Sufficient Algorithms

Charlotte Xia

February 3, 2024

Contents

1 Fibnacci Problem

1.1 Problem Description

The well known Fibnacci Problem is defined as follows:

Definition 1.1.

$$F_n = \begin{cases} F_{n-1}+F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases} \tag{1}$$

1.2 Solution

1.2.1 Naive Case and Time Complexity

Intuitively, we count the next number by adding the previous two numbers.

Algorithm 1: fib1(n)

```
1 if n=0: return 0
2 if n=1: return 1
3 return fib1(n-1)+fib1(n-2)
```

However, it has an exponential time complexity.

Lemma 1.1. $T_n > F_n$,where T_n stands for the steps needed to calculate F_n.

Proof. Firstly, we have:

$$T(n) = T(n-1) + T(n-2) + 3 \textbf{ for } n > 1$$

where 3 extra steps are needed for 2 'if' judgements and the addition operation.
In the Base Step, for $n = 0$, we have $T_0 = 1 > 0 = F_0$, and for $n = 1$, we have $T_1 = 2 > 1 = F_1$.
In the Inductive Step, suppose $T_i > F_i$ for $i = 1, 2, \ldots, n$.

$$\begin{aligned} T_{n+1} &= T_n + T_{n-1} + 3 \\ &> F_n + F_{n-1} + 3 \quad \text{(induction hypothesis)} \\ &= F_{n+1} + 3 \\ &> F_{n+1}. \end{aligned}$$

Therefore, by mathematical induction, we have proved that $T_n > F_n \ \forall n \in N$. □

Furthermore, according to the middle school knowledge of geometric progression, we can formulate a concrete expression of F_n, which is approximately $F_n = 2^{0.694n} = 1.6^n$, which gives out an exponential time complexity.

1.2.2 Dynamic Programming and Big Number Addition

We notice that every time we calculate F_n, we have to **recalculate** F_{n-1} and F_{n-2}, which is a waste of time. By creating an array to store the past values, we've come with an polynomial solution ,where approximately, every T_n is obtained by the one step addition of T_{n-1} and T_{n-2}.

However, when n is large enough, the number of digits of F_n will exceed the maximum number of digits that a computer can store(eg. 64 digits), which brings out the problem of big number addition. At present we assume that adding two n-digit numbers produces complexity of $O(n)$, with $O(\log n)$ methods displayed in Therefore, for $n > 200$, consider $T_{\frac{n}{2}} > 2^{0.694\times 100}$. Adding T_{n-1} and T_n takes $O(n)$ time. In total, T(n) takes $O(n^2)$ time.

Algorithm 2: fib2(n)

```
create an array f[0... n]
f[0] = 0, f[1] = 1
forall i ≥ 2 do
    f[i] = f[i − 1] + f[i − 2]
end
```

1.2.3 Matrix Multiplication and Fast Power Algorithm

We can rewrite (1) as:

$$\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix},$$

which implies

$$\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Therefore, it only remains to compute T^n for

$$T = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}, T_n = \begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix}$$

. Let $M(n), A(n)$ be the time complexity for computing the multiplication and addition of two n-bit integers respectively. Notice that $M(n) \geq n$, as we at least need to read the inputs. We have seen $A(n) = O(n)$. Therefore, $M(n) = \Omega(A(n))$.

We will design an algorithm to compute T^n and analyze the time complexity in terms of $M(n)$.

Let $k = \lceil \log_2 n \rceil$. We will compute all the $k+1$ matrices T^{2^0}, T^{2^1}, ..., T^{2^k}. Let $a_k a_{k-1} \cdots a_1 a_0$ be the binary representation of k. It is easy to see that

$$T^n = \prod_{i:a_i=1}^{k} T^{2^i} \tag{2}$$

The algorithm for computing T^n is thus split into two steps:

1. Compute T^{2^i} for $i = 0, 1, \ldots, k$.
2. Compute the product by (2).

Time Complexity for Step 1. Notice that it requires only one matrix multiplication to get $T^{2^{(i+1)}}$ from T^{2^i} if we've computed T^{2^i}, namely, $T^{2^{i+1}} = (T^{2^i}) \times T^{2^i}$. Now we analyze the time complexity for this multiplication by investigating the length of the for integers in T^{2^i}.

Lemma 1.2. The length of the integers in T^{2^i} is at most $2^{i+1} - 1$.

Proof. For the base step, if $i = 0$, then $T^{2^0} = T^1 = T$. The elements in T are less than $2^1 - 1 = 1$. For the induction step, suppose the lemma holds for $i - 1 (i > 2)$, then $T^{2^i} = (T^{2^{i-1}}) \times T^{2^{i-1}}$. Since th length of product of m-digit number and n-digit number is less than $m+n-1$, the length of element t in T is less than $(2^i - 1) + (2^i - 1) - 1 = 2^{i+1} - 3 < 2^{i+1} - 1$. □

Since multiplying two 2×2 matrices requires 8 integer multiplications and 4 integer additions, and $M(n) = \Omega(A(n))$, computing $T^{2^{i+1}}$ from T^{2^i} has complexity bounded by $c \cdot M(2^{i+1} - 1) \leq c \cdot M(2^{i+1})$ for some constant $c > 0$. Therefore, the overall time complexity for Step 1 is

$$c \cdot (M(2^2) + M(2^3) + \ldots + M(2^{2k+1})) \tag{3}$$

Lemma 1.3. For $i = 2, 3, \ldots, k$, $M(2^{i+1}) \geq 2M(2^i)$

Proof. To see this intuitively (although the following argument is not rigorous), if otherwise, we will have

$$M(2^i) < 2M(2^{i-1}) < 4M(2^{i-2}) < \ldots < 2^i M(1) = 2^i,$$

which contradicts $M(n) \geq n$ as mentioned earlier. □

Therefore, we have

$$M(2^{k+1}) \geq 2^1 M(2^k) \geq 2^2 M(2^{k-1}) \cdots \geq 2^i M(2^{k-i+1}) \cdots \geq 2^{k-1} M(2^2)$$

By dividing the inequation by 2^{k-i}, and pluging k with $\lceil \log_2 n \rceil$, we have

$$M(2^i) \leq \frac{1}{2^{k-i}} M(2^{k+1}) \leq \frac{1}{2^{k-i}} M(2n)$$

the time complexity for (3) is

$$c \cdot (M(2^2) + M(2^3) + \ldots + M(2^{2k+1})) \leq c \cdot M(2n) \left(1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^{k-1}}\right) = O(M(2n))$$

, where the last equality is due to the fact that $M(2n) \leq 4M(n)$ (even if we use naive grade school multiplication) and $1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^{k-1}} < 2$.

Time Complexity for Step 2. We will analyze the time complexity for computing

$$\prod_{i=0}^{k} S_k := T^{2^i}$$

We need k matrix multiplications for S_k. Suppose we have computed $S_i := T^{2^0} \times T^{2^1} \times \ldots \times T^{2^i}$. Obviously, each entry of S_i is dominated by T^{2^i+1}. Thus, computing $S_{i+1} = S_i \times T^{2^{i+1}}$ requires at most $c \cdot M(2^{i+2})$ time. The overall time complexity for Step 2 is also at most

$$c \cdot (M(2^2) + M(2^3) + \ldots + M(2^{2k+1})),$$

which, by previous analysis, is $O(M(n))$.

Overall Time Complexity: $O(M(n))$

Remark 1.1. An Intuitive way to calculate the overall time complexity is to draw analogy between

$$1 + 2 + 2^2 + \cdots + n < 1 + \cdots + 2^{\lceil \log_2 n \rceil} = 2^{\lceil \log_2 n \rceil + 1} - 1 < 2n$$

and

$$M(2^2) + M(2^3) + \cdots + M(2^{k+1}) \leq M(2^2) + \cdots + M(2n)$$

2 Divide and Conquer

2.1 Multiplication

2.1.1 Problem Description

For naive two n-digit multiplication with base 2, suppose n is 2's power.

$$xy = (a \times 2^{n/2} + b)(c \times 2^{n/2} + d) = ac \times 2^n + (ad + bc) \times 2^{n/2} + bd$$

$$1234 \times 5678 = (12 \times 100 + 34) \times (56 \times 100 + 78)$$

We only consider multiplication, ignoring multiplying by 100 (which can be done simply by shifting a few digits) or addition, which are linear compared to Multiplication. Eventually, we need $\log_2 n$ steps to divide into the base step (1-digit times 1-digit).With each step forward, we multiply to number of elements on the level by 4. The multiplication is done on the end base, so in total we have $4^{\log_2 n} = O(n^2)$ complexity.

2.1.2 Karatsuba algorithm

Rather than calculating ad, bc, ac, bd separately; we obtain $(ad + bc) = (a + b) \times (c + d) - ac - bd$. Therefore, instead of dividing it into 4 parts, we divide it into 3 groups, through 3 multiplication operations on $\frac{n}{2}$, we get $3^{\log_2 n} = n^{\log_2 3} = n^{1.6}$ Specially, if n is not 2's power, we add zero to max bits, which at most doubles the original length, $(2n)^1.6$. No adding zero in real coding, since carrying occurs, (eg. 64→32,(32+1)→64).

Algorithm 3: Karatsuba algorithm

Input: Two n-digit numbers x, y.
Output: their product
x_L, x_R = leftmost$\lceil n/2 \rceil$, rightmost$\lfloor n/2 \rfloor$ bits of x
y_L, y_R = leftmost$\lceil n/2 \rceil$, rightmost$\lfloor n/2 \rfloor$ bits of y
P1=multiply(x_L, y_L)
P2=multiply(x_R, y_R)
P3=multiply$(x_L + x_R, y_L + y_R)$
return $P1 \times 2^n + (P3 - P1 - P2) \times 2^{\frac{n}{2}} + P2$

Time complexity is:

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

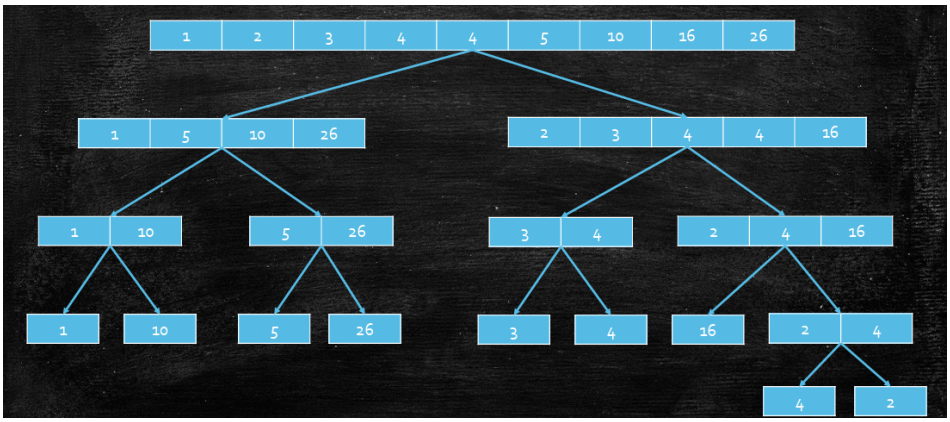


Figure 1: Merge Sort Pipeline

Due to carry, $T(\frac{n}{2})$ should be $T(\frac{n}{2}+1) = T(\frac{n}{2}) + O(n)$.

$$\begin{aligned} T(n) &= 3 \cdot (3T(\frac{n}{4}) + \frac{cn}{2}) + cn \\ &= 3^{\log_2 3} T(1) + cn(1 + \frac{3}{2} + \frac{3^2}{2^2} + \cdots + \frac{3^{\log_2 n}}{2^{\log_2 n}}) \\ &= O(n^{\log_2 3}) + O(n^{1+\log_2 1.5}) \\ &= O(n^{\log_2 3}) = O(n^{1.6}) \end{aligned}$$

1. **Toom-Cook**:$O(n^{1.465})$
   Breaking into 3 parts $5 \times \frac{n}{3} \times \frac{n}{3}$
2. **Fast-Fourier Transform**
3. **Strassen's magical idea for Matrix Multiplication**
   Squeeze eight multiplications into seven with complex calculation, better than the naive case (n for a row times a column, with n^2 numbers, in total $O(n^3)$. By normal divide and conquer, we divide one'n-size multi' into 8 $\frac{n}{2}$ multi, with total operations of $8^{\log_2 n} = O(n^3)$)

Remark 2.1. **Divide and Conquer** is a general algorithm design paradigm.

1. **Divide:** Divide the problem into small size subproblems.
2. **Recursive:** Solve small problems recursively
3. **Combine:** Combine the output of small size subproblems to get the answer for the original problem.
4. **Basic solver:** If the problem size is small enough, solve it directly.

2.2 Sorting Problem

A very familiar problem, which can be easily picked up by analyzing pipeline1.

2.2.1 Time complexity

For the 'merge' part, merging two arrays(length m and n) requires $O(m+n)$ time. For each level, we have $O(n)$ merging operations. There are $\log_2 n$ levels, so the total time complexity is $O(n \log_2 n)$

Remark 2.2. Never use asymptomatic notations in an induction-based analysis!
An example of a wrong analysis is as follows:

Proof. $T(i) = O(i)$ holds for $i = 1, \ldots, n$
Base Step: T1 =O(1) holds trivially.
Inductive Step: Suppose $T_i = O(i)$ holds for $i = 1, \ldots, n-1$
$T(n) = 2T(\frac{n}{2}) + O(n) = 2O(\frac{n}{2}) + O(n) = O(n)$ □

There are two mistakes:

Firstly, the Inductive Step is meaningless, since $O(\cdot)$ takes in a function (with variable i). Given a specific value, $O(i = n-1) = O(1)$

Secondly, when the recursion happens numerous times, a constant can be a function of n. When we write $O(n)$, what we actually mean is cn.

Therefore, assume $O(n)$ algorithm, the correct proof should be:

Proof. $T(i) \leq ci$ holds for $i = 1, \ldots, n$
Base Step: $T(1) \leq c$ holds trivially.
Inductive Step: Suppose $T(i) \leq ci$ holds for $i = 1, \ldots, n-1$

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + O(n) \leq 2T(\frac{n}{2}) + d \cdot n(\text{for some constant d}) \\ &\leq 2 \cdot c \cdot \frac{n}{2} + d \cdot n(\text{ by induction hypothesis}) \\ &= (c+d) \cdot n \end{aligned}$$

Induction fails as constant changes from c to $c+d$, so it is at least $O(n)$. □

A correct proof should be:

Proof. If $T(n) \leq 2T(\frac{n}{2}) + c \cdot n$ for constant c, $T(n) \leq B \cdot n \log n$ for some constant $B > c$.
Base Step: $T(2) \leq B \cdot 2 \log 2$ holds trivially.
Inductive Step: Suppose $T(i) \leq Bi \log i$ holds for $i = 2, \ldots, n-1$
For $i = n$, we have

$$T(n) \leq 2T(\frac{n}{2}) + cn \leq 2 \cdot B \cdot \frac{n}{2} \log \frac{n}{2} + cn = Bn \log n - Bn + cn < Bn \log n$$

□

Remark 2.3. At least $\Omega(n \log n)$ comparisons are needed.
The time complexity is largely contributed by the running time of comparison. Total possible outputs is $3^{K(n)}$, n! numbers of permutations. Therefore, $3^{K(n)} \geq n!$, $K(n) \geq \log_3 n! \geq \Omega(n \log n)$

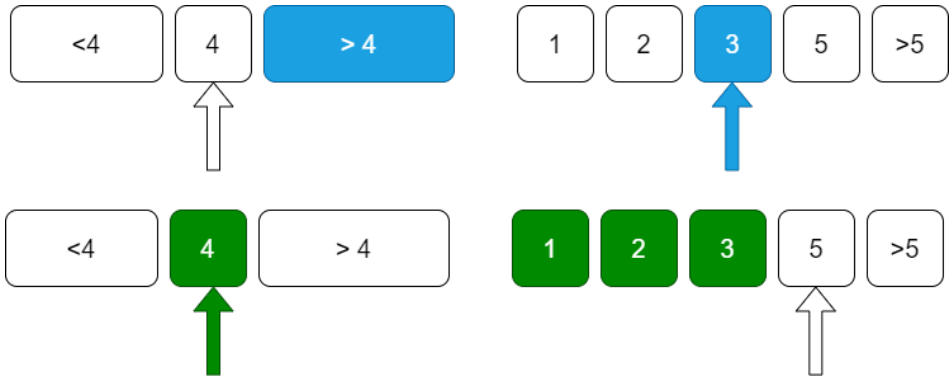


Figure 2: Inversion

2.3 Inversions

Definition 2.1. if $a_i > a_j$ when $i < j$, we call (a_i, a_j) an inversion.

The idea is that we divide the input into two subsets (A:$x_1, x_2, \ldots, x_m$; B:$x_{m+1}, x_{m+2}, \ldots, x_{m+n-1}$), and count the inversions of each subset and **across** subsets, which takes $O(mn)$ with each a_i scanning the whole list B.

A magic trick is that we **mix merging and counting together**. We first count the inversions in A and B, then sort them separately. If A,B are sorted, we change a 'comparison' complexity problem into a 'search' complexity problem. In other words, we insert the additional step of counting inversion in the mergesort, by following operations, with time complexity of O(nlogn).

Algorithm 4: countInversions

Input: S=$(x_1, x_2, \ldots, x_n)$,A:$x_1, x_2, \ldots, x_{n/2}$; B:$x_{n/2+1}, x_{n/2+2}, , x_n$
Output: Number of Inversions: count

```
count=0
if n == 1 then
    return 0
end
count+=countInversions(A)+countInversions(B)
Maintain 2 pointers i = 1, j = n/2 + 1
C={}
while i < n/2 and j < n do
    if x_i < x_j then
        C.append(x_i)
        i+ = 1
    end
    else
        C.append(x_j)
        j+ = 1
        count+ = j − n/2
    end
end
count+=(n/2-i)×(j-n/2)
S=C
return count
```

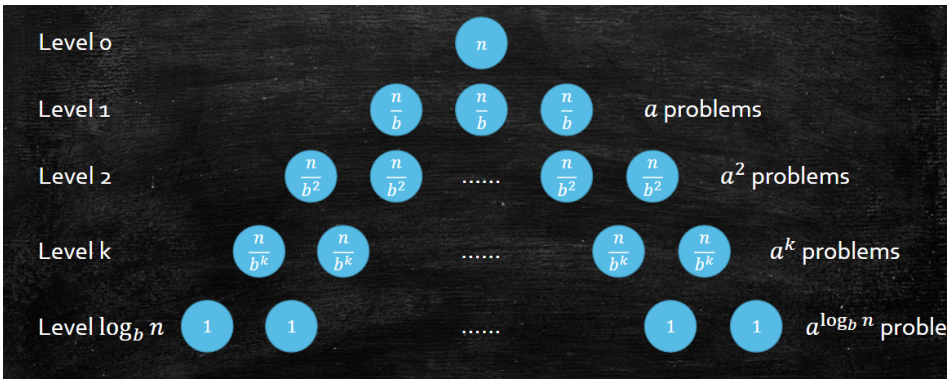


Figure 3: Master Theorem

2.4 Master Theorem

Theorem 2.1. If $T(n) = aT(\frac{n}{b}) + O(n^d)$, where a: num of subproblems; b:subproblem size n/b; d: combining time complexity.
$O(n^d)$:combining time complexity

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \\ O(n^d \log n) & \text{if } a = b^d \end{cases} \tag{4}$$

Proof. The running time of solving size-1 problem is $O(a^{\log_b n}) = O(n^{\log_b a})$. The total running time for combining is $c\dot{(}n^d) + c\dot{(}(\frac{n}{b})^d)\dot{a} + \cdots a^k\dot{c}(\frac{n}{b^k})^d + \cdots a^{\log_b n}\dot{c}$. The size-1 problem complexity is eaten up by the last element of the combining complexity. Eventually, we have $c\dot{n}^d\dot{(}1 + \frac{a}{b^d}) + \cdots (\frac{a}{b^d})^k + \cdots (\frac{a}{b^d})^{\log_b n}$. When $a < b^d$, intuitively the head is heavier(longer when imaging the operation of every layer as a line). When $a > b^d$, intuitively the tail is heavier. When $a = b^d$, $T(n) = O(n^d)\dot{(}1 \times \log_b n) = O(n^d \log_b n)$.

□

2.5 Selection

Input: A set S of n integers $x_1, x_2, \ldots, x_n$ and an integer k.
Output: The k-th smallest integer x^* among $x_1, x_2, \ldots, x_n$.
naive case: Sorting and choosing the k-th element: O(nlogn)
alg 1: quick sort? have the first number to be the pivot.
alg 2: Divide $x_1, x_2, \ldots, x_n$ into three subsets, recursively pick out the subset x^* is in.

Algorithm 5: Select

```
Choose an arbitrary value v among x_1, x_2, ...,
Divide x_1, x_2, ..., x_n into three subsets L, M, R
if k ≤ |L| then
    return Select(L,k)
end
else if |L| < k ≤ |L| + |M| then
    return v
end
else if |L| + |M| < k then
    return Select(R,k − |L| − |M|)
end
```

Proof. **the scale of the problem shrinks:**

First, we show that the algorithm always terminates. If it does not terminate in the current recursive iteration, it will either call Select(L,k) or Select(R, $k - L - |M|$). It suffices to show that $|L| < n$ and $|R| < n$ (so that the problem size is strictly decreasing). This is guaranteed by $v \notin L$ and $v \notin R$ (v is in the set, so at least v is excluded).

the algrithm is correct inductively:

Suppose the correct value is output for any array S of length n and any $k \in 1, 2, \ldots, n$.

Base Step: n = k = 1, which is obviously true.

Inductive Step: Suppose the algorithm correctly outputs the k-th smallest value for any array S with any length $l \in 1, 2, \ldots, n-1$ and any $k \in 1, 2, \ldots, n$.

Select(L,k) returns the k-th smallest value in L and Select(R, $k - |L| - |M|$) returns the $(k - |L| - |M|)$-th smallest value in R by induction hypothesis. For any input array of length n and any integer v in it, it is straightforward to check that the k-th smallest value in L is the k-th smallest value in S if $L \geq k$, and the remaining two cases are also correct. □

2.5.1 Time Complexity*

For dividion,$\Theta(n)$ with each element compared to v. For recursion, $T(|L|)$ or $T(|R|)$ or $O(1)$.
For the worst case, consider $1, 2, ..., n$,where k=n. The algorithm is $O(n^2)$.

Figure 4: The white an are doesn't matter terms. We only consider the last step when $a_m, m \in [i, j]$ will be chosen as a pivot. There are (j-i+1) choices, and only 2 choices (namely, a_i or a_j) can result in comparison between a_i and a_j.

Remark 2.4. Randomness in input and randomness in algorithm
Commonly, we choose to analyze the mean time complexity of a random algorithm. Because in real case, randomness of input is a strong condition(block-like features commonly).

By estimation, for every layer, n comparisons. If we are lucky enough to have $\frac{1}{3}$ in the minimum separation, then $n\frac{2}{3}^n > 1$, $k < \log_1 .5n$.

Definition 2.2. $E[x] = \sum_{\omega\in\Omega}^{n} \Pr(\omega) \cdot X(\omega)$

Theorem 2.2. Linearity of expectation

$$E|\sum_{i=1}^{n} X_i| = \sum_{i=1}^{n} E[X_i]$$

holds when $X_i, \dots X_n$ are dependent.

eg. For dices, $E[x] = \frac{1}{6} \cdot (1 + 2 + \cdots + 6)$. When rowing two dices, by the Linearity of expectation, $E[x] = 3.5 + 3.5 = 7$. If two dices output the same number magically, we still have $E(x) = (2 + 4 + 6 + ... + 12)/6 = 7$.

For our case, since comparison takes up most of the running time, we calculalte the average times of comparison. Firstly, if a_i, a_j are compared, one must be pivot and the other goes to the next layer of the division tree, which will only be compared once; When we consider the ordered sequence, a_i, a_j are compared iff no element betweeen them are chosen as pivots, else they will be in different subsets, as is shown in 4.

$$E[X_{ij}] = \Pr(X_{ij} = 1) \cdot 1 + \Pr(X_{ij} = 0) \cdot 0 = Pr(X_{ij} = 1) = \frac{2}{j - i + 1}$$

$$i = 1 : \frac{2}{2} + \frac{2}{3} + \cdots + \frac{2}{n} = \sum_{i=1}^{n} \frac{2}{i} = 2 \cdot \sum_{i=1}^{n} \frac{1}{i} = 2 \cdot \log n$$

$$i = 2 : \frac{2}{2} + \frac{2}{3} + \cdots + \frac{2}{n-1} \cdots$$

To sum up, worst-case $O(n^2)$, expected average running time $\Theta(n \log n)$

2.5.2 Median of the medians(1973)

How to pick a **good pivot**? In other words, how to make sure $|L|$ and $|R|$ are approximately equal?

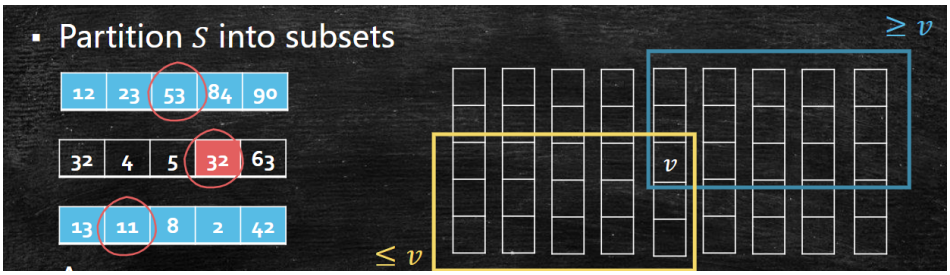


Figure 5: Median of the medians

- Partition S into subsets with size 5: O(n)
- Find the medians of each subset: $O(n/5)$ (since operations in a fixed-sized set is O(1)).
- Find the median of the medians: $T(n/5)$.
- Elements remaining: According to 5, there is $n - \frac{n}{5} \cdot \frac{1}{2} \cdot 3 = \frac{7n}{10}$ elements left.

$$T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n)$$

Proof. Proof by first guessing: $T(n) = O(n)$
Assume that $T(n) \leq cn$
$T(\frac{n}{5}) < \frac{cn}{5}$,$T(\frac{7n}{10}) < \frac{7cn}{10}$, $T(n) = T(\frac{n}{5}) + T(\frac{7n}{10}) + O(n) < 0.9cn + bn \leq cn$. which holds when we set c to be bigger then b.

□

If the size of subset is i=2m+1, $m \in N$. To obtain the median of medians, we have $O(\frac{n}{2m+1})$. There are $n - \frac{n}{2m+1} \cdot \frac{1}{2} \cdot (m+1) = \frac{(3m+1)n}{2(2m+1)}$ elements left.

$$T(n) = T(\frac{n}{2m+1}) + T(\frac{(3m+1)n}{2(2m+1)}) + O(n) = T(\frac{(3m+3)n}{2(2m+1)})$$

However, in real case, the const is much bigger than 'quick sort' approach with a random pivot.

2.6 Clostest Pair

We can improve the naive algorithm by **sorting**. Imagine in 1-dim case, sorting ($O(n \log n)$) plus comparison between neighbouring points ($O(n)$) results in an $O(n \log n)$ algorithm.

Consider 2-dim case, apparently using only x sequence or y sequence cannot lead us to the right answer. Some tricks are implemented:

1. **First sort by x, then by y**
   The only use of sorting by x is to divide the original dots into x-neighbouring subsets. After the division, any changes inside the subsets are allowed, so we merge the 'sort by y-axis step' into the 'conquer step', which eliminates the $O(n \log n)$ sorting operation in each layer.

2. **Bound the search plane**
   If we've found the minimum distance inside A and B separately, we only need to consider the distance between points across A and B. Additionally, $|x_i - x_{mid}| < dist$,which forms the $2\delta - strip$ shown in fig. Similarly, during the conquer step, for each point p_i, we only consider

p_j that satisfies $|y_i - y_j| < dist$. By proposition 2.1, we magically found that only the distance between **four** neighbouring points are considered for each point (as is shown in fig 7)

Proposition 2.1. At most 4 points can appear in a $\delta \times \delta$ square, so that the distance between any two points are no smaller than δ.

Proof. Divide the square into four smaller squares, as in shown in fig 6. Since two points are at most $\frac{\delta}{\sqrt{2}} < \delta$ apart, at most one point can exist in each small square. So in total, at most four points appear in the square. □

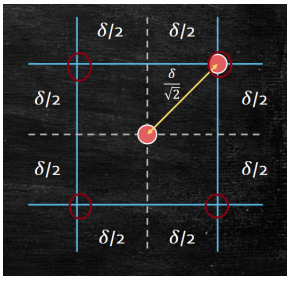


Figure 6: At most 4 points appear in the square

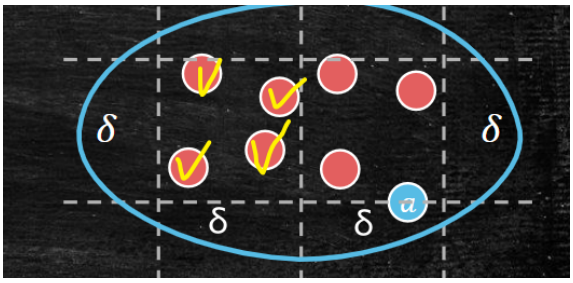


Figure 7: Only four neighbouring points need to be considered for each point a

This brings out an $O(n \log n)$ alogorithm. The sorting by x-coordinate takes $O(n \log n)$, and the 'divide and conquer' step amounts to adding constant steps(since each point only has four neighbouring points to count) to MergeSort.

2.7 Fast Fourier Transform and Polynomial Multiplication*

See maintext book for more.

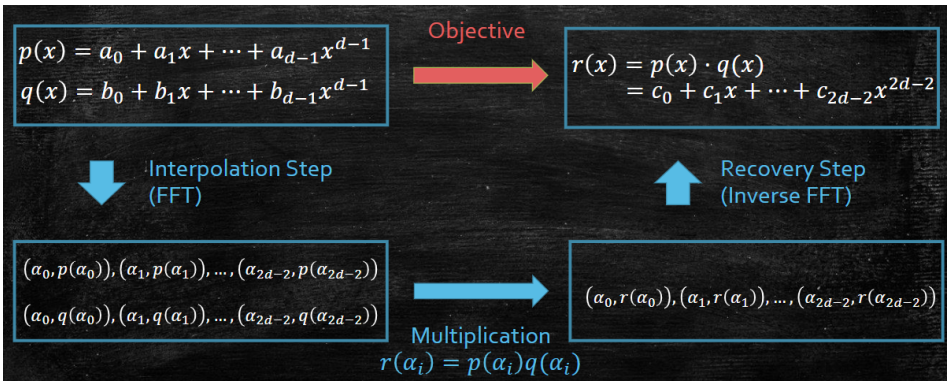


Figure 8: FFT

Algorithm 6: FindClosestPair

Input: A list of pairs P: $p_1 = (a_1, b_1), \ldots p_i = (a_i, b_i), \ldots, p_n = (a_n, b_n)$
Output: distance of closest two pairs in P: dist

```
if |P| == 1 then
    return
end
Sort P by x-coordinate.
Divide P into two equally sized subsets A,B, the point in the middle p_mid = (a_mid, b_mid).
dist=min(FindClosestPair(A),FindClosestPair(B))
Merge A and B by y-cordinate to form C.
foreach c_i = (a_i, b_i) in C do
    if |a_i − a_mid| < dist then
        add c_i to S
    end
end
foreach s_i = (a_i, b_i) in S do
    j=i+1
    while |b_i − b_j| < dist do
        dist=min(dist, ‖b_i − b_{i+1}‖)
        j=j+1
    end
end
return dist
```

3 Graph

Terminologies

1. vertex, vertice, edge, edges, index, index
2. Adjacency Matrix, Adjacency List
3. reachability, connected component
4. Tree: a connected and acyclic graph, containing backedge and Tree edge.
5. DAG: Directed Acyclic Graph
6. Sink(Source): A collection of vertice that doesn't have outgoing(ingoing) edges.
7. Partition: $C_1, C_2, \ldots, C_n$ is a partition of C iff their union is C and they don't intersect.
8. cut: Given a partion A,B of P, value defined by $c(A, B) = |E(A, B)|$(number of edges across two sets).

3.1 Depth First Search

3.1.1 Existence of cycles

Use dfs to traverse all the vertice in a graph. First reach a vertex, then visit its connected component. Most effective when stored in an adjacency list. Dfs can form a tree(forest).
Specially, we analyze the case of directed graph. For directed graph, there exists 4 kinds of edges:

1. **Tree Edges** Edges in the DFS search: (u, v) where marked(v) = false

2. **Forward Edges** Edges that point to non-child descendants
3. **Back Edges** Edges that point to ancestors
4. **Cross Edges** All the other edges: edges point to vertices on other tree doesn't exist for undirected graph, since if (c, f) occurs it will only be tree edge.

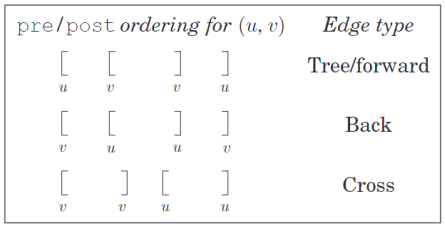


Figure 9: Directed Graph

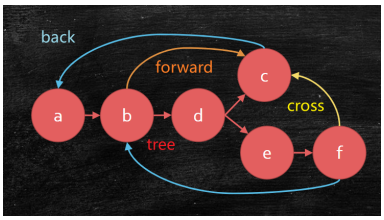


Figure 10: colored edges

Correctness of Algorithm:
If there exists a back edge in directed graph, a cycle exists.
If a cycle exists, then a backedge must be found. Since a tree is acyclic.
By algorithm 7 plus the condition of the existence of a backedge ($pre[v] < pre[u]$ and $post[u] < post[v]$), depicted in 9

Algorithm 7: DFS on Directed Graph

```
1  clk=0
2  foreach v ∈ V do
3  |  if marked[v]=False then
4  |  |  explore(v)
5  |  |  clk++ # used to calculate the number of connected components
6  |  end
7  end
8  Function explore(u):
9  |  pre[u]=clk
10 |  clk++
11 |  visited[u]=false
12 |  foreach (u, v) ∈ E do
13 |  |  if marked[v]=false then
14 |  |  |  explore (v)
15 |  |  end
16 |  end
17 |  post[u]=clk
18 |  clk++
```

3.1.2 Application: Topological Ordering

A DAG must have topological ordering, since:

1. Every DAG have a sink.
   Intuitively, if we start at v, since there exists an outgoing edge for every vertex. Since DAG doesn't have cycles, in finite steps, we'll visit every vertex,

2. Find a topological order.
   When deleting sink vertice and edges pointing to them, we obtain a new DAG. By repeting the operation of finding sink (visiting sink at last), and deleting sink to construct new graph, and finding sink in the new graph... We successfully find a topological ordering by following the inverse sequence of the sinks.

Time Complexity:
Finding a sink takes $O(|V|)$, removing a sink to update graph with $|V|$ rounds.
A trick is that the vertice with the earliest post ordering is a sink. If not, the outgoing point points to another vertex that finishes earlier. (consider the fact that no edge (u,v) exists in DAG that satisfied $post[v] > post[u]$. See 9, backedges satisfy, but doesn't exist in DAG) Therefore, the topological ordering is the inverse order of finishing points, which brings the time complexity of DFS down to $|E| + |V|$.
Correctness:
If (y,x) exists, y is not a sink since x is in the graph, contradiction!

3.1.3 Strongly Connected Components for Directed Graph

All the SCC forms a partition of C. i.e. for every v, $\exists C_i$ so that $v \in C_i$. For completeness, by construction, every single v can be formed as a SCC. For non-intersection, if $v \in C_i, C_j$, C_i and C_j together forms a bigger C.C., as every vertex from C_i can reach points from C_j through v.

- Note that we can't simply employ DFS to find C.C., as although you find reachable points from a certain vertice v, the vertice you explore may not be reachable from each other.
- Note that unlike topological ordering, the node with the earliest post time doesn't necessarily occur in sink SCC, as the vertice with outer edges are previsited where our algorithm ends in somewhere else.i.e.,for graph 11 consider DFS in sequence 5,6,7,8, apparently, the 8 has the smallest post value, but is not in sink.
- However, the vertice with biggest post time must be in source SCC. Assume u has the largest finish time, which must be a root of one DFS tree. Suppose u isn't in source SCC, i.e. $\exists v$ s.t. (v, u) exists. v isn't in the tree of u, as (u, v) doesn't exist. ,v cannot start earlier than u, v isn't in another DFS tree.
- Consider the meta-graph, where connected components form a big node, it must be DAG, else contradict with biggest SCC, therefore, there must be a sink. Also, a way to find whether v is reachable from u is to check the connectivity of bigger SCCs, U and V.

Algorithm 8: Strongly Connected Component

Input: DAG:G
Output: num of SCC
1 Deduce G's inverse G^R from G
2 Run DFS on G^R to find source SSC, which is sink SSC of G, record it in the descending finish time as F.
3 DFS on G in the order of F.

Time Complexity: Naive Case: Find a sink, delete it and repeat again...which takes $O(|V||E|)$. Better Case: Doing DFS twice, shown in algorithm 8, taking $O(|V| + |E|)$.

3.2 Breadth First Search

Difference between BFS and DFS: Firstly it cannot tell the difference between crossedge and backedge, and it has no forward edge(as the only chance of visiting unvisited nodes forms a tree edge , i.e. nodes in layer 2 can only reach unvisited nodes and index them with 3). Additionally, BFS cannot detect cycles(=detect backedge), or find SCCs(based on cycles), or do topological order.
Shortest path Algorithms:

- BFS: unweighted graphs
- Dijkstra: positively weighted graphs
- Bellman-Ford: general weighted graphs. (update all the edges)
- Floyd-Warshall: all-pair shortest path

3.2.1 Unweighed Shortest Path in Undirected Graph

Algorithm 9: alg:bfs

```
1  Queue Q;
2  Function bfs(,Q,u):
3      for each v ∈ V marked[v] ← false
4      Q.push(s) marked[s] ← true while Q is not empty do
5          u=Q.top();Q.pop();
6          foreach (u, v) ∈ E do
7              if marked[v]=false then
8                  marked[v]=True
9                  Q.push(v)
10             end
11         end
12     end
```

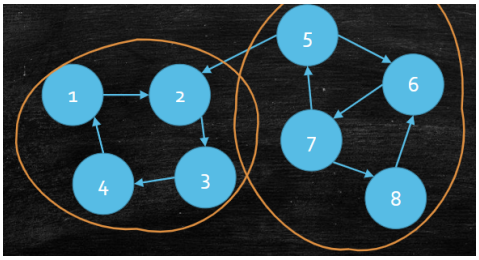


Figure 11: SCC

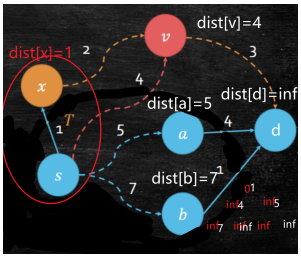


Figure 12: Correctness of Dijkstra's algorithm

3.2.2 Dijkstra: SP. with positive weight

Remark 3.1. difference between quantity and numerical value
Pay attention to the length of a numerical value. i.e., input length is n, the actual number complexity. Storing n quantities takes n storage space, but storing a n-length numerical value takes $\log n$ bits. Therefore, an increase in bits takes exponencial time complexity. $O(|V| + |E|)$, $|\cdot|$ is a quantity $O(w_max)$ is a numerical value.

- Works by keep dragging the closest vertex to the current SPT and updating the minimum path length.
- Dijkstra algorithm is based on the assumption that evrey step out of SPT(shortest path tree) at least increases the path length, so it only solves graphs without negative edges.
- A trick is to use heap (binary-heap, d-heap, Fibonachi-heap) to store the shortest distance from other vertice to s, so that we can quickly obtain the closest node from SPT, without the need to traverse all outer edges again. Creating the heap takes $O(|V|\log|V|)$, picking the top vertex takes $O(\log|V|)$, and updating all the nodes takes $O((|V|+|E|)\log|V|)$. i.e, for figure 10, consider the first step. We update x,a,b,v and the closest vertex x natually pops to the top.

Correctness of algorithm: If there exists $x \notin SPT, v \in SPT$, s.t. $dist(s \to x \to v) < dist(v)$, $dist(x) < dist(v)$, x should be in SPT,contradiction!

Algorithm 10: Di(j)kstra

```
Input: G=(V,E),s
Output: dist(u) is the shortest distance from s to u
foreach u ∈ V do
    dist(u)=inf
    pre(u)=Nan # initialize
end
dist(s)=0
H=makequeue(V)
while H is not empty do
    u=H.top();H.pop()
    foreach (u,v) ∈ E do
        if dist(v) > dist(u) + l(u,v) then
            dist(v) = dist(u) + l(u,v) # update the shortest dist
            pre(v)=u# record previous node
            update(H,v) # update the heap
        end
    end
end
```

3.2.3 Bellman-Ford: shortest path with negative weight

Dijkstra's algorithm works because the dist value it maintains is either overestimates or exactly correct. The key idea of Bellman-Ford is that it updates all the edges (kind of 'edgewise' operation) each time (different from Dijkstra's algorithm who updates nodes one step connected to SPT), which takes $O(|V||E|)$.

Correctness of algorithm:

1. After k rounds, dist(v) is the shortest path of all k-edge-paths.($dist[u_k] \le d(u_1, u_2, \ldots, u_k)$)
   For base step, dist[s] is the shortest dist of all 0-edge-paths.
   For induction step, suppose it is true for $k-1$ rounds, $dist[u_{k-1}] \le d(u_1, \ldots, u_{k-1})$. By Bellman-ford update condition, $dist[u_k] \le d(u_1, \ldots, u_{k-1}) + w(u_{k-1}, v)$, the righthand side means all the other k-edge-paths.
2. For graphs without negative cycle, all the shortest paths will have at most $|V|-1$ edges.
   If there are more than $|V|$ paths, a vertex must be visited twice, but by erasing the whole cycle results in a shorter path, contradiction!

Algorithm 11: Bellman-Ford

```
Input: G=(V,E),s
dist[s]=0, dist[x]=inf for vertex other than s
pre(v)=Nan
while some dist[x] updates do
    # update at most |V| rounds. foreach (u, v) ∈ E do
        if dist[v] > dist[u] + w(u, v) then
            dist[v]=dist[u]+w(u,v)
            pre(v)=u
        end
    end
end
```

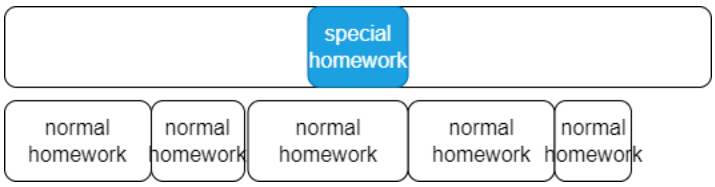


Figure 13: Diliberately constructed case

3. Negative cycle exists iff some dist is updated in the last($|V|^{th}$) round.

4. If no vertice updates in V^{th} round, then no vertice will be updated then. But if a single vertice is not updated.

5. If a vertex is upgraded after V rounds, then its path must include a negative cycle. A negative cycle can be found by tracing the recorded shortest path until two repeated nodes occurs. Tricks include running till the $2|V|^{th}$ round and see the vertice upgraded in $|V|^{th}$ and $2|V|^{th}$.

4 Greedy Algorithm

There are two kinds of algorithms, for exact optimal algorithms, prove by induction that you are on the way to an optimal solution. For approximation algorithms, you can either directly compare with optimal solution or find the upper/lower bound.

Homework++ and Partition

Definition: Given some homework with release time(r_i), deadline time(d_i) and duration of time(s_i). Give out whether we have a feasible solution to finish all the homework.

If r_i are the same, we do those with earliest d_i first.

If d_i are the same, we do those with the earliest r_i first.

If greedy algorithm cannot promise a feasible solution, then no solution occurs.

In essence, by Complexity Theory, the problem is NP-hard, as the deliberately designed example can be transformed into a Partition Problem (Partition the set into two parts, whose sum are equal), which we claim is NP-Hard. In 13, we only have two types of work. n normal homework starts at 0 and ends at $w + 1$, Special homework has release time $w/2$, deadline $w/2 + 1$ and size 1. Given an input $a_1, \ldots, a_n$,let $w = \sum_{i=1}^{n} a_i$.

4.1 Prim and Kruskal: Minimum spanning tree

Definition: A Minimum spanning tree (MSP) is the spanning tree(a tree that connects to all the vertice) with the minumum total weight.
Property: Shift edge operation on a cycle doesn't affect connectivity.

4.1.1 Prim's Algorithm

Prim's Algorithm:
Adding the edge out of the previous Tree T_i with the minimum weight to form T_{i+1}.
Correctness of algorithm:
The key idea for Prim is to guarantee that the tree with n edges T_n is a subset of **global** MST T. In inductive step, if T_i is part of T, then we can construct T_{i+1}, s.t. $T_{i+1} \in T$. Consider the first edge e out of T_i, if $e = (u, v) \notin T$, then adding e to T constructs a cycle C. Since e rather than $e' \in C, e' \notin T_i, e' \in T$, $w(e') \geq w(e)$. Since T is the smallest spanning tree, $T - w(e') + w(e) \geq T$, $w(e) \geq w(e')$, therefore, $w(e) = w(e')$. Therefore, by shifting the two edges guarantees T_{i+1} is a MST component of T.
Time complexity: $O((|E| + |V|) \log(|V|))$.

4.1.2 Kruskal's Algorithm

Kruskal's algorithm:
Similar to Prim's algorithm, it adds the smallest weighted edge that doesn't form a cycle with T_{i-1} to form T_i.
Correctness of Kruskal:
Similar to correctness of Prim, but the shift edge operation happens between components in a forest.
Time complxity:
$O(|E| \log(|E|))$ for sorting edges, and $|E|$ iterations of finding the representative element. A faster data structure is by **unified set**, which is $O(|E| \log(|V|))$

4.1.3 unified set

store groups by their representative element (the children are represented by the root vertice of their tree).

- Find: return the representative element in a group.$(2|E|)$

- Union: Merge two groups.$(|V| - 1)$

$\pi(x)$ points to the parent of x, and $rank(x)$ means the height of the tree rooted at x.

- **Path Compression:** Set the parent of the nodes checked to representative element.
- **time compexty analysis by Amortized Cost:**

  check cycle:

Algorithm 12: Kruskal's Algorithm

```
Sort the edge set E to ascending order.
X={}
For each u ∈ V, makeset(u). # |V| create group
For each (u, v) ∈ E in ascending order
if find(u) ≠ find(v) then
    Add edge (u, v) to X
    union (u,v)
end
Function makeset(x):
    π(x) = x
    rank(x) = 0
Function find(x):
    while π(x) ≠ x do
        π(x)=find (π(x))
    end
    return x # path Compression
Function union(x,y):
    r_x = find(x) # r_x means root(representative) of x
    r_y = find(y)
    if r_x = r_y then
        return # already in the same group
    end
    if rank(r_x) > rank(r_y) then
        π(r_y) = r_x # merge the smaller tree to the bigger tree
    end
    if rank(r_x) = rank(r_y) then
        rank(r_y) = rank(r_y) + 1
    end
```

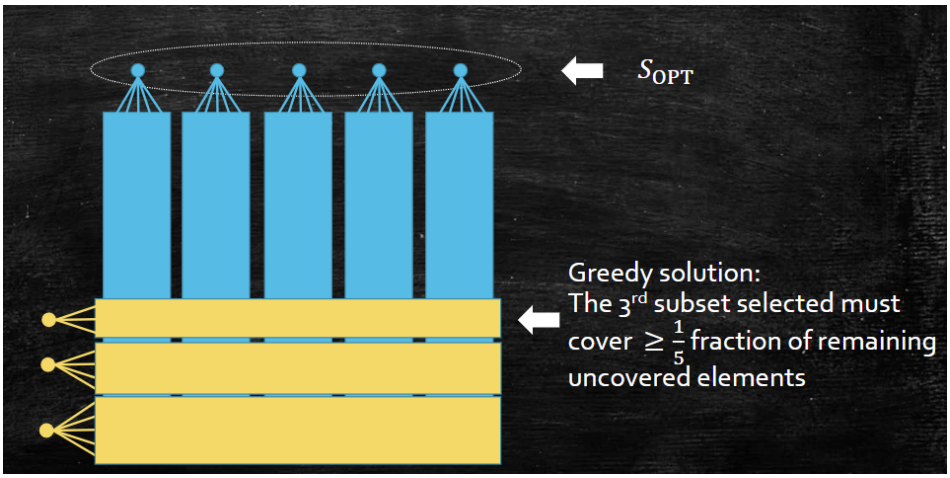


Figure 14: Directed Graph

4.2 Set Coverage

definition:

set Cover: Find a sub-collection $S \subseteq T$ with minimum $|S|$ such that $\bigcup_{A_i \in S} A_i = U$.

max k coverage: Given $k \in N$, find a sub-collection $S \subseteq T$ with $|S| \leq k$ that maximizes element number it covers. Denote $f(S) = |\bigcup_{A_i \in S} A_i|$.

Also viewed in bipartite graph, with the left set be set index, and the right set be the node index. Edges across two sets means that *node* $\in$ *set*.

Remark 4.1. Approximation Algorithm:
For maximize problem, algorithm $\mathcal{A}$ is called α-approximation algorithm if $\frac{\mathcal{A}(I)}{OPTIM(I)} \geq \alpha$. Normally, time complexity should be polynomial.

We deliberately construct the worst case: $|W_1| \geq \frac{1}{5}$biggest of remaining sets. $W_1 \cup W_2 = 1/5 + 1/5(1-1/5) = 1-(1-1/5)^2$. $f(S) \geq (1-(1-1/k)^k)$

Lemma 4.1. Denote $S_l = \{A_1, \dots A_l\}$ as the out put of greedy after l iterations, $S^* = \{O_1, \dots, O_k\}$ be **any** collection of k subsets.
$f(S_l) \geq (1-(1-1/k)^l) f(S^*)$

Proof. For base step, $l = 1$,by greedy nature, $f(S1 = A1) \geq f(O_i)$ for all O_i. Thus $f(S_1) \geq \frac{1}{k}\sum_{O_i \in S^*} f(O_i) = (1-(1-\frac{1}{k})^1) f(S^*)$ This implies when calculating the sum, union elements are calculated more than once. Also, the size of the first chosen set is larger than the average size of the optimal solution. Denote $\Delta(O_i|S_t) = f(S_t \cup O_i) - f(S_t)$.
$\Delta(A_{t+1}|S_t) \geq \frac{1}{k}\sum_{i=1}^{k} \Delta(O_i|S_t) \geq \frac{1}{k}\Delta(S^*|S_t)$.
For inductive step, suppose $f(S_t) \geq (1-(1-1/k)^t) f(S^*)$
$f(S_{t+1}) - f(S_t) \geq \frac{1}{k}(f(S^* \cup S_t) - f(S_t)) \geq \frac{1}{k}(f(S^*) - f(S_t))$.
$f(S_{t+1}) \geq \frac{1}{k} f(S^*) + (1-\frac{1}{k}) f(S_t) \geq \frac{1}{k} f(S^*) + (1-\frac{1}{k})(1-(1-1/k)^t) f(S^*) = ((1-(1-1/k)^{t+1}) f(S^*))$. □

For max-k-coverage, we at most choose k sets. $\lim_{k\to\infty}(1-(1-1/k)^l) = 1-1/e$.

For set cover problem, it is a lnn approximation. Suppose $|S^*| = k$ is optimal(f(S^*)=n). $f(S) \geq (1-(1-1/k)^{k \cdot lnn}) f(S^*) > (1-1/e^{lnn}) f(S^*) = n-1$ Therefore, $f(S) = n$.

4.3 Huffman Coding

1. prefix-free coding: each encoding is not a prefix of other encodings, i.e.,no vertice is an ancestor of others.
2. optim prefix code(with minimum avg_len):$avg\_len = \sum w(\sigma) \cdot len(\sigma)$, where w is the frequency and len is the length of the encoding.
3. Huffman Tree is constructed by replacing two vertice with minimum w with a new parent vertice whose weight is their weight sum repeatedly.
4. The correctness is guaranteed. An intuitive way is the more frequent an encoding appears, the shorter the encoding is. Denote T as the Huffman Tree, and T' the tree that switched u, v, where $len(u) < len(v)$, $w(u) > w(v)$. avg_len(T)-avg_len(T')=$w(u) \cdot len(u) + w(v) \cdot len(v) - w(v)len(u) - w(u)len(v) = (w(u) - w(v))(len(u) - len(v)) < 0$.

   Next, we'll guarantee that every next step is optim, i.e. T_{i-1} is part of T. For base step,... For inductive step: If we merge A and B in T. If A is paired with C in T^*, then $w(C) > w(A)$,$w(B) \leq w(C)$. If $w(B) = w(C)$, changing B and C preserves avg_len. If $w(B) < w(C)$, $len(B) \geq len(C)$ by above proof. If $len(B) == len(C)$, swapping B and C also preserves avg_len. If $len(B) > len(C)$, D is not processed, $w(D) > max(w(A), w(B))$, but $len(D) = len(B) > len(C)$, contradict with the fact that T^* is optimal(by above proof).

   Another intepretation is by sorting inequality, namely, the sum of $\sum ordered\_sequence \leq \sum random\_sequence \leq \sum inversed\_sequence$.
5. **time complexity analysis** By using binary-heap, every time we conduct pop twice to obtain the top two min vertice and a push to add the new parent vertice in the tree, which takes $O(n \log n)$.

4.3.1 K-Centers

Find a set of k centers, $S = \{s_1, \ldots, s_k\} \subseteq V$, $|S| = k$ that minimizes $f(S) = \max_{v \in V} \min_{s \in S}(s, v)$ which means the maximim distance of any vertex $v \in V$ to its closest center $s \in S$. Note: k-center(2), k-means(9), k-median(5) are all np-hard, but have relatively good approximation algorithms.

Remark 4.2. (V,d) is a matrix space if:

1. d(v1,v2)=0 iff v1=v2
2. d(v1,v2)=d(v2,v1)
3. $d(v1, v3) + d(v3, v2) \geq d(v1, v2)$

e.g., G=(V,E,$w > 0$)

Algorithm: Iteratively pick the center farthest to the existing centers. Pick the first by random. by Dijkstra for weighted graph and BFS for unweighted.
Note: Intuitively, we want to choose the vertex with largest degree. However, this guarantees that more vertice are closer to center, but isn't align with our objective of minimizing the dist(furthest vertice).

The algorithm is **2-approximation**.

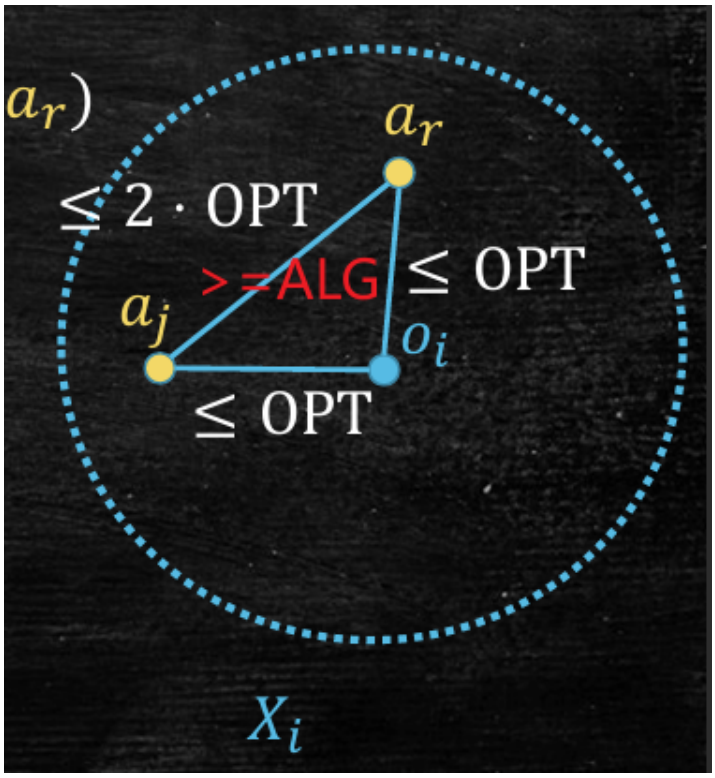


Figure 15: K-center

Denote $OPT = \max \min d(o_1, v)$. $A = \{a_1, \ldots a_k\}$ for algorithm result, $X_i = \{v_1, \ldots v_i\}$ as the set of vertice closest to o_i, optim solution $O = \{o_1, \ldots, o_k\}$

Case 1:Suppose $A \cap X_i \neq \varnothing$ for all i. Then each X_i contains a center, as the biggest length between two vertice in X_i is smaller than $2OPT$ by triangle inequality.

Case 2:Suppose $\exists A \cap X_i = \varnothing$. Suppose we choose a_j and $a_r \in X_i$. Let a_r be the second center chosen in X_i. By greedy choice, before a_r was chosen, the algorithm answer $ALG = d(A, a_r) \leq d(a_j, a_r)$. Additionally, $d(a_j, a_r) \leq d(a_j, o_i) + d(a_r, o_i) \leq 2OPT$.

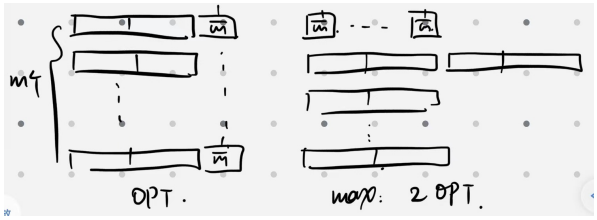
Also, it is the best algorithm, by simply considering (u-v-w),while choosing u first.

In essence, k-center is an NP-Hard problem, by changing it into a claimed NP-Hard problem dominating set. **A dominated set** is a subset of vertices S such that, for any $v \in V$ S, there is a vertex $u \in S$ that is adjacent to v. A dominated set problem asks to output whether a dominated set exists. If $\exists$ a polynomial time $(2 - \epsilon)$-approximation algorithm $\mathcal{A}$ for K-center. Denote the algorithm with slight modification running on (G, k) to output dominated set result as If $\mathcal{A}'$. If $\mathcal{A}'(G, k) = Yes$, $OPT \leq 1$. If $\mathcal{A}'(G, k) = No$, $ANS \geq 2$,$OPT \geq 2/(2 - \epsilon)$. Therefore, G has a dominated set of size k iff K-center solution of (G, k) output by A has cost 1. Also called gap(amplifying)-reduction.

Remark 4.3. Two ways to solve NPH problem, either by hyristic algorithm (with no approximation lower bound or guarantee of optimal solution) or approximation, restrictions on the input,

4.3.2 local search: max-cut

- Start with any partition $\{A, B\}$. If moving a vertex u from A to B or from B to A increases $c(A, B)$, move it. Terminate until no such movement is possible



- **Time Complexity** checking whether to update takes $O(|E|)$, every update you earns at least an edge, so it takes $O(|E|)$ rounds. The total algorithm is $O(|E|^2)$, polynomial!
- **Approximation** When the algorithm terminates, $\forall u$, at least 0.5deg(u) edges are in the cut, which leads to a 0.5-approximation.

$$c(A,B) \geq \frac{1}{2}\sum_{u\in V}\frac{1}{2}deg(u) = \frac{1}{2}|E|$$

- better approximation algorithm: 0.878-approximation(semi-definite programming)

Scheduling

n jobs, w_1, w_2 time to complete, m processors, minimize make-span(the time when all the jobs are completed)

2 processors, a bunch of works leads to partition problem(NP-Hard).

Greedy intuitive: iteratively put a job on a processor with minimum load (earliest finish time) Without sorting, a counter example is that with m 1-work and m $\frac{1}{m}$-work, with m processors, when $m \to \infty$the optimal solution is 1, but we obtain 2. By sorting first, placing bigger tasks first, we get $\frac{4}{3}-$ apprximation.

5 Dynamic Programming

From smaller subproblems to larger subproblems.

- **DAGs:** guarantees topological order, i.e. each sub-problem only depends on previous sub-problem

- **Induction:** solve sub-problems in topological order, implemented by a for loop.

- **Memoization:** (not memorization)"backward" recursion with "storing"

Different from Divide and Conquer, where the scale decreases exponentially by a **'magic' merge operation**, DP reduces the scale a little bit.

Difference between Induction and Memoization: Theoretically, time-complexity is the same. Memoization has the advantage of not needing to compute **topological order**, and some **not-computed sub-problems**. But in coding, 'throwing recursive call' is time costing.

5.1 Longest Increasing Subsequence

Formation of a DAG: establish a node i($i \in [1,n]$) for each element a_i, if $i < j$ and $a_i < a_j$, then $(a_1, a_j) \in E$. Then the problem is to find the longest path in the DAG. Construct L(i) by:

$$L(j) = 1 + \max L(i) : (i,j) \in E$$

Time Complexity is $O(|V||E|) \leq O(n^2)$.
Correctness of Algorithm:

We'll prove that L(i) is the length of LIS ending at a_i.

For base step, L(0)=0 is the length of LIS for an empty sequence.

For induction step, assume that L(i) is the length of LIS ending at a_i, Let S be the LIS ending at a_i and $T = S\backslash\{a_i\}$, the last number of T is a_t. Prove by $LIS[i] \geq |S|$, and $LIS[i] \leq |S|$. Since T is a LIS ending at a_t, $|T| = L(t)$, so $|S| = L(t) + 1$. $L(i) = 1 + \max_{a_j < a_i, j<i}\{L(j)\} \geq L(t) + 1 = |S|$.

On the other hand, $L(i) = 1 + \max_{a_j < a_i, j<i} L(j) + 1 \leq 1 + |T| = |S|$, so $L[i] \leq |S|$.

Therefore, $L(i) = |S|$.

5.2 Edit Distance

Edit distance is the minimum number of editsinsertions, deletions, and substitutions of charactersneeded to transform the first string into the second. We denote $E(i, j)$ as the editing distance between $x[1..i]$ and $y[1..j]$.

$$E(i, j) = \min(1 + E(i-1, j) + E(i, j-1) + E(i-1, j-1))$$

diff(i,j)=0 iff x[i]=y[j]. Forming a $m \times n$ table, and each cell is computed only once, which results in **Time Complexity** of $O(mn)$.

Algorithm 13: Edit Distance

Input: Two strings $x[1..m]$ and $y[1..n]$
Output: Edit distance $E(m, n)$

```
for i = 1 to m do
    E(i, 0) = i;
end
for j = 1 to n do
    E(0, j) = j;
end
for i = 1 to m do
    for j = 1 to n do
        E(i, j) = min(1 + E(i − 1, j), 1 + E(i, j − 1), diff(i, j) + E(i − 1, j − 1));
    end
end
return E(m, n);
```

Palindrome is a string that reads the same backward as forward, e.g., madam, racecar, abba, etc. Likewize, in the palindrome problem, $H(i, j)$ to denote the biggest palindrome length.

$$H(i, j) = \begin{cases} H(i+1, j-1) + 2, & \text{if } A[i] = A[j], \\ \max\{H(i, j-1), H(i+1, j)\}, & \text{o.w.} \end{cases} \tag{5}$$

5.3 Nnapsack Problem

Denote $s[i] = v[i]/c[i]$, s means the ratio of value to cost.

1. indivisibility
   If it is divisible, if you choose the object with the largest xingjiabi each time, it is optimal, else you can switch two objects and get a better one, which results in a better solution

   If it is indivisible, it is essentially an NP-Hard problem. As a problem where r of all objects are equal and v=totalSum/2 is a Partition Problem.

For **time-complexity**, it is not a polynomial problem, with $O(nW)$, where W is a numerical value rather than size.

We can control the precision of W by rounding, but the new optim solution you obtain may be infeasible in the original problem.

Howvever, if you round value, the feasibility of solutions doesn't change. Let $V = \max v_i$, $OPT \leq nV$. Denote $A[i, v]$ as the minimum cost cost(S) if we select till the i^{th} item with exactly value v.

$$A(i+1, v) = \begin{cases} \min A[i, v], c_{i+1} + A[i, v - v_{i+1}] if v[i+1] < v \\ A(i, v) \qquad \text{o.w.} \end{cases} \tag{6}$$

$$f[i][v] = \max f[i-1][v], f[i-1][v - c[i]] + w[i]$$

Where $f[i][v]$ means putting the first i objects into a v-capacity Knapsack. For every object i, you either put it in the bag, where the left capacity is $v - c[i]$, or not put it in, which is equal to $f[i-1][v]$.

Remark 5.1. Fully Polynomial Time Approximation Scheme(FPTAS) gives a $(1-\epsilon)-$approximation with running time polynomial in term of n and $\frac{1}{\epsilon}$

Algorithms that can **scale down** the problem to constant length and use naive explore algorithm are commonly FPTAS.

Polynomial Time Approximation Scheme(PTAS) gives a $(1-\epsilon)-$approximation with running time polynomial in term of n and constant ϵ.

Some have fixed constant ratio

5.4 Floyd-Warshall Algorithm and All pairs shortest paths

Floyd-Warshall Algorithm finds the shortest path between any two vertice on a directed, real number weighted(but no negetive-weighted cycles) graph.

Run bellman-Ford for all vertices takes $O(|V|^2|E|)$, but the algorithm reduces to $O(|V|^3)$.
Denote $dist_k(i, j)$ as the distance between i and j by using intermediate vertices from $1, \ldots, k$, not including start and end vertice. In implementation, also denoted as $d(i, j, k)$.

Algorithm: Gradually expand the set of permissible intermediate nodes by updateing one node at a time, updating the shortest path lengths at each stage. Eventually this set grows to all of V, at which point all vertices are allowed to be on all paths, and we have found the true shortest paths between vertices of the graph.

$$dist_{k_0+1}(i, j) = \min dist_{k_0+1}(i, j), dist_{k_0}(i, k_0 + 1) + dist_{k_0}(k_0 + 1, j)$$

$d(i, k, k-1) = d(i, k, k)$ holds because no negative cycle exists.

A trick is that we can reduce **space complexity** to $O(|V|^2)$ by using two $|V| \times |V|$ matrices, one for k and one for $k-1$. original: $d(i, j, k) = \min(d(i, j, k-1), d(i, k, k-1) + d(k, j, k-1))$ Now: $d(i, j) = \min(d(i, j), d(i, k) + d(k, j))$ Whether $d(i, k)$ and $d(k, j)$ are updated doens't matter, as $d(i, k, k-1) = d(i, k, k)$

5.5 DP with Exhaustice Searches

Given an $m \times n$ chessboard, how many different ways we can place kings such that no pair of them are in the attack range of each other.

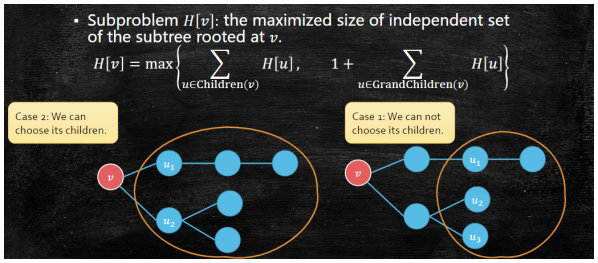
Naive exhaustive search takes $O(2^{mn})$, and a DP-based algorithm takes $O(4^n m)$

Algorithm 14: Floyd-Warshall Algorithm

For each pair $i, j \in V$, set

$$d(i,j,0) = \begin{cases} 0 & \text{if } i = j \\ w(i,j) & \text{if } (i,j) \in E \\ \infty & \text{o.w.} \end{cases} \tag{7}$$

```
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            d(i,j,k) = min(d(i,j,k-1), d(i,k,k-1) + d(k,j,k-1));
            # or similarly d(i,j) = min(d(i,j), d(i,k) + d(k,j))
        end
    end
end
```



5.6 maximum independent set on trees

A set of vertices S is an independent set if no two vertices in S are adjacent. Consider a tree structure

Time complexity Each $H(u)$ is looked up exactly twice, on for its parent and one for its grandparent. This is faster than naive case, which is $O(2^n)$, as it contains many sub-optim solutions. Greedy Algorithm? Choose the leafs and delete them and their parents. If the leaf node ins't chosen in the optimal solution, then its parent must be chosen, or either choosing it or its parent adds the maximum independent set.

6 Network Flow

6.1 Maximum flow and Ford-Fulkerson algorithm

Flow network is a directed graph $G = (V, E)$ with a source s, a sink t, and a capacity function c.

- **capacity constraints:** for each $e \in E$,$f(e) \leq c(e)$
- **flow conservation:** for each $u \in V \setminus \{s, t\}$, $\sum_{v:(v,u)\in E} f(v,u) = \sum_{w:(u,w)\in E} f(u,w)$
- **flow value:** $v(f) = \sum_{v:(s,v)\in E} f(s,v)$

Residual network $G_f = (V, E_f)$ of G induced by f:

- **residual capacity:** $c^f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \text{ is in the original graph and it is not full} \\ f(v,u) & \text{if } (v,u) \in E \end{cases}$
- **residual network:** $E_f = \{(u,v) \in V \times V : c^f(u,v) > 0\}$

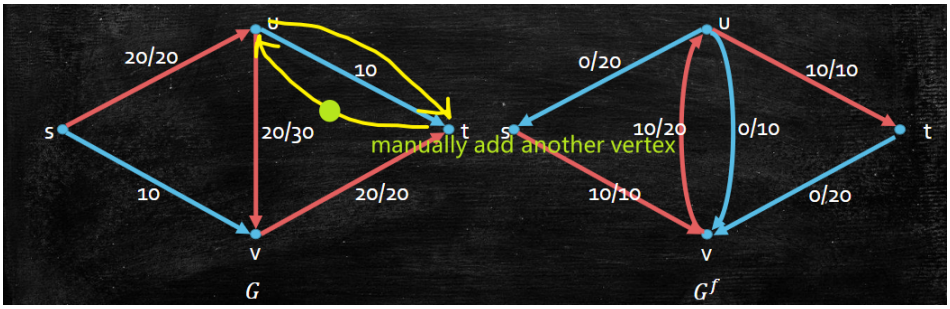


Figure 16: Residual network

Algorithm 15: Ford-Fulkerson algorithm

Input: G=(V,E),s,t,c
Output: f

initialize f s.t. $\forall e \in E$: f(e)=0; initialize $G^f = G$;
# Note that if both (u, v) and (v, u) exists, to avoid recursion , we add an extra vertex, as is shown in 16.
while *there is an s-t path p on* G^f **do**
    find an edge $e \in p$ with minimum capacity b
    foreach $e = (u, v) \in p$ **do**
        if $(u, v) \in E$ **then**
            update $f(e) = f(e) + b$
        end
        if $(v, u) \in E$ **then**
            update $f(e) = f(e) - b$
        end
        update G^f
    end
end
return f

6.1.1 Correctness of Algorithm

Denote $c(S,T) = \sum_{e=(u,v),u\in P_s,v\in P_t} w(e)$, i.e. the number of edges from P_s to P_t. **Minimum cut problem** minimizes $c(S,T)$.

Theorem 6.1. **Max-Flow-min-cut Theorem:**
The value of the maximum flow is exactly the value of the minimum cut:

$$max_f v(f) = min_{S,T} c(S,T)$$

.

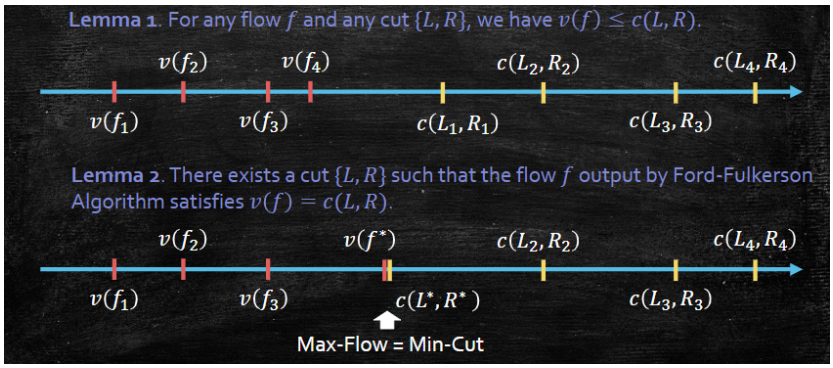


Figure 17: Proof of $f(S,T) = c(S,T)$

A visualization of proof is shown in 17.

- $max_f v(f) \le min_{S,T} c(S,T)$
  To prove this, we will first prove $v(f) = f(S,T) - f(T,S)$. If this holds, then $v(f) = f(S,T) - f(T,S) \le f(S,T) = \sum w(u,v) \le \sum c(u,v) = c(S,T)$.
  $f^o(u)$ is flow leaving u, $f^i(u)$ is flow entering u. $\sum_{u\in S}(f^o(u) - f^i(u)) = f^o(s) + \sum_{u\in L\setminus s} 0 = v(f)$
  $\sum_{u\in S \text{ or} T}(f^o(u) - f^i(u)) = \sum_{u\in S,v\in T} f(u,v) - \sum_{u\in T,v\in S} f(u,v)$. which is intuitive. Consider three kinds of edge, if $u,v \in S$, $f^o(u) - f^i(v) = 0$. If $u \in S, v \in T$, $f^o(u) - f^i(v) = f(u,v)$. If $u \in T, v \in S$, $f^o(u) - f^i(v) = -f(u,v)$. Putting them together we get the equation above.
- $\exists f$ produced by Ford-Fulkerson Algorithm satsifies:$v(f) = c(S,T)$ Denote S as vertice reachable from s in G^f.We'll prove that $f(S,T) = c(S,T)$, and $f(T,S) = 0$. If $v(f) \le c(S,T)$, at least an edge across the cut isn't full, so (u,v) must occur in the residual network, u is reachable from v, which contradict with the definition of partition. Similarly ,we can prove $f(T,S) = 0$

Therefore, we not only obtain a way to calculate maximum flow, but also minimum cut.

6.1.2 Time Complexity

Notice that min-cut is polynomial, but max-cut is NP-Hard. If capacities are integers, then each while-loop at least increases f by one, so it must terminate(after at most f_max times). If each $c(e)$ is an integer, then there exists a maximum flow f such that $f(e)$ is an integer for each e.(exists a max flow version whose edges are all integer), as Ford-Fulkerson Algorithm can produce an integer-capacity flow. With rational capacities, we can scale all numbers to integer. No need to consider irrational number, as they indeed will be presented as rational numbers in computer. We do dfs once from s,

points reachable from s is less than E, $O(|V| + |E|) = O(|E|)$, with f_max rounds, $O(|E| \cdot f_max)$(not polynomial). Consider the following case 18, if at first you get (u, v) iteratively, you will run 100000000 times, but if you get $s - u - t$ and then $s - v - t$, then you can end in 2 rounds.

For better choice of edge, we introduce Edmonds-Karp Algorithm.

6.1.3 Edmonds-Karp Algorithm

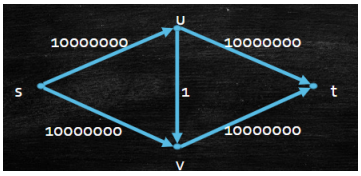


Figure 18: Residual network

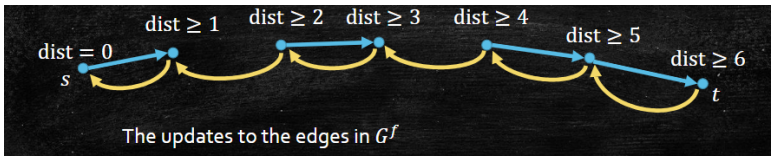


Figure 19: keyObservation

In every update, at least an edge satuate in G thus disapearing in G^f. Consider the new yellow edges in 19, they won't decrease the distance between two vertice, so $dist(u)$ (=number of edges(steps) needed to reach u from s) is non-decreasing through out the algorithm. So the distance will only be increased $|V| + 1$ times, and terminates after $O(|V|^2)$

Denote f_i as the current-flow in i^{th} iteration, G^{f_i} is the residual graph after i^{th} iteration of finding path. An edge (u, v) is critical(satuated) if the amount of flow pushed along p is $c^{f_i}(u, v)$. We will proof that the number of times an edge becomes critical isn't big. Consider the things happening bewteen an edge becomes critical twice. (u,v) in G, (v,u) in G^f, (v,u) in G, But $dist(v) > dist(u)$ by BFS, $dist^{i+j}(u) = dist^{i+j}(v) + 1 \geq dist^i(v) + 1 \geq dist^i(u) + 2$

- distance of u to s increases by 2 between during the interval of an edge being "critical" twice.
- Distance takes value from $\{0, 1, \ldots, |V|, \infty\}$ and never decrease
- When an edge is critical, the distance of its head increases by 1
- each edge can only be critical for $O(|V|)$ times
- At least one edge becomes critical in one iteration
- total number of iterations is $O(|V||E|)$
- each iteration takes $O(|E|)$ times
- Overall time complexity takes $O(|V||E|^2)$

6.2 Applications of Maximum Flow

6.2.1 Dinner Table Assignment

Students from m different universities participate in a conference. Each university i has r_i students. n tables, each table i can be shared by at most c_i students The problem has been transfered into whether max-flow has value $\sum_{i=1}^{m} r_i$

6.2.2 Tournament

Four teams, whose wins are as follows: A=40, B=38, C=37, D=29, and the matches unfinished is marked on the graph. If f is 8, then theretically D can win.

Algorithm 16: Edmonds-Karp Algorithm

Input: G=(V,E),s,t,c
Output: f

initialize f s.t. $\forall e \in E$: f(e)=0; initialize $G^f = G$;
while *there is an s-t path p on* G^f **do**
  find a path p by **BFS**;
  find an edge $e \in p$ with minimum capacity b
  foreach $e = (u, v) \in p$ **do**
    if $(u, v) \in E$ **then**
      update $f(e) = f(e) + b$
    end
    if $(v, u) \in E$ **then**
      update $f(e) = f(e) - b$
    end
    update G^f
  end
end
return f

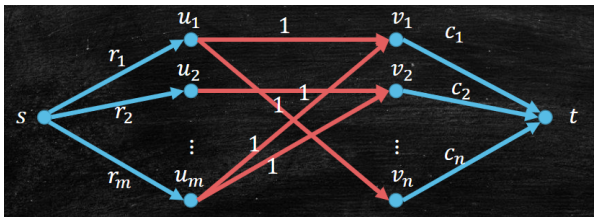


Figure 20: Dinner Tabel Assignment

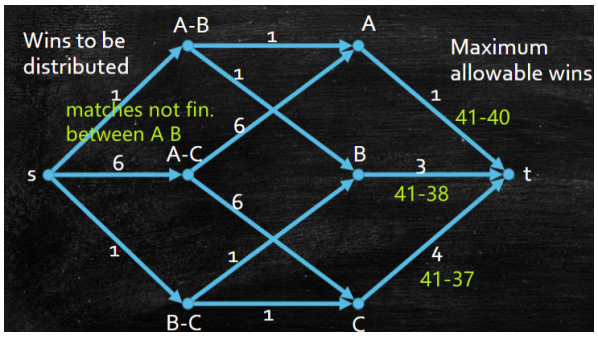


Figure 21: Tournament

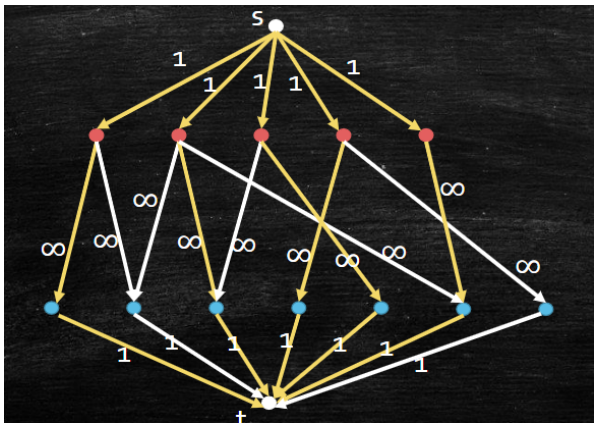


Figure 22: change maximum matching into max flow

6.2.3 Maximum Bipartite Matching

Example given in fig 22. For vertice starting from s or ending at t, capacity is 1. For other vertice, capacity is ∞. The maximum flow is equal to the maximum matching.

6.3 Big Theorem for Maximum Flow on Bipartite Graph

We will prove that the following four problems are equivalent for Bipartite Graph:

- **Maximum Flow:** Max flow from s to t satisfying capacity and flow conservation constraints.
- **Minimum Cut:** $c(S,T)$ is the min sum of edge weight from s to t with minimum total capacity.
- **Maximum Independent Set:** $S \subseteq V$, no edge between any two vertices in S.
- **Minimum Vertex Cover:** $S \subseteq V$, S contains at least one endpoint of each edge.

The number of Maximum Flow= The number of minimum cut

The number of smallest vertex cover=$|V|$-The number of maximum independent set

The main pipeline is shown in fig 23. We will prove the following four statements:

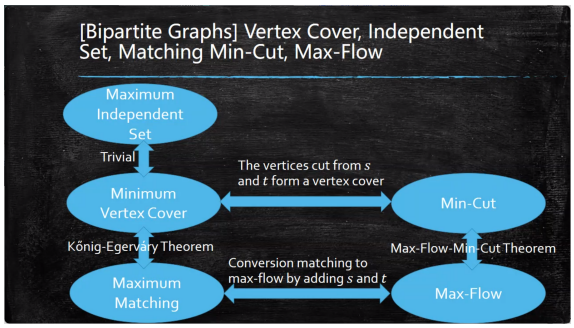


Figure 23: The proof pipeline of vertex cover, independent set, max flow and min cut on Bipartite Graph

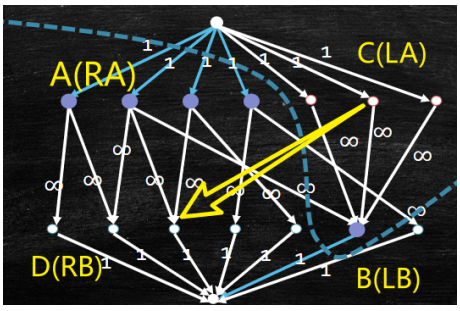


Figure 24: num of white edges ending at t are the max flow; blue edges are the min cut; purple vertice are the minimum cover; white vertice are the maximum independent set

- Maximum Independent Set ⇔ Minimum Vertex Cover
  S is an Independent set iff $V \setminus S$ is a vertex cover. Intuitively, when finding an independent set, edges are $\bullet - \circ$ or $\circ - \circ$, while finding a vertex cover, edges are $\circ - \bullet$ or $\bullet - \bullet$. For every edge in an independent set S, at least one of its endpoint must be in $V \setminus S$, else an edge exists between vertice in S. Consider $V \setminus S$, it forms a vertex cover, since all vertice connected to S are in $V \setminus S$. Therefore, maximum independent set is equal to $|V|$-minimum vertex cover.

- Min Vertex Cover ⇔ Min-Cut
  By the operation of max flow, $c(\{s\} \cup C \cup B, \{t\} \cup D \cup A) = c(s, A) + c(B, t)$, since $c(B, D) = c(C, A) = 0$ by the definition of Bipartite Graph, $c(B, A) = 0$ by the direction of edges, $c(C, D) = 0$, else min-cut goes to infinity. $c = |A| + |B|$, since $w(s, a \in A) = 1$. $A \cup B$ forms a vertex cover in the original undirected Bipartite Graph. If there exists $e = (u, v)$, $u, v \notin A \cup B$, $u \in C, v \in D$, contradiction! Suppose $A \cup B$ is a vertex cover, then there is no edge from C to D, else an edge isn't covered, contradiction! $\{s\} \cup C \cup B, \{t\} \cup D \cup A$ is a cut whose size is $|A| + |B|$ by similar approach. Therefore, min-cut is equal to min vertex cover.

- Min-Cut ⇔ Max-Flow

  Proven by Max-Flow-Min-Cut Theorem. Notice that when algorithm terminates, the points reachable from s but not t forms F, where minimum cut is $c(F, V \setminus F)$. The vertice set of cut-edges forms a vertice cover, as its complement is an independent set.

- Max Flow ⇔ Maximum Matching constructed in fig22. All matching problems can be solved in polynomial time

For a general graph, although finding a min vertex cover is np-hard, we can still find a 2-approximation based on maximal matching:

- The set of endpoints for all edges in a maximal matching is a vertex cover.
  Since for any edge $e = (u, v)$, one or both of u,v must be an endpoint of an edge in M. Otherwise, $M \cup \{e\}$ is still a matching, and M is not maximal

- For any maximal matching M, the size of any vertex cover is at least $|M|$.
  Since at least a vertex in a matching will be chosen.(Consider the extreme case)

- Since $|S(G)| = 2|M|$, $OPT(G) \geq |M|$, we successfully find a 2-approximation

7 Linear Programming

1. For Simplex Method, magically, average case polynomial time if add random Gaussian noise to the constraints. But with ellipsoid Method or Interior Point Method, it is polynomial time.

2. A linear program can be solved in a polynomial time.

3. Whenever a problem can be formulated by a linear program, it is polynomial-time solvable

7.1 Duality Problem

Use the linear combination of constraints to get the objective function.

1. For **Weak Duality Theorem**, Primal $OPT < DualOPT$

2. for **Strong Duality Theorem**, Primal OPT = Dual OPT.

3.

Maximum matching

$maximize \sum_{e \in E} x_e (\uparrow x) \sum_{e=(u,v) \in E} x_e \leq 1 (\forall u \in V)(Ax = b, A : incidencematrix, b : \uparrow)$for vertex u, at most one edge is selecte

Consider a triangle case, the maximum matching is 1, but the maximum independent set is 2. But by algorithm, for maximum matching and min vertex cover, edges(vertice) have 0.5, but is meaningless. minimum vertex cover

$$min \sum u \in V x_u x_u + x_v \geq 1 x_u \geq 0$$

The problem is, how to guarantee that the optim. sol. is int?

7.2 Strong LP-Duality and max-flow-min-cut

- LP for max-flow problem:

$$\begin{aligned} maximize \sum_{v:(s,v) \in E} f(s,u) & \\ f(u,v) \leq c(u,v) & \quad (y_{uv}) \\ \sum_{v:(v,u) \in E} f(v,u) - \sum_{w:(u,w) \in E} f(u,w) \leq 0 & \quad (z_u^+) \\ - \sum_{v:(v,u) \in E} f(v,u) + \sum_{w:(u,w) \in E} f(u,w) \leq 0 & \quad (z_u^-) \\ f(e) \geq 0 & \end{aligned}$$

- dual describes the **fractional** version of min-cut problem, denote $z_u = z_u^+ - z_u^-$

$$\begin{aligned} minimize \sum_{(s,u) \in E} c_{uv} y_{uv} & \\ y_{su} + z_u \geq 1 & \\ y_{vt} - z_v \geq 0 & \\ y_{uv} - z_u + z_v \geq 0 (u, v \in V \setminus \{s,t\}) & \\ y_{uv} \geq 0 & \end{aligned}$$

- dual program have **integral** optimum Proven by totally unimodular matrix theorem.(discuss from whether no 1, one 1, -1 and 1 occurs in the matrix, maybe by induction) If A is totally unimodular, then feasible solution of $Ax = b, x \geq 0$ is integral.
- apply Strong Duality Theorem to show max-flow-min-cut We will prove that A is totally unimodular.

8 P,NP,NP-Completeness

8.1 Basic Concepts

We have the following important definitions:

1. **Decision Problems**: those with output yes or no.
   Technically, a decision problem is a function $f : \sum^* \leftarrow \{0,1\}$, where $\sum^* = \bigcup_{n=0}^{\infty} \sum^n$, where $\sum^n$ is the set of all strings using alphabets in $\sum$ with length n, $x \in \sum^*$ is an instance.

2. $\mathcal{P}$: A problem is in $\mathcal{P}$ (usually easy) iff. it can be solved **polynomially**.
   Technically, there is a **polynomial** algorithm $\mathcal{A}$, such that $\mathcal{A}(x) = f(x)$ holds, or there exists a polynomial time TM $\mathcal{A}(x)$ (always terminates within $p(|x|)$ steps) that outputs the correct answer.

   $\mathcal{NP}$: Problems whose 'yes instance'(also called certificate) can be easily verified if a hint(an assignment of variables) is given, and any 'no instance' cannot be varified as 'yes'. (In test, you can just provide a certificate, and the others are straight forward)

3. $\mathcal{NP}$**-Hard**: All other NP problems can reduce to such problem, but they may not be in $\mathcal{NP}$.
   NP-hardness can describe optimization problems, e.g. Maximum Independent Set is NP-hard

   $\mathcal{NP}$**-Completeness**: The hardest problems in $\mathcal{NP}$, which are at least as hard as any other problem in $\mathcal{NP}$, or other problems reduce to them, $A \to NPC$.
   For decision problems: NP-complete = NP-hard + (in NP)

   $\mathcal{NP}$**-Intermediate**: Problems between $\mathcal{P}$ and $\mathcal{NP}$-Complete, with no natural examples, (since we haven't proven $P = NP$ yet)

4. **Reduction:** $A \to B$ means (karp) reduction from A to B; a mapping from A to B (see fig 25); $A \leq_k B$, B is harder than A; use B to construct A; input instance I, if A(I)=1, then B(I)=1; if A(I)=0, then B(I)=0. See fig 25.

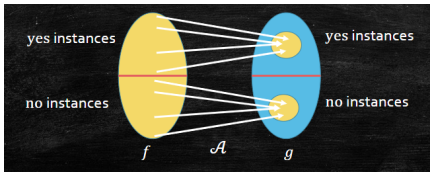


Figure 25: Reduction in Mapping's View

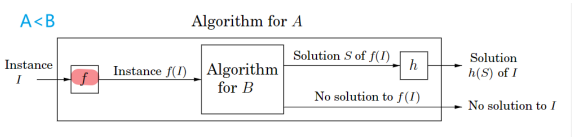


Figure 26: Reduction in algorithm's View: Key is to see B as a 'solver'

5. **Turing Machine**: state diagram with a tape; what the pointer reads at present determines the next state; special states: starting, halting (accept, reject). see fig 27

Examples for problems in $\mathcal{P}$:

- **Path:** Given a directed graph G and two vertices s and t, decide if there is a path from s to t.
- **k-flow:** Given a directed graph G, two vertices s and t, and flow capacity, decide if there is a path from s to t with at least k edges.
- **Prime:** Given n encoded in binary string, decide if n is a prime number.

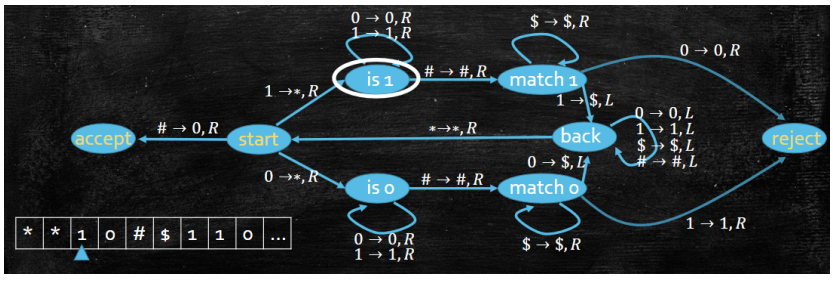


Figure 27: Given x,y with the same length, check whether they are identical; the lower pipeline checks whether the both strings are at 1, and the higher pipeline checks whether the both strings are at 0.

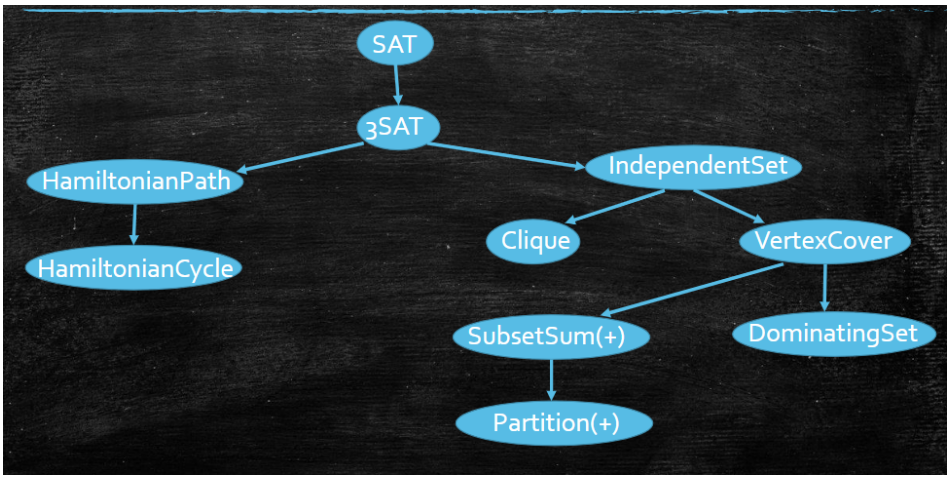


Figure 28: pipeline of our proof

Proof. $P \subseteq NP$

If a decision problem is in P, there exists polynomial algorithm $\mathcal{A}$ s.t. $A(x) = y$. If it is in NP, its output can be verified in polynomial time. We verify the answer through feeding input x into A directly, i.e. through algorithm $A'(x, c) = A(x)$. If $f(x) = 1$, then $\exists y$ s.t. A' accepts (x,y). If $f(x) = 0$, then $\forall y, A'$ rejects (x,y). Thus, $f \in A'$. □

8.2 5 NP-Completeness Problems

We will prove that following examples are all NP problems as they reduce to each other. First prove the pipeline in fig 28. The other way round is proven by Cook-Levin Theorem.

- SAT[ϕ]: Boolean Satisfiability Problem, given a CNF(conjunctive normal form: 'AND' of many clauses of 'OR', i.e.$\phi = (x_1 \vee x_2) \wedge (x_3 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_4)$)
  Reject if x does not encode a CNF formula ϕ or y does not encode a valid Boolean assignment; Check if y makes ϕ evaluated to true. Accept if it does, and reject if it does not
- Vertex Cover[$G = (V, E), k \in Z^+$]: Given an undirected unweighted graph G, decide if the graph has a vertex cover(contains at least a vertice of each edge) of size k.
- Independent Set[$G = (V, E), k \in Z^+$]: decide if the graph has an independent set(at most a vertice of each edge is chosen) of size k.
- Subset Sum[$S = \{a_1, \ldots, a_n\}, k \in \mathcal{Z}^+$]: Decide if there is a sub-collection $T \subseteq S$ such that $\sum_{a_1 \in T} a_i = k$
- Hamiltonian Path[$G = (V, E)$]: Given an undirected graph, decide whether it contains a Hamiltonian path (longest path; path containing each vertex exactly once).
- $SAT \to 3SAT$
  We transfer SAT expression into 3-SAT by introducing a connector y_i:
  $$x_1 \vee x_2 \vee \neg x_3 \vee \neg x_4 = (x_1 \vee x_2 \vee y_1) \wedge \neg y_1 \vee \neg x_3 \vee \neg x_4$$
- $3SAT \to$ Independent Set
  Construct Independent Set graph G' from 3-SAT formula ϕ:

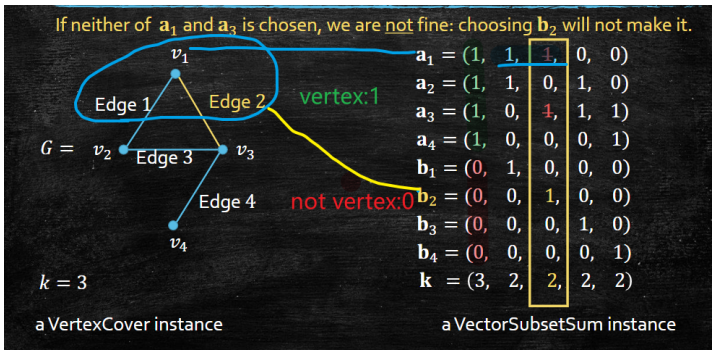


Figure 29: Vertex Cover $\leq_k$ VectorSubsetSum. Simple put, column sum=2 under such construction only implies that at least a vertice in each edge is chosen.

- For each variable x_i, construct a triangle T_i with vertices $x_i, \neg x_i, y_i$.
- For each clause C_j, add a vertex z_j and connect it to all vertices in T_i corresponding to literals in C_j.
- Set $k = m$, where m is the number of clauses in ϕ.

If ϕ is a yes instance, each clause must have a literal with value true. Pick exactly one vertex representing a true literal. Since S is an independent set, $|S| = k$, so $g(G, k) = 1$ If ϕ is a no instance, assume (G,m) is a yes instance. By triangle, you choose **exactly** one vertice in every triangle. Assigning true to literals representing the chosen vertices, and assign random value to others produces a certificate for 3-SAT.

- Independent Set → Vertex Cover

  The complement of Independent Set is Vertex Cover.

- Vertex Cover → SubsetSum (VertexCover → VectorSubsetSum → SubsetSum)
  - Vertex Cover$\leq_k$ VectorSubsetSum

    See 29 for the construction.
  - VectorSubsetSum$\leq_k$ SubsetSum

    We can convert a vector $a = (a[0], \ldots, a[m])$ to a large number by a large enough base that avoids carry, so each vector is uniquely represented by a number. Additions with vectors are now equivalent to additions with numbers.

- SubsetSum → Partition Given subsetsum problem (S,k). Suppose it is a 'yes' instance, denote sum(S)=s $\sum_{i<m} a_i = k$, $\sum_{m\leq i\leq k} a_i = s - k$. We hope $\sum_{m\leq i\leq k} + b = k$, $b = 2k - s$
- 3SAT → Hamiltonian Path*
  - 3SAT $\leq_k$ Directed Hamiltonian Path If ϕ has a yes instance, we will prove that there exists a Hamiltonian path staring from s to t: For each clause, choose a representative true literature. Go from s to t, and visit each v_j from its representative by taking a detour Prove the other way by contrapositive. If the graph has a Hamiltonian path, then ϕ is a yes instance. Each v_j has to be visited by a detour from a variable
  - Directed Hamilton Path $\leq_k$ Hamiltonian Path
- Hamiltonian Path → Hamiltonian Cycle

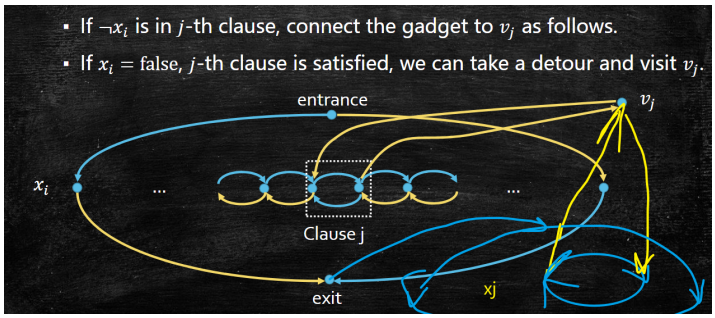


Figure 30: 3SAT $\leq_k$ Directed Hamiltonian Path

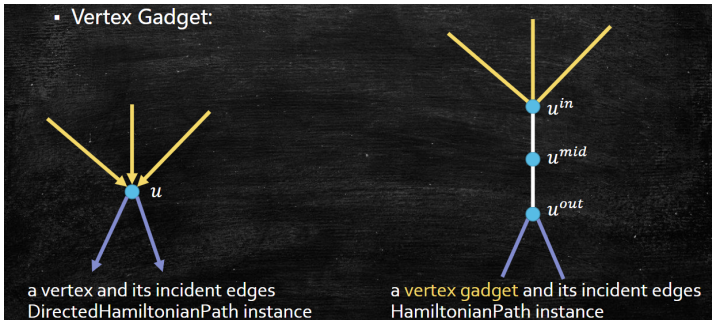


Figure 31: Directed Hamiltonian Path $\leq_k$ Hamiltonian Path

- By Cook-Levin Theorem, SAT is NP-complete, so they are all NP-complete.

  In essense, the theorem proves that a CNF formula is sufficient to simulate the execution of a Turing Machine. See **??**. For the neighbouring rows, the content differs by one bit. Since the TM operates in polynomial time, the number of columns are polynomial. Then the change can be computed in 'clause', so a CNF formula is sufficient to simulate the execution of a Turing Machine.

8.3 Summary of a NP Completeness Proof

1. Prove $f \in NP$.
2. Construct the reduction $g \leq_k f$. For a fixed instance x of g. Design transfer function ϕ, $y = \phi(x)$.
3. (Completeness) x is yes $\rightarrow$ y is yes
4. (Soundness) x is no $\rightarrow$ y is no. Or, proving the contrapositive y is yes $\rightarrow$ x is yes is often easier.