# Algorithm Design and Analysis (Spring 2024)

1. (25 points) Prove the following generalization of the master theorem. Given constants $a \geq 1, b > 1, d \geq 0$, and $w \geq 0$, if $T(n) = 1$ for $n < b$ and $T(n) = aT(n/b) + n^d \log^w n$, we have

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

2. (25 points) Let us consider the (randomized) quick sort algorithm, where we choose pivots uniformly at random. How to analyze its expected time complexity?

   (a) (25 points) Prove its expected time complexity is $O(n^c)$ for some $c < 2$ by using a similar method in the lecture. (You can choose the constant $c$ you want.)

   (b) (0 points, have fun!) Prove its expected time complexity is $O(n \log n)$.

   Tips: Consider the $i$-th smallest element $x_i$ and the $j$-th ($j > i$) smallest element $x_j$. Prove that they will be compared in quick sort if and only if $x_i$ or $x_j$ is the "first" appeared pivot among $i$ to $j$. What is its probability?

3. (25 points) Given an $n \times m$ 2-dimensional integer array $A[0, \ldots, n-1; 0, \ldots, m-1]$ 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 have to find all of them).

   (a) (10 points) 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) (15 points) 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) (0 points, have fun!) Generalize your algorithm such that it works for general $m$ and $n$. The running time of your algorithm should *smoothly* interpolate between the running times for the first two parts.

   Tips: You can first think about whether the local minimum must appear, and try to find out the reason.

4. (25 points) An integer triple $(x, y, z)$ forms a *good order* if $y - x = z - y$. Given an integer $n$, we call a permutation of $\{1, \ldots, n\}$ (denoted by $X = \{x_1, x_2, \ldots, x_n\}$) is *out-of-order* if it has the following property: for every $i < j < k$, the triple $(x_i, x_j, x_k)$ does not form a *good order* (i.e., $x_j - x_i \neq x_k - x_j$).

   (a) (3 points) Prove that an integer triple $(x, y, z)$ forms a *good order* only if $x$ and $z$ are both even or both odd.

   (b) (5 points) Write down a permutation of $\{1, \ldots, 4\}$, and quickly prove it is *out-of-order* by the claim above.

   (c) (3 points) Prove that an integer triple $(x, y, z)$ where $x, y, z$ are all odd forms a *good order* only if $((x+1)/2, (y+1)/2, (z+1)/2)$ forms a *good order*.

   (d) (14 points) Design a divide and conquer algorithm runs in $O(n \log n)$ time to output an *out-of-order* permutation of $\{1, \ldots, n\}$. (The running time can be improved to $O(n)$.)

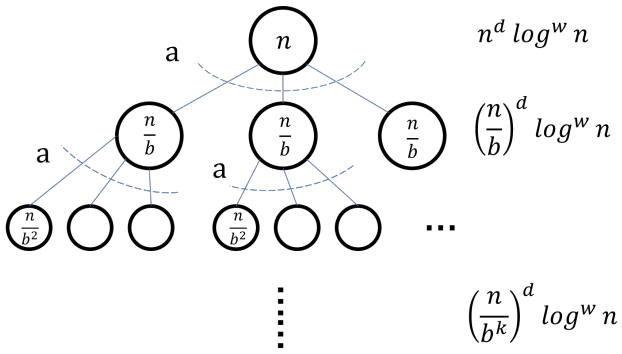
5. How long does it take you to finish the assignment (including thinking and discussion)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Please write down their names here.

# AI2615 Homework 1

Tao Rui 522030910024

March 23, 2024

## 1 Prove the Generalized Master Theorem




As the diagram showing:

$$\begin{aligned} T(n) &= n^d \log w_n + a \cdot \left(\frac{n}{b}\right)^d \log^w \left(\frac{n}{b}\right) + \cdots + a^k \left(\frac{n}{b^k}\right)^d \log^w \left(\frac{n}{b^k}\right) \\ &= n^d \left[\log_n + \left(\frac{a}{b^d}\right) \log^\omega \left(\frac{n}{b}\right) + \left(\frac{a}{b^d}\right)^2 \log^w \left(\frac{n}{b^2}\right) + \cdots + \left(\frac{a}{b^d}\right)^k \log \left(\frac{n}{b^k}\right)\right] \end{aligned}$$

where $k$ subject to the final condition: $\dfrac{n}{b^k} < b \ , \ k \geqslant \log_b n$

① $a \leqslant b^d$

As the function $g(i) = \log^\omega \left(\frac{n}{b^i}\right)$ is monotonic decrease. We can expand all terms $\log^\omega \left(\frac{n}{b^i}\right) \ , \ i \geqslant 1$ to the max term $\log^\omega n$

$$T(n) \leqslant n^d \log \omega \cdot \left[1 + \left(\frac{a}{b^d}\right) + \left(\frac{a}{b^d}\right)^2 + \cdots + \left(\frac{a}{b^d}\right)^k\right]$$

①**.1** If $a = b^d$, the summation turns to be $(k+1)$, which is $\log_b n = C \cdot \log n$

$$T(n) \leqslant C \cdot n^d \log \omega_n \cdot \log n = O\left(n^d \log^{\omega+1} n\right)$$

**①.2** If $a < b^d$, The summation turns to be as follow

$$T(n) \leqslant n^d \log^\omega n \frac{1 - \left(\frac{a}{b^d}\right)^{k+1}}{1 - \frac{a}{b^d}}$$

Replace $(k+1)$ by $\log_b a$, and bring $n^d$ into it:

$$= C \cdot \log^\omega n \left(n^d - a^{\log_b n}\right)$$

Use the the identical equation $a^{\log_b n} = n^{\log_b a}$, and the assumption $a < b^d$ we know that $\left(n^d - a^{\log_b n}\right)$ is dominated by $n^d$

$$T(n) \leqslant C \cdot n^d \log^\omega \Rightarrow T(n) = O\left(n^d \log^\omega n\right)$$

**②** $a > b^d$

Notice the function $h(i) = \left(\frac{a}{b^d}\right)^i \log^\omega \left(\frac{n}{b^i}\right)$ is not monotonous. Take the derivative:

$$h'(i) = -i^2 \log b + i \log n - \frac{a\omega}{b^d} \log b$$

$$i_{1,2} = \frac{\log n \pm \sqrt{\log^2 n - \text{Const}}}{2 \log b} \text{ , where Const } = \frac{4a\omega}{b^d} \log b$$

**Lemma:** The maximum term is obtained at $i = i_2$ but not at $i = 0$

**★ Proof ★**

$$h(i_2) - h(0) = \left(\frac{a}{b^d}\right)^{log_b n - 1} \log^\omega b - \log^\omega n$$
$$= n^{log_b a - d} \cdot \frac{b^d}{a} \log^\omega b - \log^\omega n$$

Since $log_b a - d > 0$ at this situation, the inequality $C \cdot n^\Delta - \log^\omega n > 0$ for any $\Delta > 0$ holds, which means $h(i_2) > h(0)$, the maximum term is obtained at $i = i_2 = \log_b n - 1$

■

Apply the limitation $n \to \infty$, we obtain that $i_1$ is slightly larger than 0, $i_2$ is slightly smaller than $\log_b n$. **But consider the fact that $\mathbf{h(\log_b n) = 0}$, then the right-side maximum term is obtained at $\mathbf{i = \log_b n - 1}$**

Further, we need compare the left-side(0) and right-side($\log_b n - 1$) maximum. By the Lemma proved above, the entire sequence's max term is obtained at $i = \log_b n - 1$

Now we can do the scaling:

$$\begin{aligned} T(n) &\leqslant n^d \left(\frac{a}{b^d}\right)^{i_2} \log^{\omega}\left(\frac{n}{b^{i_2}}\right) \\ &= n^d \cdot \frac{b^d}{a} \frac{n^{log_b a}}{n^d} \cdot \log^{\omega} b \\ &= C \cdot n^{\log_b a} = O(n^{\log_b a}) \end{aligned}$$

Combine ①.1, ①.2, ② we prove the generalized master theorem

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

# 2 Randomized Quick Sort Analysis

Consider the first pivot choice, it's uniformly distributed to choose any $n$ numbers as pivot.


If we choose the i-th pivot, we divide problem into two scale of $i-1$ and $n-i$, the next layer cost will be $T(i-1)+T(n-i)+cn$ since we need linear time to merge them together.

Therefore, the average time cost is:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} p(i)\left(T(i-1)+T(n-i)+cn\right) \\ &= \frac{1}{n}\sum_{i=1}^{n}(T(i-1)+T(n-i)+cn) \\ &= cn + \frac{2}{n}\sum_{i=0}^{n-1} T(i) \end{aligned}$$

Now we transform this into a calculable series problem. First multiple $n$ on both side, and substitute $n \to n-1$

$$\begin{aligned} nT(n) &= 2\sum^{n-1} T(i) + cn^2 \\ (n-1)T(n-1) &= 2\sum^{n-2} T(i) + c(n-1)^2 \end{aligned}$$

Subtract them, and divide $n(n+1)$:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1} \quad (\clubsuit)$$

Now we concentrate on solving this series ($\clubsuit$). It implies $\left\{\frac{T(i)}{i+1}\right\}$ is a near-arithmetic series.

$$\begin{aligned}\frac{T(n)}{n+1} &= \frac{2c}{n+1} + \frac{2c}{n} + \frac{2c}{n-1} + ... + \frac{2c}{3} + \frac{T(1)}{2} \\ &= \text{Const} + 2c\sum_{i=3}^{n+1}\frac{1}{i} \\ &= \text{Const} + 2c \cdot \log n + \gamma - \frac{3}{2}\end{aligned}$$

The last equality is due to Euler's summation techniques, and $\gamma = 0.5772...$ refers to Euler's constant. Then eliminate all the constant, we obtain:

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

Of course the expectation after randomizing is faster than $O(n^c)$ , $\forall c \in (1, 2)$

# 3 Find Local Minimum in Matrix

We will assume $n = 2^N$. Without loss of generality, we can complement $n$ to some power of 2 by **INT_MAX**. In 2-dimension case, which will be shown below, the $n = 2^N$ and $m = 2^M$ is not required. But we need to expand a circle outside with **INT_MAX** for convenience.

Now we firstly prove two Lemma that guarantee the existence of local minimum under border condition.

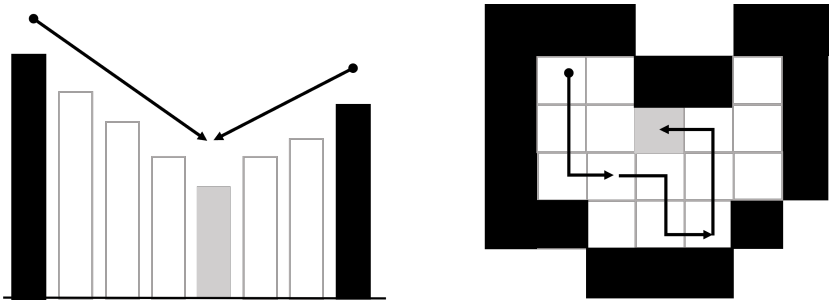


Figure 3.1: Existence of local minimum under bigger-border condition

**Lemma 1:** If a 1-dimensional space has bigger elements on its left and right side, there must exist a local minimum inside this space.

★ **Proof** ★

(Shown in 3.1 left)Suppose there's no local minimum. Because the left-side is bigger element, if the 1st one is not local minimum, the 2nd one should decrease a bit. Recursively, every element from left side should decrease one by one. Similarly, every emelent from right side should decrease one by one. When the two sequence meet at k-th element. $a_k < a_{k-1}$ and $a_k < a_{k+1}$, which means $a_k$ is a local minimum.
Which is contradictory to assumption, thus there must exist a local minimum.
■

**Lemma 2:** If a 2-dimensional space is surrounded by bigger elements, there must exist a local minimum inside this space(Not restrict to $m \times n$ space, but with any closed space).

★ **Proof** ★

(Shown in 3.1 right)Similarly as 1-dimensional case, suppose there's no local minimum. We start from any border. Since the border are bigger than the aside one, the inner layer should decrease a bit(to not let out layer be minimum). We follow this downstairs flow. The flow will at last end in somewhere because there are finite distinct numbers. That ending is the local minimum.
Which is contradictory to assumption, thus there must exist a local minimum.
■

## 3.1 1-Dimensional Case(m=1)

The 1D algorithm is shown as following pseudocode 1.

### 3.1.1 Correctness

The idea is check whether a position is local minimum from the center of array. What **Lemma 1** says is that: it's guaranteed the existence of a local minimum at any section with bigger number on both side.

Thus, if the center of array is not local minimum, we just check which side of it is smaller. **Choose that side, we get a $\frac{n}{2}$ length array still with bigger number on both side, which still guarantee the existence of local minimum.**

1D case: $m = 1$

```
0: function CHECK(l, r)
  if l equals r then
    return n[l]
  end if
  mid ← (l+r)/2
  if n[mid] is local minimum then
    return n[mid]
  else if n[mid-1] < n[mid+1] then
    return CHECK(l, mid − 1)
  else
    return CHECK(mid + 1, r)
  end if
  end function=0
```

Recursively, each time we will either find the local minimum, or get a smaller section with bigger number on both side, which is guaranteed to have local minimum inside.

In the end, when the length of section reach 1, it's guaranteed by previous induction that this single position is local minimum.

#### 3.1.2 Complexity

Check whether a single point is local minimum costs constant time $c$. And each time we will divide length by 2. Thus

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + c \times 1 \\ &= T(1) + c \times (1 + 1 + ... + 1) \\ \Rightarrow T(n) &= O(\log n) \end{aligned}$$

### 3.2 2-Dimensional Case(m=n)

The 2D algorithm is shown as following pseudocode 2. And 3.2 shows how to find next conquered area in general.

#### 3.2.1 Correctness

The **Lemma 2** implies that any area with bigger number in border, are certainly to have local minimum inside.

In order to find such sub-area, we divide $n \times n$ into 4 areas and check all the points on border(refers to border outside and middle row and middle column), as shown in 3.2.

If any points are local minimum, return it. Otherwise, we find the smallest points among border. **Consider the fact: ① it's not local minimum ② it's smaller than aside borders. ⇒ It's must bigger than at least one**

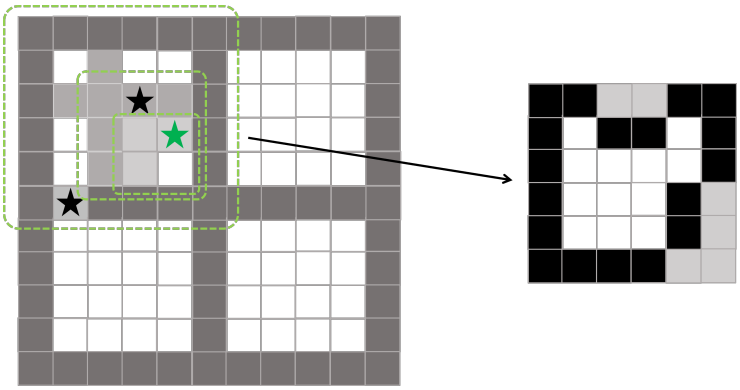


Figure 3.2: Recursively find smallest points in *critical_area*, and compare the two points aside to decide what's next area to be checked

**element inside area.** What's more, for every points on border, if it's not local minimum, it must smaller or bigger than the number inside the area. If the border is bigger, it's great that already build a 'wall'. If the border is smaller, we can push the border either inside or outside one cell to reach the 'wall'.

Finally, we will form a closed space like 3.2 shows. By the **Lemma 2**, it's guaranteed to have local minimum inside this area which has been narrowed down into $\frac{n}{2}$. In the end, when the size of section reaches $1 \times 1$, it's guaranteed by previous induction that this single position is local minimum.

### 3.2.2 Complexity

Since checking all the border, middle row and middle column will only cost $4(n-1) + 2(n-2) = c \cdot n$ linear times. And each time we will divide the problem into $\frac{n}{2}$ size, because we only check $\frac{1}{4}$ area next time.

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + c \cdot n \\ &= T(1) + c \cdot n \left(1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^{\log n}}\right) \\ &= C_1 + C_2 \cdot n \cdot (1 - \frac{1}{2^{\log n}}) \end{aligned}$$

As $n \to \infty, T(n) = O(n)$

2D case: $m = n$

```
0: function CHECK(r1, r2, c1, c2)
  if r1 equals r2 then
  return n[r1][c1]
  end if
  mid_row ← (r1+r2)/2
  mid_col ← (c1+c2)/2
  critical_area ← {Border, mid_row, mid_col}
  for point ∈ critical_area do
    if n[point] is local minimum then
      return n[point]
    end if
  end for
  // Here means no local minimum in critical_area
  smallest ← smallest point in critical_area
  Decide which side of smallest is smaller
  Go into that square:
  if 1-quadrant then

    return CHECK(r1, mid_row, mid_col, c2)
  else if 2-quadrant then

    return CHECK(r1, mid_row, c1, mid_col)
  else if 3-quadrant then

    return CHECK(mid_row, r2, c1, mid_col)
  else if 4-quadrant then

    return CHECK(mid_row, r2, mid_col, c2)
  end if
  end function=0
```

# 4 Out-of-Order Sequence

## 4.1 Prove that an integer triple $(x, y, z)$ forms a good order only if $x$ and $z$ are both even or both odd.

**★ Proof ★**

If $(x, y, z)$ forms a good order, then $y-x = z-y = \text{Const} \Rightarrow z = x+2\cdot\text{Const}$, which means $x$ and $z$ are both even or both odd.

■

## 4.2 Write down a permutation of $\{1, 2, 3, 4\}$ and quickly prove it is out-of order by the claim above.

**★ Proof ★**

For example, the permutation $\{1, 3, 2, 4\}$. Two odd number are put left-side, while two even numbers are put right-side. No matter how we randomly choose three numbers, it must be in the order that **Odd-Odd-Even** or **Odd-Even-Even**, which are both not good-order proved in 4.1. Thus the permutation is out-of-order.

■

## 4.3 Prove that an integer triple $(x, y, z)$ where $x, y, z$ are all odd forms a good order only if $(\frac{x+1}{2}, \frac{y+1}{2}, \frac{z+1}{2})$ forms a good order.

**★ Proof ★**

If $(x, y, z)$ forms a good order, consider that they are both odd $y - x = z - y = 2d$. Then we can note $y = x + 2d$ , $z = x + 4d$, and suppose $x = 2k - 1$
Then the chain is:

$$\left(\frac{2k}{2}, \frac{2k + 2d}{2}, \frac{2k + 4d}{2}\right) \Rightarrow (k, k + d, k + 2d)$$

which forms a good order.

■

## 4.4 Design a divide and conquer algorithm runs in $O(n \log n)$ time to output an out-of-order permutation of $1, ..., n$. (Can be improved to $O(n)$)

### 4.4.1 Algorithm and Correctness

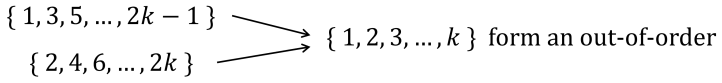



Figure 4.1: Divide the problem

We'd better not put odd and even numbers together, **therefore we can never satisfy the necessary condition for good-order: $\mathbf{x}, \mathbf{z}$ should be both odd or even**

Without loss of generality, suppose $n = 2k$ is an even number. We divide it into odd and even parts, shown as 4.1.

By question 4.3 we know that if we make $(\frac{1+1}{2}, \frac{3+1}{2}, ..., \frac{2k-1+1}{2})$ not a good-order sequence, then $(1, 3, ..., 2k-1)$ is not a good-order sequence as well.

In the same way, if we make $(\frac{2}{2}, \frac{4}{2}, ..., \frac{2k}{2})$ not a good-order sequence, then $(2, 4, ..., 2k)$ is not a good-order sequence as well.

And these two sequences are exactly $(1, 2, 3, ..., k)$, which can be solve by recursively call this function.

when $k \leqslant 4$, by question 4.2, we know that we can simple return $(1, 3, 2, 4)$ which is out-of-order.

### 4.4.2 Complexity

Each time we divide the $2k$ size problem into **one** $k$ size problem. After solving the $k$ size problem, we need linear time to compute the original sequence.

i.e. If we get the out-of-order permutation of $(1, 2, 3, ..., k)$ as $(a_1, a_2, ..., a_k)$, then the original odd out-of-order sequence will be $(2a_1 - 1, 2a_2 - 1, ..., 2a_k - 1)$, and the even out-of-order sequence will be $(2a_1, 2a_2, ..., 2a_k)$. This requires $O(n)$ time.

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + cn \\ &= T(4) + cn \cdot \left(1 + \frac{1}{2} + ... + \frac{1}{2^{\log n - 2}}\right) \\ &= 1 + c'n \left(1 - \frac{1}{2^{\log n}}\right) \\ \Rightarrow T(n) &= O(n) \end{aligned}$$

# 5 Reflection

4 hours in thinking. 1 hour in discussing and consulting references. 8 hours in drawing diagram and writing latex (somewhat due to OCD).

I suppose I will rank the difficulty to 4, especially as the first homework.

**Reference** Data Structures and Algorithm Analysis in C. Mark A.Weiss.

**Collaborator** Zhu Shengjia (Discussion for Question 3)

# Algorithm Design and Analysis 2024 Spring
# Assignment 2

1. (20 points) Please prove the Super Plan of the Strongly Connected Component Algorithm presented in the lecture is correct.

2. (20 points) Given a directed graph $G = (V, E)$, and a reward $r_v$ for all $v \in V$. The revenue of a vertex $u$ is defined as the maximum reward among all vertices reachable from $u$ (including $u$ itself). Design a linear time algorithm to output every vertex's revenue.

3. (20 points) 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 $t$. (A path fails if any edge on the path fails. The most reliable path means the path with the lowest failure probability.)

4. (20 points) Let $G = (V, E)$ be an undirected connected graph. Let $T$ be a depth-first search tree of $G$. Suppose that we orient the edges of $G$ as follows: For each tree edge, the direction is from the parent to the child; for every non-tree (back) edge, the direction is from the descendant to the ancestor. Let $G'$ denote the resulting directed graph.
   (a) (5 points) Give an example to show that $G'$ is not strongly connected.
   (b) (5 points) Prove that if $G'$ is strongly connected, then G satisfies the property that removing any single edge from $G$ will still give a connected graph.
   (c) (5 points) Prove that if $G$ satisfies the property that removing any single edge from $G$ will still give a connected graph, then $G'$ must be strongly connected
   (d) (5 points) Give an efficient algorithm to find all edges in a given undirected graph such that removing any one of them will make the graph no longer connected.

5. (20 points) Let $G = (V, E)$ be a graph and $s$ be a vertex such that there is a path from $s$ to each $u \in V$. We say $G$ is a *good graph* if there exists a tree $T = (V, E')$ that share the same vertex set $V$ with $G$ such that $T$ is both a depth-first search tree and a breadth-first search tree.

   (a) (5 points) If $G$ is an undirected graph, prove that $G$ is a good graph if and only if $G$ is a tree.

   (b) (10 points) If $G$ is a good directed acyclic graph, prove or disprove that the vertex set $V$ can be sorting in an array $\mathcal{L}$ such that $\mathcal{L}$ is both an ascending order of the distances from $s$ and a topological order.

   (c) (5 points) Prove or disprove the converse of (b). That is, if $G$ is a directed acyclic graph where the vertex set $V$ can be sorting in an array $\mathcal{L}$ such that $\mathcal{L}$ is both an ascending order of the distances from $s$ and a topological order, then $G$ is a good graph.

6. How long does it take you to finish the assignment (including thinking and discussing)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Write down their names here.

# AI2615 Homework 2

Tao Rui 522030910024

April 14, 2024

## 1 Prove the Super Plan of SCC

**Lemma:** If $SCC_1$ can reach $SCC_2$ in $G^R$, then $\max_{v \in SCC_1} finish[v] > \max_{u \in SCC_2} finish[u]$.

**★ Proof ★**

Firstly, the latest finish vertex in one SCC1 must be earliest start vertex. Since the first one is "entrance", the entrance cannot end before all vertices in SCC are done.
Suppose latest finish vertex in SCC1 is $x_1$ and SCC1 can reach SCC2 by path $u \to ... \to v$. No matter what the exploration like in SCC2, and no matter what other SCC that SCC2 can reach, they will not back to SCC1 until finish SCC2, otherwise there's a big ring.
Thus the finish time of $u$ is latest than SCC2, which implies:

$$\max_{v \in SCC_1} finish[v] = finish[x] > finish[u] > \max_{v \in SCC_2} finish[v]$$

■

We first DFS the $G^R$ and maintain a sorted list by the finish time. Then the latest finish vertex must in the head SCC of $G^R$, denote as $SCC_0$.

Then the second latest finish vertex in $SCC_1 \in \{V - SCC_0\}$ must be the head SCC of $\{G^R - SCC_0\}$. Because of **Lemma**, if $\max_{v \in SCC_1} finish[v]$ is the largest, then no other SCC can reach $SCC_1$, which implies $SCC_1$ is the head SCC of $\{G^R - SCC_0\}$.

Repeatedly doing this, we know that regardless of the head SCC found previously, the next latest finish vertex must in a SCC that no other SCC can reach, which is the next head SCC.

Thus the "super plan" algorithm keep exploring vertices by descending order of the finish time can indeed find all tail SCC of $G$ in order. By the tail SCC's order, simply doing DFS can find all SCC.

# 2 Directed Graph Vertice's Revenue

## 2.1 Intuition

What this problem wants us to output can be reansformed to: firstly find all SCC, then, for each SCC outputs the maximum revenue for every vetices in it.

The simple idea is we can firstly use the super plan in **Problem 1** to find the sequence of finish time. And use this sequence to do DFS to find all SCC component. During each finding, we can by the way record the maximum revenue of those vertices in same SCC.

---

**1: DFS set start/finish time**
- Use the Super Plan in **Problem 1** to obtain a sorted sequence of finish time.

**2: Compare max revenue in each SCC**
- DFS $G$ by the descending order of the sequence.
- If visited $v$, continue.
- Else, $CurrentMax = \max(CurrentMax, r_v)$, and record this vertex

**3: Output**
- For each SCC, print the vertices in it and the $CurrentMax$ of it.

---

## 2.2 Correctness

The "Super Plan"'s correctness has been proved by Problem 1, thus we have legally obtained the SCC components. In step 2, we DFS all the SCC (each time DFS the current tail SCC), during each SCC, we record the max revenue $r_v$ in this SCC by one-to-one compare. Finally, those maximum values can be output together with vertices in each SCC.

## 2.3 Complexity

The "Super Plan" will cost $O(|V| + |E|)$. The second turn of DFS will cost another $O(|V| + |E|)$. Comparing each revenue will be linea $O(|V|)$. Thus total time complexity is $O(|V| + |E|)$

# 3 Find Path with Lowest Failure Probability

The problem can be solved by Dijkstra with some marginal adaption. Note that $p(u, v)$ is the multiplication of all edges in the path between $u$ and $v$. What we need to find is the maximum of such multiplication.

**1: Initialize**
- $T \leftarrow \{s\}$
- $tdist[s] \leftarrow 0, tdist[v] \leftarrow \text{inf}$ for all $v$ other than $s$
- $tdist[v] \leftarrow p(s, v)$ for all $(s, v) \in E$

**2: Explore**
- Find $v \notin T$ with largest $tdist[v]$
- $T \leftarrow T + \{v\}$

**3: Update tdist**
- $tdist[u] = \min(tdist[u], tdist[v] + w(u, v))$ for all $(u, v) \in E$

**4: Repeat step 2 until T = V**

## 3.1 Correctness

The proof is exactly same as the proof of original Dijkstra, thanks to the important fact that **"Addition on $\mathbb{R}^+$" and "multiplication on $(\mathbf{0}, \mathbf{1}]$" has exactly same property in terms of monotonicity.** — they are monotonic increasing and decreasing respectively.

Prove by induction, suppose we start from $s$. If at some moment, the set $T$ contains all the maximized success possibility path. Then finding the vertex $x$ with max $tdist[x]$ (the max success possibility from $s$ to $x$), which connects some $t$ inside $T$ and $x$ outside $T$. Then the path $s \to ... \to t \to x$ must be global maximum. Because ① path that not pass $t$ is smaller since $tdist[x]$ is maximum at this round. ② path that pass other $x_1$ to $x$ will be smaller since the path to $x_1$ is already larger than $s \to ... \to t$.

## 3.2 Complexity

If we don't use any heap to optimize the sorting process, our original complexity is shown bellow:

- Step 2: Cost from $O(|V| - 1)$ to $O(1)$ in each iteration.
- Step 3: Cost $O(1)$ to update.
- Step 4: The finding minimum operation will cost $O(|V|)$ rounds. The updating operation will cosr $O(|E|)$ round iteration.

Thus the total time complexity is $O(|V|^2 + |E|)$

# 4 Connected Graph DFS Tree

## 4.1 Counter example that $G'$ not strongly connected

If we give a undirected graph as following (Eliminate all the arrow will be the case). Firstly, it's easy to check the graph $G$ is connected. But if we start exploring from Ⓐ and form the DFS tree as follow, it's obvious that Ⓔ, Ⓕ cannot back to Ⓐ. Thus $G'$ is not strongly connected.

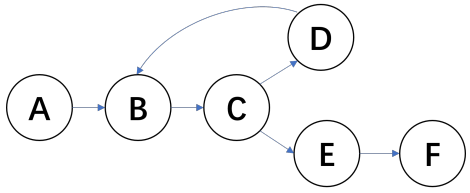



Figure 4.1: Directed $G'$ of undirected counter example

## 4.2 Property: removing any single edge from $G$ is still connected

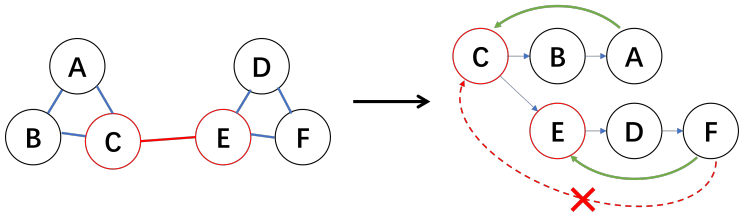



Figure 4.2: Proof by Contradiction

Now prove from contradiction: If there's an edge that can destroy connection. Like Ⓒ-Ⓔ as shown. Then we can explore the DFS tree from one of the two points, for example Ⓒ.

Consider two fact: (1). The two part is connected respectively (2). Beside Ⓒ-Ⓔ, there's no edge connecting two part, otherwise $G$ is still connected.

Therefore, **there's no back-edge from the $\{\mathbf{E}, \mathbf{D}, \mathbf{F}\}$ part to vertex Ⓒ**. Thus in the directed graph $G'$, any starting vertex from $\{E, D, F\}$ cannot reach any vertex in $\{A, B, C\}$. $G'$ Not strongly connected, which leads to contradictory. We prove the property must to be right.

## 4.3 Prove the Reversed Property

We find that in 4.1 even though Ⓓ-Ⓑ is connected by a back-edge, Ⓓ is still isolated from Ⓐ. That's because Ⓑ cannot reach Ⓐ as well, so Ⓓ's path

ends here. This above intuition reminds us to check the connectivity from leaf to root.

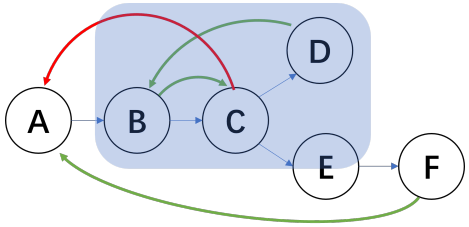



Figure 4.3: Intuition of how to maintain $G'$ strongly connected

From any leaf, such as Ⓓ. To reach the root vertex, it has to need to climb up the tree by connecting to any ancestor. **If no back-edge connects the leaf, then delete its tree-edge, it will be isolated in original undirected graph, which contradicts to the condition that graph should still be connected after deleting one edge.**

Now the leaf linked by a back-edge to some ancestor above on the DFS tree, such as Ⓑ. **Then all the vertices under Ⓑ to leaf can be reached by each other.**

So we can regard all these vertices beneath as a "super vertex"(Shown by shadow). **And this super vertex must have a back-edge from inside to some ancestor above.** Otherwise, delete this super vertex's highest tree-edge, then it's isolated by the above tree, which contradicts to the condition that this undirected graph should be connected after deleting one edge.

Repeat this process, each time enroll the vertices that beneath the back-edge's ending into "super vertex". Until reach the root.

**As conclusion, we prove that for any DFS tree, there must exist a path from any leaf to root.** Thus all the vertices are reachable by each other, since they can simply reach leaf first, then back to root by the path proved above, and go down the DFS tree to the target vertex. $\Rightarrow G'$ is stongly connected.

## 4.4 Give an Algorithm to Find All Single Cut-edge

### 4.4.1 Intuition

The intuition is that we can iteratively shrink vertices from below, and whenever we find no back-edge point to ancestor, the tree-edge here is a single cut-edge.

What we maintain is a recorder of what's the earliest ancestor this "super vertex" can reach.

### 4.4.2 Correctness

By the recorder's definition, the correctness is easy to prove.

**1: DFS Explore**
- start DFS explore from given vertex $u$
- If the vertex is not reached, set $start[u] \leftarrow time$ && $recorder[u] \leftarrow time$ && $time++$
- Keep DFS first.

**2: Update $recorder[u]$ from below**
- For all neighbors $v$ of $u$ doing the following updating
- If $v$ is father of $u$, continue
- If $v$ is not father of $u$ but visited, $recorder[u] \leftarrow \min(recorder[u],\ start[v])$
- Else, $recorder[u] \leftarrow \min(recorder[u],\ recorder[v])$

**3: Check single cut-edge**
- For all edges $(u, v)$, check if $start[u] < recorder[v]$, then it's a single cut-edge.

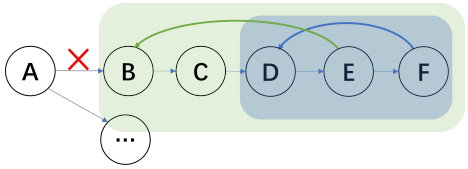



Figure 4.4: How cut-edge takes shape

For all sons $v$ of vertex $u$. If $recorder[v] \leqslant start[u]$, means from this neighbor $v$ can reach their father or earlier ancestor by some path(not passing the edge connecting $v$'s father), which further implies this can be shrinked into a new "super vertex".

If $recorder[v] > start[u]$, then from $v$ can only reach $recorder[v]$ where is later than $u$, which further means $v$ cannot reach $u$ and above. In this situation, cut off the edge $(u, v)$ can isolate the graph at least as $u$-part and $v$-part. $(u, v)$ is a single cut-edge.

### 4.4.3 Complexity

The total DFS will cost $O(|V|+|E|)$. Total updating *recoder* will be complex as total edge number, $O(|E|)$. Thus total algorithm's complexity is $O(|V|+|E|)$

# 5 Good Graph

## 5.1 Undirected $G$ is Good Graph iff $G$ is Tree

① If $G$ is a tree, which means edge number is $|V| - 1$ and there's no rings in graph.

Given a starting vertex, the BFS tree is fixed. Consider that this is a tree, **there will be no cross-edge or back-edge in any kind of exploring.**

Therefore, the DFS exploring of this BFS tree is the same, since no "fork junction" exist.

② If $G$ is a good graph. Let's firstly generate a BFS tree as following (since BFS tree is fixed with given starter).

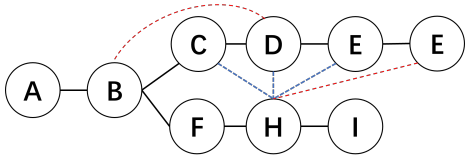



Figure 5.1: BFS Tree's Edge Property

Now consider how the DFS exploration on this graph can result in different tree? **There must exist some edge that links two different branch, so that DFS will merge these two branch.**

Now discussion on edge's type:

- back-edge : Which will not exist in undirected graph's BFS tree.
- front-edge : Which is same as back-edge in undirected graph.
- cross-edge : As 5.1 shown, the cross-edge can only exist between two vertices that $|\text{height}(u) - \text{height}(v)| \leqslant 1$, otherwise the BFS process will reach by this shorter path.

As conclusion, there may be some cross-edge in the BFS tree. However, if a cross-edge exist between two branches SubTree($u$) and SubTree($v$), then the DFS process will merge these two brach, since they are reachable by each other. Which means if there exist any cross-edge, the DFS tree is certainly not same as BFS tree.

## 5.2 Counter Example for (b)

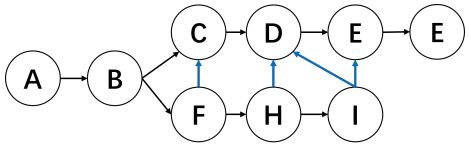



Figure 5.2: All cross-edge must have same direction

**Intuition:** The property of good graph in DAG.

Before giving the counter example of (b)(c), we first think of what does good-DAG look like.

Similarly as (a), we first generate a BFS tree due to the excellent property that it's fixed by given starter. Then we make discussion on edge's type :

- back-edge : Which will not exist in directed graph without rings.
- front-edge : Which will not exist, otherwise the BFS path will be this front-edge.
- cross-edge : The cross-edge can still just exist between two vertices that $|\text{height}(u) - \text{height}(v)| \leqslant 1$, otherwise the BFS process will reach by this shorter path. **However, there can be two directions of cross-edge,** from $u \to v$ or $v \to u$

The last difference is important. Since it allows the BFS tree of DAG to have cross-edge — **Only in the condition that all the cross edge between two branches SubTree**$(u)$ **and SubTree**$(v)$ **are be same direction.**

As shown in 5.2, although if we do the DFS and explore Ⓑ-Ⓕ first, we will still merge these two branch. But we can choose to explore Ⓑ-Ⓒ first, which still maintain two isolated branches.

■

Now we can introduce our counter example. All the construction idea is from intuition above.

If G is a good directed acyclic graph, what we get from intuition is that the BFS tree can still have cross-edge, as long as we guarantee all the cross-edge are same direction.

So how to contradict the ascending order of the distances from $s$ ? **We just link the deeper point to shallower point, then the topological order will be different from distance order.**

The example 5.3 shows a BFS tree, if we firstly explore Ⓐ-Ⓑ in DFS, then the DFS tree is exactly same as BFS tree, since Ⓑ branch have no cross-edge to Ⓒ branch. Thus 5.3 shows a good graph.

But If we write out it's **one and only** topological order: $A \to C \to E \to B$, we find that it's not an ascending order of the distances from Ⓐ, since Ⓑ should be closer than Ⓔ

## 5.3 Counter Example for (c)

if G is a directed acyclic graph where the vertex set V can be sorting in an array L such that L is both an ascending order of the distances from Ⓐ and a

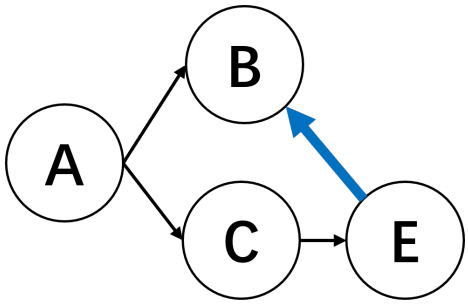



Figure 5.3: Counter Example for (b)

topological order. What lesson we learn from (b) is that there cannot exist a edge that linking the deeper point to shallower point.

Combining this with the intuition, **we can add cross-edge in different direction but in the same height.**

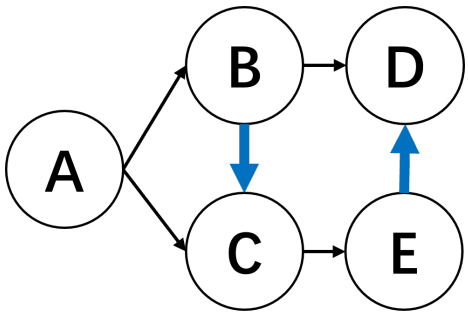



Figure 5.4: Counter Example for (c)

As shown in 5.4, the DAG's one and only topological order is $A \to B \to C \to E \to D$, and this order satisfy the ascending order of the distances from Ⓐ as well, so it satisfy the condition.

**But since it has two directions of cross-edge, no matter which branch to choose in a DFS exploration, the two branch is certainly to be merged.** So it's not same as BFS tree and not a good graph.

# 6 Reflection

Can't remember total time cost, too much.

The difficulty may be ranked as 4. The proof in graph is seemingly more logical than others; but workload is too hard this time.

**Reference** Consulting https://oi-wiki.org/graph/cut/ for $Q4$ ($d$)

**Collaborator** Inspired by Zhu Shengjia for $Q5$, and form my own explanation.

# Algorithm Design and Analysis

# Assignment 3

# Deadline: April 21, 2024

1. (30 points) You are given $n$ jobs, where each job $j$ has its processing time $p_j$ and weight $w_j$. We need to process all of them on one machine. We use $C_j$ to denote $j$'s completion time in a schedule. Our goal is to find the best schedule that minimizes the average weighted completion time (i.e., min $\frac{\sum_{j=1}^{n} w_j C_j}{n}$). Design a greedy algorithm for it.

2. (30 points) **Amortized Cost of ADD.** Let us consider the following situation. An integer (initially zero) is stored in binary. We have an operation called ADD that adds one to the integer. The cost of ADD depends on how many bit operations we need to do. (one bit operation can flip 0 to 1 or flip 1 to 0.) The cost can be high when the integer becomes large. Use amortized analysis to show the amortized cost of ADD is $O(1)$. Although there are different ways to prove it, you must use the potential function method for this question.

**Algorithm 1** Find a maximal independent set with maximum weight

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

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

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

3. (40 points) In the class, we learned Kruskal's algorithm to find a minimum spanning tree (MST). The strategy is simple and intuitive: pick the best legal edge in each step. The philosophy here is that local optimal choices will yield a global optimal. In this problem, we will try to understand to what extent this simple strategy works. To this end, we study a more abstract algorithmic problem of which MST is a special case.

   Consider a pair $M = (U, \mathcal{I})$ where $U$ is a finite set and $\mathcal{I} \subseteq \{0,1\}^U$ is a collection of subsets of $U$. We say $M$ is a *matroid* if it satisfies

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

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

   (a) (8 points) Let $M = (U, \mathcal{I})$ be a matroid. Prove that maximal independent sets are of the same size. (A set $A \in \mathcal{I}$ is called *maximal* if there is *no* $B \in \mathcal{I}$ such that $A \subsetneq B$.)

   (b) (8 points) Let $G = (V, E)$ be a simple undirected graph. Let $M = (E, \mathcal{S})$ where $\mathcal{S} = \{F \subseteq E \mid F \text{ does not contain a cycle}\}$. Prove that $M$ is a matroid. What are the maximal sets of this matroid?

   (c) (8 points) Let $M = (U, \mathcal{I})$ be a matroid. We associate each element $x \in U$ with a nonnegative weight $w(x)$. For every set of elements $S \subseteq U$, the weight of $S$ is defined as $w(S) = \sum_{x \in S} w(x)$. Now we want to find a maximal independent set with maximum weight. Consider the following greedy algorithm.

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

(d) (8 points) Prove that the greedy algorithm returns a maximal independent set with maximum weight. (Hint: Can you see that Algorithm 1 is just a generalization of Kruskal's algorithm?)

(e) (8 points) Let $U \subseteq \mathbb{R}^n$ be a finite collection of $n$-dimensional vectors. Assume $m = |U|$ and we associate each vector $\mathbf{x} \in U$ with a positive weight $w(\mathbf{x})$. For any set of vectors $S \subseteq U$, the weight of $S$ is defined as $w(S) = \sum_{\mathbf{x} \in S} w(\mathbf{x})$. Design an efficient algorithm to find a set of vectors $S \subseteq U$ with maximum weight and all vectors in $S$ are linearly independent.

4. How long does it take you to finish the assignment (including thinking and discussing)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Write down their names here.

# AI2615 Homework 3

Tao Rui 522030910024

May 5, 2024

## 1 Job Assignment

### 1.1 Intuition and Algorithm

We start from computing the neighbouring tasks. Suppose their already exist $T$ time before the task $(w_i, p_i)$ and $(w_j, p_j)$. We denote $Y$ as the final cost.

$$Y(i \text{ before } j) = w_i(T + p_i) + w_j(T + p_i + p_j)$$
$$Y(i \text{ after } j) = w_j(T + p_j) + w_i(T + p_j + p_i)$$

The situation that **i before j** is optimal is: $w_j p_i - w_i p_j \leqslant 0$, i.e. $\dfrac{w_i}{p_i} \geqslant \dfrac{w_j}{p_j}$

**Therefore we obtain a intuition of how a task can be regarded as "Better" – it has a larger $\dfrac{\mathbf{w}}{\mathbf{p}}$**

Now we construct the order of finishing job from random start. For any order, if there exist any two **neighbouring tasks** $i$ and $j$ such that $\dfrac{w_i}{p_i} \leqslant \dfrac{w_j}{p_j}$, we can simply exchange these two order. It reduce $Y$ by $w_j p_i - w_i p_j$, and more importantly, it does no impact to following tasks, since they still cost $p_i + p_j$ as a whole.

Thus we repeat this "Bubble-sort" like operation. We finally obtain a strictly decreasing order of $\left\{\dfrac{w}{p}\right\}$.

### 1.2 Correctness

Now we prove the optimization of this decreasing order tasks.

① Firstly, there must be no "neighbouring increasing tasks", proved by intuition. What's more, there must be no "increasing tasks" with gap: If we do three tasks in order $(i,\ j,\ k)$, but with $\dfrac{w_k}{p_k} \geqslant \dfrac{w_i}{p_i} \geqslant \dfrac{w_j}{p_j}$. Then we can change the order $(j,\ k)$, then $(i,\ k)$, by the intuition's supporting respectively. Finally, the stable order will always be strictly decreasing in $\left\{\dfrac{w}{p}\right\}$.

② Secondly, if there's some tasks with same $\frac{w}{p}$, we claim that the order doesn't matter. Consider two tasks $(i,\ j)$, with a time gap $t_2$ and several tasks weight $w_2$ in general.

$$Y_1 = w_i(t_1 + p_i) + w_2(t_1 + p_i + t_2) + w_j(t_1 + p_i + t_2 + p_j)$$
$$Y_2 = w_j(t_1 + p_j) + w_2(t_1 + p_j + t_2) + w_i(t_1 + p_j + t_2 + p_i)$$
$$\Rightarrow \Delta Y = w_2(p_i - p_j) + t_2(w_j - w_i) + (w_j p_i - w_i p_j) \quad (\clubsuit)$$

Notice the fact that $\frac{w_i}{p_i} = \frac{w_j}{p_j}$, **and also holds for all task between** $(i,\ j)$. **Thus we can merge all tasks in $\mathbf{t_2}$ to a "big task" with $\mathbf{w' = w_2},\ \mathbf{p' = t_2}$.** What's more, $\frac{w_i}{p_i} = \frac{w_j}{p_j} = \frac{w'}{p'}$, denote the ratio as $k$

$$(\clubsuit) \Rightarrow \Delta Y = (w' - p'k)(p_i - p_j) + (w_j p_i - w_i p_j) = 0 + 0$$

Which implies that tasks with same $\frac{w}{p}$ can be arbitrarily ordered.

Combining ① and ② we prove the correctness of this "bubble like" algorithm, such that optimal order should be decreasing order of $\frac{w}{p}$.

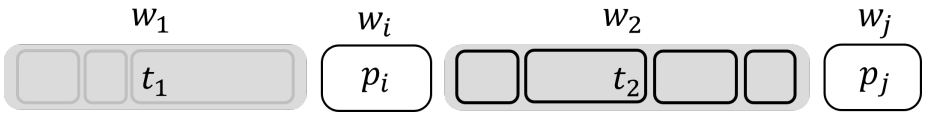



Figure 1.1: Tasks with same $\frac{w}{p}$ - Merge into Big Task with Same $\frac{w}{p}$

### 1.3 Complexity

All we need to do is to sort the sequence of $\left\{\frac{w}{p}\right\}$ in decreasing order. Since we can calculate all $\frac{w}{p}$ in $O(n)$, and sort them in $O(n \log n)$, the total time cost is $O(n \log n)$.

## 2 Amortized Cost of ADD

We define the potential function $E(x)$ for number $x$ as the number of 1. Since 1 is the cause of additional carry, we should punish the number of 1 in potential function.

Suppose $E(x) = k$, and $k$ is made up of $(k_l,\ k_{l-1},\ ...,\ k_1)$ separated 1s. Now doing the $ADD$ will cost $k_1 + 1$ times, but also eliminates $k_1$ 1s and add one more 1 due to carry. Thus the total cost is $k_1 + 1 - (k_1 - 1) = 2$. Thus amortized cost for $n$ times $ADD$ is $2 \times n = O(n)$. Each operation takes $O(1)$.

# 3 Matroid

## 3.1 All Maximal Independent Sets Are of Same Size

It's trivial by exchange property. Suppose $A$, $B$ are two independent sets but $|A| < |B|$, then $\exists x \in B - A$ , $A \cup \{x\} \in \mathcal{I}$, which implies that $A$ is not maximal. We can repeat this process until every independent sets are of same size.

## 3.2 Matroid in Graph

We need to prove the set of "None cycle edges" is independent set.

① **hereditary property** For all $A \in \mathcal{I}$ and $B \subseteq A$. Since $A$ contains none-cycle edge, any subset of these edges must not form cycle, too. Therefore the subset $B$ still contains a set of none-cycle edges, which makes $B \in \mathcal{I}$.

② **Exchange property** Suppose there are two independent sets $A$, $B$ and $|A| < |B|$. Since there is no cycle, the $|\mathbf{A}|$ edges must be a tree that connects $|A| + 1$ vertices. **So set B at least connects one more vertex $x$ than A**. Choose this $x$, and the edge that connects $x$ in $B$ (Denote as $e$). We claim that $A \cup e$ still forms no cycle. — Because if adding $x$ to the connect component results a cycle, there must exist a path to $x$ before, which contradictory to the fact that $A$ doesn't connects $x$ previously.

Combine ①②, $(E, \mathcal{S})$ is matroid.

## 3.3 Prove the Greedy Algorithm – Part 1

| | $w_1$ | $w_2$ | $w_3$ | $w_4$ | ⋯ | $w_k$ |
|---|---|---|---|---|---|---|
| $S$ | √ | Remain n-1 elements | | | | |
| $S'$ | × | Remain n elements | | | | |

Figure 3.1: Maximum Maximal $S'$ must contain largest $w_1$

(All weights are already in decreasing order ) Suppose there's a maximal set $S'$ without the largest weight element $w_1$. Then add $w_1$ into $S'$, we know that $S' \cup \{w_1\} \notin \mathcal{I}$. Now consider which elements together causing this violation of $\mathcal{I}$? It must be a set of elements $\{w_1,\ w_{j_1},\ w_{j_2},\ ...,\ w_{j_l}\}$. What's more, **removing one of these elements $S' \cup \{w_1\} - w_i$ can make it belongs to $\mathcal{I}$ again**.

That's because if we have to remove two or more elements, the whole independent set's size is smaller than $|S|$. By exchange property, we can add other element in then.

Therefore, after adding $w_1$ into $S'$, we can remove one other element to make new set $\in \mathcal{I}$ again, but the total weights increases. Thus the maximum weight $w_1$ must exist in maximum maximal $S'$.

**For Specific Graph Matroid – MST**
Denote the edge with max weight as $e_0$, and $e_0$ connects $(v_1,\ v_2)$. If $e_0$ doesn't exist in final set, then there must be another path that connects $(v_1,\ v_2)$, for example $v_1 - u - ... - v_2$. Then replace $(v_1, u)$ by $e$, the whole edge sets still forms no cycle, and is still connected, but the total weight is increasing by $weight(e_0) - weight((v_1, x))$. So the maximum maximal $S'$ must contain $e_0$.

## 3.4 Prove the Greedy Algorithm – Part 2

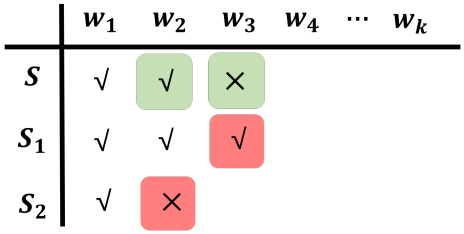



Figure 3.2: The Greedy Strategy is Optimal

We know that the largest element is certain to chose, now consider the first element that differs from Optimal $S$

Which has two possible situations:

**$S$ doesn't choose but $S_1$ chooses** This is impossible, since why $S$ doesn't choose $w_i$ is because $w_i$ violates the requirements of $\mathcal{I}$. So that if $S_1$ chooses $w_i$, $S_1 \notin \mathcal{I}$.

**$S$ chooses but $S_2$ doesn't choose** Similarly with 3.3, we add $w_i$ to $S_2$, which makes $S_2$ not belongs to $\mathcal{I}anymore$. Consider the fact that $w_i$ is chose in optimal set $S$, $w_i$ **is certain not violate the requirements with** $\{w_1,\ w_2,\ ...,\ w_{i-1}\}$. So the violation must be with at least one element in $\{w_j\},\ j > i$. Then we delete $w_j$, $S_2 \cup \{w_i\} - \{w_j\} \in \mathcal{I}$, and with a larger weights. Therefore $S_2$ is not maximum maximal set.

Concluding two situations, we know that the $S$ generating by such greedy strategy is with maximum weights.

## 3.5 Matroid in Matrix

First prove the "linear independent" relation of vectors leads to a matroid $(U, \mathcal{I})$, where $\mathcal{I} = \{F \subseteq U \mid F \text{ is linear independent}\}$.

① **hereditary property** It's trivial that a subset of several linear independent vectors are still linear independent.

② **Exchange property** If $|A| < |B|$, we know that the space rank of $A$ is smaller than space rank of $B$. Thus there must exist some vector in space $B$ that $A$ cannot represent. Otherwise, all the vectors in $B$ can be represented by $A$, which leads to the contradiction that $B$ is independent and has bigger rank than $A$.

Therefore $(U, \mathcal{I})$ is a matroid. We can simply apply **Algorithm 1**.

**Time Complexity** Sorting $|U|$ weights cost $O(m \log m)$. Each time we need to find out whether adding a new vector still holds linear independent, which can be checked by Gaussian elimination. In $m$ loops, total Gaussian elimination cost is $1^3 + 2^3 + ... + n^3 = O(n^3)$, until we already choose $n$ vectors. Therefore, the plain implementation is $O(m \log m + m \times n^3)$

# 4 Reflection

Subjectively speaking, the difficulty has slightly decreased than previous homework, I will rank 3 this time. But writing Latex still cost over 5 hours.

The matroid has been shortly introduced in Discrete Math, as a bonus homework. (Although that has nothing to do with problem 3...)

**Reference** None

**Collaborator** None

# Algorithm Design and Analysis

# Assignment 4

# Deadline: May 19, 2024

1. (30 points) Let $G$ be a tree with $n$ vertices. In this problem, we assume that it takes $O(1)$ time to store and multiply two integers. A vertex cover of a graph is a set of vertices that includes at least one endpoint of every edge of the graph.

   (a) (15 points) Design an $O(n)$ time algorithm to count the number of vertex covers in $G$. Prove the correctness of your algorithm and analyze its time complexity.

   (b) (15 points) Design an efficient algorithm to count the number of *minimum* vertex covers in $G$. Prove the correctness of your algorithm and analyze its time complexity.

2. (40 points) **Collecting Points.** You have an array of integer points $x_1, x_2, \ldots, x_n$, where each $x_i$ can be negative. Your goal is to move along the array to collect points, following specific rules:

   - In the first round, you choose the index to start.
   - In subsequent rounds, you can move to the right for a selected length or stop collecting. That is, assuming you are originally standing at index $i$, you can select a move length $k$ and move to $i + k$.
   - The objective is to maximize the total collected points.

   (a) (10 points) Single-Length Move: Develop an $O(n)$ DP algorithm to maximize the collected points when only allowed to move one step to the right in each round.

   (b) (15 points) Variable-Length Move: Extend the game to allow moves of length $k$ in the range $[A, B]$ in each round, where $A$ and $B$ are two input positive integers. Propose an $O(n^2)$ DP algorithm to maximize the collected points.

   (c) (15 points) Optimized Algorithm: Enhance the algorithm's efficiency to $O(n)$.

3. (30 points) **Min Cost Guideline.** Consider $n$ different ordered elements $X = \{x_1 \leq x_2, ... \leq x_n\}$, and we are given the searching frequency of each element, denoted by $a_1, a_2, ..., a_n$. We plan to design a search guide for users, which consists of a start point $c_1 \in X$ and two functions $S(c) : X \to X$ and $L(c) : X \to X$, where $S(c) < c$ and $L(c) > c$. When users follow the guide to search $x_j$, they will first check $c_1$; if $x_j = c_1$, they succeed. Otherwise, they should move to $c_2 = S(c_1)$ if $x_j < c_1$; or $c_2 = L(c_1)$ if $x_j > c_1$. Then, they repeat the process by comparing $x_j$ and $c_2$. Finally, for each specific $x_j$, users will have a specific checkpoint sequence $c_1, c_2, c_3, ..., c_k = x_j$. We use $\ell(j)$ to denote the length of the sequence, and we aim to design a good search guide to minimize $\sum_{j=1}^{n} w_j \cdot \ell(j)$ by an efficient DP algorithm.

4. How long does it take you to finish the assignment (including thinking and discussing)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Write down their names here.

# AI2615 Homework 4

Tao Rui 522030910024

May 19, 2024

## 1 Minimum Vertex Cover

### 1.1 Count Number of Vertex Covers in $G$

#### Algorithm and Correctness

Since $G$ is a tree, if a parent is not chosen in set, it's children has to be chosen. Thus we can divide the problem by dividing the situation into two sub-situation: Choose Root or NOT Choose Root.

If we choose the root, the sub-problem will become exactly same problems with all children, since the children can be chosen or not arbitrarily, just like a new problem.

If we do not choose the root, all the children have to be chosen. Thus the grandchildren are free to be chosen or not, the sub-problems become the new problems with grandchildren.

We gradually obtain the basic case: If a node is leaf, the number of vertex number is 2.

What's important here is how to combine those sub-problems: If a node has $k$ children, with $n_1, n_2, ..., n_k$ vertex covers respectively, and each child has $k_i$ children, with $n_{11} \sim n_{1k_1}, ..., n_{k1} \sim n_{kn_k}$ vertex covers respectively, then the vertex cover number of this node is $\prod_{i=1}^{k} n_k + \prod_{i=1}^{k} \prod_{j=1}^{k_i} n_{ij}$.

We can simplify the computing by storing each node's vertex cover number. Each node can be regarded as a state, $N(v)$ refers to the number when $v$ can be arbitarily chosen.

The topological order is from leaf to above, since $N(v)$ requires $N(x)$ for all $v$'s children and grandchildren.

The basic case: For all leaves, $N(leaf) = 2$, and for all parents of leaves, $N(leaf'sparent) = \prod_{l} N(l) + 1 = 2^{\#\text{Leaf}} + 1$

The pseudocode is as follow. (From code point of view, the $\prod$ can be done by $for$ loop, initializing the multiplication by 1. This can deal with the situation when one node doesn't have grandchildren. The $\prod$ will be 1 in that situation.)

- Record the vertex cover number of each node's subtree in list $number[v]$.
**Do the Post-order Traversal:**
- If the $v$ is leaf, $number[v] \leftarrow 2$
- else, $number[v] \leftarrow \prod_{i=1}^{k} number[v_k] + \prod_{i=1}^{k} \prod_{j=1}^{k_i} number[v_{ik_j}]$

**Return** $number[root]$

### Complexity

The total traversal in tree costs $O(n)$. Each node's vertexx conver number will only be charged twice, since it can only be children of one node and grand children of one node. So total time cost is $O(n)$.

## 1.2 Count Number of Minimum Vertex Covers in $G$

### Intuition

Suppose we have a minimum vertex cover set. From greedy point of view, **no matter whether we choose the last third node in set, we can always abandon the leaves and choose leaves' parents**. Since $\#leaves \geqslant \#leaves'\ parent$, such exchange is always better.

Therefore, no leaves should be chosen in OPT, all last second layer should be chosen instead.

From the edges point of view, **if the last second layer nodes are all chosen, we can simply regard the last third layer as leaf situation**. And we repeatedly doing this greedy choice.

### Algorithm

- Record the **min-vertex-cover number** of each node's subtree in list $number[v]$.
- Record the **vertex number** of each node **if not choose** $v$ in list $nc[v]$.
- Record the **vertex number** of each node **if choose** $v$ in list $c[v]$.
**Do the Post-order Traversal:**
If $nc[v] == c[v]$: $number[v] = \prod_{i=1}^{k} number[v_k] + \prod_{i=1}^{k} \prod_{j=1}^{k_i} number[v_{ik_j}]$
If $nc[v] > c[v]$: $number[v] = \prod_{i=1}^{k} \prod_{j=1}^{k_i} number[v_{ik_j}]$
If $nc[v] > c[v]$: $number[v] = \prod_{i=1}^{k} number[v_k]$

**Return** $number[root]$

**Complexity**

Same as 1.1, if we compute all the states from below to above(from leaves to root), we will visit each state twice. Since each state updating takes $O(3) = O(1)$ times on max, total time cost is $O(n)$.

# 2 Collecting Points

## 2.1 Single Length Move

**Intuition**

Notice that the total collected number summation can be determined by ① Start Point and ② End Point. Thus we can encode each state by End Point, and traverse all Start Point to maximize total summation.

**Algorithm and Correctness**

Define `dp[i]` as the maximal summation when ending at $i$. It's apparent that `dp[0] = num[0]`, and $\max_{i\in\{1,..,n-1\}} dp[i]$ is the answer.

**If we stop at $i$, we have only two possible ways can lead we here:** ① We start at $i$. ② We come from `num[i-1]` (Since we can only move one step each time).

Therefore the transmission function is: $dp[i] = \max\{num[i], dp[i-1] + num[i]\}$. Which can be rewrite into more expand-friendly way:

$$dp[i] = num[i] + \max\{0,\ dp[i-1]\}$$

Above claim guarantees the correctness of this DP algorithm, since each state's max has considered every possible situation.

This DP-algorithm can be calculated easily, since the topological order is simply from index 0 to $n-1$.

---

- $dp[0] \leftarrow num[0]$
**For** i **in** {1,...,n-1}:
- $dp[i] \leftarrow num[i] + \max\{0,\ dp[i-1]\}$
**Return** $\max_{i\in\{0,..,n-1\}} dp[i]$

---

**Complexity**

Determining $n$ DP states costs $O(n)$ time, since each state only compares two numbers' maximal relation. At last outputting $\max dp[i]$ still cost $O(n)$. Thus total time complexity is $O(n)$.

## 2.2 Variable Length Move

### Algorithm and Correctness

Similarly, the transmission idea is same. The only difference is: If we end at $i$, we may ① Start at $i$, or ② Come from $[i-B, i-A]$, since the last move ranges in $[A, B]$.

Therefore, we simply change the range of max, Anything else is same.

$$dp[i] = num[i] + \max_{i-B \leqslant j \leqslant i-A} \{0, dp[j]\}$$

Above claim still guarantees the correctness of this DP algorithm, since each state's max has considered every possible situation.

---

- $dp[0] \leftarrow num[0]$
**For** i **in** {1,...,n-1}:
- $dp[i] \leftarrow num[i] + \max_{i-B \leqslant j \leqslant i-A} \{0, dp[j]\}$
**Return** $\max_{i \in \{0,..,n-1\}} dp[i]$

---

### Complexity

In $n$ loops, we need compare $B-A$ numbers to determine which one is maximum. Thus total time cost is $O(nk)$, where $k$ may range from 1 to $n$. The algorithm may perform as $O(n^2)$.

## 2.3 Improve by Monotone Queue

### Algorithm

What we need in calculating each state is to find maximum among $k$ numbers. These $k$ numbers will update only one of them each time.

Using monotone queue, we abandon those smaller $dp[i]$ with earlier position in available range, since they will be out of range first and never be used.

---

- $dp[0] \leftarrow num[0]$
- $(sm[i](i \in [0, A-1]) = dp[i])$
**For** i **in** {1,...,n-1}:
- **Append back** $dp[i-A]$ to $sm$
- Start comparing from back to front: If $dp[i-A] \geqslant sm[-1]$, **Pop back** $sm$
- $dp[i] \leftarrow num[i] + \max\{0, sm[0]\}$
**Return** $\max_{i \in \{0,..,n-1\}} dp[i]$

---

**Complexity**

Getting the available maximum, which is at the front of the queue, takes $O(1)$. Since each index will only be added to queue once, and will only be popped once, total number's comparison takes $O(n)$. Thus total complexity is $O(n)$.

# 3 Min Cost Guideline

## 3.1 Algorithm

It's reasonable that the average length of checking sequence can be determined by three things: ① Starting point. ② Which Section is in. ③ The jump function $S(c_i), L(c_i)$. So naturally we encode states by two descriptors: Section Left & Right and Start Point. i.e. we define $AverageLength(l,\ r,\ i)$ as the weighted average length of checking elements starting from $x_i$ in $\{x_l, ..., x_r\}$.

The algorithm is originated from the following transition equation:

$$\begin{aligned} &AverageLength(l,\ r,\ i) = \\ &\quad a_i \times 1 + \left(\sum_{k=1}^{i-1} a_k\right) \min_{S(x_i)\in\{x_l,\ldots,x_i\}} (1 + AverageLength(1,\ i-1,\ S(x_i))) \\ &\quad + \left(\sum_{k=i+1}^{n} a_k\right) \min_{L(x_i)\in\{x_i,\ldots,x_r\}} (1 + AverageLength(i+1,\ r,\ L(x_i))) \quad (\clubsuit) \end{aligned}$$

The original problem is equivalent to $\min_{i\in\{1,\ldots,n\}} AverageLength(1,\ n,\ i)$, where the optimal $i$ is the chosen starting point. To solve these states and to achieve final problem, we point out the topological order of all these sub-problems: **Solving from the increasing order of sections' length.**

i.e. Firstly solve all the problems with $r - l = 1$, then solve all the problems with $r - l = 2$, and so on. Since all the problems can break up to $n$ sub-problems with shorter length section.

## 3.2 Correctness

Inductively, for the section with length $k$, suppose we already know all the optimal jumping function in section length $k - 1$.

After first checking, the index will either jump to left-hand side or right-hand side, due to the relation between $x_j, c_1$. What's important, both two sides' section length is shorter, which means the sub-problem and all following sub-sub-problems are all solved.

If we enumerate all the possible jumping destination of $S(c_1)$ and $L(c_1)$ and find the minimum length among them, we will get the global minimum of sub-problem of section length $k$.

### 3.3 Complexity

The basic cases that section length is 1 or 2, can be computed in $O(1)$.

The total states number is the sections number $\frac{n(n-1)}{2}$. In a section with length of $j-i$, we need $j-i-1$ comparing to determine the minimum between all the possible jump functions' results. Thus total time cost is

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} j-i+1 \\ &= \sum_{i=1}^{n-1} \left[ \frac{(i+1+n)(n-i)}{2} - (n-i)(i-1) \right] \\ &\sim \sum_{i=1}^{n-1} \frac{n^2}{2} + \frac{i^2}{2} - n_i \\ &\sim O(n^3) \end{aligned}$$

## 4 Reflection

I suppose I will rank the difficulty as 4 for average, 4.5 for Q1, 3.5 for Q2, 3 for Q3. It's highly depending on some "lucky inspiration" in designing DP algorithm.

**Reference** None

**Collaborator** None

# Algorithm Design and Analysis

# Assignment 5

# Deadline: June 6, 2024

1. (35 points) **[Matrix Deletion]** Consider a 01 matrix $A_{k\times n} = (x_{ij}), k \leq n$ (a matrix $A$ whose elements are 0 or 1 and size is $k \times n$). $S_i = \{j \mid x_{ij} = 1\}$ record the non-zero position of row $i \in [k]$. We want to delete the rows step by step until the matrix is empty following a specific deletion rule. $D$ is a set initialized to $\emptyset$. The deletion rule is as follows:

   1. You can delete a row $i$ only if row $i$ hasn't been deleted before and $S_i - D \neq \emptyset$.
   2. Each time you delete a row, you should then delete an arbitrary column $j \in S_i - D$ and add $j$ to set $D$.

   If a matrix can be deleted to empty following the rule above, we call it a *good* matrix, and $D$ is called a set of *deletion basics* of that *good* matrix. For example, the following matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

   is a good matrix with the following deletion process: delete (row 2, column 1), (row 1, column 5), (row 4, column 2), (row 3, column 3). The *deletion basics* set $D = \{1, 2, 3, 5\}$.

   However, the following matrix is not a *good* matrix.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

   (a) (15 points) Design a polynomial time algorithm to decide if the 01 matrix $A_{k\times n}$ where $k \leq n$ is a *good* matrix.

   (b) (20 points) Given two 01 matrices $A_{k\times n}$ and $B_{k\times n}$ where $k \leq n$. Design a polynomial time algorithm to decide if $\exists D\ subset[n], |D| = k$, such that $A$ and $B$ share the same *deletion basics* set $D$.

   For each part, prove the correctness of your algorithm, and analyze its time complexity.

2. (35 points) Consider solving a maximum flow problem $(G = (V, E), s, t, c)$ by using the **Dinic Algorithm**. In this problem, we assume that the capacities for all edges are 1: $c(e) = 1$ for each $e \in E$.

   (Notice: 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 $(v, u) \in E$. Every vertex is reachable from $s$.)

   (a) (15 points) Prove the algorithm runs in $O(|E|^{3/2})$ time.

   (b) (10 points) Let $f$ be the flow after $2|V|^{2/3}$ iterations of the algorithm. Let $D_i$ be the set of vertices at a distance $i$ from $s$ in the residual network $G^f$. Prove that there exists $i$ such that $|D_i \cup D_{i+1}| \leq |V|^{1/3}$.

   (c) (10 points) Analyze the time complexity by using both $|V|$ and $|E|$.

3. (30 points) In this question, we will prove König-Egerváry Theorem, which states that, in any bipartite graph, the size of the maximum matching equals the size of the minimum vertex cover. Let $G = (V, E)$ be a bipartite graph.

   (a) (6 points) Explain that the following is an LP-relaxation for the maximum matching problem.

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

   (b) (6 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) (6 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 incidents.)

   (d) (6 points) Use results in (a), (b), and (c) to prove König-Egerváry Theorem.

   (e) (6 points) Give a counterexample to show that the claim in König-Egerváry Theorem fails if the graph is not bipartite.

4. How long does it take you to finish the assignment (including thinking and discussing)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Write down their names here.

# AI2615 Homework 5

Tao Rui 522030910024

June 7, 2024

## 1 Matrix Detection

### 1.1 Detect Good Matrix

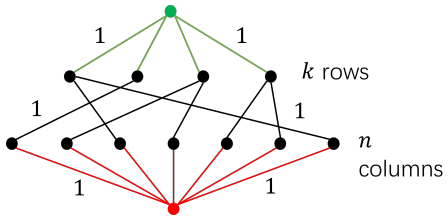



Figure 1.1: Convert Detection into Maximum Flow of Graph

---

**Build Graph** $G$
- For matrix $k \times n$, build the first layer with $k$ nodes and second layer with $n$ nodes.
- The first layer connects to *source* and second layer connects to *target*, all edge weight $= 1$.
- **For** $S_i$ **in** $S$:
- **If** $x_{ij} = 1$: connect node $k_i$ and $n_j$ with edge weight $= 1$

**Solve Maximum Flow on** $G$
- $f = Dinic(G, s, t)$
- **If** $f = k$: good matrix
- **else**: Not good matrix

---

**Algorithm**

**Correctness** The correctness of converting this problem to maximum flow problem is by **Hall's Marriage Theorem**.

Consider the deletion rule as a selecting or matching problem between rows and columns. The requirement "Only can delete row $i$ that has unique column $j$ with $x_{ij} = 1$" ensures each element $x_{ij} = 1$ can only help reduce one row. Therefore, in order to delete all $k$ rows, we have to choose out $k$ elements 1 with unique columns.

The correctness of building the maximum flow graph is as follow. The source connects $k$ nodes with $v = 1$, indicates each row can be deleted once. The target connects $n$ nodes with $v = 1$, indicates each column can be chosen to delete corresponding row once. The $i$-th node in $k$ is connected to the $j$-th node in $n$ if $x_{ij} = 1$, and the value may be set to 1, convenient for proof.

**Complexity** Building the graph at most takes $O(nk) = O(n^2)$ time, since looking at all elements that $x_{ij} = 1$ will cost $O(nk)$. If we use Dinic's algorithm to solve the max-flow on this bipartite matching graph, it will cost $O(|E| \cdot \sqrt{|V|}$, where $|V| = n+k$, $n+k \leqslant |E| \leqslant nk$. Thus Dinic will cost $O(n^2 \cdot \sqrt{n}) = O(n^{5/2})$. Since $n^{5/2} > n^2$, total time cost is $O(n^{5/2})$.

## 1.2 Detect Sharing Same Deletion Basics

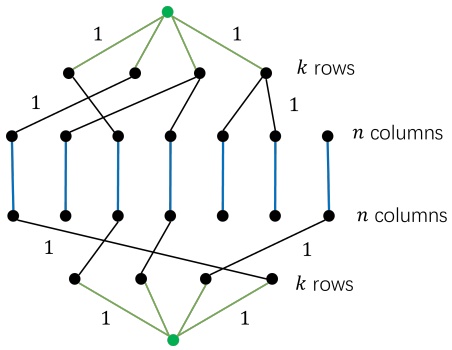



Figure 1.2: Merge Two Graph by Direct Link Between Same Column

### Algorithm

**Correctness** The idea is same as checking the good matrix. Each part of the merged graph will result in a valid flow, which is a valid matching between row and column, which is a series of valid deletion operations.

**What's new is to connect the middle layers(indicates columns) one by one. This guarantee each column can only be chosen once simultaneously by both matrix.**

**Build Graph** $G$
- For matrix $k \times n$, build the first layer with $k$ nodes and second layer with $n$ nodes. And copy these nodes and reverse them for another matrix.
- The first layer connects to *source* and last layer connects to *target*, all edge weight $= 1$.
- **For** $S_i$ **in** $S_1$:
- **If** $x_{ij} = 1$: connect node $k_{i1}$ and $n_{j1}$ with edge weight $= 1$
- **For** $S_i$ **in** $S_2$:
- **If** $x_{ij} = 1$: connect node $k_{i2}$ and $n_{j2}$ with edge weight $= 1$
- Connect the second layer and third layer one by one, with edge weight$= 1$

**Solve Maximum Flow on** $G$
- $f = Dinic(G, s, t)$
- **If** $f = k$: Have same deletion basics
- **else**: Not Have same deletion basics

**Complexity** Building the graph at most takes $O(2nk) = O(n^2)$ time.

Since this graph still share the same critical property with bipartite graph: Each edge has weight 1, each turn the finding of block flow will at most cost $O(|E|)$, since each edge will be at most checked once and deleted.

Thanks to another critical property: $V$
$\{s, t\}$ exist one degree. Thus after $|V|$ turns, the number of vertex disjoint path = number of edge disjoint path = current maximum flow, which is at most $\frac{|V|}{\sqrt{|V|}}$. Thus it will cost at most $O(\sqrt{|V|})$ turns to stop.

So running Dinic's alogrithm on this graph still cost s $O(|E| \cdot \sqrt{|V|} = O(n^{5/2})$. Therefore total complexity is $O(n^{5/2})$.

# 2 Dinic Algorithm

## 2.1 Prove Running Time $O(|E|^{3/2})$

With the property that capacity for all edge is 1, we can conclude that finding blocking flow in each turn will at most cost $O(|E|)$, since each edge will only be checked once and deleted.

Now prove the iteration will terminate at most after $2\sqrt{|E|}$ turns.

Suppose $G^f$ is the residual network after $\sqrt{|E|}$ rounds. Then the shortest path from $s$ to $t$ in $G^f$ is at least $\sqrt{|E|}$. **Since each edge's capacity is 1, all the blocking paths is disjoint.** The maximum flow = number of edge disjoint path. Since each edge disjoint path has to cost at least $|E|$ edges, total number of path is at most $\frac{|E|}{\sqrt{|E|}} = \sqrt{|E|}$.

Each turn of updating residual network increases the flow at least 1, therefore there remain at most $\sqrt{|E|}$ rounds of updating.

Total time complexity is $O(|E|\sqrt{|E|}) = O(|E|^{3/2})$

### 2.2 Prove $|D_i \cup D_{i+1}| \leqslant |V|^{1/3}$

After $2|V|^{2/3}$ rounds iterations, the minimum path length is at least $2|V|^{2/3}$. Since all the edge has capacity of 1, we cannot use each edge more than once.

Suppose $\forall i, |D_i \cup D_{i+1}| \geq |V|^{1/3}$, which indicates that $\forall i, |D_i| > \frac{1}{2}|V|^{1/3}$.

Since the minimum path from $s$ to $t$ is $2|V|^{2/3}$, the $i$ is count from 0 to $2|V|^{2/3}$. Thus the total vertex number, which contains all possible $D_i$, is larger than $2|V|^{2/3} \times \frac{1}{2}|V|^{1/3} = |V|$. Which leads to contradictory, must $\exists i, |D_i \cup D_{i+1}| \leqslant |V|^{1/3}$.

### 2.3 Analyze Complexity Using $|V|, |E|$

Running the Dinic algorithm, each round of finding block flow will cost at most $O(|E|)$ time, due to the all-1 edge capacity.

After $2|V|^{2/3}$ rounds, we will prove there exists at most another $|V|^{2/3}$ rounds to terminate.

By question (b), $\exists i, |D_i \cup D_{i+1}| \leqslant |V|^{1/3}$. We can construct the cut by dividing all points in $s$, $D_1, D_2, ..., D_i$ into left side, and remain points into right side.

Note that there will not be any edges from $D_j,\ j < i$ to right side, otherwise the right-side's points should have smaller distance to $s$.

Thus the only edges from left side to right side, is the edges from $D_i$ to $D_{i+1}$. Since $|D_i| + |D_{i+1}| \leqslant |V|^{1/3}$, the edge number between two sets are $|D_i| \times |D_{i+1}| \leqslant \left(\frac{|D_i| + |D_{i+1}|}{2}\right)^2 \leqslant \frac{|V|^{2/3}}{4}$.

Therefore the minimum cut is less than $\frac{|V|^{2/3}}{4}$, which indicates the maximum flow in residual network is less than $\frac{|V|^{2/3}}{4}$, which further indicates the remain iterations will terminate after at most $O(|V|^{2/3})$ rounds.

Thus the total time complexity is $O(|E||V|^{2/3})$.

## 3 König-Egerváry Theorem

### 3.1 LP-Relaxation for Maximum Matching Problem.

Suppose the variables $x_e \in \{0, 1\}$ represents whether an edge is chosen in the matching problem. Since it's maximum matching is bipartite graph, the number of edges is the number of matching. Thus the objective is to maximize

$\sum e \in E x_e$. And the constraints are, each node can at most be chosen once, $\forall v \in V, \sum_{e=(u,v)} = 0$ or 1.

The relaxation of this ILP is to relax $x_e$ to continuous value $x_e \in [0, 1]$. Thus the objective is not changed, and constraints change to: for each node, total selection of this node is smaller than 1. Which is $\forall v \in V, \sum_{e=(u,v)} \leqslant 1$, and of course $x_e \geqslant 0$.

Thus we prove it's the LP-relaxation for maximum matching problem.

## 3.2 Dual Problem

The objective is $x_{(v_1,v_i)} + x_{(v_1,v_j)} + x_{(v_2,v_k)} + ... + x_{(v_w,v_i)} + ....$ Now consider the constraints, each edge $x_{(v_i,v_j)}$ will exist in two constraints only, which is the constraint about $v_i$ and $v_j$. What's more, the factor of $x_{(v_i,v_j)}$ and $x_{(v_j,v_i)}$ are both 1.

Suppose we multiply $y_i$ to constraint $i$, the LP will become:

$$\begin{aligned} \textbf{min} & \sum_{i \in |V|} y_i \\ \textbf{s.t.} \ & y_i + y_j \geqslant 1 \quad \forall (i,j) \in E \\ & y_i \geqslant 0 \qquad \forall e \in E \end{aligned}$$

We can interpret this as minimum vertex cover. $y_i$ represents each vertex's chosen weight, which has been relaxed to $[0, 1]$. Correspondingly, the constraints require each edge has total chosen weight larger than one. And Of course $y_i \geqslant 0$.

Thus we prove this dual LP is the relaxation of minimum vertex cover.

## 3.3 Prove Incident Matrix Unimodular

Prove this by induction: ① Basic case: If incident matrix size is $1 \times 1$, $\det A = 0$ or 1, holds.

② Suppose $k \times k$ matrix holds, consider the matrix with size $k+1 \times k+1$. The matrix can be divided into three cases: One column all zero; One column with single 1; All column has two 1.

If one column is all zero, $\det A = 0$. If one column has single 1, $\det A_{k+1 \times k+1} = (-1)^i \times \det A_{k \times k}$. By deduction, we known that $\det A_{k \times k} \in \{-1, 0, 1\}$, therefore $\det A_{k+1 \times k+1} \in \{-1, 0, 1\}$ still holds.

If all column has two 1, consider two facts:

- Each edge connects only two vertices, thus one column can have at most two 1.
- Each edge must connect two vertices in different side $A$ and $B$.

Therefore, each column must have one 1 in the $A$ side and one 1 in the $B$ side. Adding all rows of $A$ side, we will obtain a row of 0/1. Adding all rows of $B$ side, we will also obtain a row of 0/1, which are one-to-one correspondent

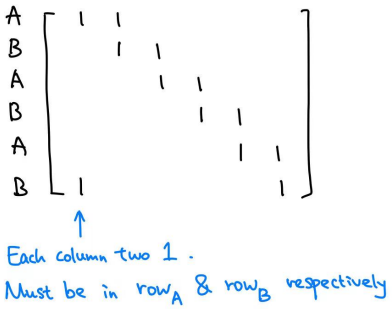



Figure 3.1: All column has two 1

to the previous row. Thus we minus the $A$ row by $B$ row, we will generate an all-zero row, which indicates that $\det A = 0$.

Combine three situations of ②, we prove that incident matrix of bipartite graph is unimodular by induction.

## 3.4 Prove König-Egerváry Theorem

Consider the two LP problems' factor matrix.

The LP-relaxation for the maximum matching problem has $|E|$ variables and $|V|$ constraints, the factor matrix is $|V| \times |E|$. For each row, the 1 will appear at those columns which that edge contains this vertex. This is the definition of incident matrix. Thus the factor matrix of LP-relaxation for the maximum matching problem is a incident matrix $A$, which is unimodular.

The LP-relaxation for the minimum vertex cover problem has $|E|$ constraints and $|V|$ variables, the factor matrix is $|E| \times |V|$. For each row, there will be only two 1, appear at the two columns that incident to this edge. Thus the factor matrix of LP-relaxation for the maximum matching problem is transpose of a incident matrix $A^\top$, which is unimodular.

Consider that both LP problem's factor matrix is unimodular, their optimal value can both be obtained by interger. What's more, by strong duality theorem, their optimal value is same, which indicates that they can obtain a same optimal value by integer, i.e. the size of the maximum matching equals the size of the minimum vertex cover.

## 3.5 Counter Example When Not Bipartite Graph

Suppose a graph with three nodes, since there doesn't exist "two type", the incident matrix may have some unexpected edges, $\det A = 2$.

In this situation, the minimum vertex cover is 2. The maximum matching is 1, since choosing any two edges will cause one vertex to be chosen twice. Which indicates König-Egerváry theorem fails.


Figure 3.2: Counter Example of Non-Bipartite Graph

# 4 Reflection

Difficulty: 5, though several similar prove scheme has been taught in class.

**Reference** None

**Collaborator** For question 1.(b) and 2.(c), obtain inspiration from Zhu Shengjia.

# Algorithm Design and Analysis

# Assignment 6

# Deadline: June 16, 2024

1. (30 points) Given an undirected graph $G = (V, E)$ with $n = |V|$, decide if $G$ contains a clique with size exactly $n/2$. Prove that this problem is NP-complete.

2. (40 points) Given an undirected graph $G = (V, E)$ and an integer $k$, decide if $G$ has a spanning tree with maximum degree at most $k$. Prove that this problem is NP-complete.

   (a) (20 points) Prove it is NP-complete when $k$ is an input.

   (b) (20 points) Prove it is NP-complete when $k$ is any fixed constant.

3. (30 points) Given a collection of integers (can be negative), decide if there is a subcollection with a sum of exactly 0. Prove that this problem is NP-complete.

4. How long does it take you to finish the assignment (including thinking and discussion)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Please write down their names here.

# AI2615 Homework 6

Tao Rui 522030910024

June 8, 2024

## 1 $\frac{n}{2}$ Size Clique Existence is NP-complete

Firstly we prove checking the existence of a clique with size $\frac{n}{2}$ is NP problem. For a graph $G$ as input $x$, and any $\frac{n}{2}$ chosen vertices as input $y$. We can simply check vertex-by-vertex whether every vertex has edge to other $\frac{n}{2}-1$ vertices in $O(n^2/4)$ time. This verifier decides whether given $\frac{n}{2}$ vertices are clique in $O(n^2)$ time, which means this is NP problem.

Secondly, we try to Karp-reduce Independent Set problem to this clique check problem.

① If $k = \frac{n}{2}$, the reduction is direct. Since a independent with size $k$ in $G$ is equal to a clique with size $k$ in reversed graph $\bar{G}$. And we already have the solver to check existed clique with size $k = \frac{n}{2}$.

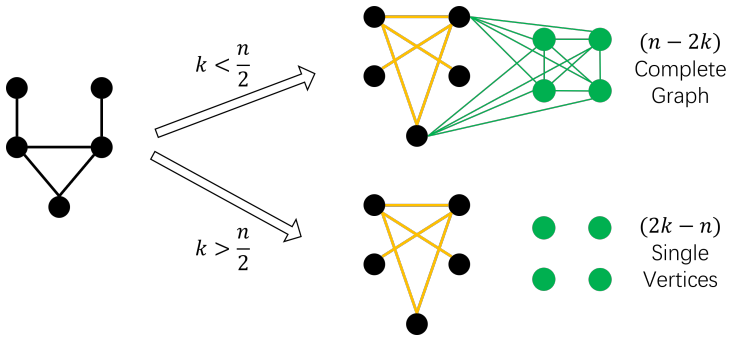



Figure 1.1: Independent Set Reduce to Clique by Adding Vertices to $\bar{G}$

② If $k < \frac{n}{2}$. In $\bar{G}$ we can only check the existence of clique with size $\frac{n}{2}$ larger than expected $k$. We can add $n-2k$ new vertices as a complete graph, and connects each new vertex to every previous vertex. We denote as

$$G_{complete} \coloneqq \bar{G} + \mathbb{K}_{n-2k} + \{(u,v) : u \in \mathbb{K}_{n-2k},\ v \in \bar{G}\}$$

In $G_{complete}$, the new $n-2k$ vertices are sure to exist in any possible clique, thus the checking of a clique size $\frac{|G_{complete}|}{2} = n-k$ is equal to checking a clique size $(n-k)-(n-2k) = k$ in previous $\bar{G}$, and further equivalent to existence of a independent set size $k$ in original $G$.

③ If $k > \frac{n}{2}$. We can add $2k-n$ new single vertices to the reversed graph $\bar{G}$. These new single vertices will never appear in any clique, since they have no edges connecting to other. Thus the adding operation only increase the problem's vertex size. We denote this graph as

$$G_{single} \coloneqq \bar{G} + \{v_i : \textbf{For i in range}(2\text{k-n})\}$$

The checking of a clique size $\frac{|G_{single}|}{2} = k$ is exactly equal to checking a clique size $k$ in previous $\bar{G}$, since those single vertices do nothing. And this further equivalent to existence of a independent set size $k$ in original $G$.

What's more, the construction of $\bar{G}$ and $G_{complete}$ and $G_{single}$ cost at most $O(|E|) = O(n^2)$ time, simply enumerate all vertices and adding new edge. Thus the reduction of input is polynomial. Combine ①②③, we Karp-reduce the independent set problem to checking existence of clique with size $\frac{n}{2}$. Since Independent Set is NP-hard, this NP problem is NP-complete.

# 2 Spanning Tree with Degree $k$

## 2.1 $k$ is Input

We try to Karp-reduce the Hamilton Path problem to Spanning Tree with Input $k$. Suppose the input is graph $G$ and we want check a Hamilton Path existence, it's equivalent to check existence of a spanning tree with max degree 2.

By **Lemma**, the reduction of the problem do not need changes on input. $HamiltonPathExist(G) = SpanningTreeK(G, 2)$. Since checking given edges whether form a spanning tree costs $O(|E|) = O(n^2)$ times (Traverse edges, and record whether each vertex is covered $[1, k]$ times), Spanning Tree with Input Degree k is NP-complete.

**Lemma:** Hamilton Path is equivalent to Spanning Tree with max degree 2

**★ Proof ★**

① If we have a Hamilton path, each vertex is achieved once, thus the number of edges connected to it is at most 2. Only two vertices has degree 1, which is the starter and ending (Otherwise there exist cycle). Therefore the total edge number is $|V|-1$, and they cover all vertices. This indicates that the Hamilton path is exactly a spanning tree of $G$, and also satisfies max degree $\leqslant 2$.
② If we have a spanning tree with max degree $\leqslant 2$. Since there's no cycle, there must be at least one vertex $v_0$ with degree 1. Start from $v_0$. For $i > 1$, $d(v_i) \in \{1, 2\}$: If we reach a vertex with degree 2, we can further go to next vertex; if degree is 1 but haven't traverse all vertices, it's must not be a spanning tree of $G$, since all visited vertices has no extra degree to link elsewhere, the visited vertices is disconnected with non-visited vertices. Therefore, we can go along the vertices with all degree = 2, until the last one. This path is exactly Hamilton path on $G$.

### 2.2 $k > 2$ is Constant

We try to Karp-reduce the previous problem to Spanning Tree with constant $k_0$.

For each previous vertex in $G$, we add $k_0 - 2$ new vertices connected to it, results in a new graph $G'$. Since the spanning tree requires all vertices to be covered, each previous vertex has to at least choose those $k_0 - 2$ edges. The remain degree limitation is 2, and remain graph is $G$, which is the original problem we want to solve. Thus $SpanningTree2(G(V, E), 2) = SpanningTreeK0(G', k_0)$,

Since $k_0$ is given constant, constructing new graph $G'$ needs $O(nk_0)$ time, which is linear, Karp-reduction holds. What's more, the verifier is same as checking with max degree 2, both cost $O(|E|) = O(n^2)$ time. Therefore the Spanning Tree with max constant degree k is NP-complete.

## 3 Subset Zero Sum

**Intuition** We want to Karp-reduce the original Subset Sum problem to this problem. A natural idea is to add a new number $-k$, $\mathbb{A}' = \mathbb{A} \cup \{-k\}$. However, this cannot force $-k$ to be chosen, a counter example is as follow: $\mathbb{A} = \{1, -1\}$ and sum requires 100, which is impossible. But if we transfer the problem to $\mathbb{A}' = \{1, -1, -100\}$, the sum of 0 is obtained by original number $1, -1$.

To force the new adding number $-k$ to be chosen, we can simply consider the problem with all positive numbers. The Subset Sum with Positive has already been proved to be NP-complete, reduced from Independent Set with Size k. Thus the reduction $SS_{positive} \leqslant_k SS_{zero}$ proves $SS_{zero}$ is NP-complete.

**Karp-Deduction** If $\mathbb{A}$ is all positive (Delete those original 0 in $\mathbb{A}$ since they are useless), the summation of zero in $\mathbb{A}'$ must require $-k$, since $-k$ is the only negative number.

$$A = \{ a_1, a_2, \dots, a_n \}$$
$$\Rightarrow \quad A' = \{ a_1, a_2, \dots, a_n, -k \}$$

Figure 3.1: Positive-Subset-Sum Reduce to Subset-Zero-Sum

**NP-Complete** The Subset Zero Sum can be verified in $O(n)$ time, since we can add the given chosen $k$ numbers and check whether it's 0. Thus the problem is NP problem. Combine the NP-hard we have proved above, the Subset Zero Sum is NP-complete.

# 4 Reflection

Difficulty: 2.5. It's the first time for me an algorithm homework has been done within 5 hours. :) Considering the overall workload this semester, this course **has to** be open in first eight weeks. Thanks for grading my homework meticulously.

**Reference** None

**Collaborator** None