Homework #1

AI2615: Algorithm Design and Analysis

$*$ If there is any problem, please contact TA Jinyi Wang.

1. (a) $T(n) = 5T(n/4) + n$.
   (b) $T(n) = 7T(n/7) + n$
   (c) $T(n) = 49T(n/25) + n^{3/2}logn$.
   (d) $T(n) = 3T(n-1) + 2$

Solution. Refered from **Haoxuan Xu**

(a) $a = 5, b = 4, d = 1$. Hence, $a > b^d$, by master theorem, we can get $T(n) = \Theta(n^{log_4 5})$

(b) $a = 7, b = 7, d = 1$. Hence, $a = b^d$, by master theorem, we can get $T(n) = \Theta(nlogn)$

(c) $T(n) = O(n^{3/2}logn) \cdot (1 + 49 \cdot (\frac{1}{25})^{3/2} + \ldots 49^k(\frac{1}{25^k})^{3/2} + \ldots 49^{log_{25}n} \cdot (\frac{1}{25^{log_{25}n}})^{3/2})$
$= O(n^{3/2}logn) \cdot (1 + (49/125) + \ldots (49/125)^k + \ldots (49/125)^{(log_{25}n)})$.
Hence, $T(n) = \Theta(n^{3/2}logn)$

(d) $T(n) + 1 = 3(T(n-1) + 1)$, from which we can get $T(n) + 1 = 3^n(T(0) + 1) = \Theta(3^n)$.
Hence, $T(n) = \Theta(3^n)$.

$\square$

2. **A k-way merge operation.** Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements. Design an efficient algorithm using divide-and-conquer (and give the time complexity).

Solution. Referred from **Junqi Xie**.
We can recursively divide the k sorted arrays into 2 parts, and merge the 2 resulting arrays into a single one, just as the merging process in merge sort. In the following pseudo code, Merge is a function for merging 2 sorted arrays, just as in merge sort.

```
1: function MERGE_LIST($L$)
2:     $len \leftarrow$ LEN($L$)
3:     if $len \leq 1$ then
4:         return $L$
5:     else
6:         $l \leftarrow$ MERGE_LIST($L[: len/2]$)
7:         $r \leftarrow$ MERGE_LIST($L[len/2 + 1 :]$)
8:         return MERGE($l, r$)
9:     end if
10: end function
```

Since $T(k) = 2T(k/2) + O(n)$, the overall time complexity

$$T(k, n) = \Theta(nk \log k)$$

□

3. You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values-so there are $2n$ values total-and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the n-th smallest value. However, the only way you can access these values is through queries to the databases. In a singlequery, you can specify a value k to one of the two databases, and the chosen databasewill return the $k-th$ smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible. Give an algorithmthat find the median value using $O(logn)$ queries.

Solution. Referred from **Haoxuan Xu**
We can consider two databases as two sorted arrays a_1, a_2(index starts from 0), and through each query we can input k and get $a_1[k-1]/a_2[k-1]$.

Algorithm 1: Median value search algorithm

```
Input: 2 sorted arrays $a_1, a_2$, each with $n$ elements.
 1: Initialization: $l_1, l_2 \leftarrow 0, r_1, r_2 \leftarrow n-1$.
 2: repeat
 3:     $m_1 \leftarrow (l_1+r_1)/2, m_2 \leftarrow (l_2+r_2)/2, evenFlag \leftarrow ((r_1-l_1)\%2=0)$.
 4:     Inputing $m_1+1, m_2+1$, get $a_1[m_1], a_2[m_2]$ through two queries.
 5:     if $a_1[m_1]=a_2[m_2]$ then
 6:         return $a_1[m_1]$.
 7:     else if $a_1[m_1]>a_2[m_2]$ then
 8:         $r_1 \leftarrow m_1, l_2 \leftarrow m_2+evenFlag$.
 9:     else
10:         $l_1 \leftarrow m_1+evenFlag, r_2 \leftarrow m_2$.
11:     end if
12: until $l_1=r_1$.
13: $m \leftarrow \min(a_1[l_1], a_2[l_2])$.
14: return $m$.
Output: The median $m$.
```

Correctness. If the median of the two arrays is equal, then it is also clearly the median of the two arrays and can be returned directly. If not, each recursion will sieve the smaller quarter and the larger quarter of the input range, narrowing the possible range to the middle half, and finally it will definitely come to the base case, where both arrays have only one number left, and the smaller one is the median value of the two arrays.

Complexity Since $T(n)=T(n/2)+O(1)$, the overall time complexity

$$T(n)=O(\log n)$$

□

4. Show that the QuickSort algorithm runs in $O(n^c)$ time on average for some constant $c < 2$ if the pivot is chosen randomly.

   Remark: The algorithm actually runs in $O(nlogn)$ time on average, but for this question it suffices to give an analysis better than $O(n^2)$.

   Proof. Referred from **Yuhao Zhang**
   In each round $w.p.\frac{1}{3}$, the pivot is greater than 1/3 and less than 2/3 of the n numbers (good case). In this case,it takes $O(n)$ time to process , and both the left and right sides have at most $\frac{2n}{3}$ numbers. It indicates $T(n) \le 2T(\frac{2n}{3}) + O(n)$.
   And $w.p.\frac{2}{3}$, the pivot is less than 1/3 or greater than 2/3 of the n numbers (bad case). In this case, it takes $O(n)$ time to process, and. It takes at most $T(n)$ time in the following steps. So it means that:

$$\begin{aligned}
E[T(n)] &\le \frac{1}{3}(2T(\frac{2n}{3}) + O(n)) + \frac{2}{3}(T(n) + O(n)) \\
E[T(n)] &\le \frac{1}{3}(2E[T(\frac{2n}{3})] + O(n)) + \frac{2}{3}(E[T(n)] + O(n)) \\
\frac{1}{3}E[T(n)] &\le \frac{2}{3}E[T(\frac{2n}{3})] + O(n) \\
E[T(n)] &\le 2E[T(\frac{2n}{3})] + O(n)\ .
\end{aligned}$$

   Then by Master theorem, $a = 2, b = \frac{3}{2}, d = 1, d < \log_b a < 1.71, E[T(n)] = O(n^{1.71})$. So we find a constant $c = 1.71 < 2$.
   Note: You can also consider $(2/5, 3/5)$ (or others) as a partition

   □

5. Given an nm 2-dimensional integer array $A[0,\dots n-1; 0,\dots m-1]$ where $A[i,j]$ denotes the cell at the i-th row and the j-th column, a local minimum is a cell $A[i,j]$ such that $A[i,j]$ is smaller than each of its four adjacent cells $A[i-1,j]$, $A[i+1,j]$, $A[i,j-1]$,$A[i,j+1]$. Notice that $A[i,j]$ only has three adjacent cells if it is on the boundary, and it only has two adjacent cells if it is at the corner. Assume all the cells have distinct values. Your objective is to find one local minimum (i.e., you do not haveto find all of them).

   (a) Suppose $m = 1$ so A is a 1-dimensional array. Design a divide-and-conquer-based algorithm for the problem above. Write a recurrence relation of the algorithm, and analyze its running time.

   (b) Suppose $m = n$. Design a divide-and-conquer-based algorithm for the problem above. Write a recurrence relation of the algorithm, and analyze its running time.

   (c) Generalize your algorithm such that it works for generalmandn. The running time of your algorithm should smoothly interpolate between the running times for the first two parts.

Solution. Referred from **Biaoshuai Tao**
(a) Consider the altitudes at $A[n/2]$ and $A[n/2+1]$. It is not hard to observe that if $A[n/2] > A[n/2+1]$, then there exists a water pool in $A[n/2+1,\dots,n]$; if $A[n/2] < A[n/2+1]$, then there exists a water pool in $A[1,\dots,n/2]$. Therefore, by looking for the altitudes at these two positions, we can reduce our searching space by half. Similarly, if we recursively look for the two middle positions, we can always reduce the searching space by half. This gives us the recursive Algorithm 1, where FindPool(Start, End) finds a pool between position "Start" and "End", and computing FindPool$(1, n)$ gives us the result.

Algorithm 2: FindPool(Start, End)

```
1: If End=Start, output Start or End and terminate
2: $m = \lceil 0.5(\text{Start} + \text{End})\rceil$
3: If $A[m] > A[m+1]$, FindPool($m+1$, End)
4: If $A[m] < A[m+1]$, FindPool(Start, $m$)
```

It is straightforward that the recurrence relation of time complexity is

$$T(n) = T(n/2) + c,$$

and master theorem gives us $T(n) = O(\log n)$.

(b) We describe the first few steps to illustrate the algorithm. Consider the $(n/2)$-th column $A[n/2, 0], A[n/2, 1], \dots, A[n/2, n-1]$. We first find the position in this column with the minimum altitude, say, at $A[n/2, 3]$. If the left and right adjacent positions $A[n/2-1, 3]$ and $A[n/2+1, 3]$ both have altitudes larger than $A[n/2, 3]$, then we have already found a pool $A[n/2, 3]$. If at least one of the two adjacent positions has smaller altitude, say, $A[n/2-1, 3]$, then we know there must exist a pool on the left half of the map. This is because the $(n/2)$-th column in this case serves as a "wall", for which the flow $A[n/2, 3] \to A[n/2-1, 3]$ can never pass through. Therefore, we have reduce the searching space by half of size.

In the second step, we are going to find the minimum altitude in the $(n/2)$-th row on the left half of the map, which is the one in $A[0, n/2], A[1, n/2], \dots, A[n/2, n/2]$. Say, the minimum is $A[5, n/2]$. Again, if both the upper and lower positions of $A[5, n/2]$ have larger altitudes, we have already found a pool. If not, we need to compare the altitude of this minimum $A[5, n/2]$ to the previous one we found, namely $A[n/2, 3]$. If the altitude at $A[5, n/2]$ is larger, then the

$(n/2)$-th row serves as a wall to stop the flow from $A[n/2, 3]$ from passing through, and we know there is a pool at the quarter of the map $A[0, 1, \ldots, n/2; 0, 1, \ldots, n/2]$. If the altitude at $A[n/2, 3]$ is larger, then the $(n/2)$-th column serves as a wall instead, and the flow from $A[5, n/2]$ cannot passing through the $(n/2)$-th column. We can then decide which of the upper and lower quarters of the left half map to be kept in the next step recursion by exam which of $A[5, n/2 + 1]$ and $A[5, n/2 - 1]$ has a altitude larger than $A[5, n/2]$. (If both are larger, we can choose either quarter.) In all the situation, the searching space is further reduced by half.

We can continue this process of "cutting" the searching space by half each time, and the "cuts" are made vertically and horizontally in alternation. Notice that we need to record the position of the current lowest position on the "wall" (as the start of flow), and its altitude, such that whenever a new "cut" is made, we keep the position with the smaller altitude as the start of flow. The process stops when we find a pool, either in the case when the lowest position on the new "wall" is itself a pool, or in the case when there is only one position left in the searching space.

By our construction, we can always be sure that there exists a pool in the searching space which is reduced by half at each step. So there is no doubt that we can finally find a pool. As for the time complexity, since in each step we need to search for a minimum at a new "cut", whose size is reduced by half in every two steps, we have

$$\begin{aligned}
T(n) &= n + T(n/2) && \text{(The first vertical "cut")}\\
&= (n + n/2) + T(n/4) && \text{(The second horizontal "cut")}\\
&= (n + n/2 + n/2) + T(n/8) && \text{(The third vertical "cut")}\\
&= (n + n/2 + n/2 + n/4) + T(n/16) && \text{(The forth horizontal "cut")}\\
&\vdots\\
&= n + n/2 + n/2 + n/4 + n/4 + n/8 + n/8 + \cdots + 1\\
&= O(n).
\end{aligned}$$

(c) For $m \neq n$, we apply the similar technique in (b) by cutting the searching space by half at each step. There are three ways to cut: 1) always cut vertically; 2) always cut horizontally; 3) cut alternatively (like in (b)). By a similar analysis, the time complexity for these three cutting scheme are respectively

$$T_{\text{vertical}} = O(m \log n) \qquad T_{\text{horizontal}} = O(n \log m) \qquad T_{\text{alternative}} = O(m + n).$$

Given value m and n, we can decide which of $m \log n$, $n \log m$ and $m + n$ is smaller, and then perform the corresponding scheme. Thus, the algorithm first makes the decision by comparison, and then perform the cutting scheme based on the decision. Let $f(n) = \min(m \log n, n \log m, m + n)$, we have the time complexity here is $O(f(n))$. We can see that the running times here smoothly interpolate between the running times of (a) and (b), as taking $m = 1$ we have $O(f(n)) = O(\log n)$ as it is in (a), and taking $m = n$, we have $O(f(n)) = O(m + n) = O(n)$ as it is in (b).

□

Assignment 2 Answer

Course: *AI2615 - Algorithm Design and Analysis*
Due date: *Nov 15th, 2021*

Problem 1

Here is a proposal to find the length of the shortest cycle in an unweighted undirected graph: DFS the graph, when there is a back edge (v, u), it forms a *cycle* from u to v, and the length is $level[v] - level[u] + 1$, where the level of a vertex is its distance in the DFS tree from the root. This suggests the following algorithm:

- Do a DFS and keep tracking the level.
- Each time we find a back edge, compute the cycle length, and update the smallest length.

Please justify the correctness of the algorithm, prove it or provide a counterexample.

Answer Referred by Huangji Wang This algorithm is not perfectly true. Counterexample:

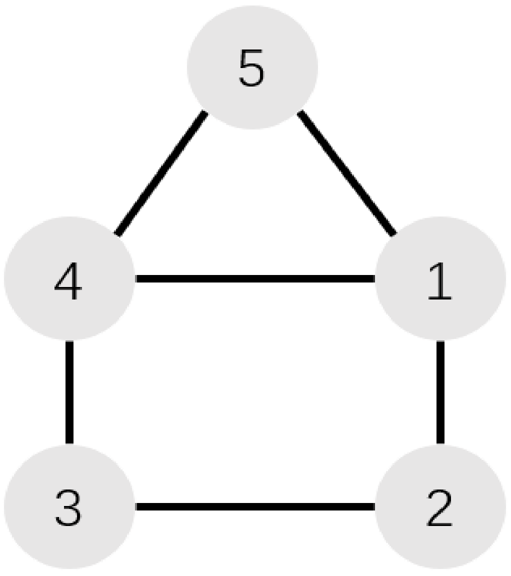


Figure 1: Counterexample

We can always find one path like this picture above. If we do the DFS from the initial vertex $v(1)$, we will form the path as:

- initialize ans = $+\infty$

- (1)
- (1) -> (2)
- (1) -> (2) -> (3)
- (1) -> (2) -> (3) -> (4)
- (1) -> (2) -> (3) -> (4) -> (1) **ans = min(ans, level[4] - level[1] + 1) = 4**
- (1) -> (2) -> (3) -> (4)
- (1) -> (2) -> (3) -> (4) -> (5)
- (1) -> (2) -> (3) -> (4) -> (5) -> (1) **ans = min(ans, level[5] - level[1] + 1) = 4**
- (1) -> (2) -> (3) -> (4) -> (5)
- (1) -> (2) -> (3) -> (4)
- (1) -> (2) -> (3)
- (1) -> (2)
- (1)
- **ans = 4**

However, if we look at the graph, we can easily find that the smallest cycle is $v(5)- > v(1)- > v(4)$ and $ans = 3$. Thus, the algorithm may miss some important cycles, leading to the wrong answer.

My suggestion on how to improve this algorithm is to do DFS without visited arrays. Though it will cost much time than we think, it can find every cycle in the graph and it cannot miss any cycle.

The best way to handle with this problem is to use **union-find disjoint sets** which is very useful to find cycles in (unweighted) undirected graph.

Problem 2

Given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has a weight $p(u, v)$ in range $[0, 1]$, that represents the reliability. We can view each edge as a channel, and $p(u, v)$ is the probability that the channel from u to v will not fail. We assume all these probabilities are independent. Give an efficient algorithm to find the most reliable path from two given vertices s and t. Hint: it makes a path failed if any channel on the path fails, and we want to find a path with minimized failure probability.

Answer Referred by YiChen Tao. The reliability of a path $v_1 - v_2 - \ldots - v_k$ can be expressed as $\prod_{i=2}^{k} p(v_{i-1}, v_i) = \exp[\sum_{i=2}^{k}(\ln p(v_{i-1}, v_i))]$. Hence, to maximize the reliability of a path is to maximize the value of $\sum_{i=2}^{k}(\ln p(v_{i-1}, v_i))$, which is equivalent to minimizing $\sum_{i=2}^{k}(-\ln p(v_{i-1}, v_i))$. So the problem converts into a shortest path problem, where we take $-\ln p(u, v)$ as the weight of the edge (u, v). Since $p(u, v) \in [0, 1]$ holds for all u, v, the edge weight $-\ln p(u, v)$ is always positive, so the problem can be addressed using the Dijkstra algorithm.

Here is a brief explanation of the algorithm:

1. Initialize
   - $w(u, v) = -\ln p(u, v)$ for all (u, v).
   - $T = \{s\}$.
   - $tdist[s] \leftarrow 0$, $tdist[v] \leftarrow \infty$ for all v other than s.
   - $pred[s] \leftarrow -1$, $pred[v] \leftarrow 0$ for all v other than s.
   - $tdist[v] \leftarrow w(s, u)$, $pred[v] \leftarrow s$ for all $(s, v) \in E$.
2. Explore and Update
   - Find $v \notin T$ with smallest $tdist[v]$.
   - $T \leftarrow T + \{v\}$
   - For all $(v, u) \in E$, if $tdist[v] + w(v, u) < tdest[u]$,then $tdist[u] \to tdist[v] + w(v, u)$ & $pred[u] \to v$.
   - Repeat the steps above until t is added to T.
3. Result: Form the path from s to t by backtracking the *pred* array.

Answer Referred by Hanmo Zheng. Since weight $p(u, v)$ in range [0,1] represents the probability that the channel will not fail. The higher $p(u, v)$ is, the lower is the probability path from u to v failed. Thus, we need to find the way from s to t with the highest $p(s, t)$, $(p(s, t) = p(s, a_1) * p(a_1, a_2) * \cdots * p(a_k, t)$, where $s, a_1, a_2, \cdots, a_k, t$ is the way from s to t). Then we design an algorithm with reference to Dijkstra Algorithm as follows:

1. **Initialize**
   - $T = \{s\},$
   - $tprob[s] = 1, tprob[v] \leftarrow 0$ for all v other than s
   - $tprob[v] \leftarrow w(s, v)$ for all $(s, v) \in E$
2. **Explore**
   - Find $v \notin T$ with biggest $tprob[v]$
   - $T \leftarrow T + \{v\}$
3. Update tprob[u]
   - $tprob[u] = max\{tprob[u], tprob[v] * w(v, u)\}$ for all $(v, u) \in E$

Now we show by the steps above, we can get the path with the minimized failure probability. Firstly, since there is a path from s to t and the algorithm will explore all the nodes connected from s, thus $t \in T$ in the end. Besides, if there is another path from s to v and it's failure probability is less than the path we get.

Suppose the way we find is $s, a_1, a_2, \cdots, a_k, t$, and the other path is $s, b_1, b_2, \cdots, b_l, t$,

$$p(s,a_1) * p(a_1,a_2) * \cdots * p(a_k,t) < p(s,b_1) * p(b_1,b_2) * \cdots * p(b_l,t) \tag{1}$$

. However, by the algorithm we explore the node which hasn't been explored with the biggest tprob. Then since p in range [0,1], we have

$$\begin{aligned} p(s,b_1) * p(b_1,b_2) * \cdots * p(b_l,t) &\le p(s,b_1) * p(b_1,b_2) * \cdots * p(b_{l-1},b_l) \\ p(s,b_1) * p(b_1,b_2) * \cdots * p(b_{l-1},b_l) &\le p(s,b_1) * p(b_1,b_2) * \cdots * p(b_{l-2},b_{l-1}) \\ &\cdots \\ p(s,b_1) &\le p(s,b_1) * p(b_1,b_2) \end{aligned} \tag{2}$$

After $a_1, a_2, \cdots, a_k$ has been explored, suppose i is the smallest number that $b_1, b_2, \cdots, b_{i-1}$ has been explored but b_i isn't. Then by inequality (2) and the update method, we can see

$$\begin{aligned} tprob[t] &\le p(s,a_1) * p(a_1,a_2) * \cdots * p(a_k,t) \\ &< p(s,b_1) * p(b_1,b_2) * \cdots * p(b_l,t) \\ &< p(s,b_1) * p(b_1,b_2) * \cdots * p(b_{i-1},b_i) \le tprob[b_i] \end{aligned} \tag{3}$$

, then we will explore b_i before explore t by the algorithm. Thus $b_1, b_2, \cdots, b_l$ will all been explored before we explore t. After b_l was explored, tprob[t] will be update by the update method and inequality (1). And it is updated by the latter path, which leads to a contradiction. Thus the algorithm above is true.

Problem 3

We have a connected undirected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain a T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same tree T. Prove that $G = T$. (In other words, if T is both a DFS tree and a BFS tree rooted at u, then G cannot contain any edges that do not belong to T).

Answer Referred by Yichen Tao. We prove the conclusion by two steps:

1. G does not contain any cycles, *i.e.* G is a tree rooted at u.
2. $G = T$.

G does not contain any cycles. We prove this by contradiction. Assume that there is a cycle $v_1 - v_2 - \ldots - v_k$ in the graph G. Denote the DFS tree by T_D and the BFS tree by T_B.

Suppose that the first node of the cycle to be visited when we DFS the graph is v_f. Then all the nodes in the cycle will be visited when we explore v_f, indicating that $v_1, v_2, \ldots, v_k$ will be on the same path from the root to a leaf.

However, for BFS tree, the case is different. Again we suppose the first node visited in the cycle is v_f, then v_{f-1} and v_{f+1} will be on the same level of the BFS tree. So the DFS tree and BFS tree cannot be the same, which contradicts to the given condition, and the assumption does not hold.

G=T. Now that we already have that G is a tree, the conclusion is rather obvious. If G contains an edge that does not belong to T, in other words, T has at least one edge less than G, then T cannot be a connected graph. So it's impossible that G has any edges that do not belong to T, and our conclusion is proved.

Problem 4

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

$$r(c) = \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}} \tag{4}$$

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.

(a) If we guess a ratio r, can we determine whether $r^* > r$ or $r^* < r$ efficiently?

(b) Based on the guessing approach, given a deisred 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 and $R = \max_{(u,v)\in E}(p_u/c_{uv})$.

Answer Referred by HaoXuan Xu.

(a) First, we are going to assign weight to each edge in E when given the guess ratio r, which will be helpful for out later analysis.

$w(u,v) = r \cdot c_{uv} - p_v$.

Then, we will prove two lemmas.

Lemma 1. If there is a cycle of negative weight, then $r < r^*$.

Proof. Suppose the cycle of negative weight is C, then we have $\sum_{(u,v)\in C} r \cdot c_{uv} - p_v < 0$, which implies $r < \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}}$. Hence, r is smaller than the profit-to-cost ratio of a cycle, which must be no greater than the maximum, and we get $r < r^*$.

Lemma 2. If all cycles in the graph have strictly positive weight then $r > r^*$.

Proof. As all cycles have strictly positive weight, then for all cycles $C \in G$, $\sum_{(u,v)\in C} r \cdot c_{uv} - p_v > 0$, which implies $r > \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}}$ holds for all cycles, including the cycle with maximum profit-to-cost ratio. Hence, $r > r^*$.

With two lemmas above, we can detect whether the graph has a negative cycle to to determine whether $r < r^*$. If there is, then $r < r^*$. After, we have $r \geq r^*$, and we can detect whether the graph has a zero cycle to determine whether $r > r^*$. If there is, then r cannot be strictly greater than r^*, which means $r = r^*$, if there isn't, $r > r^*$.

And now, all we need to do is implement the algorithm to find if there is a negative cycle or zero cycle in a given graph.

The first task can be done with Bellman-Ford algorithm, which has been introduced in class, just run the algorithm for $|V|$ rounds, find that if there is still a vertex updating, if so, there is a negative. And the time complexity is $O(|V||E|)$. We will not show the algorithm here as it is clearly explained in slides. After running the Bellman-Ford algorithm, we can get that if there is a negative cycle and the shortest distance from s if not conerning the negative cycles, which is useful to find that if there is a zero cycle.

Then we introduce the following algorithm to dectect zero cycles. And we will prove another lemma to prove its correctness.

Algorithm 1 Zero cycle detection algorithm

```
Input: $G = (V, E)$, $s$, shortest distance array $dist$
 1: Construct a new graph $G^0 = (V^0, E^0)$, $V^0 = V$, $E^0 = \emptyset$.
 2: for all $(u, v) \in E$ do
 3:     if $dist[v] = dist[u] + w(u, v)$ then
 4:         Insert $(u, v)$ to $E^0$.
 5:     end if
 6: end for
 7: if cycle $C$ identified in $G^0$ then
 8:     return True and $C$
 9: else
10:     return False
11: end if
Output: True and zero cycle $C$ if it exists in $G$, False if there is no zero cycle in $G$.
```

In the above algorithm, we can use DFS to determine if there is a cycle in G^0, and the time complexity of DFS is $O(|V| + |E|)$, the time complexity of G^0 construction is $O(|E|)$, so the time complexity for the zero cycle detection algorithm is $O(|V| + |E|)$.

Correctness of it comes from the following lemma.

Lemma 3. If G doesn't have negative cycle, and G^0 has a cycle then G must has a zero cycle.

Proof. Obviously, if there is a cycle C in G^0, then it must also exist in G. And because G doesn't have negative cycle, C can be zero or positive. And now, we'll prove that if C is positive in G, it will not exist in G^0.

For the cycle C, $\sum_{(u,v)\in C} w(u, v) = \sum_{(u,v)\in C} dist[u] + w(u, v) - dist[v] > 0$. As the sum is positive, there must be one edge (u, v) such that $dist[u] + w(u, v) - dist[v] > 0$, which will not be inserted to E^0. Hence the positive cycle will not appear in G^0. So, if there is a cycle C in G^0, it must be zero cycle in G.

Therefore, we can detect whether a cycle exists in G^0 to determine whether a zero cycle exists in G.

In conclusion, we can use the negative-cycle-detect algorithm and zero-cycle-detect algorithm to compare r and r^*.

(b) With the above method to compare r and r^* and given the initial range $(0, R)$, we can use binary search to continuously compress the range of r^*.

Algorithm 2 Max profit-to-cost ratio found algorithm

```
Input: $G = (V, E)$, desired accuracy $\epsilon$, initial upper bound $R$
 1: $lb \leftarrow 0, rb \leftarrow R, r \leftarrow \frac{lb+rb}{2}$.
 2: repeat
 3:   $tempr \leftarrow r$.
 4:   for $(u, v) \in E$ do
 5:     $w(u, v) \leftarrow tempr \cdot c_{uv} - p_v$.
 6:   end for
 7:   if NegativeCycleDetect(G, W) then
 8:     $lb \leftarrow tempr$.
 9:   else
10:     if ZeroCycleDetect(G, W) then
11:       return tempr and zero cycle $C$.
12:     else
13:       $rb \leftarrow tempr$.
14:     end if
15:   end if
16:   $r \leftarrow \frac{lb+rb}{2}$.
17: until $|tempr - r| > \epsilon$
Output: $r^*$ and its corresponding cycle $C$.
```

Correctness. Obviously, $r^* \leq max_{(u,v)\in E}(p_u/c_{uv}$, hence the initial upperbound is larger enough. And we have proved the correctness of two algorithm to compare r and r^*. So the binary search algorithm correctness is obvious, it will give us r^* with acceptable accuracy.

Time complexity ayalysis. We do $log\frac{R}{\epsilon}$ iterations at most. The time complexity of each iteration is $O(|V||E|) + O(|E|) = O(|V||E|)$. So, the total time complexity is $O(log\frac{max_{(u,v)\in E}(p_u/c_{uv})}{\epsilon} \cdot |V||E|)$.

Problem 5

Consider if we want to run Dijkstra on a bounded weight graph $G = (V, E)$ such that each edge weight is integer and in the range from 1 to C, where C is a relatively small constant.

(a) Show how to make Dijkstra run in $O(C|V| + |E|)$.

(b) Show how to make Dijkstra run in $O(\log C(|V| + |E|))$. Hint: Can we us a binary heap with size C but not $|V|$?

Answer Referred by Haoxuan Xu.

(*a*) If we want to reduce the time complexity from $O(|V|^2 + |E|)$ to $O(C|V| + |E|)$, we need to make the unexplored vertex with smallest distance in $O(C)$ in average instead of $O(|V|)$. As each edge weight is bounded by C, the longest distance is bounded by $C|V|$ when two vertices are connected. Besides, as we are maintaining a SPT during the Dijkstra algorithm, the distance of vertex added to explored set is non-decreasing, we can use an array of doubly linked list *bucket* to store vertices of distance, which means $bucket[dist] = \{$vertices whose distance to source is $dist\}$, and the array size is $|C|V$. We need the data structure of $bucket[dist]$ has following properties:

(a) Check if it's empty in $O(1)$.

(b) Delete/Add an element in $O(1)$.

So, we can choose doubly linked list or any data structure sastisfying beyond properties.

Algorithm 3 Improved Dijkstraj algorithm

```
Input: G = (V, E), s, 1 ≤ w(e) ≤ $C$ for all e ∈ E.
 1: Initialize dist(v) = ∞, pre[v] = nil for all v ∈ V. bucket ← An array of size $C|V|$,
    whose element data structure is doubly linked list.
 2: dist(s) = 0, bucket[0] = { s }, index = 0.
 3: while index != $C|V|$ do
 4:   if dist[index] is empty then
 5:     index ← index + 1.
 6:     continue.
 7:   end if
 8:   u ← bucket[index] top element, and remove it from bucket[index]
 9:   for all (u, v) ∈ E do
10:     preDist ← dist[v].
11:     if dist[v] >dist[u] + w(u, v) then
12:       dist[v] ← dist[u] + w(u, v).
13:       pre(v) = u.
14:       Add v into bucket[dist[v]]
15:       if preDist ≠ ∞ then
16:         remove v from bucket[preDist]
17:       end if
18:     end if
19:   end for
20: end while
```

Hence, we traverse index from 0 to $C|V|$, the internal time complexity of each loop except the internal for loop is $O(1)$, and visit all edges through the internal for loop only once. So, the time complexity of the improved algorithm is $O(C|V| + |E|)$.

(*b*) As the hint says, we need to improve the dijkstra algorithm with heap by reducing the heap size from $|V|$ to C. We use the idea inspired by the algorithm in **(a)**, let the elements of heap be buckets, which is are sets containing all vertices has the same distance to s. And now, we are going to prove that, when we choose an unexplored vertex from a new $bucket[d]$, there will be at most C buckets in use, which are $bucket[d], \dots bucket[d+C-1]$.

We will prove it by induction.

(a) **Base case.** The initial state is that there is only one bucket in use, $bucket[0]$, obviously satisfying the statement.

(b) **Induction.** Suppose we are choosing from a new $bucket[d]$, there are C buckets in use, which are $bucket[d], \dots bucket[d+C-1]$. And now, we will choose a vertex from $bucket[d]$, then the distance of updated adjacent vertices must lie in $(d+1, d+C)$, which means there will be a new $bucket[d+C]$. Then, we choose vertices from $bucket[d]$ contiguously until is empty. Then, we will choose a vertex from a new bucket $bucket[d+1]$, and now, there are still C buckets in use, $bucket[d+1], \dots bucket[d+C]$. And now, we finish the induction.

Hence, we have proven the previous statement, and it's easy to see that there are at most $C+1$ buckets in use with that. And now, we have bounded the size of the heap to $C+1$, the time complexity for finding minimum will be $O(|V|log(C+1)) = O(|V|logC)$, the time complexity for updating will be $O(|E|log(C+1)) = O(|E|logC)$, and the total time complexity for the improved Dijkstraj algorithm will be $O(logC(|E|+|V|))$.

SHANGHAI JIAOTONG UNIVERSITIY

Assignment 3 Suggested Answer

Course: *AI2615 - Algorithm Design and Analysis*
Due date: *Dec 1, 2021*

Problem 1

Give a linear-time algorithm that takes as input a tree and determines whether it has a *perfect matching*: a set of edges that touch each node exactly once.

Answer referred by Yichen Tao . Here is the algorithm:

Algorithm 1 Check perfect matching

```
Require: A tree $T = (V, E)$.
Ensure: Whether $T$ has a perfect matching and the prefect matching $M$ if it has one.
  $Match \leftarrow \emptyset$
  while $E$ is not empty do
    for All leaf nodes $v_L$ in $V$ do
      Find the parent of $v_L$ : $v_P$
      $M \leftarrow M + \{(v_L, v_P)\}$
      Remove all edges connecting to $v_P$ from $E$.
      Remove $v_L$ and $v_P$ from $V$.
    end for
  end while
  if $V$ is empty then
    return $True$ & $M$
  else
    return $False$
  end if
```

Proof of Correctness. Notice that if the tree has a perfect matching, the number of nodes should be even. Since the algorithm deletes two nodes at a time, it will always return *False* when the number of nodes is odd. So we only consider the case when the number of nodes is even, and denote the number of nodes by $2n$. We prove the correctness by induction over n.
Base case. When $n = 1$, the tree consists only of a root and a leaf node, and the algorithm is obviously correct.
Induction hypothesis. The algorithm is correct for a tree with $2(n-1)$ nodes.
Induction step. We need to prove that the algorithm works for a tree T with $2n$ nodes. Denote the tree after removal by T', which the algorithm can make correct decision about according to the induction hypothesis. If T has a perfect matching M, we should have $(v_L, v_p) \in M$, for otherwise there is no node able to match with v_L. So $M - (v_L, v_p)$ is a perfect matching of T', which can be correctly obtained according to induction hypothesis. If T does not have a perfect matching, the algorithm surely cannot find one.

Time complexity. Since each node and edge is deleted at most once, the time complexity of the algorithm is $O(|V| + |E|)$.

Problem 2

Consider you are a driver, and you plan to take highways from A to B 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 with surplus supply, they are located on a line together with A and B. The i-th gas station is located at d_i that means the distance from A to the station, and its price is p_i for each unit of gas, each unit of gas exactly support one unit of distance. The car's tank is empty at the beginning and so you can assume there is a gas station at A. Design efficient algorithms for the following tasks.

(a) Determine whether it is possible to reach B from A.

(b) Minimize the gas cost for reaching B.

Answer Referred by Yuhao Zhang. 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 $O(n \log n)$ 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.

1. First, we find the nearest cheaper gas station j (if exists) after i. (i.e., we find the first gas station j after i such that $p_j < p_i$.) If j does not exist, then we manually fix $j = n + 1$, and $d_j = D$, to represent the location of B, we can also view B as the $(n + 1)$-th station with distance D and the cheapest price.

2. If $d_j \le d_i + C$, we refuel the tank to $d_j - d_i$ and pay $\max\{d_j - d_i - G, 0\} \cdot p_i$.

3. Otherwise, if $d_j > d_i + C$, we refuel the tank to C and pay $(C - G) \cdot p_i$.

Correctness of The Greedy. 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 $G(0) = \emptyset$ 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 contain as a subset $G'(i) = G(i - 1) \cup g'(i)$, where $g'(i) \neq g(i)$. Consider the two choice made by our algorithm at 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 $\sum_{k=i}^{j-1} \geq g(i)$. Then, we can increase $g'(i)$ to $g(i)$ and decrease totally $g(i) - g'(i)$ unit 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 $g'(i)$ 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 $G(i) = G(i-1) \cup g(i)$.

The Running Time. 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 most pay $O(n)$ time to find j, so the time complexity is $O(n^2)$ totally.

Improve the running time 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 $O(n)$.

Problem 3

The problem is concerned with scheduling a set J of jobs $j_1, j_2, \ldots, j_n$ on a single processor. In advance (at time 0) we are given the earliest possible start time s_i, the required processing time p_i and deadline d_i of each job j_i. Note that $s_i + p_i \leq d_i$. It is assumed that s_i, p_i and d_i are all integers. A schedule of these jobs defines which job to run on the processor over the time line, and it must satisfy the constraints that each job j_i starts at or after s_i, the total time allocated for j_i is exactly p_i, and the job finishes no later than d_i. When such a schedule exists, the set of jobs is said to be feasible. We allow a preemptive schedule, i.e., a job when running can be preempted at any time and later resumed at the point of preemption. For example, suppose a job j_i has start time 6, processing time 5 and deadline 990, we can schedule j_i in one interval, say, $[6, 11]$, or over two intervals $[7, 8]$ and $[15, 19]$, or even three intervals $[200, 201]$, $[300, 301]$, $[400, 403]$.

(*a*) Give a set of jobs that is not feasible, *i.e.*, no schedule can finish all jobs on time.

(*b*) Design an efficient algorithm to determine whether a job set J is feasible or not. Let D be the maximum deadline among all jobs, *i.e.*, $D = \max_{i=1}^{n} d_i$, you should analyze the running time in terms of n and D. (Tips, you need to prove that if the input job set J is feasible, then the algorithm can find a feasible schedule for J.)

(*c*) Design an efficient algorithm with the running time only in terms of n.

Answer referred by Haoxuan Xu.

(*a*) $(s_1, p_1, d_1) = (0, 5, 6)$, $(s_2, p_2, d_2) = (0, 5, 6)$. Obviously it's not feasible, because they both start at 0, and their deadlines are 6, but the total processing time needed is $5 + 5 = 10 > 6$, so it's not feasible.

(*b*) The algorithm is similar to the one in question 2. We use an array T with size D to maintain that the allocation of time, $T[t] = 1$ meaning that there is one job allocated at time t, $T[t] = 0$ meaning that there is no job allocated at time t.

First of all, we need to sort the jobs in ascending order according to the deadlines, and initialize the array T with 0. Then, take out jobs from the sorted array in order. For example, we now take out a job j_i with s_i, p_i, d_i. Then we begin from s_i to d_i to check that whether there are p_i time are unallocated. More specifically, we can use an index t_i initialized with s_i, current time allocated ct_i initialized with 0, continuously checking that whether $T[t_i]$, if $T[t_i] = 1$, $t_i += 1$, if $T[t_i] = 0$, set $T[t_i] = 1$, $t_i += 1$, $ct_i += 1$. If we find $ct_i == p_i$, we break and continue to check the next job. If we find $t_i > d_i$, then the set J is not feasible and return false. If all jobs are checked to finish, we return true.

Time Complexity. The firsr part of sorting J takes $O(nlogn)$. The second part of checking one job takes at most $O(d_i - s_i) = O(D)$. So checking every jobs takes $O(nD)$ at most. In conclusion, the time complexity of the algorithm is $O(nlogn + nD)$.

Correctness. We need to prove that if the input J is feasible, then the algorithm can find a feasible schedule. But we can also prove that if the algorithm return False and find that the input J is not feasible, then J is really infeasible, there is no possible schedule to finish them, and if the algorithm return True, obviously there

Algorithm 2 FeasibleJobSet Algorithm

```
Input: Job set $J$, $J[i] = (s_i, p_i, d_i)$.
 1: Sort $J$ according to deadlines $d$ in ascending order using quick sort algorithm, get
    the maximum deadline $D$.
 2: Initialize array $T$ of size $D$ with 0.
 3: for all $j_i$ in $J$ do
 4:     $ct_i \leftarrow 0, t_i \leftarrow s_i$
 5:     while True do
 6:         if $T[t_i] = 0$ then
 7:             $ct_i \leftarrow ct_i + 1$, $T[t_i] = 1$.
 8:         end if
 9:         $t_i \leftarrow t_i + 1$.
10:         if $ct_i = p_i$ then
11:             break
12:         end if
13:         if $t_i > d_i$ then
14:             return False
15:         end if
16:     end while
17: end for
18: return True
```

is a possible schedule and we can get it by setting $T[t_i]$ from $\{0, 1\}$ to allocated job id and unallocated flag like -1.

Now, we prove the former statement about situation that returning False. When it returns False at job j_i and its $t_i > d_i$, then which means every time unit before d_i is allocated but there is still some time need to finish j_i. The only way to finish j_i is that we set $p_i - ct_i$ time units before d_i to finish j_i instead of their origin allocated jobs. Suppose the origin allocated job is j_j, whose deadline is earlier than j_i, $d_j \leq d_i$. If we do so, then d_j can only be scheduled after d_i, but its deadline is earlier, so we cannot do the change. In conclusion, there is no way to finish j_i before d_i, and the job set J is really infeasible. Now, we finish the proof of the correctness.

(c) I think the main cost is that for one time unit t, there might be many jobs which checks for it, so we need to improve it.

The first way in my mind is that t_i shouldn't add 1 once and check again, it should jump to the next available time unit. So, we might borrow the idea or data structure for malloc function in memory management, use a explicit free list to store all free time units, each continuous free time period is abstracted into a whole, i.e. a struct containing the start of previous free time period, the start of the next free time period, the size of itself. But the time complexity is a little bit complex I'm not sure that it's $O(nlogn)$.

The second way is that we directly allocate time unit instead of jobs and checking each time unit, and this idea is from Shuyang Jiang and Yichen Tao. We use a time t to denote current time, and maintain two priority queues q_s, q_d, one for all unmeet jobs(j_i with $s_i > t$) order by the start time, one for all hung jobs(j_i with $s_i < t$) but priority is lower than the current scheduled job order by the deadline. Then we initialize the current time with 0, and continue to do so, suppose we have a job j_c in scheduling, and get job j_i from q_s and next start time ns, then we add the job with later deadline to q_d, and have the new current job j'_c, then we allocate all time before ns to j'_c, if it can be finished before ns, then continue to get a job from j_j from q_d, and allocate its time until we get ns, then we get a new job from q_s, repeat to do so.

Time Complexity. The time for selecting one job is at most $O(logn)$, and we select each jobs $O(n)$, so the total time complexity is $O(nlogn)$.

Problem 4

Makespan Minimization. Given m identical machines and n jobs with size $p_1, p_2, \dots, p_n$. How to find a feasible schedule of these n jobs on m machines to minimize the *makespan*: the maximal completion time among all m machines?
Recall that we have introduced two greedy approaches in the lecture.

- **GREEDY:** Schedule jobs in an arbitrary order, and we always schedule jobs to the earliest finished machine.
- **LPT:** Schedule jobs in the decreasing order of their size, and we always schedule jobs to the earliest finished machine.

We have proved that GREEDY is 2-approximate and LPT is 1.5-approximate, can we finish the following tasks? (Please write down complete proofs.)

(a) Complete the proof that LPT is 4/3-approximate, (*i.e.* LPT $\leq 4/3 \cdot$ OPT).
(b) Prove that GREEDY is $(2-1/m)$-approximate, (*i.e.* GREEDY $\leq (2-1/m) \cdot$ OPT).
(c) Prove that LPT is $(4/3 - 1/3m)$-approximate, (*i.e.* LPT $\leq (4/3 - 1/3m) \cdot$ OPT).

Answer referred by Yichen Tao.

(a) Without loss of generality, suppose that $p_1 > p_2 > \dots > p_n$, and that the last finished job is p_n, since if there are jobs that finishes last but is not scheduled last, deleting them will only make the case better. We consider two cases here: $p_n \leq \frac{1}{3} \cdot$ OPT and $p_n > \frac{1}{3} \cdot$ OPT.

If $p_n \leq \frac{1}{3} \cdot$ OPT, the analysis is similar to that of the 3/2-approximation. Suppose p_n starts at time t according to LPT schedule. We can know that all machines are busy before the time t, so total workload $\sum p_i$ should be no less than tm. Hence we have OPT $\geq t$, and ALG $= t + p_n \leq$ OPT $+ \frac{1}{3} \cdot$ OPT $= \frac{4}{3} \cdot$ OPT.

Otherwise, when $p_n > \frac{1}{3} \cdot$ OPT we can prove an even stronger conclusion: LPT $=$ OPT. We prove this by contradiction. Suppose job with size p_l is the first scheduled job violating OPT. Then before scheduling this job, each machine should have at least one and at most two jobs scheduled, since all jobs are larger than $\frac{1}{3} \cdot$ OPT. Suppose there are i machines with one job scheduled on, and these jobs should have their sizes larger than OPT $- p_l$. We can these jobs "*long jobs*" and the other jobs "*short jobs*". There are $(m-i)$ machines with two short jobs, and p_l is still there waiting to be scheduled, so the number of short jobs is $2(m-i)+1$ if we do not consider the job after the l-th.

To this point, we find that it is impossible to satisfy OPT with any schedule even considering only the first l jobs. The i *long jobs* would have taken up i machines because the sizes of all other jobs are no less than p_l. There are only $(m-i)$ machines left. As the job sizes are all larger than $\frac{1}{3} \cdot \text{OPT}$, each machine can handle at most 2 short jobs. Thus the left machines can only handle $2(m-i)$ *short jobs*. However, there are $2(m-i)+1$ short jobs, which leads to contradiction. Hence, our assumption is false, and the conclusion is proved.

(b) Again, suppose that the last finished job is the one with cost p_n. Denote the starting time of p_n by t.

We can prove that $t \le \frac{1}{m}\sum_{i=1}^{n-1} p_i$. $\frac{1}{m}\sum_{i=1}^{n-1} p_i$ is the time when all machines finish the first $(n-1)$ jobs at the same time. Then when the finishing time of the m machines are not the same, the shortest one should be shorter than $\frac{1}{m}\sum_{i=1}^{n-1} p_i$. According to our algorithm, the n-th job will be assigned to this earliest finishing machine, so t cannot be larger than $\frac{1}{m}\sum_{i=1}^{n-1} p_i$.

Thus, $\text{GREEDY} = t + p_n \le \frac{1}{m}\sum_{i=1}^{n-1} p_i + p_n = \frac{1}{m}\sum_{i=1}^{n} p_i + (1-\frac{1}{m})p_n$.
We then apply two obvious facts: $\text{OPT} \ge \frac{1}{m}\sum_{i=1}^{n} p_i$; $\text{OPT} \ge p_n$. Hence, we can get $\text{GREEDY} \le (2-\frac{1}{m})\text{OPT}$.

(c) Again, suppose that $p_1 > p_2 > \ldots > p_n$, and that the last finished job is p_n. Two cases are considered here: $p_n \le \frac{1}{3}\cdot\text{OPT}$ and $p_n > \frac{1}{3}\cdot\text{OPT}$.

When $p_n \le \frac{1}{3}\cdot\text{OPT}$, similar to the analysis in (b), we have $\text{LPT} = t + p_n \le \frac{1}{m}\sum_{i=1}^{n-1} p_i + p_n = \frac{1}{m}\sum_{i=1}^{n} p_i + (1-\frac{1}{m})p_n \le \text{OPT} + (1-\frac{1}{m})\cdot\frac{1}{3}\text{OPT} = (\frac{4}{3}-\frac{1}{3m})\cdot\text{OPT}$.

In another case, when $p_n > \frac{1}{3}\cdot\text{OPT}$, it can be asserted that LPT gives the optimal solution according to our analysis in (a).

To sum up, LPT is $(\frac{4}{3}-\frac{1}{3m})$-approximate.

Assignment 4 Suggested Answer

Course: *AI2615 - Algorithm Design and Analysis*
Due date: *Jan, 2022*

Problem 1

Given two strings $A = a_1a_2a_3a_4 \dots a_n$ and $B = b_1b_2b_3b_4 \dots b_n$, how to find the longest common subsequence between A and B? In particular, we want to determine the largest k, where we can find two list of indices $i_1 < i_2 < i_3 < \dots < i_k$ and $j_1 < j_2 < j_3 < \dots < j_k$ with $a_{i_1}a_{i_2} \dots a_{i_k} = b_{j_1}b_{j_2} \dots b_{j_k}$. Design a DP algorithm for this task.

Answer referred by Haoxuan Xu . We use a 2d array, $LLST[n+1][n+1]$ to implement the dynamic programming, $LLST[i][j]$ means the longest common subsequence length of $A[0:i], B[0:j]$ (note that the index range grammer is same as python, i.e. "hello"[0:2] = "he")

First, if $i = 0$ or $j = 0$, then one string is empty, the LLST(length of longest common subsequence) is obviously 0. Then, for the ith character of A a_i, jth character of B B_j ($i \neq 0, j \neq 0$), there are either same or different. If they are same, they can be added to the common subsequence, the new longest common subsequence length will be $LLST[i][j] + 1$; if they are different, then the new longest common subsequence length will be $max(LLST[i-1][j], LLST[i][j-1])$. And we can see from the state transition equation that the solution of $LLST[i][j]$ need to solve $LLST[i-1][j-1], LLST[i-1][j], LLST[i][j-1]$ in advance. So we only need to solve the subproblems from $i = 0$ to n, and from $j = 0$ to n. The longest common subsequence length of A, B will be $LLST[n][n]$.

If we want to get the corresponding longest common subsequence, we can use an extra 2d array to maintain it, $LST[n+1][n+1]$, $LST[i][j]$ means the longest common subsequence of $A[0:i], B[0:j]$, and continusously update it.

Time Complexity. There are totally $(n+1)^2$ states and the update for each state takes $O(1)$. So the total time complexity is $O(n^2)$.

Correctness.

(Base case) $i = 0$ or $j = 0$, then one string is empty, the longest common subsequence length is obviously 0.

(Induction) Our inductive hypothesis is that $LLST[i][j]$ correctly represents the longest common subsequence length of $A[0:i]$ and $B[0:j]$ for $i = p-1, j = q-1$, $i = p-1, j = q$ and $i = p, j = q-1$. Then, suppose we are finding $LLST[p][q]$, if $A[p-1] = B[q-1]$, obviously they can match up, then its corresponding length is $LLST[p-1][q-1] + 1$, but we can also choose not to match them up, and try to find the LST of $A[0:p]$ and $B[0:q-1]$ or $A[0:p-1]$ and $B[0:q]$, but obviously the longest common subsequence lenght of these two pairs of strings if smaller than $A[0:p]$ and $B[0:q]$, as we have one more additional character and one more choice. So, $LLST[p][q] = LLST[p-1][q-1] + 1$. And if $A[p-1] \neq B[q-1]$, they can't match

Algorithm 1 LongestCommonSubsequenceLength Algorithm

```
Input: string $A$ with size $n$, string $B$ with size $n$
1: Initialize $LLST[n][n]$ with 0, initialized $LST[n][n]$ with "".
2: for $i \leftarrow 0$ to $n$ do
3:    for $j \leftarrow 0$ to $n$ do
4:       if $i = 0$ or $j = 0$ then
5:          continue
6:       end if
7:       if A[i-1] = B[j-1] then
8:          $LLST[i][j] \leftarrow LLST[i-1][j-1] + 1$
9:          $LST[i][j] \leftarrow LST[i-1][j-1] + A[i-1]$
10:      else
11:         $LLST[i][j] \leftarrow \max\{LLST[i-1][j], LLST[i][j-1]\}$
12:         $LST[i][j] \leftarrow$ corresponding $LST[i-1][j] / LST[i][j-1]$
13:      end if
14:   end for
15: end for
16: return $LLST[n][n], LST[n][n]$
```

up, we have to throw one of them and try to find the LST of $A[0:p]$ and $B[0:q-1]$ or $A[0:p-1]$ and $B[0:q]$, and get the maximum of them, its correctness is obvious. And the correctness of maintaining LST is obvious by the correctness of $LLST$.

(Computation Order) We solve subproblems from $i = 0$ to n, $j = 0$ to n, which guarantees that when we solve $LLST[i][j]$, $LLST[i-1][j-1]$, $LLST[i-1][j]$, $LLST[i][j-1]$ have been solved.

Problem 2

Given two teams A and B, who has already played $i + j$ games, where A won i games and B won j games. We suppose that they both have 0.5 independent probability to win the upcoming games. The team that first win $n \geq \max\{i, j\}$ games will be the final winner. Design a DP algorithm to calculate the probability that A will be the winner.

Answer referred by Yichen Tao. Define the subproblem $p_A(x,y) (i \leq x \leq n, j \leq y \leq n, x + y \neq 2n)$ as the probability that A is the final winner, provided that A has won x games and B has won y games. Three cases are considered here:

- $x = n \wedge y < n$. Since A has already won, $p_A(x,y) = 1$.
- $x < n \wedge y = n$. Here B has already won n games, so $p_A(x,y) = 0$.
- $x < n \wedge y < n$. In this case, at least one more game will be played. The possibilty that A win or lose the next game is both $\frac{1}{2}$. Hence, according to Law to Total Probability, we should have $p_A(x,y) = \frac{1}{2}p_A(x+1,y) + \frac{1}{2}p_A(x,y+1)$.

By the analysis above, we have the following recurrence relation:

$$p_A(x,y) = \begin{cases} 1, & x = n \wedge y < n \\ 0, & x < n \wedge y = n \\ \frac{1}{2}p_A(x+1,y) + \frac{1}{2}p_A(x,y+1), & x < n \wedge y < n \end{cases} \quad (1)$$

By calculating the values of $p_A(x,y)$ in a certain order (which will be shown below), the value of $p_A(i,j)$ is the answer to the problem.
The formalization of the algorithm is shown in 2:

Algorithm 2 Calculating the probability that A be the final winner.

```
Require: Number of games A has already won: $i$. Number of games B has already
  won: $j$. Number of winnings needed to be the final winner: $n$.
Ensure: The probability that A be the final winner.
  for $x := n$ to $i$ do
    for $y := n$ to $j$ do
      if $x = n$ then
        $p_A(x,y) \leftarrow 1$
      else if $y = n$ then
        $p_A(x,y) \leftarrow 0$
      else
        $p_A(x,y) \leftarrow \frac{1}{2}p_A(x+1,y) + \frac{1}{2}p_A(x,y+1)$
      end if
    end for
  end for
  return $p_A(i,j)$
```

Proof of Correctness. The correctness of the algorithm can be proved by induction.
Base case. The value of $p_A(n,n)$ does not matter the calculation after that since it will not be involved in any further calculation. The reason that $p_A(n,y) = 1$ and $p_A(x,n) = 0$ is obvious and has already been explained above.
Induction Hypothesis. Suppose that the calculation of $p_A(x+1,y)$ and $p_A(x,y+1)$ (when $x,y < n$) are both correct.
Induction. Denote the event "A wins the $x+y+1$-th game" by E; the event "A is the final winner" by F. Apparently, we should have:
$P(E) = P(\overline{E}) = \frac{1}{2}$
$P(F|E) = p_A(x+1,y); P(F|\overline{E}) = p_A(x,y+1)$
Therefore, $p_A(x,y) = P(F) = P(E)P(F|E) + P(\overline{E})P(F|\overline{E}) = \frac{1}{2}p_A(x+1,y) + \frac{1}{2}p_A(x,y+1)$, which is consistent with what we do in the algorithm.
The values of $p_A(x,y)$ is calculated in the following order: $p_A(n,n), p_A(n,n-1),\ldots, p_A(n,j)$; $p_A(n-1,n),\ldots,p_A(n-1,j);\ldots;p_A(i,n),\ldots,p_A(i,j)$. So when $p_A(x,y)$ is being calculated, the two possibly needed term $p_A(x,y+1)$ and $p_A(x+1,y)$ should have been calculated already.

Time Complexity. Calculating each entry of p_A needs $O(1)$, and $(n-i)(n-j)$ entries are calculated. So the overall time complexity is $O((n-i)(n-j))$.

Problem 3

Formalize the improved DP algorithm for the Longest Increasing Subsequence Problem in the lecture, prove its correctness, and analyze its time complexity.

Answer referred by Haoxuan Xu. The description of the improved algorithm is explained in the lecture, so I directly give the formal algorithm.

Algorithm 3 LIS Improved Algorithm

```
Input: A sequence of number a with size $n$.
1: $sm \leftarrow$ '-' with size $n$, $maxLen \leftarrow 0$
2: for $i \leftarrow 0$ to $n-1$ do
3:     if $maxLen = 0$ or $a[i] > sm[maxLen-1]$ then
4:         $maxLen \leftarrow maxLen + 1, sm[maxLen] \leftarrow a[i]$
5:     else
6:         Binary search $sm[0 : maxLen]$ to get the largest j such that $sm[j] < a[i]$
7:         $sm[j+1] \leftarrow a[i]$
8:     end if
9: end for
10: return $maxLen$
```

Time Complexity. We totally do n iterations, and in each iteration, the binary search costs $O(logn)$ time, and if we append the number to the end of sm, the time cost is $O(1)$, hence, the time cost for each iteration is $O(logn)$. In conclusion, the time complexity is $O(nlogn)$.

Correctness. Here, we define the subproblems as maintain sm correctly, which means $sm[len]$ is the smallest ended number for an increasing subsequence with length len during iterations.

(Base case) The base case is that the orgin sm is empty, so when adding the first number to it, it must be the smallest ended number for an increasing subsequence with length 1, as no other number can be chosen, its correctness is obvious.

(Induction) The inductive hypothesis is that $sm[len]$ is the smallest ended number for an increasing subsequence with length len for $len < maxLen$ by using $a_1...a_{i-1}$.

Considering a_i, there are two cases. First, $a_i > sm[maxLen]$, we can just add $maxLen$ and append $a[i]$ to the end, which is still increasing, whose correctness is obvious. In case 2, we find the the largest number $sm[j]$ which is smaller than $a[i]$, then obviously, $sm[0 : j]$ will not be updated as they are all smaller than $a[i]$ and sm is the smallest ended number, and we update $sm[j+1]$ to $a[i]$, because we can form a new increasing subsequence with length $j+1$ by adding $a[i]$ to the previous incresing subsequence with length j. Besides, we cannot change $sm[j+2]$ or later. Take $sm[j+2]$ for example, as $a[i] <= sm[j+1]$, it's smaller than the number before ended number with length $j+2$, so we cannot change the ended number. So, the correctness of sm is guaranteed in iterations.

Hence, we have proven that the correctness of sm is maintained during iterations by induction.

(Computation Order) The order in which the subproblems are solved is from $i = 0$ to n, so we can traverse from $i = 0$ to n to solve them.

Problem 4

Optimal Indexing for A Dictionary. Consider a dictionary with n different words $a_1, a_2, \ldots, a_n$ 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.

Answer referred by Yichen Tao. First, the notations involved are stated. $cost[i][j]$ is the minimum cost of the binary search tree (BST) including words $a_i, a_{i+1}, \ldots, a_j$. $root[i][j]$ is the root of the above BST. $sum[i][j]$ is the sum of weight of words $a_i, a_{i+1}, \ldots, a_j$, *i.e.* $sum[i][j] = \sum_{x=i}^{j} w_x$.
While calculating the cost of BST involving $a_i, \ldots, a_j$, if we choose a_r ($i \le r \le j$) as the root node, $a_i, \ldots a_{r-1}$ will be on the left subtree, and $a_{r+1}, \ldots, a_j$ on the right subtree. Suppose that $cost[i][r-1]$ and $cost[r+1][j]$ has already been calculated, all nodes except a_r should have their levels increased by 1 in the new BST. Hence, the cost of the BST with root a_r is $sum[i][j] + cost[i][r-1] + cost[r+1][j]$. The remaining problem is finding the r that minimizes the cost, and we address it by traversing all possbile values of r.
By the analysis above, we have the following recurrence relation:

$$cost[i][j] = \begin{cases} w_i, \ \ i = j \\ sum[i][j] + \min_r\{cost[i][r-1] + cost[r+1][j]\}, \ \ i < j \end{cases} \tag{2}$$

$$root[i][j] = \begin{cases} i, \ \ i = j \\ \arg\min_r\{cost[i][r-1] + cost[r+1][j]\}, \ \ i < j \end{cases} \tag{3}$$

After that, the value of $cost[1][n]$ is the optimal cost of the BST, and the BST can be formed with $root[i][j]$. 4 shows the formalized algorithm:

Proof of correctness. We prove the correctness by induction.
Base case. When $i = j$, there is only one node a_i in the BST. So the cost is w_i, and only a_i can be the root.
Induction hypothesis. The calculation of all $cost[x][y]$ with $y - x < j - i$ is correct.
Induction. Consider calculating $cost[i][j]$ and $root[i][j]$. Since we traverse all possible roots and choose the one giving the least cost as $root[i][j]$, it should give the optimal BST including nodes $a_i, \ldots, a_j$.
By the analysis above, all values of $cost[i][j]$ and $root[i][j]$ are correct, so the value of $cost[1][n]$ is the cost of the optimal BST, and the tree formed using $root$ is also the optimal binary search tree.

Algorithm 4 Finding the optimal binary search tree.

```
Require: $n$ different words $a_i$ with weight $w_i$.
Ensure: The binary search tree.
  Calculate all values of $sum[i][j]$.
  $cost[i][i] \leftarrow w_i$; $root[i][i] \leftarrow i$ for all $i$.
  for $l := 2$to $n$ do
    for $i := 1$to $n - l + 1$ do
      $j \leftarrow i + l - 1$
      $sum[i][j] \leftarrow sum[i][j] + \min_r\{cost[i][r-1] + cost[r+1][j]\}$
      $root[i][j] \leftarrow \arg\min_r\{cost[i][r-1] + cost[r+1][j]\}$
    end for
  end for
  Form the BST recursively with $root$.
  return the BST formed.
```

Time complexity. If we calculate $sum[i][j]$ following the idea of DP (which is rather simple and will not be dilated upon here), the initialization takes $O(n^2)$. For the calculation part, all $1 \leq i \leq j \leq n$ cases are considered, so there are $O(n^2)$ cases in all. Each case takes $O(n)$ as we need to traverse all possible r. So the overall time complexity is $O(n^3)$.

Problem 5

Precedence Constrained Unit Size Knapsack Problem. Let us recap the classic knapsack problem (unit size): We are given a knapsack of capacity W and n items, each item i has the same size $s_i = 1$ and is associated with its value v_i. The goal is to find a set of items S (each item can be selected at most once) with total size at most W (*i.e.*, $\sum_{i \in S} s_i = |S| \leq W$) to maximize the total value (*i.e.*, $\sum_{i \in S} v_i$). Now we are given some additional precedence constraints among items, for example, we may have that item i can only be selected if we have selected item j. These constraints are presented by a tree $G = (V, E)$, which says that a vertex can only be selected if we have selected its parent, (*i.e.*, we need to select all his ancestors). The task is also to maximize the total value we can get, but not violating the capacity constraint and any precedence constraints.

(a) Design a DP algorithm for it in $O(nW^2)$ time.

(b) Can you improve the algorithm and analysis to make it run in $O(n^2)$ time? (When W is sufficiently large like $W = \Theta(n)$, it is an improvement.)

Answer.

(*a*) referred by Xiangge Huang

Using DP algorithm, we define $F(i,j)$ is each situation to choose items. Due to the dependence between parent and children in trees, we let i represents the relationship between each item and j represents the volume. Firstly we consider the root, which must be chosen with size 1, so we should choose the biggest value in the left tree and let it add the value of the root, and we do the same thing in the following subtrees. So we have the formula $F(i,j) = \max_{0\leq k\leq j}(F(i,j), F(i,j-k) + F(t,k))$. And the basic step is when i is a leaf, $F(i,1) = v_i$, because each item size is 1, to make value bigger we must have only left 1 when we reach the leaf, and when $j = 0$, that is the bag is full, we have $F(i,0) = 0$. We describe our algorithm as below.

```
//precedence bag
construct f[num_of_nodes][W]
void bag(node u)
    f[u,0]=0, f[u,1]=v[u]
    for node i go through all u sons and i is not null:
        bag(i)
        for j from W to 0:
            $f(i,j) = \max_{0\leq k\leq j}(f(i,j), f(i,j-k) + f(t,k))$
int main()
    bag(root)
    return f[root][W]
```

We can analyze the whole process of this algorithm, due to the precedence relationship, it is natural for us to solve this problem from the top to the bottom of this tree. And for a special node u, if our bag does not have any space, we can't add it, so $f(u,0) = 0$, and if our bag only has 1 place, we add it directly, so $f(u,1) = 1$, and this always happens when u is a leaf. And when we arrive at a node, we know its statue is decided by its sons, so we use recursion to solve its sons initially, and when we go back from sons to parent, we will choose the best subtrees through our formula. So we can get the maximal value in root finally, that is $f(root, W)$. And because the parent only focus on its son, this must be a DAG.

Obviously for all recursion part, we should go through the whole tree, so it costs $O(n)$, and in combination process, we should loop j from W to 0 and k from 0 to j separately, and it costs $O(W^2)$, so the total time complexity is $O(nW^2)$.

(b) referred by Xiangge Huang

We can simplify this algorithm by changing the way we explore the tree. Because we calculate all subtrees of a parent, which leads to a big cost, we optimize our algorithm is this part. We can let $G(u,j)$ means the whole forest containing u, u sons and u brothers, so the path that we loop the tree can be shorten. And the formula can be

$$G(i,j) = \max\left(G(b,j), G(i,j), \max_{x+y=j-1}(G(s,x) + G(b,y) + v_i)\right)$$

The b means the brothers of i and the s means the sons of u. Because we link a node with all its brothers and sons, obviously that each subpart of this tree is equal, and we can choose the best part among its sons and brothers as the final value. Also, x and y means the left bag size, which should be less than or equal to current subtree.

We describe our algorithm as below.

```
//improved bag
construct g[num_of_nodes][W]
void bag(node u)
    g[u,0]=0, g[u,1]=v[u]
    b=brother[u], s=son[u]
    if s is not null: bag(s)      //size[s]^2
    if b is not null: bag(b)      //size[b]^2
    size[u]=size[s]+size[b]+1
    for k from 1 to W: g[u][k]=g[b][k] //choose best brother
    for x from 0 to size[s]:    //choose both son and brother
```

```
        for y from 0 to size[b]:    //size[s]size[b]
            j=x+y+1
            $g(u,j) = max\ (g(u,j), g(s,x) + g(b,y) + v[u])$
int main()
    bag(root)
    return g[root][W]
```

We analyze the time complexity first. Also, we also should go through all node, but in each loop, we know the complexity is $(size[s])^2 + (size[b])^2 + (size[s]size[b])$, which is related to $bag(s)$, $bag(b)$ and the next x and y part. We know $(size[s])^2 + (size[b])^2 + (size[s]size[b]) \leq (size[s] + size[b] + 1)^2 = (size[u])^2$. Consider we start from the root, so the time complexity will not be larger than n^2. So the total time complexity is $O(n^2)$.

We reconstruct our algorithm to make it simpler.

```
//new improved bag
construct g[num_of_nodes][W]
void bag(node u)
    g[u,0]=0, f[u,1]=v[u]
    size[u]=1
    for node i go through all u sons and i is not null:
        bag(i)
        size[u]+=size[i]
        for j from size[u] to 1:
            for t from j-1 to 0:
                $g(u,j) = max\ (g(u,j), g(u,j-t) + g(i,t))$
int main()
    bag(root)
    return g[root][W]
```

Similarly, this algorithm is still $O(n^2)$ based on the former analysis. And it still builds from the bottom of the tree, but due to the truth in each layer we consider its sons and brothers together, we only run the corresponding size in each loop and recursion. And in each transformation part, the node with its brothers situates in the equal situation and is only decided by its sons. And we always choose the maximal value in each choice to fill the bag, so we can get the best option and value finally, and this is also a DAG.

Algorithm Design and Analysis (Fall 2021)
Assignment 5

1. (0 points) Given an $n \times n$ matrix A whose entries are either 0 or 1, you are allowed to choose a set of entries with value 1 and modify their value to 0. Design an efficient algorithm to decide if we can make A invertible (A is invertible if the determinant of A is not 0) by the above-mentioned modification. Prove the correctness of your algorithm and analyze its time complexity. Notice that your algorithm only needs to output "yes" or "no" and does not need to find the minimum number of modified entries. (Hint: Prove that this is possible if and only if there exist n entries with value 1 such that no two entries are in the same row and no two entries are in the same column.)

   We first prove the following lemma.

   Lemma 1. *We can choose a set of entries from* 1 *to* 0 *to make A invertible if and only if there exist n entries with value* 1 *such that no two entries are in the same row and no two entries are in the same column.*

   Proof. Suppose it is possible to choose a set of entries from 1 to 0 to make A invertible. By the definition of determinant, we have

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} a_{i,\sigma_i},$$

   where the summation is taken over all possible permutations $\sigma \in S_n$. Since we can only change an entry from 1 to 0 (not from 0 to 1), we must have at least one permutation σ such that $a_{i,\sigma_i} = 1$ for each $i = 1, \ldots, n$. Otherwise, we have $\prod_{i=1}^{n} a_{i,\sigma_i} = 0$ for all permutations no matter how we make changes to the entries, and this will make $\det(A) = 0$. Therefore, there exist n entries with value 1 such that no two entries are in the same row and no two entries are in the same column.

   If there exist n entries with value 1 such that no two entries are in the same row and no two entries are in the same column, we can change all the remaining entries to 0, and the determinant of the matrix is either 1 or -1, which is invertible. □

   With the lemma, we only need to *decide* if A contains n entries with value 1 such that no two entries are in the same row and no two entries are in the same column. This problem reduces to the problem of deciding if a bipartite graph $G = (U, V, E)$ with $|U| = |V| = n$ contains a perfect matching. To see this, given A, we can construct G as follows: create n vertices for each of U and V, and create an edge between the i-th vertex of U and the j-th vertex of V if and only if $a_{ij} = 1$. Thus, an entry a_{ij} with value 1 corresponds to an edge in G. It is easy to see that G contains a matching of size n

(i.e., a perfect matching) if and only if A contains n entries with value 1 such that no two entries are in the same row and no two entries are in the same column.

Now the problem becomes deciding if G contains a perfect matching. This can be done by Hopcroft-Karp Algorithm.

Putting these together, our algorithm can be described as

1. Construct a bipartite graph $G = (U, V, E)$ by the above-mentioned procedure.
2. Decide if G contains a perfect matching.

The correctness of the algorithm is just based on Lemma 1.

The time complexity for building G is $O(n^2)$. The time complexity for Hopcroft-Karp algorithm here is $O(\sqrt{|U| + |V|} \cdot |E|) = O(n^{2.5})$. Thus, the overall time complexity is $O(n^{2.5})$.

2. (25 points) Consider the maximum flow problem $(G = (V, E), s, t, c)$ on graphs where the capacities for all edges are 1: $c(e) = 1$ for each $e \in E$. You can assume there is no pair of anti-parallel edges: for each pair of vertices $u, v \in V$, we cannot have both $(u, v) \in E$ and $(v, u) \in E$. You can also assume every vertex is reachable from s.

   (a) (13 points) Prove that Dinic's algorithm runs in $O(|E|^{3/2})$ time.

   (b) (12 points) Prove that Dinic's algorithm runs in $O(|V|^{2/3} \cdot |E|)$ time. (Hint: Let f be the flow after $2|V|^{2/3}$ iterations of the algorithm. Let D_i be the set of vertices at distance i from s in the residual network G^f. Prove that there exists i such that $|D_i \cup D_{i+1}| \leq |V|^{1/3}$.)

We have seen in the class that a single iteration of the Dinic's algorithm requires $O(|E|)$ time if all edges have capacity 1. It remains to show that the number of iterations is bounded by $O(\sqrt{|E|})$ (for part (a)) and $O(|V|^{2/3})$ (for part (b)).

(a) If the algorithm terminates after $\sqrt{|E|}$ iterations, we are done. Suppose this is not the case. Let G^f be the residual network after $\sqrt{|E|}$ iterations. The distance between s and t in G^f is at least $\sqrt{|E|}$, as we have seen in the class that this distance is increased by at least 1 after each iteration of Dinic's algorithm. The maximum flow on G^f must be a union of many edge-disjoint s-t paths, since the flow must be integral (as the edges have integer capacities) and each edge has capacity 1. Since the distance between s and t in G^f is at least $\sqrt{|E|}$, there can be at most $|E|/\sqrt{|E|} = \sqrt{|E|}$ edge-disjoint s-t paths. This implies the maximum flow on G^f is at most $\sqrt{|E|}$.

Since each iteration increases the value of the flow by at least 1, there can be at most $\sqrt{|E|}$ extra iterations. Thus, the total number of iterations is $2\sqrt{|E|} = O(\sqrt{|E|})$.

(b) If the algorithm terminates after $2|V|^{2/3}$ iterations, we are done. Suppose this is not the case. Let G^f be the residual network after $2|V|^{2/3}$ iterations. The distance between s and t in G^f is then at least $2|V|^{2/3}$. Let D_i be the set of vertices at distance i from s. Those layers D_i's are disjoint, and there are at least $2|V|^{2/3}$ layers. We have $\sum_{i=1}^{|V|^{2/3}} |D_{2i-1} \cup D_{2i}| \leq |V|$. Thus, there exist i such that $|D_{2i-1} \cup D_{2i}| \leq |V|^{1/3}$. The total number of edges between D_{2i-1} and D_{2i} is then at most $|V|^{2/3}$. Since removing all those edges disconnect t from s, the minimum cut of G^f is at most $|V|^{2/3}$. By the Max-Flow-Min-Cut Theorem, the maximum flow is at most $|V|^{2/3}$.

Since each iteration increases the value of the flow by at least 1, there can be at most $|V|^{2/3}$ extra iterations. Thus, the total number of iterations is $3|V|^{2/3} = O(|V|^{2/3})$.

3. (20 points) Given an undirected and unweighted graph $G = (V, E)$, a *vertex cover* is a subset S of vertices such that it contains at least one endpoint of every edge, and an *independent set* is a subset T of vertices such that $(u, v) \notin E$ for any $u, v \in T$. Notice that V is a vertex cover, and $\emptyset$, or any set containing a single vertex, is an independent set. A *minimum vertex cover* is a vertex cover containing a minimum number of vertices, and a *maximum independent set* is an independent set containing a maximum number of vertices.

   (a) (5 points) Prove that S is a vertex cover if and only if $V \setminus S$ is an independent set.

   (b) (5 points) Prove that S is a minimum vertex cover if and only if $V \setminus S$ is an maximum independent set.

   (c) (10 points) Design an efficient algorithm to find a minimum vertex cover, or a maximum independent set, on *bipartite graphs*. Prove the correctness of your algorithm and analyze its time complexity.

(a) If S is a vertex cover, then every edge (u, v) must satisfy $u \in S$ or $v \in S$ (or both). Therefore, $V \setminus S$ can contain at most one of u and v. This already implies $V \setminus S$ is an independent set.

If $V \setminus S$ is an independent set, then, for every edge (u, v), $V \setminus S$ can contain at most one of u and v. Thus, at least one of u or v is in S. This implies S is a vertex cover.

(b) As the sum $|S| + |V \setminus S| = |V|$ is fixed, minimizing $|S|$ maximizes $|V \setminus S|$, and maximizing $|V \setminus S|$ minimizes $|S|$.

(c) Given a bipartite graph $G = (A, B, E)$, construct a max-flow instance as follows. Create two vertices s and t. Create an edge (s, a) for each $a \in A$ with capacity 1. Create an edge (b, t) for each $b \in B$ with capacity 1. Create an edge (a, b) for $a \in A$ and $b \in B$ with capacity ∞ if and only if (a, b) is in the original bipartite graph. Then, by finding a maximum flow on the graph, we find a minimum s-t cut (L, R) with $s \in L$ and $t \in R$. Finally, we output $(R \cap A) \cup (L \cap B)$, which is a minimum vertex cover. This completes the description of our algorithm.

To prove the correctness, first notice that the vertex cover has size $\sigma := |R \cap A| + |L \cap B|$, which is exactly the size of the cut (L, R) (notice that no edge with weight/capacity ∞ can be cut in a min-cut). Suppose for the sake of contradiction there is another vertex cover S with size less than σ. We have $|S| = |S \cap A| + |S \cap B|$. There is no edge from $A \setminus S$ to $B \setminus S$, so (L_S, R_S) with $L_S = (A \setminus S) \cup (B \cap S) \cup \{s\}$ and $R_S = (A \cap S) \cup (B \setminus S) \cup \{t\}$ is a cut (note that edges are directed in the cut instance, so an edge between A and B can only connect from a vertex in A to a vertex in B) with value $|S \cap A| + |S \cap B| = |S|$, which is less than σ. This contradicts to (L, R) is a min-cut.

The time complexity is $O(\sqrt{|V|} \cdot |E|)$ for running Hopcroft-Karp Algorithm.

4. (20 points) Covert the following linear program to its standard form, and compute its dual program.

$$\begin{aligned} \text{minimize} \quad & 2x_1 + 7x_2 + x_3 \\ \text{subject to} \quad & x_1 - x_3 = 7 \\ & 3x_1 + x_2 \geq 24 \\ & x_2 \geq 0 \\ & x_3 \leq 0 \end{aligned}$$

Standard form:

$$\begin{aligned} \text{maximize} \quad & -2x_1^+ + 2x_1^- - 7x_2 + x_3 \\ \text{subject to} \quad & x_1^+ - x_1^- + x_3 \leq 7 \\ & -x_1^+ + x_1^- - x_3 \leq -7 \\ & -3x_1^+ + 3x_1^- - x_2 \leq -24 \\ & x_1^+, x_1^-, x_2, x_3 \geq 0 \end{aligned}$$

Dual Program:

$$\begin{aligned} \text{minimize} \quad & 7y_1 - 7y_2 - 24y_3 \\ \text{subject to} \quad & y_1 - y_2 - 3y_3 \geq -2 \\ & -y_1 + y_2 + 3y_3 \geq 2 \\ & -y_3 \geq -7 \\ & y_1 - y_2 \geq 1 \\ & y_1, y_2, y_3 \geq 0 \end{aligned}$$

5. (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.

   (a) (5 points) Explain that the following is an LP-relaxation for the maximum matching problem.

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

   (b) (5 points) Write down the dual of the above linear program, and justify that the dual program is an LP-relaxation to the minimum vertex cover problem.

   (c) (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.)

   (d) (10 points) Use results in (a), (b) and (c) to prove König-Egerváry Theorem.

   (e) (5 points) Give a counterexample to show that the claim in König-Egerváry Theorem fails if the graph is not bipartite.

(a) 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.

(b) The dual program is as follows:

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

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).

(c) 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.

(d) 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.

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

Algorithm Design and Analysis (Fall 2021)
Assignment 6

1. (50 Points) Consider the following variant of the scheduling problem. We have n jobs with size (time required for completion) $p_1, \ldots, p_n \in \mathbb{Z}^+$. Instead of fixing the number of machines m and minimizing the makespan, we fix the makespan and minimize the number of machines used. That is, you can decide how many (identical) machines to use, but each machine can be operated for at most $T \in \mathbb{Z}^+$ units of time. Assume $p_i \le T$ for each $i = 1, \ldots, n$. Your objective is to minimize the number of machines used, while completing all the jobs.

   (a) (20 Points) Show that this minimization problem is NP-hard.

   (b) (20 Points) Consider the following local search algorithm. Initialize the solution where n machines are used such that each machine handles a single job. Do the following update to the solution until no more update is possible: if there are two machines such that the total size of the jobs on the two machines is less than T, update the solution by using only one machine to complete all these jobs (instead of using two machines). Show that this algorithm gives a 2-approximation.

   (c) (10 Points) Provide a tight example showing that the algorithm in (b) can do 1.5-approximation at best.

(a) This optimization problem is called *the bin packing problem*. We will show that it is NP-hard to decide if we can complete all the jobs by at most *two* machines. We reduce this decision problem from Partition+.

Given a Partition+ instance $S = a_1, \ldots, a_n$, we construct an instance $(p_1, \ldots, p_n, T)$ of the bin packing problem as follows:

- for each $i = 1, \ldots, n$, $p_i = a_i$;
- set $T = \lfloor \frac{1}{2} \sum_{i=1}^{n} a_i \rfloor$.

If the Partition+ instance is a yes instance, we can partition S to S_1 and S_2 such that $\sum_{a_i \in S_1} a_i = \sum_{a_i \in S_2} a_i = \frac{1}{2} \sum_{i=1}^{n} a_i$. For the bin packing instance, we can load all the jobs corresponding to S_1 to the first machine, and all the jobs corresponding to S_2 to the second machine. Each machine needs exactly time T, so this is a valid job assignment. We can complete all the jobs with at most two machines.

Suppose we only need at most two machines for the bin packing instance. Let T_1 be the set of jobs on the first machine, and T_2 be the set of jobs on the second machine (set $T_2 = \emptyset$ if only one machine is needed). We have $\sum_{i \in T_1} p_i \le T$ and $\sum_{i \in T_2} p_i \le T$. Since T_1, T_2 is a partition of all jobs and $\sum_{i=1}^{n} p_i = \sum_{i=1}^{n} a_i = 2T$, we must have $\sum_{i \in T_1} p_i = T$ and $\sum_{i \in T_2} p_i = T$. Let S_1, S_2 be the two subsets in the Partition+

instance corresponding to T_1, T_2 respectively. It is straightforward that $\sum_{a_i \in S_1} a_i = T$ and $\sum_{a_i \in S_2} a_i = T$. Thus, the PARTITION+ instance is a yes instance.

In fact, we can prove a stronger result than just the NP-hardness.

If $\mathrm{P} \neq \mathrm{NP}$*, the bin packing problem do not admit a polynomial-time* $\left(\frac{3}{2} - \varepsilon\right)$*-approximation algorithm for any* $\varepsilon > 0$*.*

Proof. Suppose we have a polynomial-time $\left(\frac{3}{2} - \varepsilon\right)$-approximation algorithm for any $\varepsilon > 0$. We show that there exists a polynomial-time algorithm for PARTITION+. Given a PARTITION+ instance, we convert it to the bin packing instance by the above-mentioned construction. The construction can clearly be done in polynomial time. Then we run the $\left(\frac{3}{2} - \varepsilon\right)$-approximation algorithm for the obtained bin packing instance. If the output of the algorithm uses at most two machines, we know the PARTITION+ instance is a yes instance. If the output of the algorithm uses at least three machines, we know the optimal number of machines is at least $\frac{1}{3/2-\varepsilon} \times 3 > 2$. Since the number of machines is an integer, we know the optimal solution will use at least three machines, and the PARTITION+ instance should be a no instance. This gives us a polynomial time algorithm for PARTITION+. Since PARTITION+ is NP-complete, it is a contradiction to $\mathrm{P} \neq \mathrm{NP}$. □

(b) The important observation here is that, for the output of the local search algorithm, the total time for jobs on every two machines is larger than T.

Let k be the number of machines the algorithm uses, and k^* be the optimal number of machines. Let m_i be the total size of jobs on machine i for the output of the local search algorithm. We have $m_i + m_j > T$ for any $1 \leq i, j \leq k$ by our observation.

On the one hand, we have

$$\sum_{i=1}^{k} m_i \leq k^* T,$$

as $\sum_{i=1}^{k} m_i$ is the overall size of all jobs and these jobs need to be contained in those k^* machines in the optimal solution.

On the other hand, since $m_i + m_j > T$, we have

$$2\sum_{i=1}^{k} m_i = (m_1 + m_2) + (m_2 + m_3) + \cdots + (m_{k-1} + m_k) + (m_k + m_1) > kT.$$

Combining the two inequalities, we have

$$\frac{\sum_{i=1}^{k} m_i}{k^*} \leq T < \frac{2\sum_{i=1}^{k} m_i}{k},$$

which implies $k < 2k^*$.

(c) A tight example could be $(p_1, p_2, p_3, p_4) = (1, 1, 2, 2)$ and $T = 3$. The optimal solution uses two machines with p_1, p_3 on the first and p_2, p_4 on the second. If we use the local search algorithm and first combine the machine containing p_1 and the machine containing p_2, it is easy to check that at least three machines are required.

2. (50 Points) Choose *any one* of the following questions. (You are encouraged to solve as many the remaining questions as possible "in your mind".)

   (a) 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.

   (b) 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.

   (c) Given two undirected graphs G and H, decide if H is a subgraph of G. Prove that this problem is NP-complete.

   (d) 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.

   (e) 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.

   (f) 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.

   (g) 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.

I am going to start from the easiest to the hardest.

(c) 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.

(d) 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.

(f) Difficulty: $*$

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.

(a) 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.

(g) 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.

(b) Difficulty: $* * *$

[Solution 1: Textbook Solution]

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. 1 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. 1. We do this for all the clauses.

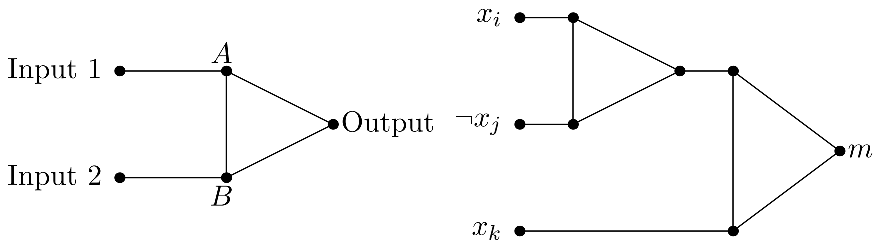


Figure 1: The OR gadget (left-hand side) and the gadget for the clause $m = (x_i \vee \bar{x}_j \vee x_k)$ (right-hand side).

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.

[Solution 2: Wenqian Wang's 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 1 (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 2. 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.

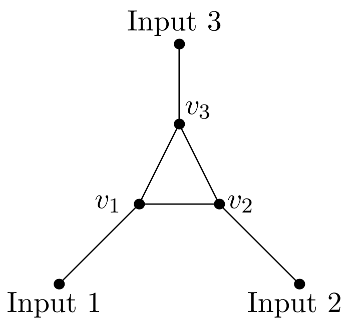
If ϕ' is a yes NotAllEqual-3SAT instance, we aim to show that at least one of ℓ_1, ℓ_2, ℓ_3 must be true, which will imply ϕ is a yes 3SAT instance. Firstly, we assume without loss of generality that $w =$ `false`. If $w =$ `true`, we can flip the values for all variables, which will also be a valid 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 3. 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. 2 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.

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. 2 demonstrates these features.



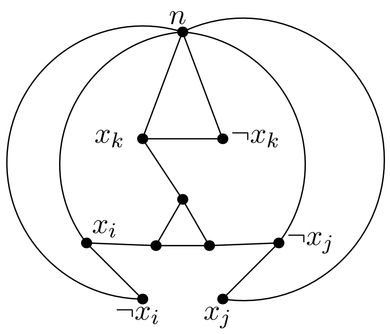


Figure 2: The not-all-equal gadget (left-hand side); the gadget for the clause $m = (x_i \vee \bar{x}_j \vee x_k)$ and how it is connected with other vertices (right-hand side).

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

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

(e) Difficulty: $***$

[Solution 1: Biaoshuai Tao's Solution]

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 $S^{c(u)}_u$ to $\mathcal{T}$;
- For each edge (u, v), include $S^{c(u,v)}_{uv}$ 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 $S^{c(u)}_u$. For each element e^r_{uv} corresponding to edge (u, v) in G, it is covered by S^r_u if $c(u) = r$, it is covered by S^r_v if $c(v) = r$, and it is covered by S^r_{uv} if $c(u, v) = r$. Since exactly one of $c(u) = r, c(v) = r, c(u, v) = r$ can happen, e^r_{uv} is covered by exactly one subset. The same holds for e^g_{uv} and e^b_{uv}. 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^r_u, S^g_u, S^b_u 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^r_u is included, we assign $c(u) = r$; if S^g_u is included, we assign $c(u) = g$; if S^b_u 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^r_u and S^r_v in $\mathcal{T}$. In this case, the element e^r_{uv} is covered by both S^r_u and S^r_v, which contradicts to that $\mathcal{T}$ is an exact cover.

[Solution 2: Jiaxin Song's 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.