1 to n.

Proof of Correctness: We prove the correctness by induction.

- **Base case**: When i=1, the only choice is to take a_1 as the subsequence. Maximum revenue is a_1.
- **Inductive hypothesis**: The calculation of all $mr[x]$ with $x < i$ is correct.
- **Induction**: Consider calculating $mr[i]$. Since we traverse the two choices to construct new subsequence to choose the largest one, and all subproblem has been calculated yet, it should give the largest revenue of subsequence ended at a_i.

[14] Tips: Refer to the "Priority Queue" technique of the k-largest number problem in the lecture.

By the analysis above, all values of $mr[i]$ are correct, so the maximum value of $mr[i]$ for $i = 1, \ldots, n$ is the maximum revenue we can get from a (1,1)-step subsequence. The subsequence corresponds to max $mr[i]$ is subsequence we want.

Time Complexity: There are $O(n)$ states we need to calculate for $mr[i]$. Each calculation of $mr[i]$ takes $O(1)$ for comparison. So the overall time complexity is $O(n)$.

(2) Algorithm:

Subproblem: Define $mr[i]$ as the maximum (L,R)-step revenue we can get from $a_1, \ldots, a_i$ ended at a_i. The initial state is mr[1]=a_1. The maximum revenue is exactly $\max\limits_{1 \le i \le n} mr[i]$.

State transition: Since $L \le i_{j+1} - i_j \le R$, the subsequence ended at a_i can be attached to subsequence ended at $a_k, i - R \le k \le i - L$. It could also start by itself, give up all subsequence before. So we need to compare all the situations of them and pick the largest one. The recurrence relation is as follows:

$$mr[i] = \max_{\substack{i-R \le k \le i-L \\ k>0}} \{mr[k] + a_i,\, a_i\}, \qquad 1 < i \le n$$

Topological Order: Since each subproblem $mr[i]$ can be solved after $mr[i-1]$ is solved, we can solve subproblems from $mr[i]$, 1 to n.

Proof of Correctness: We prove the correctness by induction.

- **Base case**: When $i = 1$, the only choice is to take a_1 as the subsequence. Maximum revenue is a_1.
- **Inductive hypothesis**: The calculation of all $mr[x]$ with $x < i$ is correct.
- **Induction**: Consider calculating $mr[i]$. Since we traverse all the choices to construct new subsequence to choose the largest one, and all subproblem has been calculated yet, it should give the largest revenue of subsequence ended at a_i.

By the analysis above, all values of $mr[i]$ are correct, so the maximum value of $mr[i]$ for $i = 1, \ldots, n$ is the maximum revenue we can get from a (L,R)-step subsequence. The subsequence corresponds to max $mr[i]$ is subsequence we want.

Time Complexity: There are $O(n)$ states we need to calculate for $mr[i]$. Each calculation of $mr[i]$ takes $O(R - L + 1) = O(n)$ for comparisons. So the overall time complexity is $O(n^2)$

(3) Algorithm:

The subproblem, initial state, state transition and topological order are the same as in (2). We just need the optimization process of finding the maximum one of $mr[i - R]$ to $mr[i - L]$ with the deque technique.

We use a deque PLL to store the potential largest $mr[k]$ which is possible for $a_j,\ j > k$ to be attached to. The updating operation is as follows.

After the updating operation each turn, PLL.front is the maximum one we need.

Proof of Correctness: The correctness of dp algorithm is just as that in (2). The correctness we need to state is the technique of queue.

Algorithm 14 Updating Priority Queue

```
function UPDATADEQUE(PLL, i, L, R)
    if PLL.front.index < i − R then
        PopFront(PLL)
    while PLL.back.value ≤ mr[i − L] do
        PopBack(PLL)
    PushBack(PLL, (index = i − L, value = mr[i − L]))
```

claim 1: PM only stores the potential largest that can be the subsequence for a_i to be attached to. The indexes in PLL are in $[i-R, i-L]$.

claim 2: PLL.front is the largest one in $i-R$ to $i-L$. Considering the algorithm, the deque is monotonically decreasing.

claim 3: The updating operation mantains the property of deque. We kick off out of range elements and old&small elements. We only push in element whose $index < i-L$.

By the analysis above, PLL optimize the process of finding max correctly.

Time Complexity: There are $O(n)$ states we need to calculate for $mr[i]$. In each calculation of $mr[i]$, we need to update the deque once and fetch PLL.front. Since each element push and pop once, the time complexity is $O(n)$. So the overall time complexity is $O(n)$.

*This solution is provided by **T.A. Wenqian Wang**.* □

4.2 *Problem 2 - Optimal Indexing for A Dictionary (25 Points)*[†‡§]

Consider a dictionary with n different words $a_1, a_2, ..., a_n$ sorted by the alphabetical order. We have already known the number of search times of each word a_i, which is represented by w_i. Suppose that the dictionary stores all words in a binary search tree T, i.e., each node's word is alphabetically larger than the words stored in its left subtree and smaller than the words stored in its right subtree. Then, to look up a word in the dictionary, we have to do $\ell_i(T)$ comparisons on the binary search tree, where $\ell_i(T)$ is exactly the level of the node that stores a_i (root has level 1). We evaluate the search tree by the total number of comparisons for searching the n words, i.e., $\sum_{i=1}^{n} w_i \ell_i(T)$. Design a DP algorithm to find the best binary search tree for the n words to minimize the total number of comparisons.

Solution. **Algorithm**:

Subproblem: Define $cost[i][j]$ to record the minimum cost of the binary search tree(BST) including $a_i, a_{i+1}, \ldots, a_j$. So we have $cost[i][j] = 0, \quad i > j$ and $cost[i][i] = w_i$. The search tree corresponds to $cost[1][n]$ is best binary search tree we want.

State transition: When considering BST involving $a_i, \ldots, a_j$, since the words are already sorted by alphabetical order, if we choose a_r as the root, $a_i, \ldots, a_{r-1}$ will be the left subtree and $a_{r+1}, \ldots, a_j$ will be the right subtree. Hence, if $cost[i][r-1]$ and $cost[r+1][j]$ has already been calculated, then all nodes in $a_i, \ldots, a_j$ except a_r should have their levels increased by 1 in the new BST rooted at a_r, which means the increase can be easily calculate by summing up their weights. The remaining problem is traverse on r to find one to minimize the cost. The recurrence relation is as follows:

$$cost[i][j] = \min_{i<=r<=j} \{cost[i][r-1] + cost[r+1][j]\}, \qquad i < j$$

Topological Order: Since we solve the problem based on smaller scaled solution, we can solve this problem through increasing order of $(j - i)$.

Proof of Correctness: We prove the correctness by induction.

- **Base case**: When $i = j$, there is only one node in BST. So the word a_i should be the root and the cost is w_i.
- **Inductive hypothesis**: The calculation of all $cost[x][y]$ with $y - x < j - i$ is correct.
- **Induction**: Consider calculating $cost[i][j]$. Since we traverse all possible root to choose one who gives the least cost of BST, and all subproblem has been calculated yet, it should give the optimal BST including nodes $a_i, \ldots, a_j$.

By the analysis above, all values of $cost[i][j]$ are correct, so the value of $cost[1][n]$ is the cost of the optimal BST, and the search tree corresponds to $cost[1][n]$ is best binary search tree we want.

Time Complexity: There are $O(n^2)$ states we need to calculate for $cost[i][j]$. Each calculation of $cost[i][j]$ takes $O(n)$ for traversing all possible r. So the overall time complexity is $O(n^3)$

*This solution is provided by **T.A. Wenqian Wang**.* □

4.3 *Problem 3 - The Longest Palindrome (25 Points)*†‡§

A *palindrome* is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).

Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`. What is the running time of your algorithm?

Solution. **Algorithm**:

Subproblem: Define $arr[i][j]$ to record the length of longest palindrome subsequence of string $a_i a_{i+1} \dots a_j$. Since all strings of length 1 are naturally palindrome, we have $arr[i][i] = 1$ and $arr[i][j] = 0,\ i > j$. The subsequence corresponds to $arr[1][n]$ is longest palindrome we want.

State transition: We just consider the palindrome subsequence of $a_i a_{i+1} \dots a_j$. If $a_i = a_j$, then the longest palindrome subsequence could be constructed by a_i, longest palindrome subsequence of $a_{i+1} \dots a_{j-1}$ and a_j. if $a_i \neq a_j$, then the longest palindrome subsequence could be either the longest palindrome subsequence of $a_i \dots a_{j-1}$ or $a_{i+1} \dots a_j$. The recurrence relation is as follows:

$$arr[i][j] = \begin{cases} arr[i+1][j-1] + 2, & i < j, a_i = a_j \\ \max\{arr[i][j-1], arr[i+1][j]\}, & a_i \neq a_j \end{cases}$$

Actually, since $arr[i][j-1] + 1 \leq arr[i][j]$, $arr[i+1][j] + 1 \leq arr[i][j]$, we don't need to compare $arr[i+1][j-1] + 2$ to other situations when $a_i = a_j$.

Topological Order: Since we solve the problem based on smaller scaled solution, we can solve this problem through increasing order of $(j - i)$.

Proof of Correctness: We prove the correctness by induction.

- **Base case**: When $i = j$, all strings of length 1 are naturally palindrome, so $arr[i][i] = 1,\ 1 \leq i \leq n$.
- **Inductive hypothesis**: The calculation of all $arr[x][y]$ with $y - x < j - i$ is correct.
- **Induction**: Consider calculating $arr[i][j]$. Since we do case by case discussion on whether elements at both ends are equal, traverse all possible situations to choose the longest palindrome subsequence length, and all subproblem has been calculated yet, it should give the length of longest palindrome sequence of $a_i \dots a_j$.

By the analysis above, all values of $arr[i][j]$ are correct, so the value of $arr[1][n]$ is the length of longest palindrome sequence of input string, and the palindrome subsequence corresponding to $arr[1][n]$ is the longest palindrome subsequence we want.

Time Complexity: There are $O(n^2)$ states we need to calculate for $arr[i][j]$. Each calculation of $arr[i][j]$ takes $O(1)$ for traversing all possible situations. So the overall time complexity is $O(n^2)$

*This solution is provided by **T.A. Wenqian Wang**.*[15] □

[15] There is also a clever way of converting this problem to a delete-only edit distance problem. And the target string is the inversion of the source string. – T.A. Zichen Zhu

4.4 *Problem 4 - Independent Sets in A Tree (25 Points)*[†‡§]

Let G be a tree with n vertices. In this problem, we assume that it takes $O(1)$ time to store and multiply two integers.

1. (20[‡] points) Design an $O(n)$ time algorithm to count the number of independent sets in G. Prove the correctness of your algorithm and analyze its time complexity.
2. (Bonus 5[‡] points) Design an efficient algorithm to count the number of *maximum* independent sets in G. Prove the correctness of your algorithm and analyze its time complexity.

Solution. **(1) Algorithm**:

Subproblem: We assume the tree G rooted at r. Define $count[i]$ to record the number of independent sets of the tree rooted at i. Since there are two independent sets construction for a leaf j, $\{j\}$ and $\emptyset$, we have $count[j] = 2$, j is leaf. Therefore, $count[r]$ is the number of independent sets in G we want.

State transition: Considering the $count[i]$ for the subtree rooted at a not leaf node i, we can derive $count[i]$ from i's sons' and grandsons' independent sets number. If we add i into the independent set, then we can't choose i's son into independent set. So the number of independent sets including i is the product of the number of independent sets of subtrees rooted at i's grandsons. If we don't add i into the independent set, then we can choose i's sons into independent set freely. So the number of independent sets excluding i is the product of the number of independent sets of subtrees rooted at i's sons. The recurrence relation is as follows:

$$count[i] = \prod count[i's\ son] + \prod count[i's\ grandson], \qquad \text{(i isn't a leaf.)}$$

Topological Order: Since we can solve $count[i]$ only after $count[i's\ \ son]$ and $count[i's\ \ grandson]$ are solved, a feasible topological order is follow the decreasing order of node's depth.

Proof of Correctness: We prove the correctness by induction.

- **Base case**: When j is leaf, there are two independent sets construction for a leaf j, $\{j\}$ and $\emptyset$, we have $count[j] = 2$.
- **Inductive hypothesis**: The calculation of all $count[i]$ which x's depth>i's depth are correct.
- **Induction**: Consider calculating $count[i]$. Since we do case by case discussion on whether root i is in independent sets and all subproblem has been calculated yet, it should give the number of independent sets in subtree rooted at i.

By the analysis above, all values of $count[i]$ are correct, so the value of $count[r]$ is the number of independent sets in G, which is exactly what we want.

Time Complexity: There are $O(|V|)$ states we need to calculate for $count[i]$. Each node would take part in its parent's and grandparent's calculation, which means the product operation take $O(|V|)$ at all too. So the overall time complexity is $O(|V|)$

(2) Algorithm:

Subproblem: The subproblem here is a little more complex than that in (1). We assume the tree G rooted at r. Define $count[i][0]$ to record the number of maximum independent sets of the tree rooted at i excluding i, $count[i][1]$ to record the number of maximum independent sets of the tree rooted at i including i.

We also need to record the size of the size of maximum independent sets. Define $size[i][0]$ to record the size of maximum independent sets of the tree rooted at i excluding i, $size[i][1]$ to record the size of maximum independent sets of the tree rooted at i including i.

Since there are two independent sets construction for a leaf j, $\{j\}$ and $\varnothing$, we have

$$count[j][0] = count[j][1] = 1,\ size[j][0] = 0,\ size[j][1] = 1, \qquad \text{(j is leaf)}$$

When it comes to the root, if $size[r][0] \neq size[r][1]$, select the count corresponding to larger one. if $size[r][0] \equiv size[r][1]$, $count[r][0] + count[r][1]$ should be the result.

State transition: Considering the subtree rooted at i. Firstly, if we add i into the independent sets, then i's sons shouldn't be in the independent sets, $count[i][1]$ can be counted by production of $count[i'sson][0]$ and $size[i][1]$ can be counted by one plus sum of $size[i'sson][0]$. The complex case is that if we don't add i into the independent sets, then $count[i][0]$ can be counted by production of i's son's number of maximum independent sets no matter whether i's son is added into the independent sets or not. The $size[i][0]$ can be counted by sum of i's son's size of maximum independent sets. The recurrence relation is as follows:

$$count[i][1] = \prod count[i's\ son][0]$$

$$size[i][1] = 1 + \sum size[i's\ son][0]$$

$$count[i][0] = \prod \begin{cases} count[i's\ son][0] + count[i's\ son][1], & size[i's\ son][0] = size[i's\ son][1] \\ count[i's\ son][0], & size[i's\ son][0] > size[i's\ son][1] \\ count[i's\ son][1], & size[i's\ son][0] < size[i's\ son][1] \end{cases}$$

$$size[i][0] = \sum max\{size[i's\ son][0], size[i's\ son][1]\}$$

Topological Order: Since we can solve $count[i]$ and $size[i]$ only after $count[i'sson]$ and $size[i'sson]$ are solved, a feasible topological order is follow the decreasing order of node's depth.

Proof of Correctness: We prove the correctness by induction.

- **Base case**: When j is leaf, there are two independent sets construction for a leaf j, $\{j\}$ and $\emptyset$, we have

$$count[j][0] = count[j][1] = 1,\ size[j][0] = 0,\ size[j][1] = 1, \qquad \text{(j is leaf)}$$

- **Inductive hypothesis**: The calculation of all $count[x][0],\ count[x][1],\ size[x][0],\ size[x][1]$ which x's depth>i's depth are correct.
- **Induction**: Consider calculating $count[i][0],\ count[i][1],\ size[i][0],\ size[i][1]$. Since we do case by case discussion on whether root i is in independent sets and the size of maximum independent sets. What's more, all subproblem has been calculated yet. It should give the number of maximum independent sets in subtree rooted at i including i or excluding i and the corresponding size correctly.

By the analysis above, all values of $count[i][0],\ count[i][1],\ size[i][0],\ size[i][1]$ are correct, we can calculate the maximum number of independent sets in total with these values using the method stated in **Subproblem** part.

Time Complexity: There are $O(|V|)$ states we need to calculate for $count[i][0],\ count[i][1],\ size[i][0],\ size[i][1]$. Each node would take part in its parent's calculation once, which means the product and sum operation take $O(|V|)$ at all too. So the overall time complexity is $O(|V|)$.

*This solution is provided by **T.A. Wenqian Wang**.*

□

Solution. **(1*) Another Solution of (1)**:

Subproblem: Let $f(i)$ be the number of independent sets of the subtree rooted at i in which node i is included, and $g(i)$ be the number in which node i is not included. Obviously $f(r) + g(r)$ is the number of independent sets in G. For leaf nodes, $f(i) = g(i) = 1$. Let $\Omega(i)$ denote the set of i's children.

State transition:

$$f(i) = \prod_{k \in \Omega(i)} g(k),$$
$$g(i) = \prod_{k \in \Omega(i)} (f(k) + g(k)).$$

Topological Order: This algorithm is proved feasible since $f(i), g(i)$ is solved only after $f(k), g(k)$ is solved, where $k \in \Omega(i)$. The topological order is the decreasing order of node's depth.

Proof of Correctness: By induction,

- **Base case**: $f(i) = g(i) = 1$ holds for leaf nodes i, given the similar reasoning as (1).
- **Inductive hypothesis**: The calculation of $f(i), g(i)$ is correct given $f(k), g(k)$ correct for $k \in \Omega(i)$.
- **Induction**: If the independent set contains node i, then it must not contain any of i's children. The number of independent sets in this case, denoted as $f(i)$, equals the product of the number of independent sets in each subtree of i's children k with k excluded. So the first transition equation holds.

  If the independent set does not contain node i, then it can contain any of i's children. In this case, for $k \in \Omega(i)$, both $f(k)$ and $g(k)$ should be counted for each subtree of k. So the number of independent sets equals the product of $f(k) + g(k)$, which exactly implies the second transition equation.

By the analysis above, $f(r) + g(r)$ is the number of independent sets in G.

Time Complexity: Similarly, for every node i, $f(i)$ and $g(i)$ is calculated exactly once. So the overall complexity is $O(|V|)$.

*This solution is provided by **T.A. Lejun Min**.* □

5 Assignment V - Flow

5.1 *Problem 1 - Lower Bounds for Randomized Algorithms (40 Points)*†

Recall the algorithm to find the k-th smallest element in an array with a random pivot. The running time of the algorithm is a random variable, which depends on the pivot we chose in each iteration. Given as input an array of length n, we also learnt that the algorithm uses $\Omega(n^2)$ steps to terminate in the worst case while on average it costs $O(n)$. In this homework, we develop an important tool to establish lower bounds for the expected running time of a best randomized algorithms for a problem.[16]

Let us fix a computational problem Π (e.g. sorting, finding the k-th smallest element, etc). Let $\mathcal{X}$ be the set of inputs of length n for some fixed n and let $\mathcal{A}$ be the set of deterministic algorithms solving Π. We assume for convenience that both $\mathcal{X}$ and $\mathcal{A}$ are finite.

1. Now suppose we are given a randomized algorithm $\overline{A}$ (like the random pivoting algorithm just mentioned, $\overline{A}$ terminates in finite steps while the termination time is random). Prove that we can find a distribution $\mathscr{A}$ over $\mathcal{A}$ so that the expected running time of $\overline{A}$ on the input x equals

$$\mathbf{E}_{A\sim\mathscr{A}}[T(A,x)],$$

   where $T(A,x)$ is the running time of A on the input x. Use this to conclude that the expected running time[17] of the best algorithm for Π (over inputs with length n) is

$$\min_{\substack{\text{distribution}\\ \mathscr{A}\text{ over }\mathcal{A}}} \max_{x\in\mathcal{X}} \mathbf{E}_{A\sim\mathscr{A}}[T(A,x)].$$

2. Use Von Neumann's minimax theorem to prove that[18]

$$\max_{\substack{\text{distribution}\\ \mathscr{X}\text{ over }\mathcal{X}}} \min_{a\in\mathcal{A}} \mathbf{E}_{X\sim\mathscr{X}}[T(a,X)] = \min_{\substack{\text{distribution}\\ \mathscr{A}\text{ over }\mathcal{A}}} \max_{x\in\mathcal{X}} \mathbf{E}_{A\sim\mathscr{A}}[T(A,x)].$$

   This is the famous Yao's minimax principle, which essentially says that in order to lower bound the expected running time of the best algorithm for Π, one only needs

to construct a distribution over all inputs (of length n) and show that every deterministic algorithm performs bad on average when the inputs are drawn from the distribution.

3. Given an array A of n numbers and a number x which is in the array. Your task is to determine the location of x, i.e., to find the index $i \in [n]$ such that $A[i] = x$. Assume probing the value of the array at a specific location costs 1 unit of time (that is, reading the value of $A[j]$ for any j costs 1 unit of time). What is the worst running time for a deterministic algorithm? How to improve it using randomization? What's the expected cost of your randomized algorithm?

4. Prove that each randomized algorithm for the above problem costs at least $\frac{n+1}{2} - \frac{1}{n}$ in expectation in the worst case. Does your algorithm match this lower bound?

Solution. **(1).** For a fixed input x, the performance of $\overline{A}$ is determined by a random seed every time we run it. Practically the performance of a sequence of its running is determined by a sequence of random seeds. Hence, we can assume that the random sequences are hardwired in a table beforehand, and each time we run $\overline{A}$, we can simply look for the corresponding seed in the table. Since $\mathcal{X}$ and $\mathcal{A}$ are both finite, the table is finite, so we can always find finite sets $S = \{A_1, A_2, \ldots, A_m\}$ and $P = \{p_1, p_2, \ldots, p_m\}$, where A_i is a deterministic algorithm with certain hardwired random seeds and p_i is the corresponding possibilities for the seeds. Hence $\sum_i p_i = 1$. Then the expected running time of $\overline{A}$ on input x can be written as

$$\mathbf{E}\left[T(\overline{A}, x)\right] = \sum_i p_i \cdot T(A_i, x).$$

Now the distribution $\mathscr{A}$ can be defined as

$$\mathscr{A} = \{A_i \mid A_i \text{ with probability } p_i, \quad \forall i \in [m]\}.$$

It's obvious that

$$\mathbf{E}_{A\sim\mathscr{A}}\left[T(A, x)\right] = \sum_i p_i \cdot T(A_i, x) = \mathbf{E}\left[T(\overline{A}, x)\right].$$

To find the expected runing time of the best algorithm for Π, firstly it should represented by the worst case of the input distribution, and secondly we need to optimize the input distribution to get the minimum value. Therefore it should be

$$\min_{\substack{\text{distribution} \\ \mathscr{A} \text{ over } \mathcal{A}}} \max_{x\in\mathcal{X}} \mathbf{E}_{A\sim\mathscr{A}}[T(A, x)].$$

[16] Note that *deterministic* algorithms are special cases of randomized algorithms.

[17] The expected running time of the best algorithm for Π is $\min_{\substack{\text{randomized algorithm} \\ \overline{A} \text{ solving } \Pi}} \max_{x\in\mathcal{X}} \mathbf{E}[T(\overline{A}, x)]$ where the randomness is over those in the execution of $\overline{A}$

[18] The LHS is the "expected time of the best deterministic algorithm against the worst input distribution".

(2). We can design a corresponding zero-sum game for this problem. Think of two players Alice and Bob playing a game where Alice decide which deterministic algorithm A to use among $\mathcal{A}$ and Bob decide which input x to choose among $\mathcal{X}$. Given their strategy A, x, The loss of Alice is $T(A, x)$, which is exactly the gain of Bob. By definition, this is a finite zero-sum game with two players. A randomized algorithm $\mathscr{A}$ is, in this case, a randomized choice/strategy of Alice to play against Bob. Bob can be thought of as selecting the input x from a certain distribution $\mathscr{X}$ over $\mathcal{X}$. By von Neumann's minimax theorem, we have

$$\min_{\substack{\text{distribution}\\ \mathscr{A} \text{ over } \mathcal{A}}} \max_{\substack{\text{distribution}\\ \mathscr{X} \text{ over } \mathcal{X}}} \mathbf{E}_{A\sim\mathscr{A}}\mathbf{E}_{X\sim\mathscr{X}}[T(A,X)] = \min_{\substack{\text{distribution}\\ \mathscr{X} \text{ over } \mathcal{X}}} \min_{\substack{\text{distribution}\\ \mathscr{A} \text{ over } \mathcal{A}}} \mathbf{E}_{X\sim\mathscr{X}}\mathbf{E}_{A\sim\mathscr{A}}[T(A,X)].$$

On the other hand, a moment's reflection should convince you that the best choice of Bob in LHS above is the *pure strategy*, namely

$$\min_{\substack{\text{distribution}\\ \mathscr{A} \text{ over } \mathcal{A}}} \max_{\substack{\text{distribution}\\ \mathscr{X} \text{ over } \mathcal{X}}} \mathbf{E}_{A\sim\mathscr{A}}\mathbf{E}_{X\sim\mathscr{X}}[T(A,X)] = \min_{\substack{\text{distribution}\\ \mathscr{A} \text{ over } \mathcal{A}}} \max_{x\in\mathcal{X}} \mathbf{E}_{A\sim\mathscr{A}}[T(A,x)].$$

Similarly in RHS above, we have

$$\min_{\substack{\text{distribution}\\ \mathscr{X} \text{ over } \mathcal{X}}} \min_{\substack{\text{distribution}\\ \mathscr{A} \text{ over } \mathcal{A}}} \mathbf{E}_{X\sim\mathscr{X}}\mathbf{E}_{A\sim\mathscr{A}}[T(A,X)] = \min_{\substack{\text{distribution}\\ \mathscr{X} \text{ over } \mathcal{X}}} \min_{a\in\mathcal{A}} \mathbf{E}_{X\sim\mathscr{X}}[T(a,X)].$$

(3). The worst running time is simply $n-1$ if we scan the array linearly and do not find x in the first $n-1$ cells, then it must be in the last cell.

The randomized algorithm is that we select a location uniformly randomly that has not been probed yet. Let the steps till the algorithm stops is $T(x)$. Then

$$\begin{aligned}
&\mathbf{Pr}\left[T(x)=k\right] = \frac{n-1}{n}\cdot\frac{n-2}{n-1}\cdot\cdots\cdot\frac{1}{n-k+1} = \frac{1}{n}, \quad 0<k<n-1\\
&\mathbf{Pr}\left[T(x)=n-1\right] = 1-(n-2)\cdot\frac{1}{n} = \frac{2}{n}.
\end{aligned}$$

These identifies conclude the proof.

So the expected time is

$$\begin{aligned}
\mathbf{E}\left[T(x)\right] &= \sum_{i=1}^{n-2}\frac{i}{n} + (n-1)\cdot\frac{2}{n}\\
&= \frac{n+1}{2} - \frac{1}{n}.
\end{aligned}$$

(4) From (2) we know that we only need to construct a input distribution to show that every deterministic algorithm performs no better than $\frac{n+1}{2} - \frac{1}{n}$ on it.

Let the input distribution, i.e. the distribution of x's position, be simply a uniformly random distribution over $[n]$. The best deterministic algorithm we can find is the linear scanning algorithm starting from the beginning of the array and stops when the current position contains x. And if we do not find x in the first $n-1$ positions, we stop, because we know that it must be in the last cell. Since x appears in each cell

with equal probability, the expected stopping time is

$$\frac{1}{n}+\frac{2}{n}+\cdots+\frac{n-1}{n}+\frac{n-1}{n}=\frac{n+1}{2}-\frac{1}{n}.$$

which exactly matches the result we get in (3).

*This solution is provided by **TA. Lejun Min** and **Prof. Chihao Zhang**.* □

5.2 *Problem 2 - Perfect Matching and Hall's condition (30 Points)*†

Let $G = (V_1 \cup V_2, E)$ be a bipartite graph where $E \subseteq V_1 \times V_2$. A perfect matching of G is a set of edges $M \subseteq E$ touching each vertex in $V_1 \cup V_2$ exactly once. That is, for every $v \in V_1 \cup V_2$, there exists exactly one edge $e \in M$ such that $v \in e$. Consider the problem of determining whether a given graph contains a perfect matching.

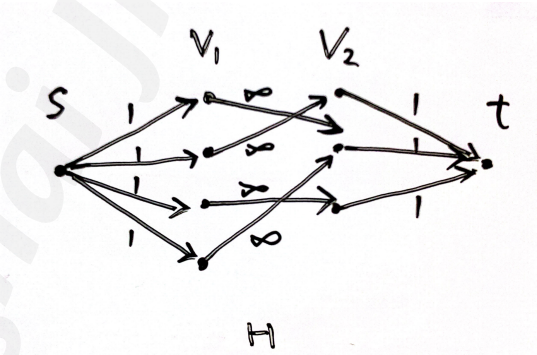
1. Show how to find the perfect matching (if any) by reducing to the problem of maxflow.
2. Hall's condition is a combinatorial characterization of the existence of perfect matching:

Proposition 1 The graph G contains a perfect matching if and only if for any $S \subseteq V_1, |N(S)| \geq |S|$, where $N(S) = \{u \in V_2 \mid \{u, v\} \in E \text{ for some } v \in S\}$ is the set of neighbours of vertices in S.

Prove above proposition.[19]

Solution. **(1).** To reduce to the problem of maxflow, first we need to construct a directed graph H with capacity and then apply maxflow algorithms on this graph.

First duplicate all the vertices in G to H. For every undirected edge $e = (v_1, v_2), v_1 \in V_1, v_2 \in V_2$ in G, construct a directed edge e' from v_1 to v_2 in H. Then let the capacity of the original edges be $+\infty$. Add a source vertex s and add an edge from s to each vertex in V_1 and assign its capacity as 1. Similarly, add a sink vertex t and add an edge from each vertex in V_2 to t and assign its capacity as 1.



We need to prove that a maxflow on H from s to t choose the corresponding edges between V_1 and V_2 that exactly form a maximum matching M of G.

1. M **is a matching**. Since every edge coming from s or ending to t has the capacity of 1, every node in V_1 and V_2 can only be chosen by one edge between V_1 and V_2.

[19] Hint: Maxflow-Mincut Theorem

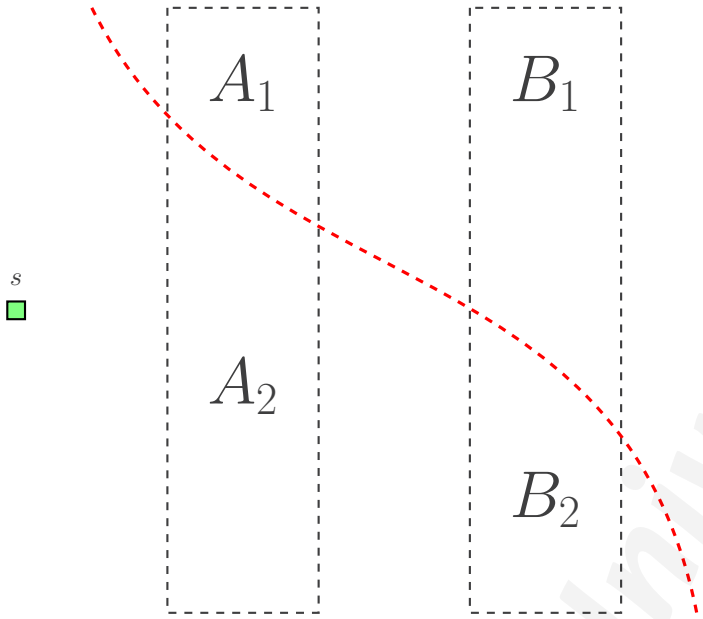


Figure 3: Min-cut of the graph

2. **A flow of value k corresponds to a matching of k edges, and vise versa.** "$\Rightarrow$" is proved by the simple fact that every edge between V_1 and V_2 can only contains a flow of 1. Otherwise it exceed the capacity when flowing into the next edge from one vertex in V_2 to t.

   "$\Leftarrow$" holds since for a matching of k edges, every vertex in V_1, V_2 are touched no more than once. Denote the set of vertice touched in V_1 as W_1 and W_2 in V_2. There can always be 1 unit of flow in edges from S to W_1, and edges from W_2 to t. By connecting these edges we get a flow of value k.

3. M **is the maximum matching.** Suppose that we have a maximum flow f with k edges on H and there is a matching with $> k$ edges, then from 2 we know that there must be a flow with value $> k$, which leads to contradiction.

Finally, if the maxflow on H goes through all the vertices in V_1 and V_2, it is a perfect matching. This is proved by the definition of perfect matching.

(2). The "only if" part is trivial, and we only prove the "if" part; that is, if every $S \subseteq V_1$ satisfies $|N(S)| \geq |S|$, then a perfect matching exists.

Let us compute the max-flow as in (1) and find its min-cut, which is $A_2 \cup B_2$ as shown in Figure 3. Clearly there is no edge between A_2 and B_1, and no edge between A_1 and B_2. The size of the cut is therefore $|A_1| + |B_2|$ as each vertex in A_1 and B_2 are connected to s and t respectively. On the other hand, we know that $|N(A_2)| \leq |B_2|$ since $N(A_2) \subseteq B_2$. Together we have

$$n = |A_1| + |A_2| \leq |A_1| + |N(A_2)| \leq |A_1| + |B_2| = \texttt{size of the mincut}.$$

By the max-flow min-cut theorem, the max-flow is at least n and therefore a perfect matching exists.

*This solution is provided by **TA. Lejun Min** and **Prof. Chihao Zhang**.* □

5.3 *Problem 3 - Person-to-Person Payments (30 Points)*†‡§

You and your $n-1$ roommates are always sharing expenses (bills, groceries, pizza, etc.) but it's terribly inconvenient to split each bill equally. You agree that each bill should be paid by one person (a possibly different one for each bill) who writes down what a subset of the roommates owe him. At the end of the year you aggregate everything in a debt network $G = (V, E, w)$ where $V = \{1, \ldots, n\}$ and $w(u, v)$ is the net (positive) amount that u owes v. If u owes v nothing then $(u, v) \notin E$. Prove that all debts can be settled with at most $n - 1$ person-to-person payments such that if u pays v then $(u, v) \in E$.[20]

Solution. What we need to do is to simplify the debt network G that may contain cycle into a forest without cycle. We design an algorithm to construct such a new network using the similar idea of Ford-Fulkerson.

1. Find a cycle C in G **ignoring the directions of the edges**. If not found, we have contructed a graph H that is cycle-free and contains at most $n - 1$ edges, we can quit the algorithm.
2. Find the edge with smallest absolute weight w in C as (u, v_0). The cycle is denoted as

$$C = (u, v_0), (v_0, v_1)(v_1, v_2), \ldots, (v_n, u).$$

3. Delete (u, v_0), then update all the edges in C following the rules below:
   - If $(v_i, v_{i+1}) \in E$ then $w(v_i, v_{i+1}) \leftarrow w(v_i, v_{i+1} - w)$. Else, $w(v_i, v_{i+1}) \leftarrow w(v_i, v_{i+1}) + w$.
   - If the weight of an edge becomes zero after update, delete this edge.
4. Go to step 1.

The correctness of this algorithm is proved as follows. In step 3, no new edges is introduced and since we choose the edge with smallest absolute weight, after update no edge would have negative value, thus remaining their original direction. Also according to the rules in step 3, the amount of debt of anybody is not changed. In the end, the algorithm can construct a forest without cycle, so there exists at most $n - 1$ edges, which means that we can clear the debt by at most $n - 1$ person-to-person payments.

*This solution is provided by **TA. Lejun Min**.* □

Alternative Solution. First of all, we can assume without loss of generality that G contains no anti-parallel edges. That is, if $(u, v) \in E$, then $(v, u) \notin E$. In particular, if we have $(u, v), (v, u) \in E$, then we can replace both edges by a single edge (u, v) with weight $w'(u, v) = w(u, v) - w(v, u)$ for the case $w(u, v) > w(v, u)$, replace both edges by a single edge (v, u) with weight $w'(v, u) = w(v, u) - w(u, v)$ for the case $w(v, u) > w(u, v)$, and just delete both edges for the case $w(u, v) = w(v, u)$.

Define a *cycle of length* r be a subgraph containing r vertices and r edges such that the r vertices $v_1, v_2, \ldots, v_r$ satisfy

[20] Hint: Prove that the debt network can be equivalently transform to a network with at most $n-1$ edges, using the idea behind Ford-Fulkerson.

- either $(v_i, v_{i+1}) \in E$ or $(v_{i+1}, v_i) \in E$ for each $i = 1, 2, \ldots, r-1$, and
- either $(v_r, v_1) \in E$ or $(v_1, v_r) \in E$,

and the r edges are precisely those r edges above. In other words, a cycle of length r is a cycle of length r in $\widetilde{G}$ where $\widetilde{G}$ is a undirected graph obtained by removing all directions of edges in G. We aim to show that **whenever there is a cycle of length r in G, we can arrange these r payments in an equivalent way such that at most $r-1$ payments are needed**.

Consider a cycle of length r with vertices $v_1, v_2, \ldots, v_r$, and suppose they are oriented in a cycle in the clockwise orientation. For each v_i, v_{i+1}, the edge between them can be either clockwise or counterclockwise. We say that two edges in this cycle have the same orientation if they are both clockwise or counterclockwise, and have the opposite orientation otherwise.

We find an edge with minimum weight in this cycle, and let (v_{i_1}, v_{i_2}) be this edge (where i_1, i_2 differ by 1 unless $\{i_1, i_2\} = \{r, 1\}$, and $i_2 - i_1 = 1$ if this edge has a clockwise orientation, and $i_1 - i_2 = 1$ if this edge has a counterclockwise orientation). We delete this edge, and update the weight of the remaining $r-1$ edges in the cycle such that for each (u, v) of these $r-1$ edges we have

- $w'(u, v) = w(u, v) - w(v_{i_1}, v_{i_2})$ if (u, v) has the same orientation as (v_{i_1}, v_{i_2}), and
- $w'(u, v) = w(u, v) + w(v_{i_1}, v_{i_2})$ if (u, v) has the opposite orientation as (v_{i_1}, v_{i_2}).

In particular, since we have chosen (v_{i_1}, v_{i_2}) with the minimum weight, the updated weights of the remaining $r-1$ edges are non-negative, and we simply remove those edges having 0 weight. By a straightforward analysis, it is easy to see that the payments defined by the new weights in this cycle is equivalent to the original r payments. Therefore, we have proved our claim that at most $r-1$ payments are needed for a cycle of length r.

Finally, we iteratively find a cycle in the graph, and eliminate the found cycle by the method above. The iterative process ends when no cycle remains in the graph, and the payments according to the updated graph is equivalent to the payments according to the original graph, as the payments before and after each update are equivalent. Since a graph with n vertices containing no cycle can have at most $n-1$ edges, we have proved that all debts can be settled with at most $n-1$ payments.

*This solution is provided by **Prof. Biaoshuai Tao**.* □

5.4 *Problem 4 - Network Flow Problems (35 Points)*[‡§]

The network flow problem only restricts the capacity of each edge. Consider the following variant, each edge e does not only have a capacity c_e, but also a demand d_e. Finally, the edge should have flow $d_e \leq f_e \leq c_e$. Please find out how to solve the following problems by reducing them to the original max flow problem.

(a) (20 points) In the original network flow problem, finding a feasible flow is easy. (a zero flow is surely feasible.) However, whether there exists a feasible flow is not straightforward in this new variant. How to determine the existence of a feasible flow by using the original max flow algorithm? ($i.e.$, each f_e should satisfy $d_e \leq f_e \leq c_e$ and the flow conservation constraint should hold at all vertices other than s and t.)

(b) (15 points) Based on the previous part, can we further find a maximum feasible flow? Please also use the original max flow algorithm.

Solution. **(a)** Suppose we are given a graph $G = (V, E)$. First, without loss of generality, we can make the following assumptions: (1) the lower bound of every edge is non-negative (Otherwise, if $d_e < 0$, we can add a reversed edge if $c_e \geq 0$, or reverse edge e if $c_e < 0$); (2) there is at most one edge between two vertices (Otherwise, we can merge the edges between two vertices into a single edge); (3) There do not exist anti-parallel edges between every two vertices.

Then, we will construct a new graph $G' = (V', E')$ in which each edge $e' \in E'$ has the same lower bound 0 and its own capacity defined as follows.

- Add two super vertices s' and t', and they are the new source node and target node of graph G', respectively.
- For each vertex v, create an edge from vertex s' to v, and the capacity of (s', v) is set as $\sum_{u \in V} d_{(u,v)}$; create an edge from vertex v to t', and the capacity of (v, t') is set as $\sum_{u \in V} d_{(v,u)}$.
- For each edge $e = (u, v) \in E$, create a edge (u, v) in G' and set the capacity of (u, v) as $c_e - d_e$.
- Create an edge from the original target vertex t to the original source vertex s with capacity ∞.

Further, we run Fold-Fulkerson algorithm on this new graph G' and get a maximum flow $f : E \to \mathbf{R}^*$ of graph G'. The following lemma implies that we can determine whether G is feasible according to the value of f.

Lemma 1. *The original graph G has a feasible flow if and only if the value of the maximum flow in G' equals to $\sum_{e \in E} d_e$.*

Proof. Sufficiency. If the original graph G has a feasible flow $g : E \to \mathbf{R}$, consider the following flow $g' : E' \to \mathbf{R}^*$ on the new graph G':

$$g'(e) = g(e) - d_e, \forall e \in E$$
$$g'((t,s)) = |g|$$
$$g'((s',v)) = \sum_{u \in V} d_{(u,v)}, \forall v \in V$$
$$g'((v,t')) = \sum_{u \in V} d_{(v,u)}, \forall v \in V$$

Since g is feasible, then $d_e \le g(e) \le c_e, \forall e \in E$. Thus, for each vertex $v \in V'$ (other than s, t),

$$\begin{aligned}\sum_{(u,v)\in E'} g'(v) &= g'((s',v)) + \sum_{(u,v)\in E', u\in V} g'(v) \\ &= \sum_{u\in V} d_{(u,v)} + \sum_{e=(u,v)\in E, u\in V} (g(e) - d_e) \\ &= \sum_{e=(u,v)\in E, u\in V} g(e) = \sum_{e=(v,u)\in E, u\in V} g(e) = \sum_{(v,u)\in E'} g'(v),\end{aligned}$$

which demonstrates g' satisfies flow conservation. Moreover, it is not hard to verify g' satisfy the capacity restriction of G'. Since the sum of the capacity of the edges incident to s' is $\sum_{e\in E} d_e$ and $\sum_{e=(s,u)\in E'} g'(e) = \sum_{e\in E} d_e$.

Necessarity. If the maximum flow of G' equals to $\sum_{e\in E} d_e$, it means each edge incident to s' is saturated. Then we can create a new flow f' on graph G, as follows:

$$f'(e) = f(e) + d_e, \forall e \in E.$$

It is also not hard to verify f' is feasible, and satisfies flow conservation since f satisfies flow conservation and $0 \le f(e) \le c_e - d_e$. □

Based on this lemma, we can check whether the maximum flow of G' equals to $\sum_{e\in E} d_e$ in order to determine the existence of a feasible flow in G.

Time complexity Analysis The total time for constructing the graph G' and running Fold-Fulkerson (Edmonds-Karp) is $O(|E|) + O(|V| \cdot |E|^2) = O(|V| \cdot |E|^2)$.

(b) Based on the algorithm above, we first check whether G has a feasible flow. If G has a feasible flow f, then we can construct a new graph $G' = (V, E')$ as follows. The vertex set of G' is the same as G, but I add some reverse edges for G'. For each edge $e = (u, v) \in E$, create edges $e_1 = (u, v)$ and $e_2 = (v, u)$ for E', and assign weights $c_e - f((u, v))$ to e_1 and $f((u, v)) - d_e$ to e_2.

We then run Edmonds-Karp algorithm on G', and it outputs a maximum flow g. Further we can construct a flow g' based on g, for each edge $e = (u, v) \in E$, assign $g'((u, v)) = f((u, v)) + g((u, v)) - g((v, u))$.

Next, we argue that g' is the maximum feasible flow of graph G:

- Since $0 \le g((u, v)) \le c_{(u,v)} - f((u, v))$ and $0 \le g((v, u)) \le f((u, v)) - d_{(u,v)}$ for each edge $e = (u, v) \in E$, then $d_e \le g'((u, v) \le c_e$. Thus, g' is a feasible flow.

- Since G' can be seen as the residual Network of the original graph G after pushing a flow f. For this reason, the value of the maximum flow of G' is equal to the value of the maximum flow of G plus $|f|$. On the other hand, since the value of g' is just equal to $|g|$ plus $|f|$, we can concludes that g' is indeed the maximum flow of G'.

According to the above explanation, we can find a maximum feasible flow based on the algorithm in (a) within $O(|V| \cdot |E|^2)$ time.

*This solution is provided by **T.A. Jiaxin Song**.* □

5.5 *Problem 5 - König-Egerváry Theorem (35 Points)*[‡§]

In this question, we will prove König-Egerváry Theorem, which states that, in any bipartite graph, the size of the maximum matching equals to the size of the minimum vertex cover. Let $G = (V, E)$ be a bipartite graph.

1. (5 points) Explain that the following is the fractional version of the maximum matching problem (i.e., replacing $x_e \geq 0$ to $x_e \in \{0, 1\}$ gives you the exact description of the maximum matching problem).

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in E} x_e \\ \text{subject to} \quad & \sum_{e:e=(u,v)} x_e \leq 1 && (\forall v \in V) \\ & x_e \geq 0 && (\forall e \in E) \end{aligned}$$

2. (5 points) Write down the dual of the above linear program, and justify that the dual program describes the fractional version of the minimum vertex cover problem.
3. (10 points) Show by induction that the *incident matrix* of a bipartite graph is totally unimodular. (Given an undirected graph $G = (V, E)$, the incident matrix A is a $|V| \times |E|$ zero-one matrix where $a_{ij} = 1$ if and only if the i-th vertex and the j-th edge are incident.)
4. (10 points) Use results in (a), (b) and (c) to prove König-Egerváry Theorem.
5. (5 points) Give a counterexample to show that the claim in König-Egerváry Theorem fails if the graph is not bipartite.

Solution. **(1).** x_e with restriction $x_e \in \{0, 1\}$ indicates whether e is selected in the matching. Thus, $\sum_{e \in E} x_e$ is the number of edges in the matching for which we want to maximize, and $\forall v \in V : \sum_{e:e=(u,v)} x_e \leq 1$ says that at most one edge incident to v can be selected.

(2). The dual program is as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} y_v \\ \text{subject to} \quad & y_u + y_v \geq 1 && (\forall (u, v) \in E) \\ & y_v \geq 0 && (\forall v \in V) \end{aligned}$$

Here, y_v with restriction $y_v \in \{0, 1\}$ indicates whether v is selected in the vertex cover. $\sum_{v \in V} y_v$ is then the size of the vertex cover, and $y_u + y_v \geq 1$ says that one or two of u, v must be selected for each edge (u, v).

(3). For the base step, each 1×1 sub-matrix, which is an entry, is either 1 or 0, which has determinant either 1 or 0.

For the inductive step, suppose the determinant of any $k \times k$ sub-matrix is from $\{-1, 0, 1\}$. Consider any $(k + 1) \times (k + 1)$ sub-matrix T. Since each edge have exactly two endpoints, each column of T can contain at most two 1s.

If there is an all zero-column in T, we know $\det(T) = 0$, and we are done.

If there is a column in T contains only one 1, then $\det(T)$ equals to 1 or -1 multiplies the determinant of a $k \times k$ sub-matrix, which is from $\{-1, 0, 1\}$ by the induction hypothesis. Thus, $\det(T) \in \{-1, 0, 1\}$.

If every column in T contains two 1s, by the nature of bipartite graph, we can partition the rows into the upper half and the lower half such that each column of T contains exactly one 1 in the upper half and one 1 in the lower half. By multiplying 1 to the upper-half rows and -1 to the lower-half rows, and adding all of them, we will get a row of zeros. This means the set of all rows of T are linearly dependent, and $\det(T) = 0$.

We conclude the inductive step.

(4). It is easy to see that the coefficients in the constraints of the primal LP form exactly the incident matrix, and the coefficients in the constraints of the dual LP form the transpose of the incident matrix. Since this matrix is totally unimodular and the values at the right-hand side of the constraints in both linear programs are integers, both linear programs have integral optimal solutions.

In addition, it is straightforward to check that any primal LP solution with $x_e \geq 2$ cannot be feasible and any dual LP solution with $y_u \geq 2$ cannot be optimal. Therefore, there exists an primal LP optimal solution $\{x_e^*\}$ with $x_e^* \in \{0, 1\}$, and there exists an dual LP optimal solution $\{y_v^*\}$ with $y_v^* \in \{0, 1\}$. We have argued in (a) and (b) that both solutions can now represent a maximum matching and a minimum vertex cover respectively. By the LP-duality theorem, the primal LP objective value for the optimal solution $\{x_e^*\}$ equals to the dual LP objective value for the optimal solution $\{y_v^*\}$. This implies König-Egerváry Theorem.

(5). The simplest counterexample would be a triangle, where the maximum matching has size 1 and the minimum vertex cover has size 2.

*This solution is provided by **Prof. Biaoshuai Tao**.* □

6 Assignment VI - NP Complete

6.1 *Problem 1 (33.3 Points)*†‡§

Given two undirected graphs G and H, decide if H is a subgraph of G. Prove that this problem is NP-complete.[21]

Solution. **Difficulty:** $*$

The problem is clearly in NP, as the set of vertices in G that form the subgraph H is a certificate.

To show it is NP-complete, we reduce it from CLIQUE. Given a CLIQUE instance $(G = (V, E), k)$, the instance (G', H') of this problem is constructed as $G' = G$ and H' is a complete graph with k vertices. It is straightforward to check that $(G = (V, E), k)$ is a yes instance if and only if (G', H') is a yes instance.

This solution is provided by ***Prof. Biaoshuai Tao****.* □

[21] Choose *any three* of the following questions. Each question carries $\frac{100}{3}$ points. (You are encouraged to solve as many the remaining questions as possible "in your mind".) The solution is started from the easiest to the hardest.

6.2 *Problem 2 (33.3 Points)*[†‡§]

Given an undirected graph $G = (V, E)$ and an integer k, decide if G has a spanning tree with maximum degree at most k. Prove that this problem is NP-complete.

Solution. **Difficulty:** $*$

The problem is clearly in NP, as the set of edges forming the spanning tree with maximum degree at most k is a certificate.

To show it is NP-complete, we reduce it from HAMILTONIANPATH. Given a HAMILTONIANPATH instance $G = (V, E)$, the instance (G', k) of this problem is constructed as $G' = G$ and $k = 2$.

If $G = (V, E)$ is a yes HAMILTONIANPATH, there is a Hamiltonian path in G, and this is also a spanning tree with maximum degree 2, so (G', k) is a yes instance.

If (G', k) is a yes instance, then a spanning tree with maximum degree 2 must also be a Hamiltonian path, so $G = (V, E)$ is a yes HAMILTONIANPATH instance.

*This solution is provided by **Prof. Biaoshuai Tao**.* □

6.3 *Problem 3 (33.3 Points)*[†‡§]

Given a ground set $U = \{1, \ldots, n\}$, a collection of its subsets $\mathcal{S} = \{S_1, \ldots, S_m\}$, and a positive integer k, the *set cover* problem asks if we can find a subcollection $\mathcal{T} \subseteq \mathcal{S}$ such that $\bigcup_{S \in \mathcal{T}} S = U$ and $|\mathcal{T}| = k$. Prove that set cover is NP-complete.

Solution. **Difficulty:** $*$

The problem is clearly in NP, as the collection $\mathcal{T}$ can be served as a certificate, and verifying whether $\mathcal{T}$ covers all the elements in U can be clearly done in polynomial time.

To show it is NP-complete, we reduce it from VERTEXCOVER. Given a VERTEXCOVER instance $(G = (V, E), k)$, we construct a SETCOVER instance $(U, \mathcal{S}, k')$ as follows. The ground set U contains $|E|$ elements corresponding to the $|E|$ edges in G; $\mathcal{S}$ contains $|V|$ elements corresponding to $|V|$ vertices in G; $k' = k$. In addition, for an edge $e = (u, v)$ in the VERTEXCOVER instance, let the element in U representing e be an element of both the subset in $\mathcal{S}$ representing u and the subset representing v. This complete the description of the construction.

It then follows immediately from our construction that G has a vertex cover of size k if and only if we can find a subcollection $\mathcal{T} \subseteq \mathcal{S}$ with $|\mathcal{T}| = k' = k$ that covers U.

*This solution is provided by **Prof. Biaoshuai Tao**.* □

6.4 *Problem 4 (33.3 Points)*[†‡§]

Given a collection of integers (can be negative), decide if there is a subcollection with sum exactly 0. Prove that this problem is NP-complete.

Solution. The problem is clearly in NP, as the subcollection of integers with sum 0 is a certificate.

To show it is NP-complete, we reduce it from SUBSETSUM+. Given a SUBSETSUM+ instance (S, k), the instance S' of this problem is constructed as $S' = S \cup \{-k\}$.

If (S, k) is a yes SUBSETSUM+ instance, there exists a subcollection T of S with sum k. Then adding $-k$ to T gives a subcollection of S' with sum 0, which means S' is also a yes instance.

If S' is a yes instance, the subcollection T' with sum 0 must contain $-k$, as $-k$ is the only negative number in S'. Then excluding $-k$ from T' gives a subcollection of S with sum k, which means (S, k) is a yes SUBSETSUM+ instance.

This solution is provided by ***Prof. Biaoshuai Tao****.* □

6.5 *Problem 5 (33.3 Points)*[†‡§]

Suppose we want to allocate n items $S = \{1, \ldots, n\}$ to two agents. The two agents may have different values for each item. Let $u_1, u_2, \ldots, u_n$ be agent 1's values for those n items, and $v_1, v_2, \ldots, v_n$ be agent 2's values for those n items. An allocation is a partition (A, B) for S, where A is the set of items allocated to agent 1 and B is the set of items allocated to agent 2. An allocation (A, B) is *envy-free* if, based on each agent's valuation, (s)he believes the set (s)he receives is (weakly) more valuable than the set received by the other agent. Formally, (A, B) is envy-free if

$$\sum_{i \in A} u_i \geq \sum_{j \in B} u_j \qquad \text{agent 1 thinks } A \text{ is more valuable}$$

and

$$\sum_{i \in B} v_i \geq \sum_{j \in A} v_j \qquad \text{agent 2 thinks } B \text{ is more valuable.}$$

Prove that deciding if an envy-free allocation exists is NP-complete.

Solution. **Difficulty:** $**$

This problem is clearly in NP, as the partition (A, B) is a certificate, and envy-freeness can clearly be verified in polynomial time.

To show it is NP-complete, we reduce it from PARTITION+. Given a PARTITION+ instance $S = \{a_1, \ldots, a_n\}$, we construct an instance of this problem with $u_i = v_i = a_i$ for $i = 1, \ldots, n$.

It is then easy to verify that envy-freeness can be rewritten as simply

$$\sum_{i \in A} a_i = \sum_{j \in B} a_j.$$

As a result, the PARTITION+ instance is a yes instance if and only if there exists an envy-free allocation.

*This solution is provided by **Prof. Biaoshuai Tao**.* □

6.6 *Problem 6 (33.3 Points)*[†‡§]

Consider the decision version of *Knapsack*. Given a set of n items with weights $w_1, \ldots, w_n \in \mathbb{Z}^+$ and values $v_1, \ldots, v_n \in \mathbb{Z}^+$, a capacity constraint $C \in \mathbb{Z}^+$, and a positive integer $V \in \mathbb{Z}^+$, decide if there exists a subset of items with total weight at most C and total value at least V. Prove that this decision version of Knapsack is NP-complete.

Solution. **Difficulty:** $**$

KNAPSACK is clearly in NP, as the subset of items satisfying the requirements is a certificate.

To show it is NP-complete, we reduce it from SUBSETSUM+. Given a SUBSETSUM+ instance $(S = \{a_1, \ldots, a_n\}, k)$, we construct a KNAPSACK instance such that $w_i = v_i = a_i$ for each $i = 1, \ldots, n$ and $C = V = k$.

If the SUBSETSUM+ instance is a yes instance, there exists $T \subseteq S$ such that the numbers in T sum up to k. Choosing the subset of items corresponding to T is a valid KNAPSACK solution, as the total weight and the total value are both exactly $C = V = k$. Thus, the KNAPSACK instance is a yes instance.

If the KNAPSACK instance is a yes instance, there exists a subset of items T such that the total value is at least $V = k$ and the total weight is at most $C = k$. Since $a_i = v_i = w_i$, we have

$$\sum_{i \in T} v_i = \sum_{i \in T} a_i \geq k$$

and

$$\sum_{i \in T} w_i = \sum_{i \in T} a_i \leq k,$$

which implies

$$\sum_{i \in T} a_i = k.$$

Thus, the SUBSETSUM+ instance is a yes instance.

*This solution is provided by **Prof. Biaoshuai Tao**.* □

6.7 *Problem 7 (33.3 Points)*[†‡§]

Given an undirected graph $G = (V, E)$ with $n = |V|$, decide if G contains a clique with size exactly $n/2$. Prove that this problem is NP-complete.

Solution. **Difficulty:** $**$

The problem is clearly in NP, as the set of $n/2$ vertices that form a clique is a certificate.

To show that the problem is NP-complete, we reduce it from CLIQUE. Given a CLIQUE instance $(G = (V, E), k)$, we construct the following half-clique instance G':

- If $k < \frac{|V|}{2}$, G' is obtained by adding $|V| - 2k$ extra vertices, connect those extra vertices to each other, and then connect each extra vertex to all vertices in V.
- If $k = \frac{|V|}{2}$, $G' = G$.
- If $k > \frac{|V|}{2}$, G' is obtained by adding $2k - |V|$ extra *isolated* vertices.

It is straightforward to check that G has a k-clique if and only if G' has a $\frac{n}{2}$-clique. The details are omitted here.

*This solution is provided by **Prof. Biaoshuai Tao**.* □

6.8 *Problem 8 (33.3 Points)*[†‡§]

In an undirected graph $G = (V, E)$, each vertex can be colored either black or white. After an initial color configuration, a vertex will become black if all its neighbors are black, and the updates go on and on until no more update is possible. (Notice that once a vertex is black, it will be black forever.) Now, you are given an initial configuration where all vertices are white, and you need to change k vertices from white to black such that all vertices will eventually become black after updates. Prove that it is NP-complete to decide if this is possible.

Solution. **Difficulty:** $**$

The problem is clearly in NP, as the set of vertices whose colors are initially changed to black is a certificate.

To show it is NP-complete, we reduce it from VERTEXCOVER. Here is an important observation. Let S be the subset of vertices whose colors are changed to black initially. We will prove that all vertices will eventually turn to black *if and only if* S is a vertex cover of this graph.

If S is a vertex cover, then for each white vertex u, all its neighbors must be black, or in other words, in S. Otherwise, if a neighbor v of u is not in S, then (u, v) is not covered by S and S cannot be a vertex cover. Given that all the neighbors for each white vertex are black, each white vertex will become black in the next update.

If S is not a vertex cover, then there exists an edge (u, v) that is not covered by S. This means u and v are both white. By our rule of update, u and v can never become black: it is never possible that all neighbors of u are black, nor it is possible for v.

Therefore, the reduction is simple. For the VERTEXCOVER instance (G, k), the instance for this problem is also (G, k). The remaining details are omitted.

This solution is provided by ***Prof. Biaoshuai Tao****.* □

6.9 *Problem 9 (5 Points)*[‡§]

Given an undirected graph $G = (V, E)$, the *3-coloring* problem asks if there is a way to color all the vertices by using three colors, say, red, blue and green, such that every two adjacent vertices have different colors. Prove that 3-coloring is NP-complete.[22]

Solution. **Difficulty:** $***$

3-coloring is clearly in NP, as a color assignment of all vertices is a certificate for a yes instance. To show it is NP-complete, we reduce it from 3-SAT.

Given a 3-SAT instance ϕ, we construct a 3-coloring instance $G = (V, E)$ as follows. We will name the three colors T, F, N which stand for "true", "false", "neutral". The reason for this naming will be apparent soon. Firstly, we construct three vertices named t, f, n and three edges $\{t, f\}, \{f, n\}, \{n, t\}$. It is easy to see these 3 vertices, forming a triangle, must be assigned different colors. Without loss of generality, we assume $c(t) = T, c(f) = F, c(n) = N$, where $c : V \to \{T, F, N\}$ is the color assignment function. We call this triangle the "palette": in the remaining part of the graph, whenever we want to enforce that a vertex cannot be assigned a particular color, we connect it to one of the three vertices in the palette.

For each variable x_i in ϕ, we construct two vertices $x_i, \neg x_i$, and three edges $(x_i, \neg x_i), (x_i, n), (\neg x_i, n)$. This ensures that it must be either $c(x_i) = T, c(\neg x_i) = F$ or $c(x_i) = F, c(\neg x_i) = T$. The former case corresponds to assigning `true` to variable x_i, and the latter case corresponds to assigning `false` to variable x_i.

After we have constructed the gadgets simulating the Boolean assignment for variables, we need to create a gadget to simulate each clause m. We use $m = (x_i \vee \neg x_j \vee x_k)$ as an example to illustrate the reduction. Firstly, we create a vertex m and connect it to the two vertices n, f, so that we can only have $c(m) = T$ (i.e., the clause must be evaluated to `true`). Then, we need to create a gadget that connects the three vertices x_i, $\neg x_j$ and x_k (constructed in the previous step) as "inputs", and connects vertex m at the other end as "output". The gadget must simulate the logical OR operation.

The gadget shown on the left-hand side in Fig. 4 simulates the logical OR operation for two inputs: if both two input vertices are assigned F, then the output vertex cannot be assigned T, for otherwise there is no valid way to assign colors for vertices A and B; if at least one input vertex is assigned T, say, Input 1 is assigned T, then it is possible to assign the output vertex T, since it is always valid to assign F to A and N to B. The gadget for the clause $m = (x_i \vee \neg x_j \vee x_k)$ can then be constructed by applying two OR gadgets, shown on the right-hand side of Fig. 4. We do this for all the clauses.

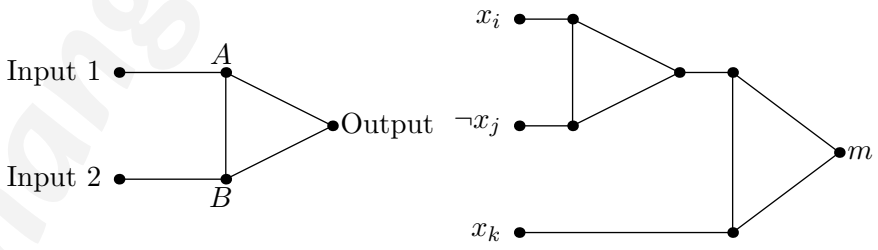


Figure 4: The OR gadget (left-hand side) and the gadget for the clause $m = (x_i \vee \bar{x}_j \vee x_k)$ (right-hand side).

[22] Choose *one* of the following two questions.

The remaining part of the proof is straightforward. We have shown that G simulate assignments to ϕ. Thus, ϕ is satisfiable if and only if there is a valid color assignment to vertices in G. Finally, it is easy to verify that the construction of G can be done in a polynomial time.

*This solution is provided by **Prof. Biaoshuai Tao** from the textbook solution.* □

Alternative Solution.

The proof for 3-coloring is in NP is the same as before. To show it is NP-complete, we introduce an intermediate problem NOTALLEQUAL-3SAT.

Problem 2 (NOTALLEQUAL-3SAT)**.** Given a 3-CNF Boolean formula ϕ, decide if there is an assignment such that each clause contains at least one true literal and one false literal.

Lemma 3. NOTALLEQUAL-3SAT *is* NP*-complete.*

Proof. The problem is clearly in NP, and we will present a reduction from 3SAT to NOTALLEQUAL-3SAT. Given a 3SAT instance ϕ, we construct a NOTALLEQUAL-3SAT instance ϕ' as follows. Firstly, introduce a new Boolean variable w. Then, for each clause $C_j = (\ell_1 \vee \ell_2 \vee \ell_3)$ in ϕ, we introduce two more variables y_j, z_j and use three clauses in ϕ' to represent C_j:

$$(w \vee \ell_1 \vee y_j) \wedge (\neg y_j \vee \ell_2 \vee z_j) \wedge (\neg z_j \vee \ell_3 \vee w). \tag{1}$$

Notice that the variable w is used for all triples of three clauses, while y_j, z_j are only used for the j-th triple.

If ϕ is a yes 3SAT instance, we will show that ϕ' is also a yes instance by showing each of the three clauses in (1) contains a `true` and a `false`. We will set $w =$ `false`. Since ϕ is a yes 3SAT instance, we know at least one of ℓ_1, ℓ_2, ℓ_3 is `true`. If $\ell_1 =$ `true`, we can set $y_j =$ `false` and $z_j =$ `false`. If $\ell_2 =$ `true`, we can set $y_j =$ `true` and $z_j =$ `false`. If $\ell_3 =$ `true`, we can set $y_j =$ `true` and $z_j =$ `true`. In each scenario, it can be check that each of the three clauses in (1) contains a `true` and a `false`. Thus, ϕ' is a yes NOTALLEQUAL-3SAT instance.

If ϕ' is a yes NOTALLEQUAL-3SAT instance, we aim to show that at least one of ℓ_1, ℓ_2, ℓ_3 must be true, which will imply ϕ is a yes 3SAT instance. Firstly, we assume without loss of generality that $w =$ `false`. If $w =$ `true`, we can flip the values for all variables, which will also be a valid assignment. Suppose for the sake of contradiction that $\ell_1 = \ell_2 = \ell_3 =$ `false`. To ensure the first and the third clauses in (1) contain at least one `true`, we need to set $y_j =$ `true` and $z_j =$ `false`. In this case, the second clause does not contain any `true`, which is a contradiction. □

Lemma 4. NOTALLEQUAL-3SAT $\leq_k$ *3-coloring.*

Proof. The main idea here is that a not-all-equal gadget is much easier to construct. The figure on the left-hand side of Fig. 5 demonstrates a not-all-equal gadget.

It is easy to check that, if all the three inputs have the same value, or the same color, the three middle vertices v_1, v_2, v_3 can only choose from two colors, which is impossible. On the other hand, if the three inputs are not all equal, then we can properly choose colors for v_1, v_2, v_3. For example, if Input 1 is T and Input 2 is F, regardless of the value of Input 3, assigning $c(v_1) = F, c(v_2) = T, c(v_3) = N$ is always valid. Thus, this gadget faithfully performs the not-all-equal job.

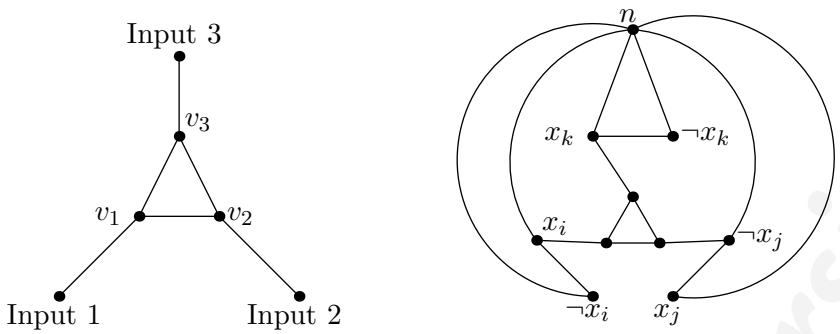


Figure 5: The not-all-equal gadget (left-hand side); the gadget for the clause $m = (x_i \vee \bar{x}_j \vee x_k)$ and how it is connected with other vertices (right-hand side).

Owing to the simplicity of the not-all-equal gadget, we do not need a triangle as a palette as before. All we need is a vertex n that is assumed (without loss of generality) to be colored with N. We connect n to the vertex representing x_i and the vertex representing $\neg x_i$. The figure on the right-hand side of Fig. 5 demonstrates these features.

The remaining parts of the proof are straightforward, and thus omitted.

Lemma 2 and Lemma 3 immediately imply 3-coloring is NP-complete.

*This solution is provided by **T.A. Wenqian Wang**.* □

6.10 *Problem 10 (33.3/5 Points)*[†‡§]

Given a ground set $U = \{1, \ldots, n\}$ and a collection of its subsets $\mathcal{S} = \{S_1, \ldots, S_m\}$, the *exact cover* problem asks if we can find a subcollection $\mathcal{T} \subseteq \mathcal{S}$ such that $\bigcup_{S \in \mathcal{T}} S = U$ and $S_i \cap S_j = \emptyset$ for any $S_i, S_j \in \mathcal{T}$. Prove that exact cover is NP-complete.

Solution. **Difficulty:** $***$

The problem is clearly in NP, as a subcollection of subsets that exactly covers U is a certificate.

To show it is NP-complete, we reduce it from 3-coloring. Given a 3-coloring instance $G = (V, E)$, we construct an exact cover instance as follows.

The ground set U is constructed as follows.

- For each vertex $u \in V$, we construct an element $e_u \in U$.
- For each edge $(u, v) \in E$, we construct three elements $e^r_{uv}, e^g_{uv}, e^b_{uv} \in U$, where the superscripts r, g, b stand for red, green and blue respectively.

The collection $\mathcal{S}$ is constructed as follows:

- For each vertex $u \in V$, we construct three subsets $S^r_u, S^g_u, S^b_u \in \mathcal{S}$ defined as follows.
  - include $e_u \in S^r_u$
  - include $e^r_{uv} \in S^r_u$ for each u's neighbor v.
  - S^g_u and S^b_u are defined similarly.
- For each edge $(u, v) \in E$, construct three subsets $S^r_{uv}, S^g_{uv}, S^b_{uv} \in \mathcal{S}$ such that they contains $e^r_{uv}, e^g_{uv}, e^b_{uv}$ respectively. Notice that each of $S^r_{uv}, S^g_{uv}, S^b_{uv}$ contains only one elements.

For a demonstrating example, suppose the 3-coloring instance is just a triangle with three vertices u, v, w. We have $U = \{e_u, e_v, e_w, e^r_{uv}, e^g_{uv}, e^b_{uv}, e^r_{vw}, e^g_{vw}, e^b_{vw}, e^r_{wu}, e^g_{wu}, e^b_{wu}\}$. The collection $\mathcal{S}$ contains 18 subsets:

$$S^r_u = \{e_u, e^r_{uv}, e^r_{wu}\}, S^g_u = \{e_u, e^g_{uv}, e^g_{wu}\}, S^b_u = \{e_u, e^b_{uv}, e^b_{wu}\}$$

$$S^r_v = \{e_v, e^r_{uv}, e^r_{vw}\}, S^g_v = \{e_v, e^g_{uv}, e^g_{vw}\}, S^b_v = \{e_v, e^b_{uv}, e^b_{vw}\}$$

$$S^r_w = \{e_w, e^r_{vw}, e^r_{wu}\}, S^g_w = \{e_w, e^g_{vw}, e^g_{wu}\}, S^b_w = \{e_w, e^b_{vw}, e^b_{wu}\}$$

$$S^r_{uv} = \{e^r_{uv}\}, S^g_{uv} = \{e^g_{uv}\}, S^b_{uv} = \{e^b_{uv}\}, S^r_{vw} = \{e^r_{vw}\}, S^g_{vw} = \{e^g_{vw}\}, S^b_{vw} = \{e^b_{vw}\}, S^r_{wu} = \{e^r_{wu}\},$$

$$S^g_{wu} = \{e^g_{wu}\}, S^b_{wu} = \{e^b_{wu}\}.$$

This whole construction can clearly be done in polynomial time.

If the 3-coloring instance is a yes instance, let $c : V \to \{r, g, b\}$ be a valid coloring. We further assign a color to an edge (u, v), denoted by $c(u, v)$, such that $c(u, v)$ is different from $c(u)$ and $c(v)$. Given a valid coloring of vertices, each edge is uniquely colored. For example, if $c(u) = r$ and $c(v) = b$, then we have $c(u, v) = g$. To show the exact cover instance we constructed is a yes instance, we consider the following subcollection $\mathcal{T} \subseteq \mathcal{S}$:

- For each vertex u, include $S_u^{c(u)}$ to $\mathcal{T}$;
- For each edge (u, v), include $S_{uv}^{c(u,v)}$ to $\mathcal{T}$.

We will prove that $\mathcal{T}$ is an exact cover.

For each element e_u corresponding to vertex u in G, it is covered by exactly one of $S_u^{c(u)}$. For each element e_{uv}^r corresponding to edge (u, v) in G, it is covered by S_u^r if $c(u) = r$, it is covered by S_v^r if $c(v) = r$, and it is covered by S_{uv}^r if $c(u, v) = r$. Since exactly one of $c(u) = r, c(v) = r, c(u, v) = r$ can happen, e_{uv}^r is covered by exactly one subset. The same holds for e_{uv}^g and e_{uv}^b. We conclude that the exact cover instance we constructed is a yes instance.

Now, suppose the exact cover instance is a yes instance. We will show that the 3-coloring instance is a yes instance. Let $\mathcal{T}$ be a valid solution to the exact cover instance. Then exactly one of S_u^r, S_u^g, S_u^b is included in $\mathcal{T}$, as these are the three subsets that contains e_u. This corresponds to a coloring $c : V \to \{r, g, b\}$: if S_u^r is included, we assign $c(u) = r$; if S_u^g is included, we assign $c(u) = g$; if S_u^b is included, we assign $c(u) = b$. It remains to show that c is a valid 3-coloring. Suppose for the sake of contradiction that there exists $(u, v) \in E$ with $c(u) = c(v)$. Assume $c(u) = c(v) = r$ without loss of generality. Then we have included both S_u^r and S_v^r in $\mathcal{T}$. In this case, the element e_{uv}^r is covered by both S_u^r and S_v^r, which contradicts to that $\mathcal{T}$ is an exact cover.

This solution is provided by ***Prof. Biaoshuai Tao****.* □

Alternative Solution. The proof that exact cover is in NP is the same as before.

To show it is NP-complete, we reduce it from 3SAT. Given a 3SAT instance ϕ, we construct an exact cover instance as follows.

The ground set U is constructed as follows:

- For each variable x_i, we construct an element $e_{x_i} \in U$.
- For each clause $C_j = (\ell_1 \vee \ell_2 \vee \ell_3)$, we construct four elements $e_{C_j}, e_{C_j}^{\ell_1}, e_{C_j}^{\ell_2}, e_{C_j}^{\ell_3} \in U$.

The collection $\mathcal{S}$ is constructed as follows:

- For each variable x_i, construct two subsets $S_{x_i}^T, S_{x_i}^F \in \mathcal{S}$ where
  - $S_{x_i}^T = \{e_{x_i}\} \cup \{e_{C_j}^{x_i} : x_i \text{ is a literal in } C_j\}$;
  - $S_{x_i}^F = \{e_{x_i}\} \cup \{e_{C_j}^{\neg x_i} : \neg x_i \text{ is a literal in } C_j\}$.
- For each clause $C_j = (\ell_1 \vee \ell_2 \vee \ell_3)$, construct seven subsets $S_{C_j}^{TTT}, S_{C_j}^{TTF}, S_{C_j}^{TFT}, S_{C_j}^{FTT}, S_{C_j}^{FFT}, S_{C_j}^{FTF}, S_{C_j}^{TFF} \in \mathcal{S}$ where
  - $S_{C_j}^{TTT} = \{e_{C_j}\}$
  - $S_{C_j}^{TTF} = \{e_{C_j}, e_{C_j}^{\ell_3}\}$
  - $S_{C_j}^{TFT} = \{e_{C_j}, e_{C_j}^{\ell_2}\}$
  - $S_{C_j}^{FTT} = \{e_{C_j}, e_{C_j}^{\ell_1}\}$
  - $S_{C_j}^{FFT} = \{e_{C_j}, e_{C_j}^{\ell_1}, e_{C_j}^{\ell_2}\}$
  - $S_{C_j}^{FTF} = \{e_{C_j}, e_{C_j}^{\ell_1}, e_{C_j}^{\ell_3}\}$

– $S_{C_j}^{TFF} = \{e_{C_j}, e_{C_j}^{\ell_2}, e_{C_j}^{\ell_3}\}$

For a demonstrating example, suppose ϕ consist of only one clause $C_1 = (x_1 \vee \neg x_2 \vee x_3)$. The ground set is $U = \{e_{x_1}, e_{x_2}, e_{x_3}, e_{C_1}, e_{C_1}^{x_1}, e_{C_1}^{\neg x_2}, e_{C_1}^{x_3}\}$. The collection $\mathcal{S}$ consists of the following 13 subsets:

$$S_{x_1}^T = \{e_{x_1}, e_{C_1}^{x_1}\}, S_{x_1}^F = \{e_{x_1}\}, S_{x_2}^T = \{e_{x_2}\}, S_{x_2}^F = \{e_{x_2}, e_{C_1}^{\neg x_2}\}, S_{x_3}^T = \{e_{x_3}, e_{C_3}^{x_3}\}, S_{x_3}^F = \{e_{x_3}\}$$

$$S_{C_1}^{TTT} = \{e_{C_1}\}, S_{C_1}^{TTF} = \{e_{C_1}, e_{C_1}^{x_3}\}, S_{C_1}^{TFT} = \{e_{C_1}, e_{C_1}^{\neg x_2}\}, S_{C_1}^{FTT} = \{e_{C_1}, e_{C_1}^{x_1}\}$$

$$S_{C_1}^{FFT} = \{e_{C_1}, e_{C_1}^{x_1}, e_{C_1}^{\neg x_2}\}, S_{C_1}^{FTF} = \{e_{C_1}, e_{C_1}^{x_1}, e_{C_1}^{x_3}\}, S_{C_1}^{TFF} = \{e_{C_1}, e_{C_1}^{\neg x_2}, e_{C_1}^{x_3}\}$$

The whole construction can clearly be done in polynomial time.

Suppose ϕ is a yes 3SAT instance. We will construct an exact cover $\mathcal{T}$ to show that the exact cover instance is a yes instance. For each x_i, if $x_i = \texttt{true}$, we include $S_{x_i}^T \in \mathcal{T}$; otherwise, include $S_{x_i}^F \in \mathcal{T}$. For each clause C_j, we check the values of the three literals, and include the corresponding subset $S_{C_j}^{XXX}$ in $\mathcal{T}$. For example, for the clause $C_1 = (x_1 \vee \neg x_2 \vee x_3)$, if the assignment is $x_1 = x_2 = x_3 = \texttt{true}$, the first and the third literals are `true` and the second is `false`, and in this case we will include $S_{C_1}^{TFT}$. We will show $\mathcal{T}$ is an exact cover.

Firstly, each "variable element" e_{x_i} is covered exactly once by either $S_{x_i}^T$ or $S_{x_i}^F$. Secondly, each "clause element" e_{C_j} is covered exactly once since we have selected exactly one of those seven $S_{C_j}^{XXX}$. Lastly, for each element $e_{C_j}^{x_i}$, it is covered exactly once: if we have not selected $S_{x_i}^T$, based on our construction of $\mathcal{T}$, it will be covered by the subset $S_{C_j}^{XXX}$ we selected; if we have selected $S_{x_i}^T$, it will not be covered by the subset $S_{C_j}^{XXX}$ we selected. The same analysis holds for each element $e_{C_j}^{\neg x_i}$: it will be covered by either $S_{x_i}^F$ or the subset $S_{C_j}^{XXX}$ we selected, but not both. We conclude that $\mathcal{T}$ is an exact cover.

Now, suppose we have an exact cover $\mathcal{T}$. We will show that ϕ is satisfiable. For each variable x_i, exactly one of $S_{x_i}^T$ and $S_{x_i}^F$ must be selected to cover e_{x_i}. This gives us a natural assignment to $x_1, \ldots, x_n$. We will show that this is a satisfiable assignment. Suppose for the sake of contradiction that there is a clause C_j where the values of all the three literals are `false`. Using the same example $C_1 = (x_1 \vee \neg x_2 \vee x_3)$. Suppose $x_1 = x_3 = \texttt{false}$ and $x_2 = \texttt{true}$. Then we have selected $S_{x_1}^F, S_{x_2}^T, S_{x_3}^F$. The three elements $e_{C_1}^{x_1}, e_{C_1}^{\neg x_2}, e_{C_1}^{x_3}$ are not covered by any of $S_{x_1}^F, S_{x_2}^T, S_{x_3}^F$, so they need to be covered by those $S_{C_1}^{XXX}$. We can only select one of those $S_{C_1}^{XXX}$: if we select more than one of those, the element e_{C_1} will be covered more than once. On the other hand, only one $S_{C_1}^{XXX}$ cannot cover all the three elements $e_{C_1}^{x_1}, e_{C_1}^{\neg x_2}, e_{C_1}^{x_3}$ by our construction (we have not included "$S_{C_1}^{FFF}$" as a subset in $\mathcal{S}$). Therefore, $\mathcal{T}$ cannot be an exact cover, which leads to a contradiction.

*This solution is provided by **T.A. Jiaxin Song**.* □