Divide and Conquer

Multiplication

- Problem: Given two n-digit numbers x and y, compute their product.
- T(n) = 3T(n/2) + O(n)
- T(n) = $O\left(n^{\log_2 3}\right)$
- ```
  1:
  2:    function KARATSUBA(x, y)
  3:        ▷ Append zeros to the left so that |x| = |y| = n = 2^k
  4:        if n = 1 then
  5:            return xy
  6:        ▷ Write x = a · 2^(n/2) + b and y = c · 2^(n/2) + d
  7:        α ← Karatsuba(a, c)
  8:        β ← Karatsuba(b, d)
  9:        γ ← Karatsuba(a + b, c + d)
  10:       return α · 2^n + (γ − α − β) · 2^(n/2) + β
  ```

Matrix Multiplication

- Magic Idea:
  - $P_1 = A(F - H)$
  - $P_2 = (A + B)H$
  - $P_3 = (C + D)E$
  - $P_4 = D(G - E)$
  - $P_5 = (A + D)(E + H)$
  - $P_6 = (B - D)(G + H)$
  - $P_7 = (A - C)(E + F)$
  - $XY = \begin{pmatrix} A & B \\ C & D \end{pmatrix}\begin{pmatrix} E & F \\ G & H \end{pmatrix}$

    $= \begin{pmatrix} P_5+P_4-P_2+P_6 & P_1+P_2 \\ P_3+P_4 & P_1+P_5-P_3-P_7 \end{pmatrix}$
- $7^{\log_2 n}$ multiplications

Sorting

- O(n log n) sorting algorithms
  - Merge Sort
  - Quick Sort
  - Heap Sort
- O(n^2) sorting algorithms
  - Bubble Sort
  - Selection Sort
  - Insertion Sort

Selection Problem

- Input: A set of n integers $x_1, x_2, ..., x_n$ and an integer k
- Output: The k-th smallest integer $x^\star$ among $x_1, x_2, ..., x_n$
- ```
  1:
  2:    function SELECT(S, k)
  3:        ▷ Choose a random value v in x_1, x_2, ...
  4:        ▷ Divide x_1, x_2, x_3, ... into three sets: L: x < v, M: x = v, R: x > v
  5:        if |L| ≥ k then
  6:            return Select(L, k)
  7:        if |L| + |M| < k then
  8:            return Select(R, k - |L| - |M|)
  9:        return v
  ```
- Running Time:
  - $\tau(n)$: Time we reduce n to $\frac{3n}{4}$
  - $T(n) = \tau(n) + T\left(\frac{3n}{4}\right)$
  - $E[T(n)] = E[\tau(n)] + E\left[T\left(\frac{3n}{4}\right)\right]$
  - $E(T(n)) = O(n)$
  - E[T(n)] = O(n)
- Medium of Medium
  - Partition S into subsets of 5 elements
  - find the medians of them
  - Finding a good pivot takes O(n) + T(n/5) time.
  - the subproblem size is at most $\frac{7n}{10}$.
  - T(n) = T($\frac{7n}{10}$) + O(n) + T($\frac{n}{5}$) = O(n)

Closest Pair

- Problem: Given n points in the plane, find the closest pair of points.
- Algorithm:
  1. Sort the points by x-coordinate
  2. Divide the points into two equal-sized sets by a vertical line $x = x_{\text{mid}}$
  3. Find the closest pair in each set recursively, let d_l, d_r be the distance.
  4. let $d = \min(d_l, d_r)$ and S' be the set of points at most d from the line.
  5. Sort the points in S' by y-coordinate
  6. For each point b, check 7 point above b, and find the closest pair.
  7. Return the closest pair.
- Running Time: $O(n \log^2 n)$

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

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

Graph

DFS

Running time: O(V+E)

- DFS Tree:
  - Undirected Graph:
    - Root: the first explored vertex
    - If we explore v from u, then v is u's child.
    - Two kinds of edges:
      - Tree edges
      - Back edges
  - Directly Graph:
    - Four kinds of edges:
      - Tree edges
      - Back edges
      - Cross edges
      - Forward edges

Application:

- Undirected Graph
  1. Reachability
  2. Connected Components(CC)
  3. T has back edges ↔ G has cycles.
- Directly Graph
  1. Topological Ordering (Directed Acyclic Graph, DAG).
     - Algorithm: sort vertices by descending order of finish time of DFS. O(|V|+|E|)
  2. Strongly Connected Components(SCC)
     - **Algorithm**

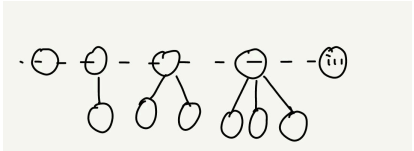
       ```
       1:  Construct G^R
       2:  DFS G^R with finish time
       3:  Choose v with the largest finish time
       4:  Explore(v) in G
       5:  When it returns, reached vertices form one SCC (|V_1|)
       6:  Remove them in both G and G^R |V| ← |V| − |V_1|
           |E| ← |E| − |ΔE|
       7:  Repeat from 2
       ```
     - Running Time: O()
     - **Algorithm(Super Plan)**

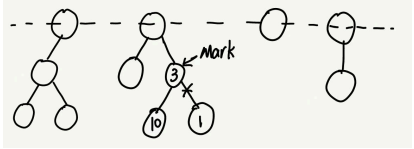
       ```
       1:  DFS G^R and maintain a sorted list by the finish time.
       2:  DFS G by the descending order of the finish time
           1. Keep explore vertices by the descending order.
           2. Do not start from a reached vertex.
       3:  Each explore() forms a SCC.
       ```

       To proof: **Lemma:** If SCC1 can reach SCC2 in G^R, then we have $\max_{v \in \text{SCC1}} \text{finish}[v] > \max_{u \in \text{SCC2}} \text{finish}[u]$
     - Running Time: O(|V|+|E|)

Shortest Path

- BFS (no weights) and Dijkstra (weighted)
- Dijkstra Algorithm
  - Core Idea: explore the unvisited vertex with the smallest distance first.
  - Running Time:
    - simple array:$O(V^2 + |E|)$
    - Binary Heap: $O(V + E \log |V|)$
    - Fibonacci Heap: O(E + V log |V|)
      - Each degree at most has one root!

        
      - The children of every vertex have the same degree property: Each degree at most has one root!
      - We only allow each non-root node to lose one child.
      - Max degree D is at most O(log n).
      - Cascading Cut

      - Amortized Analysis: Potential Function
        - $\hat{C} = C + \delta\Delta \cdot \Phi$
        - Potential Function: $\Phi = t + 2m$.

          t: #roots(extra), m: #marked nodes.
        - Update:
          - $\hat{C} = O(\#\text{CC} + 1) + \delta(\Delta\Phi) = O(\#\text{CC} + 1) + \delta((\#\text{CC} + 1) + (-\#\text{CC} + 1)) = O(1)$
          - Pop Min: $\hat{C} = O(t^- + D) + \delta\Delta t \leq O(t^- + 2D) + \delta(D - t^-) = O(D) = O(\log n)$

Shortest Path With Negative Weights

- Bellman-Ford Algorithm
  - ```
    1:
    2:    function BELLMAN_FORD(G, s)
    3:        ▷ dist[s] = 0, dist[x] = ∞ for other x ∈ V
    4:        while ∃ dist[x] is updated do
    5:            for each (u, v) ∈ E do
    6:                dist[v] ← min{dist[v], dist[u] + d(u, v
    7:        return "dist"
    ```
  - Correctness:
    - Lemma1: After k rounds, dist(v) is the shortest distance of all k-edge-path
    - Lemma2: The shortest distance of all |V|-edge-path can not be shorter than the shortest distance of all (|V|-1)-edge-path unless there is a Negative Cycle.
  - Running Time:
    - O(|V||E|)

Greedy

Minimun Spanning Tree

- Problem: Given a connected, undirected graph G = (V, E) with edge weights, find a minimum spanning tree.
- Prime & Kruskal
- **Algorithm 1:** Prime's Algorithm

  Input: A graph G = (V, E) with edge weights.

  Output: A minimum spanning tree T.

  1 $T \leftarrow \{\}, S \leftarrow \{s\}$;

  2 cost[s] = 0, cost[v] $\leftarrow \infty$ for all v other than s;

  3 cost[v] $\leftarrow w(s, v)$, pre[v] $= s$ for all $(s, v) \in E$;

  4 Find $v \notin S$ with smallest cost[v];

  5 $S \leftarrow S + \{v\}, T \leftarrow T + \{(\text{pre}[v], v)\}$

  6 cost[u] = min(cost[u], $w(v, u)$) for all $(v, u) \in E$

  7 If cost[u] is updated, then pre[u] $= v$;
- Running Time: |E| rounds Update, |V| rounds PopMin. O(|E| + |V|log|V|) for Prime's Algorithm. With Fibonacci Heap, we can do better.
- **Algorithm 2:** Kruskal's Algorithm

  Input: A graph G = (V, E) with edge weights.

  Output: A minimum spanning tree T.

  1 $T \leftarrow \{\}, S \leftarrow \{\}$;

  2 Sort edges in E by non-decreasing order of weights;

  3 For each edge $(u, v) \in E$:

  4 If $u \notin S$ or $v \notin S$, then $T \leftarrow T + \{(u, v)\}, S \leftarrow S + \{u, v\}$;
- Running Time: O(E log E) = O(E log V)for Kruskal's Algorithm.

Union Find Set (With Path Compression)

- Running Time Analysis (Amoritized Analysis):
  - We prove: any m find operations totally cost $O(m \log^\star n + n \log^\star n)$
  - Then we can say the amoritized cost is $O(\log^\star n)$ for each find operation.
  - Lemma1: Parent's rank is strictly larger than the child.
  - Lemma2: The tree of v has at least $2^{\text{rank}[v]}$ vertices.
  - Group vertices by rank: Group i: $k_i = 2^{k_{i-1}}$(1| 2| 3-4| 5-16| 17-65536| ...)
  - Across Group Charing (AGC)
  - Same Group Charging (SGC)
  - Lemma3: We have at most $\log^\star n$ groups.
  - Lemma4: Group [k+1, 2^k] at most have $\frac{n}{2^k}$ vertices.
  - each find incurs at most $\log^\star n$ AGC.
  - for a vertex v, at most $2^k - (k + 1) < 2^k$ SGC for v.
  - In a group $[k + 1, ..., 2^k]$, at most $\frac{n}{2^k} * 2^k = n$ SGC.
  - So, the total cost is $O(m \log^\star n + n \log^\star n)$ for m find operations.

Halfman code

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')=$w(u)c \cdot \text{len}(u) + w(v)c \cdot \text{len}(v) -$
    ```

$w(v)\text{len}(u) - w(u)\text{len}(v) = (w(u) - w(v))(\text{len}(u) - \text{len}(v)) < 0$.

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 $w(C) > w(A)$, $w(B)l = w(C)$. If $w(B) = w(C)$, changing B and C preserves avg_len. If $w(B) < w(C)$, $\text{len}(B) \geq \text{len}(C)$ by above proof. If $\text{len}(B) == \text{len}(C)$, swapping B and C also preserves avg_len. If $\text{len}(B) > \text{len}(C)$, D is not processed, $w(D) > \max(w(A), w(B))$, but $\text{len}(D) = \text{len}(B) > \text{len}(C)$, 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.

textbf{time complexity:} 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 $O(n \log n)$.

Makespan Minimization

- Greedy: Schedule jobs to the earliest finished machine.
- Greedy is at most 2 times of OPT in any input!
- LPT Algorithm:
  - Longest Processing Time First
  - Insert jobs into the earliest finished machine.
- Greedy is at most 4/3 times of OPT in any input!
  - When $\text{OPT} \geq 3p_n$, $\text{ALG} = t + p_n < \frac{4}{3}$ OPT
  - When $\text{OPT} < 3p_n$
    - Lemma 1: Every machine in OPT only has two jobs.
    - Lemma 2: p_1~p_m are scheduled on different machines in OPT.
    - Lemma 3: p_m~p_n is correctly scheduled based on Lemma 1.
    - Lemma 4: ALG = OPT for p_1~p_n if $\text{OPT} < 3p_n$.

DP

Guideline

1. find subproblem
2. check whether we are in a DAG and find the topological order of the DAG
3. solve & store subproblems by the topological order.

Shortest path in DAG

$$\text{dist(t)} = \min \text{dist}(v) + \omega(v, t) \text{ for (v, t) in E}$$

O(V+E)

Longest Increasing Subsequence

Algorithm 3: LIS

```
function LIS(n)
    lis[0] = 0
for i = 1 to n
    lis[i] = max_{a_j<a_i, j<i} lis[j]+1
return max_{1≤i≤n} lis[i]
```

Using priority queue

Sm [*len*] : the smallest ended number for an increasing subsequence with length *len*.

1. $a_i >$ sm[len]: It can create a longer LIS, but can't update sm[len]
2. $a_i <$ sm[len]: It must update sm[len], but can't create a longer LIS.

结合二分搜索，复杂度为 O(nlgn)

Edit Distance

Allow insertion Deletion and replacement *ED* i, j : The edit distance between *X*[1. . *i*] and *Y*[1. . *j*].

$$ED[i, j] = \min \begin{cases} ED[i-1, j-1] + 1 \text{ if } x[i] \neq y[j] \\ ED[i-1, j] + 1 \\ ED[i, j-1] + 1 \end{cases}$$

Knapsack Problems

Input: *n* items with cost *ci* and value *vi*, and a capacity *W*.

Output: Select a subset of items, with total cost at most W. The goal is to maximize the total value.

$$f[i, w] = \max\{f[i-1, w], f[i-1, w-c_i] + v_i\}$$

$$O(nW)$$

Surplus Supply

Input: *n* items with cost c_i and value v_i, and a capacity *W*.

Output: Select some items (**each items can be selected more than once**), with total cost at most W. The goal is to maximize the total value

Simple version

f [w] : the maximum value we can with w budget.

Algorithm 4: Surplus Supply

```
f[0]= ... = f[n] = 0
for w = 0 to W
    for i = 1 to n
        f[w] = max{f[w], f[w-c_i]+v_i}
return f[n, W]
```

Transfer

1. Not but it anymore: can at most use w budget before i.
2. Buy: use at most $w - c_i$ budget before i and i

$$f[i, w] = \max\{f[i-1, w]\}, f[i, w - c_i] + v_i\}$$

Largest Number in k Consecutive Numbers

Subproblem Definition

Input: *large* [*i*] : the largest number from $a_i - k + 1$ to a_i.

Output: *large* [*k*] to *large*[*n*].

Potential Largest List

- PLL[i]: the Potential Largest List for a_{i-k+1}~a_i;
- At most k numbers
- Sorted by the index
- $a_i \geq a_j$ if $i < j$
- i-k+1 ≤ Index ≤ i

Algorithm 5: Updating Priority Queue

```
function updating(a[1..n],i,k,PLL)
    if PLL.front.index≤ i-k
        PopFront()
    while PLL.back.value ≤ a[i]
        PopBack()
    PushBack(a[i])
function largest(a[1..n], k)
    PLL = NULL
    for i = 1 to n
        updating(a,i,k,PLL)
        output PLL.front
```

Time

The cost of *n* times updating has been charged to numbers!

Each number

- Charged once when it is popped out.
- Charged once when it is pushed in.

Totally: *O*(*n*).

Minimizing Manufacturing Cost

Input: A sequence of items with cost $a_1, a_2, ..., a_n$.

- Operation man(*l*, *r*): manufacture the items from *l* to *r*.

$$cost(l, r) = C + \left(\sum_{i=l}^{r} a_i\right)^2$$

Output: The minimum cost to manufacture all items

y is better than x for i means the gradient of x to y g(x,y) smaller than s(i). s(i) = $\sum_{k=1}^{i} a_k$

$$g(x, y) = \frac{f[y] + s^2(y) - f[x] + s^2(x)}{2(s(y) - s(x))}$$

Algorithm 6: Convex hall

```
let j_1..j_m be the convex hall
loop from k = m (charge to i)
while g(j_{k-1}, j_k) ≥ g(j_k, i)(charge to j_k)
    kick j_k
    k–
Until g(j_{k-1}, j_k) ≤ g(j_k, i)(charge to i)
j_{k+1} = i
```

Algorithm 7: updating f[i]

```
let j_1..j_m be the convex hall
loop from k =1
while g(j_k, j_{k+1}) ≤ s(i)(charge to j_k)
    kick j_k
    k++
Until g(j_k, j_{k+1}) ≥ s(i)(charge to i)
j_k is best
```

$f[i] = f[j] + C + \left(\sum_{k=j+1}^{i} a_k\right)^2 = f[j] + C + (s(i) - s(j))^2$

Product of Sets

Input: n sets L_n Output: The minimum number of operations to calculate $L_1 \times ... \times L_n$ $L_1 \times L_2 = \{(a, b) | a \in L_1, b \in L_2\}$

Transfer

c(i, j) = $m_i m_{i+1} ... m_j + \min_{i \leq k < j}\{c(i, k) + c(k+1, j)\}$

Time Complexity

O(n^3)

Quadrangle Inequalty

$$\forall i \leq i' \leq j \leq j', w(i, j) + w(i', j') \leq w(i', j) + w(i, j')$$

monotonicity

$$\forall i \leq i' \leq j \leq j', w(i', j) \leq w(i, j')$$

Bellman-Ford

dist [*k*, *v*] : the shortest distance from *s* to *v* among all ***k***-edgepath (path with at most ***k*** edges)

Algorithm 8: updating f[i]

```
function bellman(G,s)
dist[0,s] = 0, dist[0,x] = ∞ for other x in V
for k = 1 to |V|
    for each (u,v) in E dist[k, v] =
      min{dist[k-1,v],dist[k-1, u]+d(u,v)}
```

All Pair Shortest Path

- Input: A directed graph *G*(*V*,*E*), and a weighted function *d*(*u*, *v*) for all *u*, *v* ∈ *E*.
- Output: Distance *dist*(*u*, *v*), for all vertex pair u, *v*.
- V times Bellman-Ford $O(V^2E)$

Floyd-Warshall: *dist k*, *u*, *v* : the shortest distance from u to v that only across inter-vertices in $\{v_1, ..., v_k\}$

We can label vertices from 1 to |*V*|. *dist*[|*V*|, *u*, *v*] is exactly what we want!

Algorithm 9: Floyd-Warshall

```
function floyd_warshall(G)
dist [0, u, v] = d(u, v) for all u, v ∈ E, dist [0, u, v] =
  ∞ otherwise.
for k = 1 to |V|
    for u = 1 to |V|
        for v = 1 to |V|
            dist [k, u, v] = min{dist [k – 1, u, v] , dist [k
              – 1, u, k] + dist[k – 1, k, v]}
```

O(V^3) running time but *O*(V^2) space

Traveling Salesman Problem (TSP)

1. Input: A complete weighted undirected graph *G*, such that *d* (*u*, *v*) > 0 for each pair *u*, *v* (*u* ≠ *v*).
2. Output: the cycle of *n* vertices with the minimum weight.

f [*S*, *u*, *v*]

- The shortest path from *u* to *v* with inter-vertex exactly ***S*** ⊂ ***V*** except *u* and *v*.
- min *u* *f*[*V*, *u*, *u*] is what we want!

$$f[S, u, v] = \min_{k \in S}[S - \{k\}, u, k] + d(k, v)$$

Time Complexity

n = |V|

- O($n2^n$) subproblems
- O (n) solving

Brute-force: ***O***(***n***!)

Maximize Independent Set on Trees

f[v] = $\max\{\sum_{u \in \text{ children(v)}} f[u], \sum_{u \in \text{ grandchildren(v)}} f[u] + 1\}$

Each of its children and its grandchildren cost one.

O(n)

Greedy:
1. Choose all Leaves
2. Remove all leaves' parents
3. repeat 1

一些题目

注意证明拓扑性质

the number of vertex covers in G

$$f_u = \prod_{v \in son(u)} (f_v + g_v)$$

$$g_u = \prod_{v \in son(u)} f_v$$

the number of minimum vertex covers in G

(b) We calculate the following DP: f_v^*: the number of maximum independent set in the subtree of i when i is chosen, g_v^* for not chosen. fs_v^*: the size of maximum independent set in the subtree of i when i is chosen, gs_v^* for not chosen. The DP equation is

$$f_u^* = \prod_{v \in son(u)} min(f_v^*, g_v^*, fs_v^*, gs_v^*)$$
$$g_u^* = \prod_{v \in son(u)} f_v^*$$

$$fs_u^* = \sum_{v \in son(u)} \min\{fs_v^*, gs_v^*\} + 1$$
$$gs_u^* = \sum_{v \in son(u)} fs_v^*$$

where $min(f_v^*, g_v^*, fs_v^*, gs_v^*)$ is a function that

$$min(f_v^*, g_v^*, fs_v^*, gs_v^*) = \begin{cases} f_v^*, & fs_v^* < gs_v^* \\ g_v^*, & fs_v^* > gs_v^* \\ f_v^* + g_v^*, & fs_v^* = gs_v^* \end{cases}$$

Network Flow

二分图是可以被二着色的图 如果流有下限，可以构造一个新的图，s'到 u 为到 u 的下限的和，出边同理，边的 capacity 为上界减下界来确定是否有 feasible，如果有 feasible，可以 push 一次 feasible 后在 Gf 上寻找最大流

Algorithm

$G_f = (V_f, E_f)$ is defined as : $V_f = V$ $(u, v) \in E^f$ if one of the following holds

- $(u, v) \in E$ and $f(u, v) < c(u, v)$: in this case, $c^f(u, v) = c(u, v) - f(u, v)$
- $(v, u) \in E$ and $f(v, u) > 0$: in this case, $c^f(u, v) = f(v, u)$

Algorithm 10: Network-Flow

Initialize an empty flow f and the corresponding residual flow G_f

Iteratively

find a path on Gf,

push maximum amount of flow on Gf, and

update f and Gf,

until there is no s-t path on Gf.

Algorithm 11: FordFulkerson

```
FordFulkerson(G = (V, E), s, t, c):
1  initialize f such that ∀e ∈ E: f(e) = 0; initialize G' ← G;
2  while there is an s-t path p on G':
3      find an edge e ∈ p with minimum capacity b;
4      for each e = (u, v) ∈ p:
5          if (u, v) ∈ E: update f(e) ← f(e) + b;
6          if (v, u) ∈ E: update f(e) ← f(e) − b;
7      endfor
8      update G';
9  endwhile
10 return f
```

Time Complexity

All the capacities are integers

Lemma 1. Each while-loop iteration increase the value of f by at least 1. **Lemma 2**. If each $c(e)$ is an integer, then the max flow f_{max} is an integer.

$O(Ef_{\max})$

Theorem. If each c(e) is an integer, then the value of the maximum flow f is an integer.

Correctness: Max-Flow-Min-Cut Theorem.

The value of the maximum flow is exactly the value of the minimum cut:

$$\max_f v(f) = \min_{L,R} c(L, R)$$

$$c(L, R) = \sum_{(u,v) \in E, u \in L, v \in R} c(u, v)$$

proof

Claim $v(f) = f_{ou}(L) - f_{in}(L)$

For a cut L and R of the original graph

$f_{in}(L) = \sigma\ u,v \in E, u \in L, v \in R\ f(v, u)$

$v(f) = f_{ou}(L) - f_{in}(L)$

$f_{ou}(u) = f_{in}(u)$ for $u \in V\{s, t\}$ $f_{ou}(s) = v(f), f_{in}(s) = 0$

$v_f = \sum_{u \in L}(f_{ou}(u) - f_{in}(u)) = f_{ou}(L) - f_{in}(L)$

Lemma 1. For any flow f and any cut $\{L, R\}$ in G, we have $v(f) \le c(L, R)$.

L: reachable from s in Gf, $f_{ou}(L) = c(L, R), f_{in}(L) = 0$

Lemma 2. There exists a cut $\{L, R\}$ such that the flow f output by Ford-Fulkerson Algorithm satisfies $v(f) = c(L, R)$.

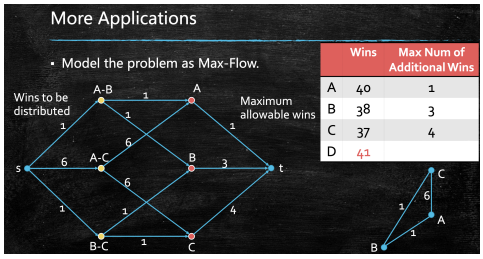
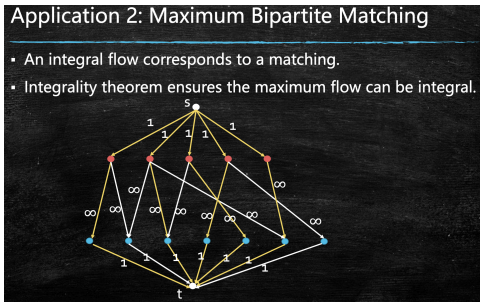
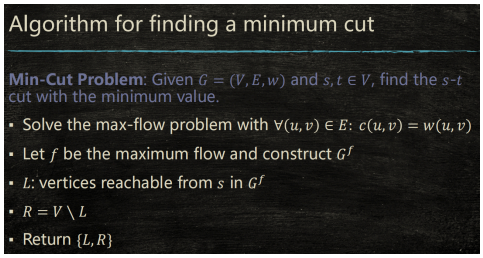
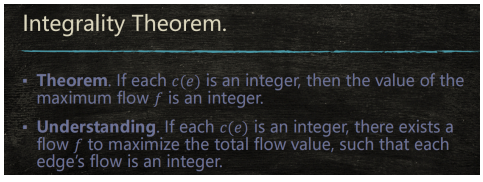


Figure 2: 比赛问题







Edmonds-Karp Algorithm

Implement FordFulkerson with BFS

Algorithm 12: Edmonds-Karp Algorithm

```
EdmondsKarp(G = (V, E), s, t, c):
initialize f such that ∀e ∈ E: f(e) = 0; initialize G_f ← G;
while there is an s-t path on G_f:
    find such a path p by BFS;
    find an edge e ∈ p with minimum capacity b;
    update f that pushes b units of flow along p;
    update G_f;
endwhile
return f
```

Time

节点的 distance 根据 BFS 有不减，只需证明严格增

The distance of u from s increases by 2 between two critical. $\text{dist}^{i+j}(u) = \text{dist}^{i+j}(v) + 1 \ge \text{dist}^{i(v)} + 1 \ge \text{dist}^{i(u)} + 2$

The max distance is V, so each edge can only be critical for V times, at least 1 edge become critical in one iteration, so the number of iteration is O(|V||E|), each iteration takes O(E), so the complexity is $O(VE^2)$

Dinic's Algorithm

Build a level graph:
- Vertices at Level i are at distance i.
- Only edges go from a level to the next level are kept.
- Can be done in O(E) time using a similar idea to BFS.

Blocking flow: push flow on multiple s-t paths, each s-t path must contain a critical edge.

Initialize f to be the empty flow and $G^f = G$.

Iteratively do the followings until dist $t = \infty$:
- Construct the level graph G_L^f for G^f
- find a blocking flow on G_L^f
- update f and G^f

Finding a blocking flow:

Iteratively do the followings, until no path from s to t:
- Run a frontward DFS.
- Two possibilities:
  - End up at t: in this case, we update f (by pushing flow along the path) and remove the critical edge
  - End up at a dead-end, a vertex v with no out-going edges in G_L^f: in this case, we remove all the incoming edges of v

Time

At least one edge is removed after each search. Total number of searches: O(E) while Each search takes at most $|V|$ steps.

Time complexity for each iteration $O(V \cdot E)$.

After one iteration, a new edge (u, v) appearing in $G^{f_{i+1}}$ (but not in G^{f_i}) must be "backward": $\text{dist}^i(u) = \text{dist}^i(v) + 1$.

We again have that dist(u) is non-decreasing

Iteration Times

Proving $\text{dist}^{\{i+1\}}(t) > \text{dist}^i(t)$:

- Consider an arbitrary $s - t$ path p in $G_L^{\{f_{\{i+1\}}\}}$ with length $\text{dist}^{\{i+1\}}(t)$.
- We have $\text{dist}^{i+1}(t)g = \text{dist}^i(t)$ by monotonicity.
- Suppose for the sake of contraction that $\text{dist}^{\{i+1\}}(t) = \text{dist}^i(t)$.
- Case 1: all edges in p also appear in G_L^i.
  - Then p is a shortest path containing no critical edges in G_L^i.
  - Contracting to the definition of blocking flow!

Proving $\text{dist}^{\{i+1\}}(t) > \text{dist}^{i(t)}$:

- Case 2: p contains an edge (u, v) that is not in G_L^i.
  - If (u, v) was not in G_L^i, then (v, u) was critical in the last iteration. We have $\text{dist}^i(u) = \text{dist}^i(v) + 1$.
  - If (u, v) was in G^i but not G_L^i, by the definition of level graph, we have $\text{dist}^i(u) \ge \text{dist}^i(v)$.
  - In both cases above, $\text{dist}^i(u) \ge \text{dist}^i(v)$.
  - We have $\text{dist}^{i+1}(u) \ge \text{dist}^i(u)$ by monotonicity,
  - and we have $\text{dist}^{i+1}(v, t) \ge \text{dist}^i(v, t)$

So we have $\text{dist}^{i+1}(t) \ge \text{dist}^{i(t)} + 1$. Besides, dist(t) maximum is |V| or ∞ so the total number of the iteration is at most V

So the time complexity is O(V^2E)

Application

A graph is regular if all the vertices have the same degree.

A matching is perfect if all the vertices are matched.

A regular bipartite graph always has a perfect matching.

Hall's Marriage Theorem

The matching problem on a bipartite graph $G = (A, B, E)$.

For a subset $S \subseteq A$, let $N(S) \subseteq B$ be the set of vertices that are incident to vertices in S.

Hall's Marriage Theorem There exists a matching of size $|A|$ if and only if $S \le |N(S)|$ for every $S \subseteq A$.

proof Given $\forall S$: $S \le |N(S)|$, suppose the maximum matching has size $M < |A|$. 分情况讨论切割在哪 If $L_A, L_B, R_A, R_b \neq \emptyset$

$M = |L_B| + |R_A|, |L_A| + |R_A| = |A|, M < |A|$

so $L_A > L_B$

If there exist an edge from $L_A to L_B$, the cut is ∞

Thus, $N(L_A) \le L_B < L_A$ contradiction

By setting the capacity to 1, the Dinic's algorithm runs in O(E $\sqrt{V}$)

Because the search takes O(1), and search at most O(E) times. v

1. If the iteration ends in $\sqrt{V}$, we have finished
2. Else: let f be the flow after $\sqrt{V}$ iterations

In each iteration, for each $v \in V \setminus \{s, t\}$, either its in-degree is 1, or its out-degree is 1.

Integrality Theorem: there exists a maximum integral flow f' in G_f.

f' consists of vertex-disjoint paths with flow 1 (from flow conservation)

So each path length $\ge \sqrt{V}, \frac{V}{\sqrt{V}} = \sqrt{V}, v(f') \le \sqrt{V}$

Linear Programming

基础的解决方式就是通过不断换元，把所有东西取到系数小于 0 范围大于 0.

Vertex is A point at the intersection of n linearly independent hyperplanes.

Edge is the intersection of $n - 1$ linearly independent hyperplanes.

Move from one vertex to another adjacent vertex along an edge:

- Relax one of the n constraint and impose another.
- The new vertex can be computed by solving a system of n linear equations.

For Variable with unrestricted signs,Introduce two variables $x+$ and $x-$ with standard constraints $x+, x- \ge 0$ and Replace x with $x+ - x-$

A linear program can be solved in a polynomial time.

Linear Program (LP)

- Standard Form LP:

$$\max (\vec{c})^T \vec{x}$$
$$\text{subject to } A\vec{x} \le \vec{b}$$
$$\vec{x} \ge 0$$

LP Relaxation

Integer Programming is NP-complete, even for the zero-one special case。 We can use the fact that LP is polynomial-time solvable to design approximation algorithm. Then "round" the fractional solution to integral one.

LP duality

Strong Duality Theorem

- Theorem [Strong Duality Theorem]. Let x^* be the optimal solution to (a) and y^* be the optimal solution to (b), then $c^T x^* = b^T y^*$.

maximize $c^T x$		minimize $b^T y$	
subject to $Ax \le b$	(a)	subject to $A^T y \ge c$	(b)
$x \ge 0$		$y \ge 0$	

Primal feasible — Primal OPT = Dual OPT — Dual feasible

Formulation as Linear Program

- The maximum flow problem can be formulated by a linear program.

$$\text{maximize} \quad \sum_{u:(s,u)\in E} f_{su}$$
$$\text{subject to} \quad 0 \le f_{uv} \le c_{uv} \qquad \forall (u,v) \in E$$
$$\sum_{v:(v,u)\in E} f_{vu} = \sum_{w:(v,w)\in E} f_{uw} \qquad \forall u \in V \setminus \{s,t\}$$

- Ford-Fulkerson Method implements the simplex method.

We also make it easier

$$\text{maximize} \quad \sum_{u:(s,u)\in E} f_{su}$$
$$\text{subject to} \quad f_{uv} \le c_{uv} \qquad \forall (u,v) \in E \qquad \to y_{uv}$$
$$\sum_{v:(v,u)\in E} f_{vu} - \sum_{w:(u,w)\in E} f_{uw} = 0 \qquad \forall u \in V \setminus \{s,t\} \qquad \to z_u$$
$$f_{uv} \ge 0 \qquad \forall (u,v) \in E$$

Compute Its Dual Program

$$\text{minimize} \quad \sum_{(u,v)\in E} c_{uv} y_{uv}$$
$$\text{subject to} \quad y_{su} + z_u \ge 1 \qquad \forall u: (s,u) \in E$$
$$y_{vt} - z_v \ge 0 \qquad \forall v: (v,t) \in E$$
$$y_{uv} - z_u + z_v \ge 0 \qquad \forall (u,v) \in E, u \neq s, v \neq t$$
$$y_{uv} \ge 0 \qquad \forall (u,v) \in E$$

- We aim to show the LP above describes the min-cut problem.
- Let OPT_{dual} be its optimal objective value. We need to show OPT_{dual} is the size of the min-cut.

Some Intuitions

$$\text{minimize} \quad \sum_{(u,v)\in E} c_{uv} y_{uv}$$
$$\text{subject to} \quad y_{su} + z_u \ge 1 \qquad \forall u: (s,u) \in E$$
$$y_{vt} - z_v \ge 0 \qquad \forall v: (v,t) \in E$$
$$y_{uv} - z_u + z_v \ge 0 \qquad \forall (u,v) \in E, u \neq s, v \neq t$$
$$y_{uv} \ge 0 \qquad \forall (u,v) \in E$$

- y_{uv} describes if edge (u, v) is cut: $y_{uv} = \begin{cases} 1 & \text{if } (u,v) \text{ is cut} \\ 0 & \text{otherwise} \end{cases}$
- z_u describes u's "side": $z_u = \begin{cases} 1 & \text{if } u \text{ is on the } s-\text{side} \\ 0 & \text{if } u \text{ is on the } t-\text{side} \end{cases}$

Totally Unimodular Matrix

- **Definition.** A matrix A is totally unimodular if every square submatrix has determinant 0, 1 or −1.
- **Theorem.** If $A \in \mathbb{R}^{m\times n}$ is totally unimodular and $\mathbf{b}$ is an integer vector, then the polytope $P = \{\mathbf{x}: A\mathbf{x} \le \mathbf{b}\}$ has integer vertices.
- A Proof Sketch.
  - If $\mathbf{v} \in \mathbb{R}^n$ is a vertex of P. Then there exists an invertible square submatrix A' of A such that $A'\mathbf{v} = \mathbf{b}'$ for some sub-vector $\mathbf{b}'$ of $\mathbf{b}$.
  - By Cramer's Rule, we have $v_i = \frac{\det(A'_i|\mathbf{b}')}{\det(A'_i)}$, where $(A'_i|\mathbf{b}')$ is the matrix with i-th column replaced by $\mathbf{b}'$.
  - $\det(A'_i) = \pm 1$ and $\det(A'_i|\mathbf{b}') \in \mathbb{Z}$. Thus, $\mathbf{v}$ is integral.

Corollary on Integrality of LP

- **Theorem.** If $A \in \mathbb{R}^{m\times n}$ is totally unimodular and $\mathbf{b}$ is an integer vector, then the polytope $P = \{\mathbf{x}: A\mathbf{x} \le \mathbf{b}\}$ has integer vertices.
- Since there always exists optimum at a vertex of the feasible region of LP, we have the following corollary.
- **Corollary.** If A is unimodular, then the optimal solution to LP (a) is integral when $\mathbf{b}$ is integral, and the optimal solution to LP (b) is integral when $\mathbf{c}$ is integral.

maximize $\mathbf{c}^T\mathbf{x}$		minimize $\mathbf{b}^T\mathbf{y}$	
subject to $A\mathbf{x} \le \mathbf{b}$	(a)	subject to $A^T\mathbf{y} \ge \mathbf{c}$	(b)
$\mathbf{x} \ge \mathbf{0}$		$\mathbf{y} \ge \mathbf{0}$	

Proving Z is totally unimodular by Induction...

- Base Step: Each cell of Z belongs to $\{0, 1, -1\}$.
- Inductive Step: Suppose every $k \times k$ submatrix of Z has determinant belongs to $\{0, 1, -1\}$. Consider any $(k+1) \times (k+1)$ submatrix Z'.
- Case 1: If a row of Z' is all-zero, then $\det(Z') = 0$.
- Case 2: If a row of Z' contains only one non-zero entry, then $\det(Z')$ equals to ± 1 times the determinant of a $k \times k$ submatrix. $\det(Z') \in \{0, 1, -1\}$ by induction hypothesis.
- Case 3: If every row of Z' has two non-zero entries (one of them is -1 and the other is 1), then $\det(Z') = 0$:
  - Adding all the column vectors, we get a zero vector.

A Framework for Proving Theorems Using Strong Duality □ Write down the primal and dual LPs. □ Justify that the primal and dual LPs describe the corresponding problems. □ If the problem described is discrete, prove that the corresponding LP always gives integral solution. – Total Unimodularity □ Apply strong duality theorem.

Minimax Theorem

- Suppose A chooses strategy first. Knowing that B will play the best response, A will choose an optimal strategy $\mathbf{x}$ that maximizes his/her utility:

B plays the best response given A's strategy $\mathbf{x}$.

$$\max_{\mathbf{x}} \overbrace{\min_{\mathbf{y}} \sum_{i,j} G_{i,j} x_i y_j}$$

Given B plays the best response, A choose a strategy maximizing the utility.

- Suppose B chooses strategy first. Similarly, the utility for A is $\min_{\mathbf{y}} \max_{\mathbf{x}} \sum_{i,j} G_{i,j} x_i y_j$

Minimax Theorem

- Minimax Theorem:

$$\max_{\mathbf{x}} \min_{\mathbf{y}} \sum_{i,j} G_{i,j} x_i y_j = \min_{\mathbf{y}} \max_{\mathbf{x}} \sum_{i,j} G_{i,j} x_i y_j$$

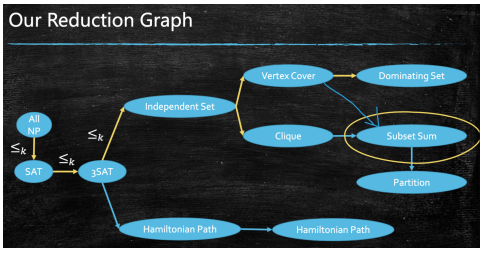
- Who chooses strategy first doesn't matter!

P, NP, NP-Completeness

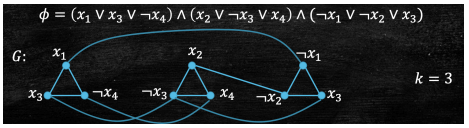
- **P** A decision problem f: $\Sigma_* \longrightarrow \{0, 1\}$ is in P, if there exists a polynomial time TM $\mathscr{A}$ such that – $\mathscr{A}$ accepts x if $f(x) = 1$ – $\mathscr{A}$ rejects x if $f(x) = 0$

  contains path,k-flow,prime
- Some famous NP-hard problems
  - SAT
  - Vertex Cover
  - Independent Set
  - Subset Sum
  - Hamiltonian Path
  - partition: divide a set to two sum-equal set
- SAT $\longleftrightarrow$ 3-SAT

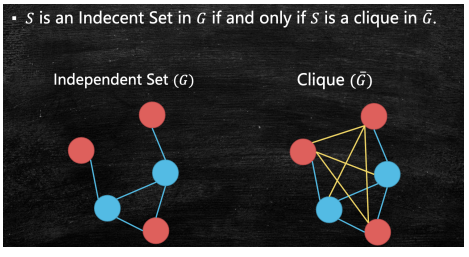
Our Reduction Graph



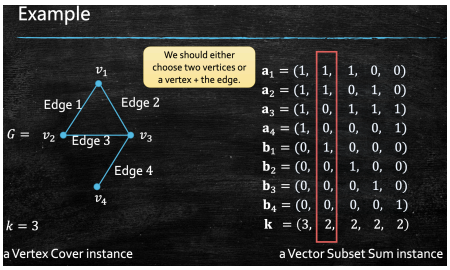
- SAT $\longrightarrow$ 3SAT
  - $(l_1 \vee l_2 \vee \cdots \vee l_k) = (l_1 \vee l_2 \vee y_1) \wedge (\neg y_1 \vee l_3 \vee y_2) \wedge \cdots (\neg y_{k-2} \vee l_{k-1} \vee l_k)$
- 3SAT $\longrightarrow$ Independent Set



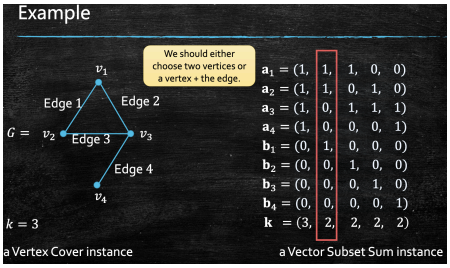
- Independent Set $\longrightarrow$ Vertex Cover
  - Choose the vertexes which are not chosen in independent set.
- Independent Set $\longrightarrow$ Clique



- Vertex Cover $\longrightarrow$ Dominating Set
  - a dominating set is a subset of vertices S such that, for any $v \in V \setminus S$, there is a vertex $u \in S$ that is adjacent to v. (G(V,E),k)
  - let all vertexes connected to each other and add new vertexes to each edge.
- Vertex Cover $\longrightarrow$ Subset Sum
  - Vertex Cover $\le_k$ VectorSubsetSum*

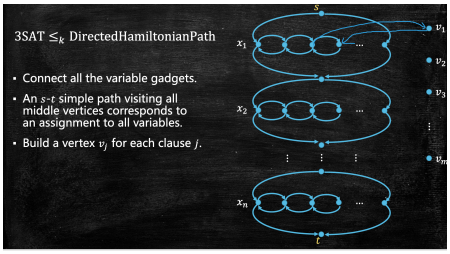


- $\text{VectorSubsetSum}^* \leq_k \text{SubsetSum}$

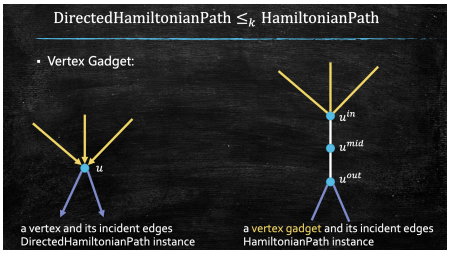


把向量看成 10 进制数

- 3SAT $\longrightarrow$ Hamiltonian Path
  - $3\text{SAT} \leq_k$ 有向图中的哈密顿路



  - 有向图哈密顿路 $\leq_k$ 哈密顿路



- 子图同构解决团，G'=G，H' is a complete graph with k vertices
- 将顶点覆盖问题归约到集合覆盖问题。具体步骤如下：
  - 给定一个顶点覆盖问题的实例 (G=(V,E),k)。构造一个新的集合覆盖问题实例
  - (U,S,k')，其中：U 是包含|E|个元素的集合，每个元素对应于G中的一条边。S 是包含 |V| 个子集的集合，每个子集对应于 G 中的一个顶点 v，子集Sv包含所有与顶点 v 相连的边。k' = k
- 无嫉妒分配问题将价值设为相等的可以解决 partition
- 背包问题的判断形式，可以将价值和大小设为 k 来解决 subsetnum 问题

三着色问题：使用 T（true）F（false）N（neutral）

构建 tfn 的三角形和 tf，fn，nt 成一个三角形，分配不同的颜色，对于一个命题，创建 x and $\neg x$并相连并连接到 n，确保有一个对有一个错。对于一个括号，创建命题 m 并连到 n，f 确保选 t,