Algorithm and Complexity

Chap0: Preliminary

Set

cardinality, proper subset, strict subset, union, intersection

difference, complement, cartesian product, power set

function

injective(one to one,单射), surjective(满射), bijective

Proof

categories

- Proof by Construction
- Proof by Contrapositive

  $p \rightarrow q \Leftrightarrow \neg q \rightarrow \neg p$

  - Proof by Contradiction
  - Proof by Counterexample
- Proof by Cases

  divide domain into distinct subsets
- Proof by Mathematical Induction

  basic step, induction hypothesis, proof of induction step

  - The Principle of Mathematical Induction

    from P(k) to P(k+1)
  - Minimal Counterexample Principle

    suppose minimal counterexample is k, then P(k-1) holds and then derive a contradiction on P(k)
  - The Strong Principle of Mathematical Induct

    from P(1~k) to P(k+1)

Chap1: Algorithm Analysis

Computational Complexity

Theory of Computation

understand the notion of computation in a formal framework.

- Computability Theory: what can be solved

  models: λ-calculus, Turing machine

  Church-Turing Thesis: a computer program $\Leftrightarrow$ a Turing machine. Computation Models can solve exactly the same class of problems.

- Computational Complexity: how much resource is necessary

  Decision Problem & Search Problem

  Time Complexity & Space Complexity

  DTM(Deterministic Turing Machine) & NTM

- Theory of Algorithm: how to design

  algorithm is a black box

  Algorithmic Thinking & Applicability of Algorithm

Time Complexity

asymptotic(渐近) running time

- O

  upper bound. $f(n) = O(g(n)) \Leftrightarrow \exists c, \exists n_0, \forall n \geq n_0, f(n) \leq cg(n)$

- Ω

  lower bound. $f(n) = \Omega(g(n)) \Leftrightarrow \exists c, \exists n_0, \forall n \geq n_0, f(n) \geq cg(n)$

  $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

- Θ

  exact picture. $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$

- o

  $f(n) = o(g(n)) \Leftrightarrow \forall c, \exists n_0, \forall n \geq n_0, f(n) < cg(n)$

- ω

  $f(n) = \omega(g(n)) \Leftrightarrow \forall c, \exists n_0, \forall n \geq n_0, f(n) > cg(n)$

Suppose $\lim_{n\to\infty} f(n)/g(n)$ **exists**.

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$ implies $f(n) = O(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$ implies $f(n) = \Omega(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ implies $f(n) = \Theta(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ implies $f(n) = o(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ implies $f(n) = \omega(g(n))$.

- $f(n) = O(g(n))$ is similar to $f(n) \leq g(n)$.
- $f(n) = o(g(n))$ is similar to $f(n) < g(n)$.
- $f(n) = \Theta(g(n))$ is similar to $f(n) = g(n)$.
- $f(n) = \Omega(g(n))$ is similar to $f(n) \geq g(n)$.
- $f(n) = \omega(g(n))$ is similar to $f(n) > g(n)$.

complexity classes

A complexity class is an equivalence class of $\mathcal{R}$. $f\mathcal{R}g$ if and only if $f(n) = \Theta(g(n))$

$f \prec g$ iff $f(n) = o(g(n))$

$$1 \prec \log\log n \prec \log n \prec \sqrt{n} \prec n^{\frac{3}{4}} \prec n \prec n\log n \prec n^2 \prec 2^n \prec n! \prec 2^{n^2}$$

note: limit definition is of great help

Space Complexity

memory needed to carry out an algorithm, excluding the space allocated to hold the input

work space of an algorithm can not exceed the running time of the algorithm, $S(n) = O(T(n))$

Complexity Analysis

optimal algorithm: if we can prove that any algorithm to solve problem Π must be $\Omega(f(n))$ and $O(f(n))$ algorithm is optimal

elementary operation: whose cost is upper bounded by a constant (e.g. arithmetic operations)

basic operation: most frequent elementary operation (e.g. visiting a node in graph traversals)

input size

- sorting and searching problems: # entries in the array or list

- graph algorithms: # vertices or edges, or both
- computational geometry: # points, vertices, edges, line segments, polygons
- matrix operations: the dimensions of the input matrices
- number theory algorithms and cryptography: # bits of the input

analysis

- best case analysis: lower bound
- worst case analysis: upper bound
- average case analysis: take all possible inputs and calculate the expected computing time

note: by default, usually we provide worst case running time for an algorithm without specification

Searching and Sorting

Algorithm	Best Case	Average Case	Worst Case	Space
Linear Search	$\Omega(1)$	$O(n)$	$O(n)$	$O(1)$
Binary Search	$\Omega(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(n)$
Quick Sort	$\Omega(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

Arithmetico-Geometric Progression (A.G.P.):

$$\begin{cases} c_n = (a_1 + (n-1)\cdot d)\cdot q^{n-1}, \\ S_n = (A\cdot n + B)\cdot q^n - B, \end{cases} \qquad A = \frac{d}{q-1}, B = \frac{a_1 - d - A}{q-1}$$

note: assume $n = 2^k$ can be helpful

example: average case for Insertion Sort

A[i] will be moved if there is A[j]<A[i] and j>i.

$P(A[i]\ moves) = \#A[j] < A[i], j > i$

$E(k\ th\ element\ moves) = \frac{k}{2}$

$$E = \sum_{k=1}^{n} \frac{k}{2} = O(n^2)$$

Chap2: Divide and Conquer

example: multiplication

$$x = \boxed{x_L}\boxed{x_R} = 2^{n/2}x_L + x_R$$
$$y = \boxed{y_L}\boxed{y_R} = 2^{n/2}y_L + y_R.$$

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

Optimization: By **Gauss**'s trick, three multiplications, $x_L y_L$, $x_R y_R$, and $(x_L + x_R)(y_L + y_R)$, suffice, as

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

$$T(n) = 3T(n/2) + O(n)$$

example: Strassen matrix multiplication

Master Theorem

If

$$T(n) = aT(\lceil n/b \rceil) + O(n^d)$$

for some constants $a > 0$, $b > 1$, and $d \geq 0$, then

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

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$T(n) = aT(n/b) + f(n)$,

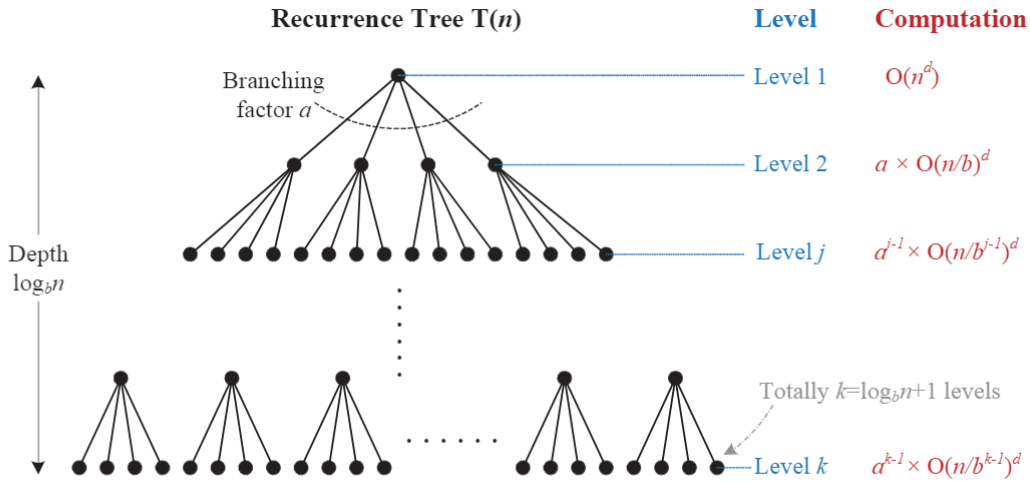
where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

note: $b^d > a$ indicates that merge part is more significant, otherwise recursion parts.

note: $T(n) = 2T(n/2) + O(n \log n)$ falls into the gap between case 2 and case 3.

proof:



Complexity of $T(n)$ = Sum up all computations at each level.

The total work done is

$$\sum_{j=1}^{\log_b n+1}\left(a^{j-1}\times O\left(\frac{n}{b^{j-1}}\right)^d\right)=\sum_{j=0}^{\log_b n}\left(O(n^d)\times\left(\frac{a}{b^d}\right)^j\right)=O(n^d)\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j.$$

It's the sum of a *geometric series (GS)* with ratio a/b^d.

(1) $\frac{a}{b^d}<1\Rightarrow d>\log_b a$:

$$O(n^d)\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j\le O(n^d)\frac{1}{1-\frac{a}{b^d}}=O(n^d).$$

(Sum of GS: $S_n=\sum_{j=1}^{n}a_1q^{j-1}=a_1\frac{1-q^n}{1-q}\le a_1\frac{1}{1-q}$ if $q<1$)

(2) $\frac{a}{b^d}=1\Rightarrow d=\log_b a$:

$$O(n^d)\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j=O(n^d)(\log_b n+1)=O(n^d\log_b n)=O(n^d\log n).$$

($\log_b n=\frac{\log n}{\log b}=\frac{1}{\log b}\log n=O(\log n)$ by changing the base)

(3) $\frac{a}{b^d} > 1 \Rightarrow d < \log_b a$: (reverse the GS in decreasing order)

$$
\begin{aligned}
O(n^d)\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j &= O(n^d)\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^{\log_b n}\cdot\left(\frac{b^d}{a}\right)^j \\
&= O(n^d)\sum_{j=0}^{\log_b n}\frac{a^{\log_b n}}{(b^{log_b n})^d}\cdot\left(\frac{b^d}{a}\right)^j \\
&\le O(n^d)\frac{n^{\log_b a}}{n^d}\cdot\frac{1}{1-\frac{b^d}{a}} \\
&= O(n^{\log_b a})
\end{aligned}
$$

$\left(a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}\right)$

Lower Bound for Sorting

sorting permutation tree

Sorting algorithms can be depicted as trees. Each of its leaves is labeled by a permutation

The maximal depth of the tree is exactly the worst-case time complexity of the algorithm.

This is a binary tree with $n!$ leaves. Thus depth must be at least $\log n!$

Chap3: Sorting Network

Sorting Network

A comparison network contains n input $\langle a_1, a_2, \ldots, a_n\rangle$ and n output $\langle b_1, b_2, \ldots, b_n\rangle$. If output is monotonically increasing for every input, then it is a sorting network.

If a comparator has two input wires of depth d_x and d_y, then its output wire have depth $\max(d_x, d_y) + 1$. Initial, the depth of every input wire is 0.

We are discussing a family of comparison networks according to the input size.

Zero-One Principle

Domain Conversion Lemma

If a comparison network transforms the input sequence $\mathbf{a} = \langle a_1, a_2, \cdots, a_n\rangle$ into the output sequence $\mathbf{b} = \langle b_1, b_2, \cdots, b_n\rangle$.

Then for any monotonically increasing function f, the network transforms the input sequence $f(\mathbf{a}) = \langle f(a_1), f(a_2), \cdots, f(a_n)\rangle$ into the output sequence $f(\mathbf{b}) = \langle f(b_1), f(b_2), \cdots, f(b_n)\rangle$

proof: For a single comparator, it holds.

A stronger statement: if a wire assumes the value a_i when the input sequence is a, then it assumes the value $f(a_i)$ when the input sequence is $f(a)$. (i.e. inside the network, wires hold the property same as output)

We prove the stronger statement by induction on the depth of each wire.

Zero-One Principle

If a sorting network works correctly with inputs drawn from $\{0, 1\}$, then it works correctly on arbitrary input numbers

proof (contradiction): Suppose there exists such a sequence $\langle a_1, a_2, \ldots, a_n \rangle$ and two elements $a_i < a_j$, but the network places a_j before a_i in output. Then we can set monotonically increasing $f(x) = [x > a_i]$. According to Domain Conversion Lemma, $f(a_j)$ should be before $f(a_i)$. Thus, it fails to sort zero-one sequence $\langle f(a_1), f(a_2), \cdots, f(a_n) \rangle$.

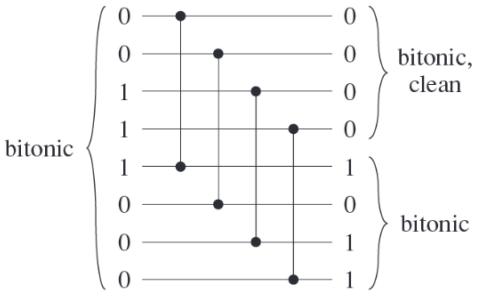
Construction of a Sorting Network

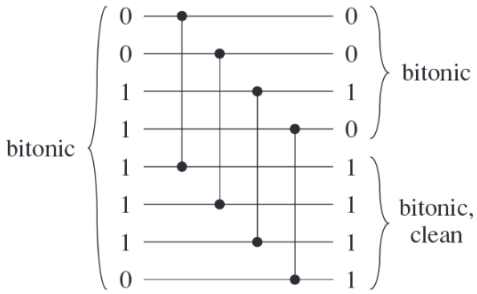
Bitonic Sorter

A bitonic sequence is a sequence that monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing. (e.g. (1,4,6,8,3,2))

Zero-one bitonic sequence have the form $0^i1^j0^k$ or the form $1^i0^j1^k$.

A half-cleaner is a comparison network of depth 1, in which input line i is compared with line $i + \frac{n}{2}$ for $i = 1, 2, \cdots, \frac{n}{2}$ (assume n is even).

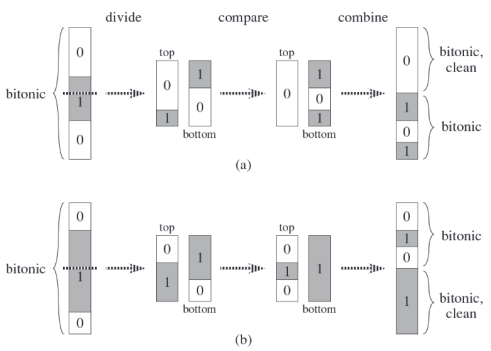


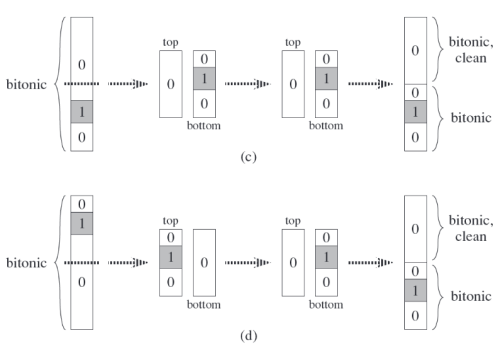


Lemma: If the input to a half-cleaner is a bitonic sequence of 0's and 1's, then the output satisfies the following properties:

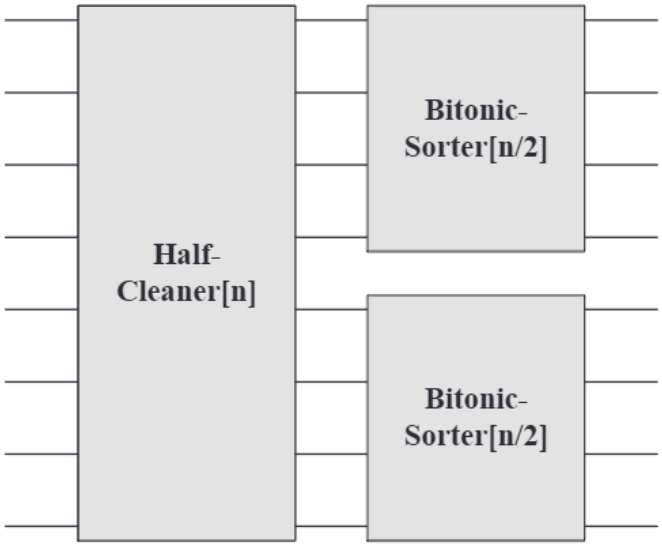
- ▷ both the top half and the bottom half are bitonic;
- ▷ every element in the top half is at least as small as every element of the bottom half, and at least one half is clean.

proof (by cases): suppose input is $0^i1^j0^k$





By recursively combining half-cleaners, we can build a bitonic sorter, which is a network that sorts bitonic sequences.



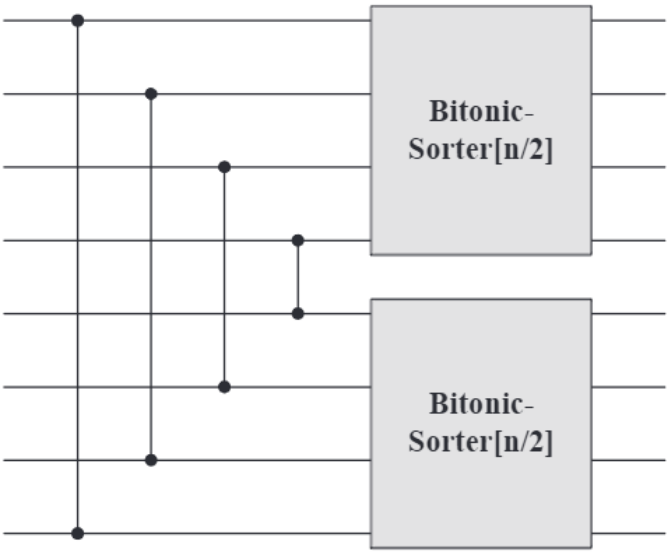
Bitonic-Sorter[n]

Let $D(n)$ be the depth, we have $D(n) = D(n/2) + 1$. Thus, $D(n) = O(\log n)$.

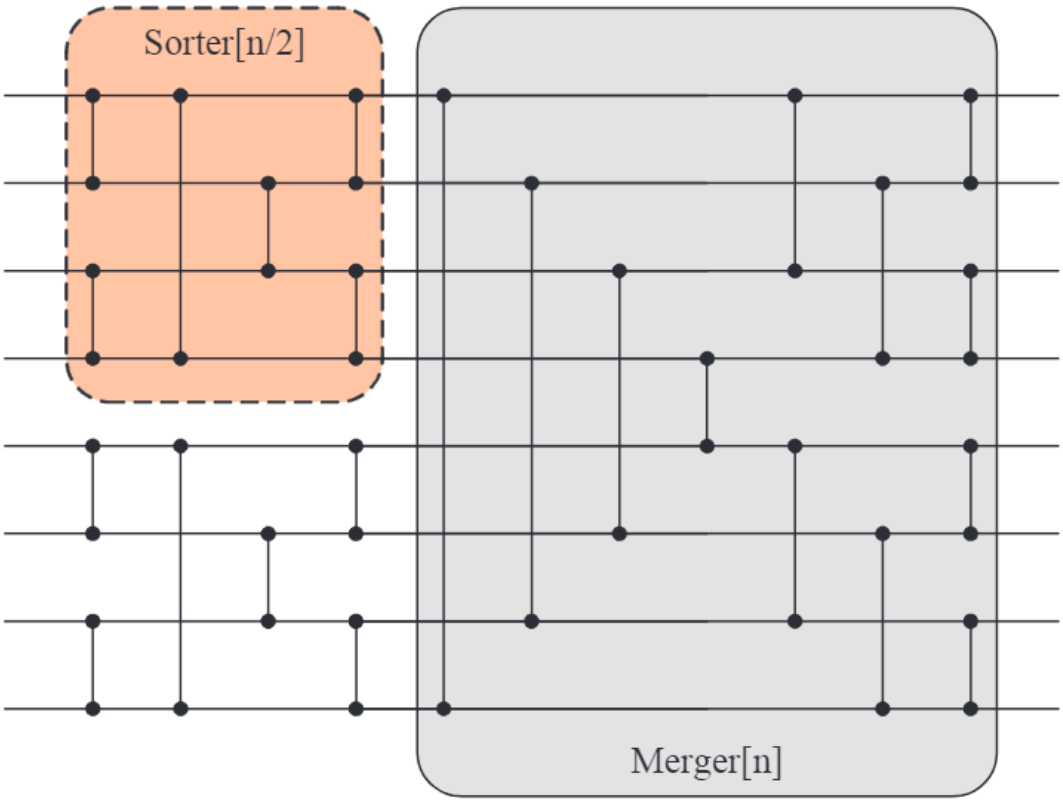
Merger

Given two sorted sequences, if we reverse the order of the second sequence and then concatenate the two sequences, the resulting sequence $\langle a_1, a_2, \cdots, a_{n/2}, a_n, a_{n-1}, \cdots, a_{n/2+1} \rangle$ is bitonic.

To achieve this, we make half cleaner compare i and $n - i + 1$. The order of the outputs from the bottom of this procedure is reversed.



Sorter



Let $D(n)$ be the depth, we have $D(n) = D(n/2) + O(\log n)$. Thus, $D(n) = O(\log^2 n)$.

Chap4: Greedy Algorithm

Greedy Analysis Strategies

- stays ahead

  show that in each step of the greedy algorithm, its solution is at least as good as any other solution. (e.g. interval scheduling.)

- structural

  discover a simple structural bound asserting every possible solution must have a certain value, and prove that your algorithms always achieves the bound. (e.g. interval partitioning.)

- exchange argument

  gradually transform any solution to the result by the greedy algorithm without hurting its quality. (e.g. minimize lateness)

Examples

Interval Scheduling

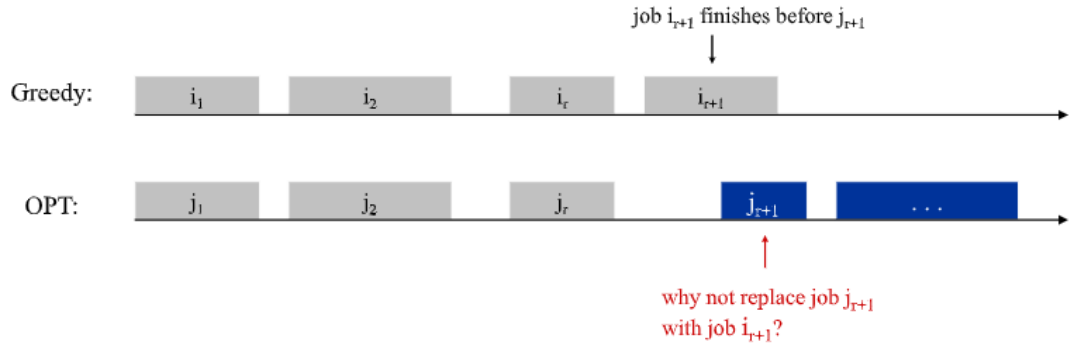
Job i starts at s_i and finishes at f_i, find maximum subset of mutually compatible(not overlap) jobs.

Theorem. Greedy Interval Scheduling algorithm is optimal.

Proof. (by contradiction) Assume greedy is not optimal.

Let $i_1, i_2, \cdots, i_k$ denote set of jobs selected by greedy.

Let $j_1, j_2, \cdots, j_m$ denote set of jobs in an optimal solution with $i_1 = j_1$, $i_2 = j_2, \cdots, i_r = j_r$ for the largest possible value of r.



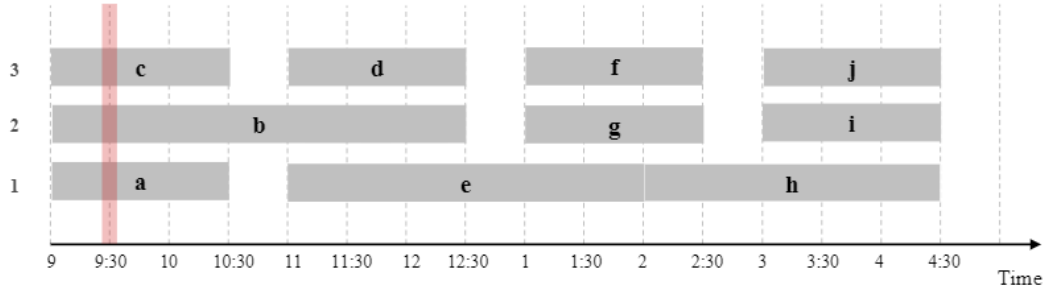
Interval Partitioning

Lecture i starts at s_i and finishes at f_i, find minimum number of classrooms to schedule all lectures.

Definition: The depth of a set of open intervals is the maximum number that contain any given time.

Key observation. Number of classrooms needed $\geq$ depth.

Example: Depth of schedule $= 3 \Rightarrow$ The schedule is optimal.



```
Sort intervals by starting time so that s_1 ≤ s_2 ≤ ... ≤ s_n;
d ← 0 ;                    // number of allocated classrooms
for j = 1 to n do
    if lecture j is compatible with some classroom k then
        schedule lecture j in classroom k;
    else
        allocate a new classroom d + 1;
        schedule lecture j in classroom d + 1;
        d ← d + 1;
return A;
```

Proof. Let d = number of classrooms that the algorithm allocates.

Classroom d is opened because we needed to schedule a job, say j, that is incompatible with all $d - 1$ other classrooms. (These d jobs each end after s_j.)

Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than s_j. Thus, we have d lectures overlapping at time $s_j + \varepsilon$. □

Minimize Lateness

Single CPU. Job i requires t_i time to finish and deadline is d_i. $l_i = \max\{f_i - d_i, 0\}$. Schedule jobs to minimize lateness $L = \max l_i$.

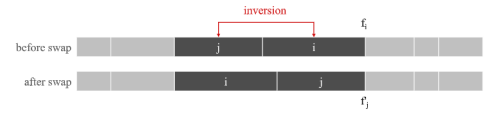
Definition. Given a schedule S, an inversion is a pair of jobs i and j such that: $i < j$ but j scheduled before i.

[as before, we assume jobs are numbered so that $d_1 \le d_2 \le \ldots \le d_n$]

Observation. Greedy schedule has no inversions.

Observation. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

Claim. Swapping two consecutive, inverted jobs reduces the number of inversions by one and does not increase the max lateness.



Proof. Let l be the lateness before the swap, and let l' be it afterwards.

- $l'_k = l_k$ for all $k \neq i, j$
- $l'_i \leq l_i$
- If job j is late:

$$\begin{aligned} l'_j &= f'_j - d_j && \text{(definition)} \\ &= f_i - d_j && (j \text{ finishes at time } f_i) \\ &\leq f_i - d_i && (i < j) \\ &\leq l_i && \text{(definition)} \end{aligned}$$

Theorem. Greedy schedule S is optimal.

Proof. Define S^* to be an optimal schedule that has the fewest number of inversions. (We can assume S^* has no idle time.)

- If S^* has no inversions, then $S = S^*$.
- If S^* has an inversion, let $i - j$ be an adjacent inversion.

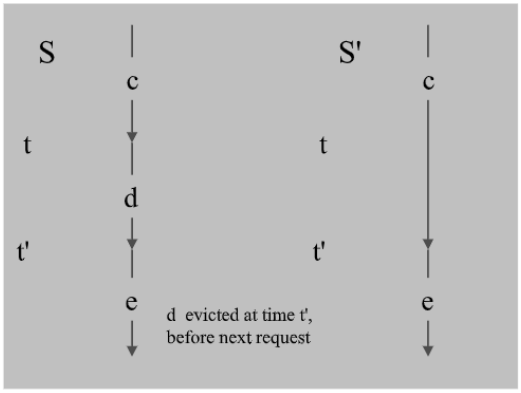
  Swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions. This contradicts definition of S^*. □

Optimal Caching

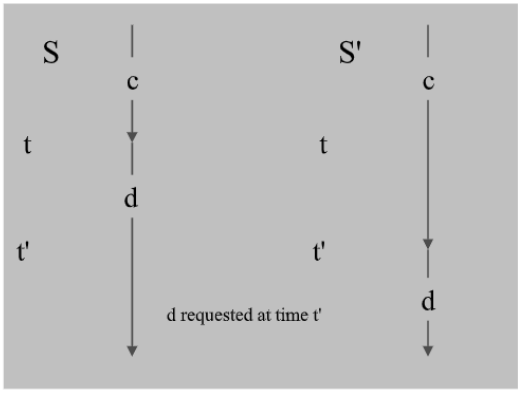
evicts the item that is not requested until farthest in the future.

Claim. Given any unreduced schedule S, we can transform it into a reduced schedule S' with no more cache replacement (insertion).

Proof. (by induction on number of unreduced† items)
Suppose S brings d into the cache at time t, without a request. Let c be the item S evicts when it brings d into the cache.



Case 1



Case 2

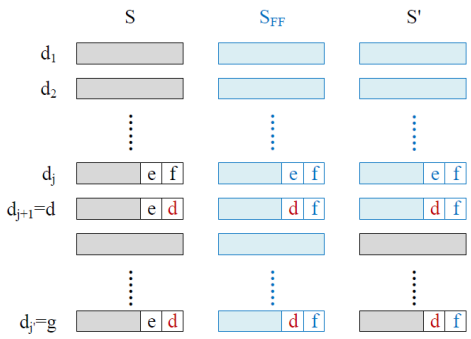
Proof. (by induction on number of requests j)

Invariant: There exists an optimal reduced schedule S that makes the same eviction schedule as S_{FF} through the first $j+1$ requests.

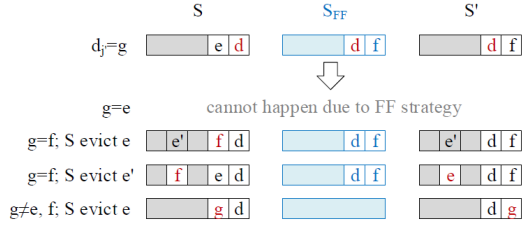
Let S be reduced schedule that satisfies invariant through j requests. We produce S' that satisfies invariant after $j+1$ requests.

Consider $(j+1)^{th}$ request $d = d_{j+1}$. Since S and S_{FF} have agreed up until now, they have the same cache contents before request $j+1$.

- Case 1: (d is already in the cache) $S' = S$ satisfies invariant.
- Case 2: (d is not in the cache; S, S_{FF} evict same element) $S' = S$ satisfies invariant.
- Case 3: (d is not in the cache; S_{FF} evicts e; S evicts $f \neq e$) Let S' agree with S_{FF} at the $j+1$ requests; we show that having element f in cache is no worse than having element e.



Let j' be the first time after $j+1$ that S and S' take a different action (must involve e or f or both), and let g be item requested at time j'.



Case 3*a*: $g = e$. Can't happen with Farthest-In-Future since there must be a request for f before e.

Case 3*b*: $g = f$. Element f can't be in cache of S, so let e' be the element that S evicts.

- if $e' = e$, S' accesses f from cache; now S and S' have same cache
- if $e' \neq e$, S' evicts e' and brings e into the cache; now S and S' have the same cache.

Case 3*c*: $g \neq e, f$. S must evict e (otherwise S' would take the same action). Make S' evict f; now S and S' have the same cache. □

note: key is to make S and S' have same cache after.

Chap5: Matroid

Key: Current maximal must belongs to one of the global optimal solutions.

Matroid

Independent System

S is a finite set and $\mathbf{C}$ is a collection of subsets of S

$(S, \mathbf{C})$ is called an independent system if $A \subset B, B \in \mathbf{C} \Rightarrow A \in \mathbf{C}$

a.k.a $\mathbf{C}$ is hereditary(遗传性) and each subset in $\mathbf{C}$ is called an independent subset.

Note that the empty set $\emptyset$ is necessarily a member of $\mathbf{C}$.

example:

Given $G = (V, E)$ and $\mathbf{H} = \{F \subseteq E \mid F \text{ is a Hamiltonian circuit or a union of disjoint paths }\}$.

Consider $(E, \mathbf{H})$. Given any $F \in \mathbf{H}$ and $P \subset F$, P is a union of disjoint paths.

Matroid

An independent system $(S, \mathbf{C})$ is a matroid if it satisfies the exchange(交换性) property:

$$A, B \in \mathbf{C} \text{ and } |A| < |B| \Rightarrow \exists x \in B \backslash A \text{ such that } A \cup \{x\} \in \mathbf{C}.$$

examples:

- graphic matroid

  Consider a (undirected) graph $G = (V, E)$. Let $S = E$ and $\mathbf{C}$ the collection of all edge sets each of which induces an acyclic subgraph of G. Then $M_G = (S, \mathbf{C})$ is a matroid.

  Exchange Property: consider $A, B \in \mathbf{C}$ with $|A| < |B|$.

  Note that (V, A) has $|V| - |A|$ connected components and (V, B) has $|V| - |B|$ connected components.

  Hence, B has an edge e connecting two connected components of (V, A), which implies $A \cup \{e\} \in \mathbf{C}$.

- uniform matroid

  A subset $X \subseteq \{1, 2, \cdots, n\}$ is independent if and only if $|X| \leq k$.

Greedy Algorithm on Matroid

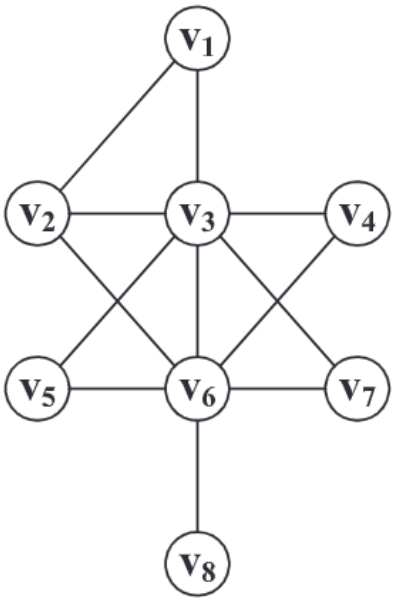
An element x is called an extension of an independent subset I if $x \notin I$ and $I \cup \{x\}$ is independent.

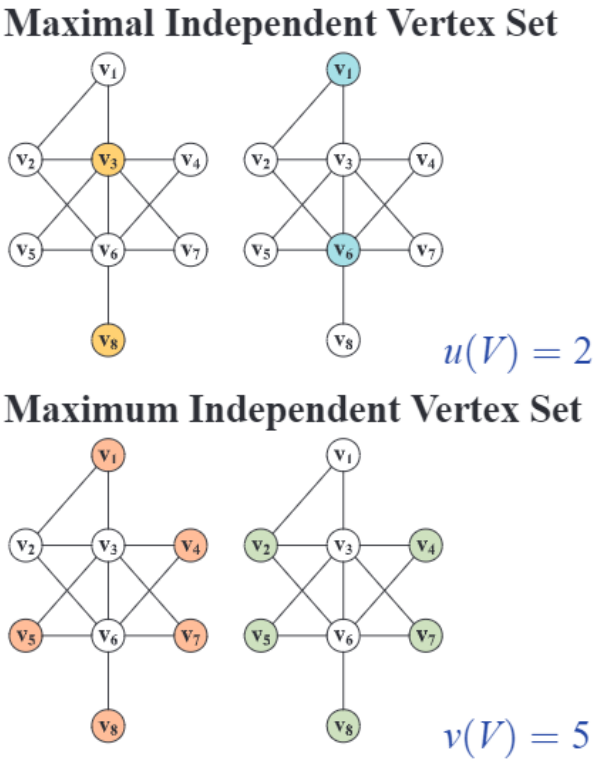
For any subset $F \subseteq S$, an independent subset $I \subseteq F$ is maximal in F if I has no extension in F.

Consider an independent system $(S, \mathbf{C})$. For $F \subseteq S$, define

$$u(F) = \min\{|I| \mid I \text{ is a maximal independent subset of } F\}$$
$$v(F) = \max\{|I| \mid I \text{ is an independent subset of } F\}$$

$u(F)$: min maximal(极大), $v(F)$: maximum(最大)





Matroid Theorem: An independent system $(S, \mathbf{C})$ is a matroid if and only if for any $F \subseteq S, u(F) = v(F)$.

proof:

$\Rightarrow$: For two maximal independent subsets A and B, if $|A| < |B|$, then there must exist an $x \in B$ such that $A \cup \{x\} \in \mathbf{C}$, contradicting the maximal of A.

$\Leftarrow$: Consider two independent subsets A and B with $|A| < |B|$. Set $F = A \cup B$. Then every maximal independent subset I of F has size $|I| \geq |B| > |A|$. Hence, A cannot be a maximal independent subset of F, so A has an extension in F.

Note: The definition of matroid could be either by exchange property or by $\forall F \subseteq S, u(F) = v(F)$

Corollary: All maximal independent subsets in a matroid have the same size.

GREEDY-MAX

In a matroid $(S, \mathbf{C})$, every maximal independent subset of S is called a basis

An independent system $(S, \mathbf{C})$ with a nonnegative function $c : S \to \mathbb{R}^+$ is called a weighted independent system. And there is a maximum weight independent subset which is a basis.

We define c as,

$$c(A) = \sum_{x \in A} c(x)$$

We want to maximize $c(I)$ for any independent system $(S, \mathbf{C})$ with cost function c

By the way, we can use $c^*(x_i) = m - c(x_i)$ to minimize $c(I)$

Algorithm 1: Greedy-MAX

Sort all elements in S into ordering $c(x_1) \geq c(x_2) \geq \cdots \geq c(x_n)$;
$A \leftarrow \emptyset$;
for $i = 1$ *to* n **do**
  if $A \cup \{x_i\} \in \mathbf{C}$ **then**
    $A \leftarrow A \cup \{x_i\}$;
output A;

If each check takes $f(n)$, the algorithm yields $O(n \log n + nf(n))$.

Greedy Theorem: Consider a weighted independent system. Let A_G be obtained by the Greedy Algorithm. Let A^* be an optimal solution. Then

$$1 \leq \frac{c(A^*)}{c(A_G)} \leq \max_{F \subseteq S} \frac{v(F)}{u(F)}$$

where $v(F)$ is the maximum size of independent subset in F and $u(F)$ is the minimum size of maximal independent subset in F.

proof:

Denote $S_i = \{x_1, \ldots, x_i\}$. (Sorted in nonincreasing order). Then we prove that $S_i \cap A_G$ is a maximal independent subset of S_i.

(By Contradiction) If not, there exists an element $x_j \in S_i \backslash A_G$ such that $(S_i \cap A_G) \cup \{x_j\}$ is independent.

However, at the beginning of the jth iteration of the loop in the Greedy-Max, x_j must be selected into A_G^{j-1}. (Since $A_G^{j-1} \cup \{x_j\}$ must be a subset of $(S_j \cap A_G) \cup \{x_j\}$, and hence, is an independent set.)

Therefore we have $|S_i \cap A_G| \geq u(S_i)$.

Moreover, since $S_i \cap A^*$ is independent, we have $|S_i \cap A^*| \leq v(S_i)$.

Now we express $c(A_G)$ and $c(A^*)$ in terms of $|S_i \cap A_G|$ and $|S_i \cap A^*|$.

$$\text{Firstly, } |S_i \cap A_G| - |S_{i-1} \cap A_G| = \begin{cases} 1, & \text{if } x_i \in A_G, \\ 0, & \text{otherwise.} \end{cases}$$

Therefore,

$$\begin{aligned} c(A_G) &= \sum_{x_i \in A_G} c(x_i) \\ &= c(x_1) \cdot |S_1 \cap A_G| + \sum_{i=2}^{n} c(x_i) \cdot (|S_i \cap A_G| - |S_{i-1} \cap A_G|) \\ &= \sum_{i=1}^{n-1} |S_i \cap A_G| \cdot (c(x_i) - c(x_{i+1})) + |S_n \cap A_G| \cdot c(x_n) \end{aligned}$$

Similarly,

$$c(A^*) = \sum_{i=1}^{n-1} |S_i \cap A^*| \cdot (c(x_i) - c(x_{i+1})) + |S_n \cap A^*| \cdot c(x_n)$$

Define $\rho = \max_{F \subseteq S} \frac{v(F)}{u(F)}$. Then we have

$$\begin{aligned} c(A^*) &= \sum_{i=1}^{n-1} |S_i \cap A^*| \cdot (c(x_i) - c(x_{i+1})) + |S_n \cap A^*| \cdot c(x_n) \\ &\leq \sum_{i=1}^{n-1} v(S_i) \cdot (c(x_i) - c(x_{i+1})) + v(S_n) \cdot c(x_n) \\ &\leq \sum_{i=1}^{n-1} \rho \cdot u(S_i) \cdot (c(x_i) - c(x_{i+1})) + \rho \cdot u(S_n) \cdot c(x_n) \\ &\leq \sum_{i=1}^{n-1} \rho \cdot |S_i \cap A_G| \cdot (c(x_i) - c(x_{i+1})) + \rho \cdot |S_n \cap A_G| \cdot c(x_n) \\ &= \rho \cdot c(A_G). \end{aligned}$$

Note: The implies the result of greedy isn't too bad.

Corollary: GREED-MAX yield optimal solution on matroid.

example: Kruskal Algorithm

Theorem: An independent system $(S, \mathbf{C})$ is a matroid iff for any cost function $c(\cdot)$, the Greedy-MAX algorithm gives an optimal solution.

proof:

$\Leftarrow$: For contradiction, suppose independent system $(S, \mathbf{C})$ is not a matroid. Then there exists $F \subseteq S$ such that F has two maximal independent sets I and J with $|I| < |J|$. Define

$$c(e) = \begin{cases} 1+\varepsilon & \text{if } e \in I \\ 1 & \text{if } e \in J \backslash I \\ 0 & \text{if } e \in S \backslash (I \cup J) \end{cases}$$

where ε is a sufficiently small positive number to satisfy $c(I) < c(J)$. Then the Greedy-MAX algorithm will produce I, which is not optimal.

Theorem 1. *Suppose an independent system $(E, \mathcal{I})$ is the intersection of k matroids $(E, \mathcal{I}_i)$, $1 \le i \le k$; that is, $\mathcal{I} = \bigcap_{i=1}^{k} \mathcal{I}_i$. Then $\max_{F \subseteq E} \frac{v(F)}{u(F)} \le k$, where $v(F)$ is the maximum size of independent subset in F and $u(F)$ is the minimum size of maximal independent subset in F.*

Example: Unit-Time Task Scheduling

The problem of **scheduling unit-time tasks with deadlines and penalties for a single processor** has the following inputs:

- a set $S = \{1, 2, ..., n\}$ of n unit-time tasks;
- a set of n integer deadlines $d_1, d_2, \ldots, d_n$, such that each d_i satisfies $1 \le d_i \le n$ and task i is supposed to finish by time d_i;
- a set of n nonnegative weights or penalties $w_1, w_2, \ldots, w_n$, such that a penalty w_i is incurred if task i is not finished by time d_i.

Requirement: find a schedule for S on a machine within time n that minimizes the total penalty incurred for missed deadline.

Given a schedule S, Define:

Early: a task is early in S if it finishes before its deadline.
Late: a task is late in S if it finishes after its deadline.

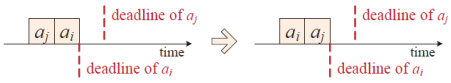
Early-First Form: S is in the early-first form if the early tasks precede the late tasks.

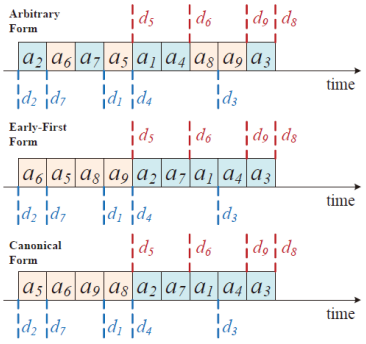
Claim: An arbitrary schedule can always be put into *early-first form* without changing its penalty value.

Canonical Form: An arbitrary schedule can always be transformed into *canonical form*, in which the early tasks precede the late tasks and are scheduled in order of monotonically increasing deadlines.

First put the schedule into early-first form.

Then swap the position of any consecutive early tasks a_i and a_j if $d_j > d_i$ but a_j appears before a_i.





The search for an optimal schedule S thus reduces to finding a set A of tasks that we assign to be early in the optimal schedule.

To determine A, we can create the actual schedule by listing the elements of A in order of monotonically increasing deadlines, then listing the late tasks (i.e., $S - A$) in any order, producing a canonical ordering of the optimal schedule.

Independent: A set of tasks A is independent if there exists a schedule for these tasks without penalty.

Let $N_t(A)$ denote the number of tasks in A whose deadline is t or earlier. $N_0(A) = 0$

Lemma: For any set of tasks A, the statements (1)-(3) are equivalent.
(1). The set A is independent.
(2). For $t = 0, 1, 2, \cdots, n, N_t(A) \leq t$.
(3). If the tasks in A are scheduled in order of monotonically increasing deadlines, then no task is late.

Proof:
$\neg(2) \Rightarrow \neg(1)$: if $N_t(A) > t$ for some t, then there is no way to make a schedule with no late tasks for set A, because more than t tasks must finish before time t. Therefore, (1) implies (2).

$(2) \Rightarrow (3)$: there is no way to "get stuck" when scheduling the tasks in order of monotonically increasing deadlines, since (2) implies that the ith largest deadline is at least i.

$(3) \Rightarrow (1)$: trivial. □

The problem can be changed to maximizing the sum of penalties of early tasks.

Theorem: Let S be a set of unit-time tasks with deadlines and $\mathbf{C}$ the set of all independent tasks of S. Then $(S, \mathbf{C})$ is a matroid.

Proof: (Hereditary): Trivial.
(Exchange Property): Consider two independent sets A and B with $|A| < |B|$. Let k be the largest t such that $N_t(A) \geq N_t(B)$. Then $k < n$ and $N_t(A) < N_t(B)$ for $k + 1 \leq t \leq n$. Choose $x \in \{i \in B \backslash A \mid d_i = k + 1\}$.

$$\text{Then,} \quad N_t(A \cup \{x\}) = N_t(A) \leq t, \quad \text{for } 1 \leq t \leq k,$$

$$\text{and} \quad N_t(A \cup \{x\}) = N_t(A) + 1 \leq N_t(B) \leq t, \quad \text{for } k + 1 \leq t \leq n.$$

Thus $A \cup \{x\} \in \mathbf{C}$. □

Thus, we can use GREEDY-MAX. By the way, we can use segment tree to check $A \cup \{x\}$ in $O(\log n)$.

Chap6: Dynamic Programming

Key: No idea whether current maximal belongs to global optimal or not. Thus, keep all possible sub-problems.

Example

Weighed Interval Scheduling

Let P(j) = largest index i<j such that job i is compatible with j.

After sorting by finish time and start time, we can compute P() with two pointers(双指针) trick.

Input: $n; s_1, \cdots, s_n; f_1, \cdots, f_n; w_1, \cdots, w_n;$
Output: Optimal weight $OPT(n)$.

Sort jobs by finish times so that $f_1 \leq f_2 \leq \cdots \leq f_n$;
Compute $p(1), p(2), \cdots, p(n)$;
$M[0] = 0$;
for $j = 1 \rightarrow n$ **do**
  $M[j] = \max\{w_j + M[p(j)], M[j-1]\}$;

Running Time: $O(n \log n)$.

Hirschberg's Alignment Algorithm

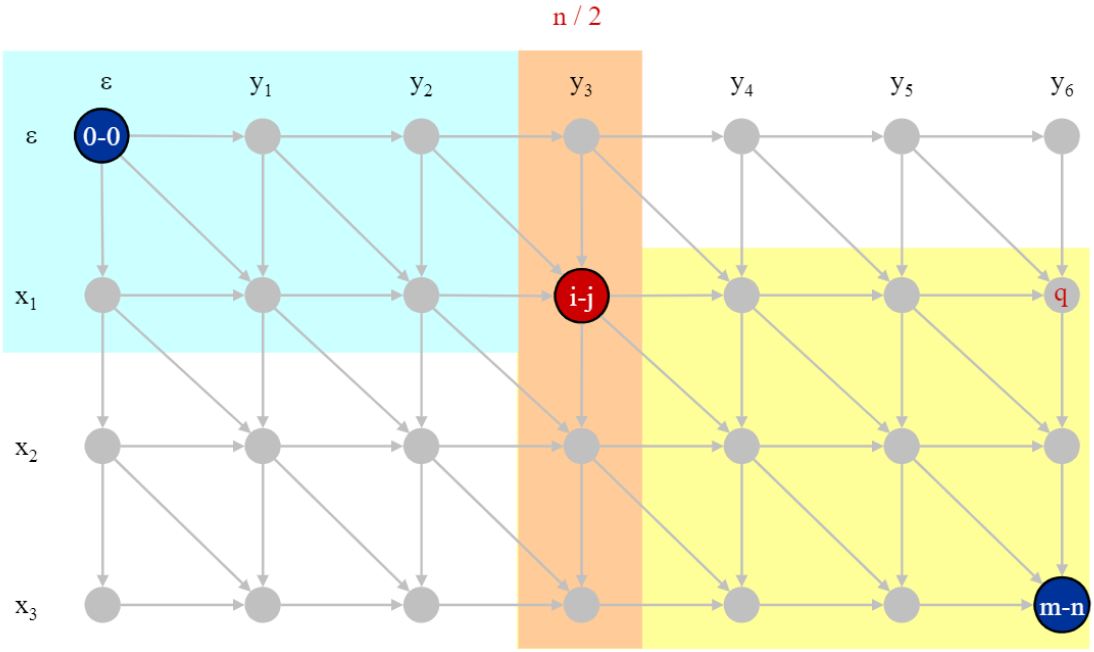
Input: $m, n, x_1 x_2 \cdots x_m, y_1 y_2 \cdots y_n, \alpha, \delta;$
for $i = 0 \rightarrow m$ **do** $M[i, 0] = i\delta$;
for $j = 0 \rightarrow n$ **do** $M[0, j] = j\delta$;
for $i = 1 \rightarrow m$ **do**
  for $j = 1 \rightarrow n$ **do**
    $M[i,j] = \min(\alpha[x_i, y_j] + M[i-1, j-1], \delta + M[i-1, j],$
    $\delta + M[i, j-1])$;
return $M[m, n]$;

Analysis: $\Theta(mn)$ time and space.

$O(m+n)$ space and $O(mn)$ time to find optimal alignment

edit distance graph: edge weight equals to cost

two dynamic programing, f from $(0,0)$ and g from (m,n)



Let q be an index that minimizes $f(q, n/2) + g(q, n/2)$. Then, the shortest path from $(0, 0)$ to (m, n) uses $(q, n/2)$.

We divide the problem and recursively compute optimal alignment in each piece

Let $T(m, n) =$ max running time of algorithm on strings of length m and n.

$T(m, n) = O(mn)$

- Base cases: $m = 2$ or $n = 2$.
- Inductive hypothesis: $T(m, n) \leq 2\text{cmn}$

$$\begin{aligned} T(m, n) &\leq T(q, n/2) + T(m - q, n/2) + cmn \\ &\leq 2cqn/2 + 2c(m - q)n/2 + cmn \\ &= cqn + cmn - cqn + cmn \\ &= 2cmn \end{aligned}$$

Chap7: Linear Programming

Standard Form of LP

$$\begin{aligned} &\max \sum_{j=1}^{n} c_j x_j \\ \text{s.t.} \quad &\sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad i = 1, 2, \cdots, m \\ &x_j \geq 0 \quad j = 1, 2, \cdots, n \end{aligned}$$

Transform to Standard Form

1. From min **to** max **:**

$$\min \quad \sum_{j=1}^{n} c_j x_j \quad \Rightarrow \quad \max \quad -\sum_{j=1}^{n} c_j x_j$$

$$\begin{array}{lrcll} \min & -2x_1 + 3x_2 & \Rightarrow & \max & 2x_1 - 3x_2 \\ s.t. & x_1 + x_2 = 7, & & s.t. & x_1 + x_2 = 7, \\ & x_1 - 2x_2 \le 4, & & & x_1 - 2x_2 \le 4, \\ & x_1 \ge 0. & & & x_1 \ge 0. \end{array}$$

2. Equality Constraint :

$$\sum_{j=1}^{n} a_{ij} x_j = b_i \quad \Rightarrow \quad \begin{cases} \sum_{j=1}^{n} a_{ij} x_j \le b_i \\ \sum_{j=1}^{n} a_{ij} x_j \ge b_i \end{cases}$$

$$\begin{array}{lrcll} \max & 2x_1 - 3x_2 & \Rightarrow & \max & 2x_1 - 3x_2 \\ s.t. & x_1 + x_2 = 7, & & s.t. & x_1 + x_2 \le 7, \\ & x_1 - 2x_2 \le 4, & & & x_1 + x_2 \ge 7, \\ & x_1 \ge 0. & & & x_1 - 2x_2 \le 4, \\ & & & & x_1 \ge 0. \end{array}$$

3. Inequality Constraint with $\ge$ **:**

$$\sum_{j=1}^{n} a_{ij} x_j \ge b_i \quad \Rightarrow \quad -\sum_{j=1}^{n} a_{ij} x_j \le -b_i$$

$$\begin{array}{lrcll} \max & 2x_1 - 3x_2 & \Rightarrow & \max & 2x_1 - 3x_2 \\ s.t. & x_1 + x_2 \le 7, & & s.t. & x_1 + x_2 \le 7, \\ & x_1 + x_2 \ge 7, & & & -x_1 - x_2 \le -7, \\ & x_1 - 2x_2 \le 4, & & & x_1 - 2x_2 \le 4, \\ & x_1 \ge 0. & & & x_1 \ge 0. \end{array}$$

4. Variables without Constraints :

x_2 is without constraints. $\Rightarrow$ Introducing x_2^+ and x_2^-

$x_2 = x_2^+ - x_2^-, x_2^+, x_2^- \ge 0.$

$$\begin{array}{lrcll} \max & 2x_1 - 3x_2 & & & \\ s.t. & x_1 + x_2 \le 7, & \Rightarrow & \max & 2x_1 - 3x_2^+ + 3x_2^- \\ & -x_1 - x_2 \le -7, & & s.t. & x_1 + x_2^+ - x_2^- \le 7, \\ & x_1 - 2x_2 \le 4, & & & -x_1 - x_2^+ + x_2^- \le -7, \\ & x_1 \ge 0. & & & x_1 - 2x_2^+ + 2x_2^- \le 4, \\ & & & & x_1, x_2^+, x_2^- \ge 0. \end{array}$$

Standard Form

Slack Form of LP

Constraints are equality constraints, and only inequality are $x_i \ge 0$.

Key: introducing slack variables s_i

$$\sum_{j=1}^{n} a_{ij} x_j \le b_i \quad \Rightarrow \quad \sum_{j=1}^{n} a_{ij} x_j + s_i = b_i, s_i \ge 0$$

we can use matrix and vectors to denote LP. With $\mathbf{A} = (a_{ij})_{m \times n}$, $\mathbf{b} = (b_1, b_2, \cdots, b_m)^T$, $\mathbf{c} = (c_1, c_2, \cdots, c_n)^T$, $\mathbf{x} = (x_1, x_2, \cdots, x_n)^T$

$$\begin{aligned} \max \quad & \mathbf{c}^T \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \le \mathbf{b} \\ & \mathbf{x} \ge \mathbf{0} \end{aligned}$$

Trick: Only requires to meet at least k conditions, then introduce m extra binary variables, and introduce a large constant M to eliminate the constraints that we don't want. (e.g. $\cdots \le b_i + (1 - v_i)M$)

Other Programming

general form: maximize $f(\mathbf{x})$, with $g_i(\mathbf{x}) \le 0$ and $h_i(\mathbf{x}) = 0$

Integer Linear Programming

with additional constraint that variables $\mathbf{x}$ must be integer

examples:

- Independent Set

$$\begin{aligned} \max \quad & \textstyle\sum_{u \in V} x_u \\ \text{s.t.} \quad & x_u + x_v \le 1 \quad \forall (u, v) \in E \\ & x_u \in \{0, 1\}, \quad \forall v \in V \end{aligned}$$

- Dominating Set

either v or at least one neighbor is in S

$$\begin{aligned} \min \quad & \textstyle\sum_{u \in V} x_u \\ \text{s.t.} \quad & x_u + \textstyle\sum_{v \in N(u)} x_v \geq 1, \quad \forall u \in V \\ & x_u \in \{0,1\}. \quad \forall u \in V \end{aligned}$$

Duality

Example

$$\begin{aligned} \max \quad & f(x_1, x_2) = x_1 + 6x_2 \\ \text{s.t.} \quad & x_1 + x_2 \leq 400 \\ & x_1 \leq 200 \\ & x_2 \leq 300 \\ & x_1, x_2 \geq 0 \end{aligned}$$

Q: How do we know whether $(100, 300)$ is optimal?

We can use y_i to multiply constraint, which yields
$(y_1 + y_2)x_1 + (y_1 + y_3)x_2 \leq 400y_1 + 200y_2 + 300y_3$

We want to use the above inequality to constrain $x_1 + 6x_2$

$$x_1 + 6x_2 \leq 400y_1 + 200y_2 + 300y_3 \text{ if } \begin{Bmatrix} y_1, y_2, y_3 \geq 0 \\ y_1 + y_2 \geq 1 \\ y_1 + y_3 \geq 6 \end{Bmatrix}$$

We should minimize $400y_1 + 200y_2 + 300y_3$ to get the tightest upper bound of $x_1 + 6x_2$. A new LP problem!

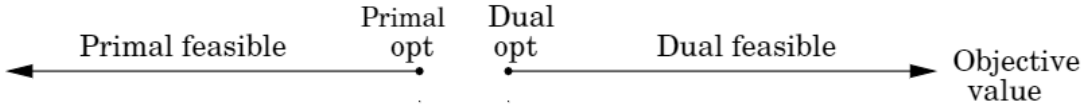
Dual Form

$$\begin{aligned} \max \quad & \mathbf{c}^T\mathbf{x} \\ s.t. \quad & \mathbf{Ax} \leq \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}. \end{aligned} \quad \Rightarrow \quad \begin{aligned} \min \quad & \mathbf{y}^T\mathbf{b} \\ s.t. \quad & \mathbf{y}^T\mathbf{A} \geq \mathbf{c}^T, \\ & \mathbf{y} \geq \mathbf{0}. \end{aligned}$$

Primal Form Dual Form

Duality Theorem

dual LP is an upper bound of primal LP



Weak Duality Theorem: Let x be any feasible solution to the primal LP, and let y be any feasible solution to its dual LP. Then $\sum_{j=1}^{n} c_j x_j \leq \sum_{i=1}^{m} b_i y_i$.

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} c_j x_j \\ s.t. \quad & \sum_{j=1}^{n} a_{ij} x_j \le b_i, \quad \forall i \\ & x_j \ge 0. \quad \forall j \end{aligned} \quad \Rightarrow \quad \begin{aligned} \min \quad & \sum_{i=1}^{m} b_i y_i \\ s.t. \quad & \sum_{i=1}^{m} a_{ij} y_i \ge c_j, \quad \forall j \\ & y_i \ge 0. \quad \forall i \end{aligned}$$

Proof.

$$\sum_{j=1}^{n} c_j x_j \le \sum_{j=1}^{n} (\sum_{i=1}^{m} a_{ij} y_i) x_j = \sum_{i=1}^{m} (\sum_{j=1}^{n} a_{ij} x_j) y_i \le \sum_{i=1}^{m} b_i y_i.$$

Strong Duality Theorem: x and y are optimal solutions to primal and dual LPs respectively if and only if $\sum_{j=1}^{n} c_j x_j = \sum_{i=1}^{m} b_i y_i$.

Simplex Method

Idea: The optimal solution of LP exists on the vertex of the feasible region.

what simplex does on each iteration

- check whether current vertex is optimal
- move to the neighbor that can increase objective function

perform simplex

1. convert to slack form

   slack variables x_3, x_4, x_5 are basic variables

   Note: It is recommended to put basic variables on one side and put others on another side.

$$\begin{aligned} \max \quad & x_1 + 6x_2 \\ s.t. \quad & x_1 + x_2 \le 400, \\ & x_1 \le 200, \\ & x_2 \le 300, \\ & x_1, x_2 \ge 0. \end{aligned} \quad \Rightarrow \quad \begin{aligned} \max \quad & x_1 + 6x_2 \\ s.t. \quad & 400 - x_1 - x_2 = x_3, \\ & 200 - x_1 = x_4, \\ & 300 - x_2 = x_5, \\ & x_1, x_2, x_3, x_4, x_5 \ge 0. \end{aligned}$$

2. obtain basic solution

   set non-basic variables to 0. $\bar{x} = (\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_5) = (0, 0, 400, 200, 300)$

3. select non-basic variable

   select a non-basic variable x_i with positive coefficient to increase f

   increase x_i as much as possible

   Note: The premise is NOT changing the value of non-basic variables!

$$
\begin{aligned}
\max \quad & x_1 + 6x_2 \\
s.t. \quad & 400 - x_1 - x_2 = x_3, \\
& 200 - x_1 = x_4, \\
& 300 - x_2 = x_5, \\
& x_1, x_2, x_3, x_4, x_5 \geq 0.
\end{aligned}
$$

$$\Downarrow$$

$$
\begin{aligned}
\max \quad & x_1 + 6x_2 \\
s.t. \quad & 400 - x_1 - x_2 = x_3, \\
& 200 - x_1 = x_4, \\
& 300 - x_5 = x_2, \\
& x_1, x_2, x_3, x_4, x_5 \geq 0.
\end{aligned}
$$

- Choose the nonbasic variable x_2.
- When $x_2 \uparrow$, $x_3 \downarrow$ and $x_5 \downarrow$.

  However x_3 and x_5 should be nonnegative.
  - $x_3 \leq 0$ when $x_2 \geq 400$;
  - $x_5 \leq 0$ when $x_2 \geq 300$.
- $300 - x_2 = x_5$ is the tightest constraint for x_2.

  We transform it into $300 - x_5 = x_2$.

4. pivoting

   exchange a non-basic and a basic variable. x_2 and x_5 in this example.

   Note: remember to change equations accordingly.

$$
\begin{aligned}
\max \quad & x_1 + 6x_2 \\
s.t. \quad & 400 - x_1 - x_2 = x_3, \\
& 200 - x_1 = x_4, \\
& 300 - x_2 = x_5, \\
& x_1, x_2, x_3, x_4, x_5 \geq 0.
\end{aligned}
\quad \Rightarrow \quad
\begin{aligned}
\max \quad & x_1 + 6(300 - x_5) \\
s.t. \quad & 100 - x_1 + x_5 = x_3, \\
& 200 - x_1 = x_4, \\
& 300 - x_5 = x_2, \\
& x_1, x_2, x_3, x_4, x_5 \geq 0.
\end{aligned}
$$

$$
\bar{x} = \{0, 0, 400, 200, 300\} \quad \Rightarrow \quad \bar{x} = \{0, 300, 100, 200, 0\}.
$$

5. repeat

   until all coefficients in objective function is negative

Chap8: Amortized Analysis

give a tighter bound even under worst case

Idea: the cost of expensive operations can be "spread out"(amortized) to all operations

Average-case analysis: average over all input

Amortized analysis: "average" over operations

Methods

different methods may assign different amortized cost

Aggregate Analysis (聚合分析)

Compute the worst time $T(n)$ in total for a sequence of n operations. The amortized cost (average cost) per operation is $T(n)/n$ in the worst case.

examples:

- stack operations

  add, pop and multi-pop

  Key observation: $\#Pop \leq \#Push$; Thus, we have:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} C_i \\ &= \#Push + \#Pop \\ &\leq 2 \times \#Push \\ &\leq 2n \end{aligned}$$

  Conclusion: on average, the MULTIPOP(S, k) step takes only $O(1)$ time rather than $O(k)$ time.

- binary counter

  $000 \rightsquigarrow 001 \rightsquigarrow 010 \rightsquigarrow \cdots$

ALGORITHM 3: INCREMENT(A)

```
i ← 0;
while i ≤ k − 1 and A[i] = 1 do
    A[i] ← 0;
    i ← i + 1;
if i ≤ k − 1 then
    A[i] ← 1;
```

  $A[i]$ flips $n/2^i$ times

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} C_i \\ &= 1 + 2 + 1 + 3 + 1 + 2 + 1 + 4 + \cdots && \text{(add by row)} \\ &= \#flip(A[0]) + \#flip(A[1]) + \cdots + \#flip(A[k]) && \text{(add by column)} \\ &= n + \frac{n}{2} + \frac{n}{4} + \cdots \\ &\leq 2n \end{aligned}$$

  Amortized cost of each operation: $O(n)/n = O(1)$.

Accounting Method (核算法)

Basic idea: for each operation OP with actual cost C_{OP}, an amortized cost $\widehat{C_{op}}$ is assigned such that for any sequence of n operations,

$$T(n) = \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C}_i$$

Intuition: If $\widehat{C_{op}} > C_{op}$, the overcharge will be stored as prepaid credit; the credit will be used later for the operations with $\widehat{C_{op}} < C_{op}$.
The **requirement** that $\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \widehat{C}_i$ is essentially credit never goes negative.

examples:

- stack operations

Operation	Real Cost C_{op}	Amortized Cost $\widehat{C_{op}}$
PUSH	1	2
POP	1	0
MULTIPOP	$\min\{\|S\|, k\}$	0

Credit: the number of items in the stack.

- binary counter

OP	Real Cost C_{OP}	Amortized Cost $\widehat{C_{OP}}$
`flip(0→1)`	1	2
`flip(1→0)`	1	0

Key observation: $\#flip(0 \to 1) \geq \#flip(1 \to 0)$

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} C_i \\ &= \#flip(0 \to 1) + \#flip(1 \to 0) \\ &\leq 2\#flip(0 \to 1) \\ &\leq 2n \end{aligned}$$

Potential Function Method

Basic idea: Define a potential function as a bridge, i.e. we can assign a value to state rather than operation, and amortized costs are then calculated based on potential function.

Potential Function: $\Phi(S) : S \to R$, where S is state collection. (use "credit" as potential)

Amortized Cost Setting: $\widehat{C}_i = C_i + \Phi(S_i) - \Phi(S_{i-1})$.

$$\begin{aligned}\sum_{i=1}^{n}\widehat{C}_i &= \sum_{i=1}^{n}\left(C_i+\Phi\left(S_i\right)-\Phi\left(S_{i-1}\right)\right)\\ &= \sum_{i=1}^{n}C_i+\Phi\left(S_n\right)-\Phi\left(S_0\right)\end{aligned}$$

Requirement: To guarantee $\sum_{i=1}^{n}C_i \leq \sum_{i=1}^{n}\widehat{C}_i$, it suffices to assure $\Phi\left(S_n\right) \geq \Phi\left(S_0\right)$

examples:

- stack operations

  $\Phi(S)$ denote the number of items in stack
- binary counter

  $\Phi(S) = \#1$ in counter

  only flip one 0 to 1 in each operation

$$\begin{aligned}C_i &= \#flip_{0\to1}^{(i)} + \#flip_{1\to0}^{(i)} = 1 + \#flip_{1\to0}^{(i)}\\ \Phi(S_i) &= \Phi(S_{i-1}) + 1 - \#flip_{1\to0}^{(i)}\\ \widehat{C}_i &= C_i + \Phi(S_i) - \Phi(S_{i-1})\\ &= 1 + \#flip_{1\to0}^{(i)} + 1 - \#flip_{1\to0}^{(i)} = 2\end{aligned}$$

Example: Dynamic Table

memory-allocation strategy for vector

Insert Only

- Aggregate method

  expansions are rare

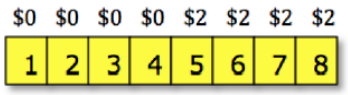
$$\sum_{i=1}^{n}C_i = n + \sum_{j=0}^{\lfloor\log n\rfloor}2^j < 3n$$

  amortized cost is 3.
- Accounting method

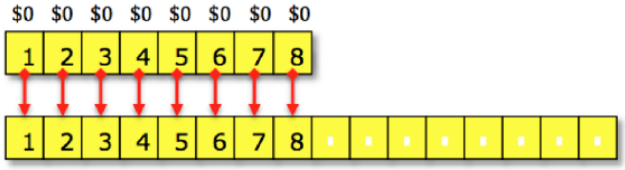
For the i-th operation, an **amortized cost** $\widehat{C}_i = \$3$ is charged.

- $\$1$ pays for the insertion **itself**;
- $\$2$ is stored for **later table doubling**, $\$1$ for copying one of the recent $\frac{i}{2}$ items, $\$1$ for copying one of the old $\frac{i}{2}$ items.

Original:



Expansion:



- Potential function

Basic idea: the **bank account** can be viewed as potential function of the dynamic set. More specifically, we prefer a potential function $\Phi : \{T\} \rightarrow R$ with the following properties:

- $\Phi(T) = 0$ immediately **after** an expansion;
- $\Phi(T) = size[T]$ immediately **before** an expansion; thus, the next expansion can be paid for by the potential.

A possibility: $\Phi(T) = 2 \times num[T] - size[T]$

table is always at least half full, so $\Phi\left(S_n\right) \geq \Phi\left(S_0\right) = 0$

Insert and Delete

load factor $\alpha(T) = num[T]/size[T]$

Trial1: $\alpha(T)$ never drops below 1/2. This may cause thrashing!

Trial2: $\alpha(T)$ never drops below 1/4

$\Phi(T)$ that is 0 immediately after an expansion or contraction, and $size[T]$ as $\alpha(T)$ increases to 1 or decreases to $\frac{1}{4}$

$$\Phi(T) = \begin{cases} 2 \times \text{num}[T] - \text{size}[T] & \text{if } \alpha(T) \geq \frac{1}{2} \\ \frac{1}{2} \text{ size } [T] - num[T] & \text{if } \alpha(T) < \frac{1}{2} \end{cases}$$

$\Phi\left(S_0\right) = 0$ and $\Phi(T)$ never goes negative

insert

Case 1: $\alpha_{i-1} \geq \frac{1}{2}$ and no expansion

The amortized cost is:

$$\begin{aligned} \widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\ &= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (2num_{i-1} - size_{i-1}) \\ &= 3 \end{aligned}$$

Case 2: $\alpha_{i-1} \geq \frac{1}{2}$ and an expansion was triggered

The amortized cost is:

$$\begin{aligned} \widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\ &= num_{i-1} + 1 + (2(num_{i-1} + 1) - 2size_{i-1}) - (2num_{i-1} - size_{i-1}) \\ &= 3 + num_{i-1} - size_{i-1} \quad \leftarrow num_{i-1} = size_{i-1} \\ &= 3 \end{aligned}$$

Case 3: $\alpha_{i-1} < \frac{1}{2}$ and $\alpha_i < \frac{1}{2}$

The amortized cost is:

$$\begin{aligned}\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\frac{1}{2}size_i - num_i) - (\frac{1}{2}size_{i-1} - num_{i-1}) \\ &= 1 + (\frac{1}{2}size_i - num_i) - (\frac{1}{2}size_i - (num_i - 1)) \\ &= 0\end{aligned}$$

Case 4: $\alpha_{i-1} < \frac{1}{2}$ but $\alpha_i \geq \frac{1}{2}$

The amortized cost is:

$$\begin{aligned}\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2num_i - size_i) - (\frac{1}{2}size_{i-1} - num_{i-1}) \\ &= 1 + (2num_i - size_i) - (\frac{1}{2}size_i - (num_i - 1)) \\ &= 1 + 0 - 1 = 0 \quad \leftarrow size_i = 2num_i\end{aligned}$$

delete

Case 1: $\alpha_{i-1} < \frac{1}{2}$ and no contraction

The amortized cost is:

$$\begin{aligned}\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\frac{1}{2}size_i - num_i) - (\frac{1}{2}size_{i-1} - num_{i-1}) \\ &= 1 + (\frac{1}{2}size_{i-1} - (num_{i-1} - 1)) - (\frac{1}{2}size_{i-1} - num_{i-1}) \\ &= 2\end{aligned}$$

Case 2: $\alpha_{i-1} < \frac{1}{2}$ and a contraction was triggered

The amortized cost is:

$$\begin{aligned}\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= num_i + 1 + (\tfrac{1}{2}size_i - num_i) - (\tfrac{1}{2}size_{i-1} - num_{i-1}) \\ &= num_{i-1} + (\tfrac{1}{4}size_{i-1} - (num_{i-1} - 1)) - (\tfrac{1}{2}size_{i-1} - num_{i-1}) \\ &= 1 + num_{i-1} - \tfrac{1}{4}size_{i-1} \quad \leftarrow num_{i-1} = \tfrac{1}{4}size_{i-1} \\ &= 1\end{aligned}$$

Case 3: $\alpha_{i-1} \geq \frac{1}{2}$ and $\alpha_i \geq \frac{1}{2}$

The amortized cost is:

$$\begin{aligned}\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\ &= 1 + (2(num_{i-1} - 1) - size_{i-1}) - (2num_{i-1} - size_{i-1}) \\ &= -1\end{aligned}$$

Case 4: $\alpha_{i-1} \geq \frac{1}{2}$ and $\alpha_i < \frac{1}{2}$

The amortized cost is:

$$\begin{aligned}\widehat{C}_i &= C_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\tfrac{1}{2}size_i - num_i) - (2num_{i-1} - size_{i-1}) \\ &= 1 + (\tfrac{1}{2}size_{i-1} - (num_{i-1} - 1)) - (2num_{i-1} - size_{i-1}) \\ &= 2 + \tfrac{3}{2}size_{i-1} - 3num_{i-1} \\ &= 2 \quad \leftarrow size_{i-1} = 2num_{i-1}\end{aligned}$$

Chap9: Graph Algorithm

Introduction to Graph Theory

complete graph K_n, complete bipartite graph $K_{m,n}$

spanning subgraph: $V' = V$

induced subgraph: $E' = \{(u, v) | u, v \in V' \wedge (u, v) \in E\}$

Handshaking Theorem: $\sum d(v) = 2|E|$

Introduction to Graph Algorithms

MST

proof method: **cycle/cut** property

exchange property: one MST can be changed to another MST, while maintain minimal total cost during procedure

Chap10: Graph Decomposition

DFS in Undirected Graph

Input: $G = (V, E)$ is a graph; $v \in V$
Output: VISITED(u) = *true* for all nodes u **reachable** from v

```
VISITED(v) = true;
PREVISIT(v);
foreach edge (v, u) ∈ E do
    if not VISITED(u) then
        EXPLORE(G, u);
POSTVISIT(v);
```

Previsit and Postvisit Orderings

DFS ordering

$PRE[v]$ record first discovery time and $POST[v]$ record departure time

Lemma: $\forall u, v \in V$, intervals $[\text{PRE}(u), \text{POST}(u)], [\text{PRE}(v), \text{POST}(v)]$ are either disjoint or one is contained within the other.

DFS in Directed Graph

DFS yields a search tree/forest.

- Tree edges: part of the DFS forest.
- Forward edges: lead from a node to a nonchild descendant in the DFS tree.
- Back edges: lead to an ancestor in the DFS tree.
- Cross edges: neither descendant nor ancestor; they lead to anode that has already been explored

PRE/POST ordering for (u, v)	Edge type
$[_u \quad [_v \quad]_v \quad]_u$	Tree/forward
$[_v \quad [_u \quad]_u \quad]_v$	Back
$[_v \quad]_v \quad [_u \quad]_u$	Cross

DAG (Directed Acyclic Graph)

Lemma: A directed graph has a cycle iff its DFS tree has a back edge.

proof: $\Rightarrow$ consider $v_0 \to v_1 \to v_2 \cdots \to v_k \to v_0$. If we first visit v_i, then $v_{i-1} \to v_i$ will be back edge.

Linearization/Topologically Sort

Order the vertices such that every edge goes from a small vertex to a large one.

Lemma: In a DAG, every edge leads to a vertex with a lower POST

proof: (1) explore through this edge, depart from child vertex earlier (2) explored earlier through other edges

Hence, a DAG can be linearized by decreasing POST numbers.

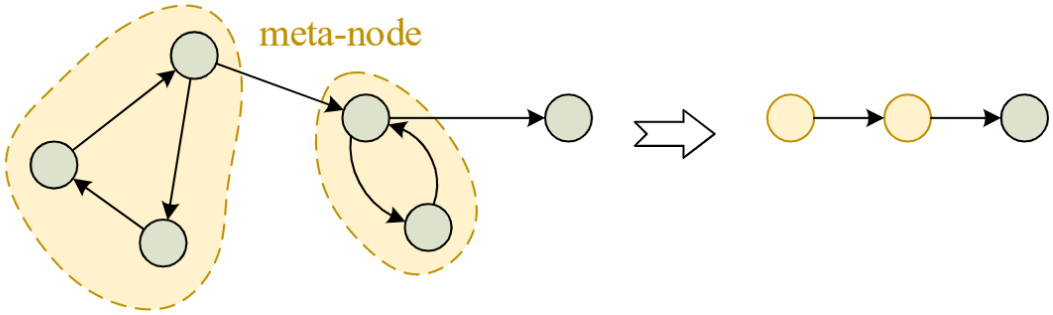
sink: vertex with smallest POST number

source: vertex with highest POST number

Lemma: Every DAG has at least one source and at least one sink

Strongly Connected Components

Lemma: Every directed graph is a DAG of its strongly connected components.



Lemma: If the $EXPLORE$ is started at node u, then it will terminate precisely when all nodes reachable from u have been visited.

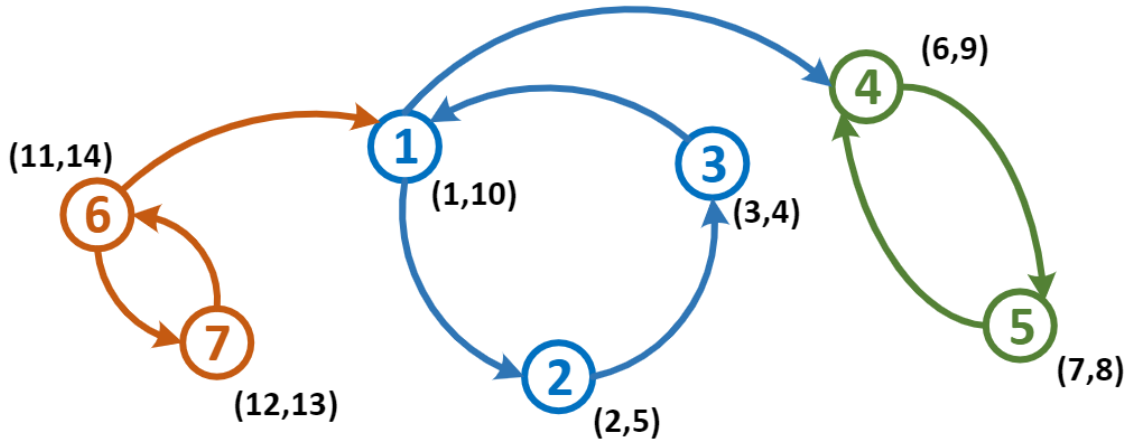
Thus, if we invoke $EXPLORE$ on a sink strongly connected component, we will get exactly that component.

Lemma: If C and C' are strongly connected components, and there is an edge from a node in C to a node in C', then the highest POST number in C is bigger than the highest POST number in C'

proof: similar to lemma in DAG

Lemma: The node that receives the highest POST number in a DFS must lie in a source strongly connected component.

Note: The smallest POST number may NOT lie in a sink strongly connected component!



- How do we find a sure sink SCC?

  consider G^R, node with highest POST number is source in G^R, which is sink in G
- How do we continue after remove sink SCC?

  find next sink SCC, i.e. next highest POST number in G^R

This yields Kosaraju Algorithm.

BFS

```
foreach u ∈ V do
    DIST(u) = ∞;
DIST(s) = 0;
Q = [s] (queue containing just s);
while Q is not empty do
    u = EJECT(Q);
    foreach edge (u, v) ∈ E do
        if DIST(v) = ∞ then
            INJECT(Q, v);
            DIST(v) = DIST(u) + 1;
```

Chap11: Shortest Path

Introduction

Definition

weight of a path: $w(P) = \sum w(v_i, v_{i+1})$

shortest path weight: $d(u, v) = \min\{w(P) \mid P \text{ is a path from } u \text{ to } v\}$

note: $d(u, v) = +\infty$ if no path from u to v exists.

Property

Optimal Substructure: A sub-path of a shortest path is a shortest path.

Triangle Inequality: $\forall v_1, v_2, v_3 \in V, d(v_1, v_2) \leq d(v_1, v_3) + d(v_3, v_2)$.

If a graph G contains a negative-weight cycle, then some shortest path may not exist.

Single-Source Shortest Paths

Dijkstra Algorithm

greedy approach, can only handle positive weight

```
foreach u ∈ V do
    INSERT(Q, u);
while Q ≠ ∅ do
    u ← EXTRACT-MIN(Q);
    S ← S ∪ {u};
    foreach v ∈ Adj[u] do
        if d[v] > d[u] + w(u, v) then
            d[v] ← d[u] + w(u, v) ;    /* Relaxation Step */
            DECREASE-KEY(Q, v);
```

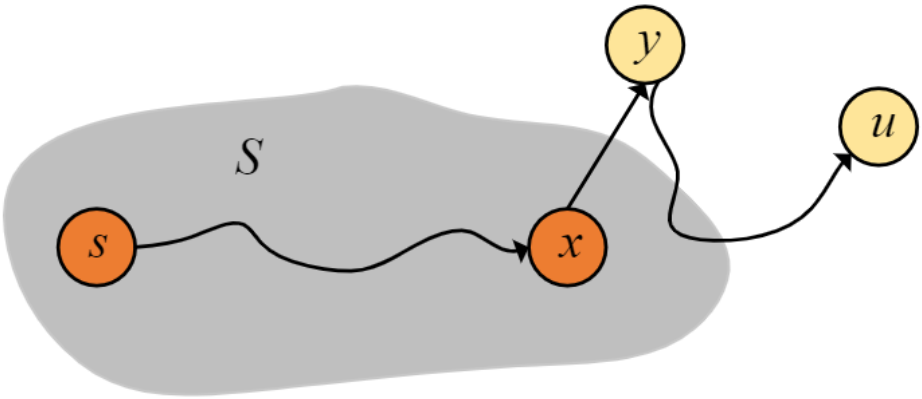
Lemma: Initializing $d[s] \leftarrow 0$ and $d[v] \leftarrow +\infty$ for all $v \in V - \{s\}$ establishes $d[v] \geq d(s, v)$ for all $v \in V$, and this invariant is maintained over any sequence of relaxation steps.

Proof: Suppose not. Let v be the first vertex for which $d[v] < d(s, v)$, and let u be the vertex that caused $d[v]$ to change, $d[v] = d[u] + w(u, v)$ and $d[u] \geq d(s, u)$

$$\begin{aligned} d[v] &< d(s, v) \\ &\leq d(s, u) + d(u, v) \\ &\leq d(s, u) + w(u, v) \\ &\leq d[u] + w(u, v) \end{aligned}$$

Theorem: Dijkstra's algorithm terminates with $d[v] = d(s, v)$ for all $v \in V$

Proof: It suffices to show that $d[v] = d(s, v)$ for every $v \in V$ when v is added to S. Suppose u is the first vertex added to S for which $d[u] \neq d(s, u)$. Let y be the first vertex in $V - S$ along a shortest path from s to u, and let x be its predecessor. (There can be many vertices between s and x or between y and u.) Since u is the first violation, we have $d[x] = d(s, x)$ and $d(s, y) = d[y] = d[x] + w(x, y)$. As y is a vertex on the shortest path to u, $d[y] = d(s, y) \leq d(s, u) \leq d[u]$. However, the algorithm yields $d[u] \leq d[y]$. Contradiction!

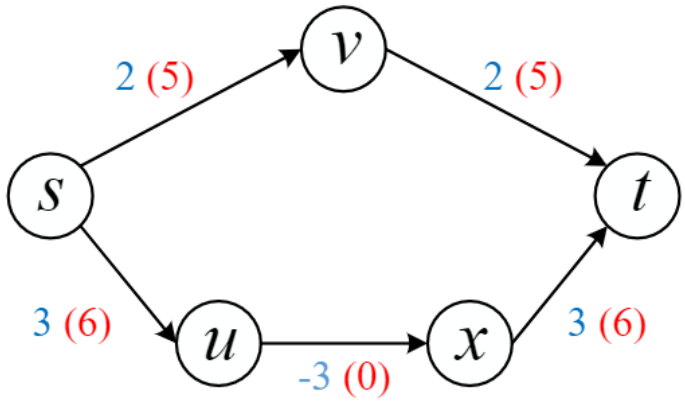


Key: find contradiction between this vertex and its predecessor.

unweighted graph: BFS. In this case, FIFO queue has the same effect as priority queue.

Bellman-Ford Algorithm

allow negative weight and naive re-weighting is not correct



Let $f(i, v)$ be the length of shortest $s - v$ path P using at most i edges.

$$f(i, v) = \min \left\{ f(i-1, v), \min_{(u,v) \in E} \{ f(i-1, u) + w(u, v) \} \right\}$$

improvements: reuse $f[v]$ and no need to check edge (u, v) again unless $f[u]$ changed

---

foreach *node* $u \in V$ **do**
    $M[u] \leftarrow \infty$;
    $predecessor[u] \leftarrow \emptyset$;
$M[s] \leftarrow 0$;
for $i = 1$ *to* $n - 1$ **do**
    foreach *node* $u \in V$ **do**
        if $M[u]$ *has been updated in previous iteration* **then**
            foreach *edge* $(u, v) \in E$ **do**
                if $M[v] > M[u] + w(u, v)$ **then**
                    $M[v] \leftarrow M[u] + w(u, v)$;
                    $predecessor[v] \leftarrow u$;
    If no $M[v]$ changed in this iteration, stop.

Lemma: If $f(n, v) = f(n-1, v)$ for all v, then no negative cycles.

Lemma: If $f(n, v) < f(n-1, v)$ for some node u, then shortest path from s to v contains a cycle W. Moreover W has negative cost.

proof: $f(n, v) < f(n-1, v) \Rightarrow P$ has exactly n edges. By pigeonhole principle, P must contain a directed cycle W. Deleting W yields a $s - v$ path with $< n$ edges $\Rightarrow W$ has negative cost.

Application: Use Bellman-Ford Algorithm to detect negative cycle. Add new vertex s and connect it to all vertices with 0 cost.

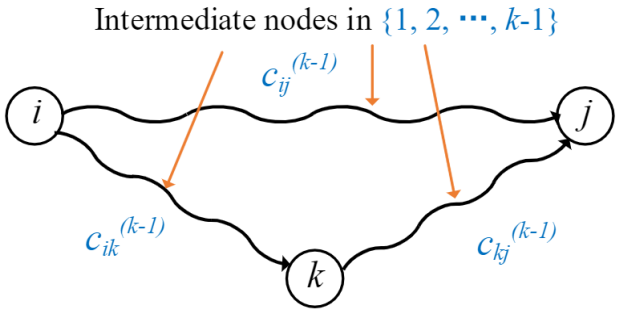
All-Pair Shortest Paths

Floyd-Warshall algorithm

Define $c_{ij}^{(k)}$ as the weight of a shortest path from i to j with intermediate vertices belonging to the set $\{1, 2, \ldots, k\}$.

Insert one new vertex each iteration.

$$c_{ij}^{(k)} = \min_k \left\{ c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)} \right\}$$




```
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            if c_ij > c_ik + c_kj then
                c_ij ← c_ik + c_kj;
```

note: k is the outer most loop

Johnson's Algorithm

Theorem: Given a label $h(v)$ for each $v \in V$, reweight each edge $(u, v) \in E$ by $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$. Then, all paths between the same two vertices are reweighted by the same amount.

Proof: $h(v)$ is similar to a potential function.

$$\begin{aligned}
\hat{w}(P) &= \sum_{i=1}^{k-1} \hat{w}\left(v_i, v_{i+1}\right) \\
&= \sum_{i=1}^{k-1} \left(w\left(v_i, v_{i+1}\right) + h\left(v_i\right) - h\left(v_{i+1}\right)\right) \\
&= \sum_{i=1}^{k-1} w\left(v_i, v_{i+1}\right) + h\left(v_1\right) - h\left(v_k\right) \\
&= w(P) + h\left(v_1\right) - h\left(v_k\right)
\end{aligned}$$

note: shortest path has the property that $d[u] + w(u, v) \geq d[v] \Rightarrow w(u, v) + d[u] - d[v] \geq 0$

Implementation: (1) use Bellman-Ford to calculate $d[x]$. (2) reweight edge and apply Dijkstra on every vertex

Chap12: Network Flow

Introduction

Definition

Description: A flow network is a tuple $G = (V, E, s, t, c)$:

- Directed graph $G = (V, E)$, with source $s \in V$ and sink $t \in V$.
- Assume all nodes are reachable from s, no parallel edges.
- Capacity $c(e) > 0$ for each edge $e \in E$.

Cut: a partition (A, B) of V with $s \in A$ and $t \in B$

Cut capacity: $cap(A, B) = \sum_{e \text{ out of } A} c(e)$ (only care about out)

Flow: An s-t flow f is a function that satisfies:

- Capacity: for each $e \in E : 0 \leq f(e) \leq c(e)$
- Conservation: for each $v \in V - \{s, t\} : \sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$

Value of a flow: $v(f) = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e)$

Property

Flow Value Lemma: Let f be any flow, and let (A, B) be any s-t cut. Then, the value of f equals to the net flow across the cut (A, B).

$$v(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e)$$

Proof: $\sum_{e \text{ out of } v} f(e) = \sum_{e \text{ in to } v} f(e)$ if $v \neq s, t$. Thus,

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e) \\ &= \sum_{v \in A} \left(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ in to } v} f(e) \right) \\ &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \end{aligned}$$

Weak Duality: Let f be any flow. Then, for any $s - t$ cut (A, B) we have

$$v(f) \leq \text{cap}(A, B)$$

proof:

$$\begin{aligned} v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\ &\leq \sum_{e \text{ out of } A} f(e) \\ &\leq \sum_{e \text{ out of } A} c(e) = cap(A, B) \end{aligned}$$

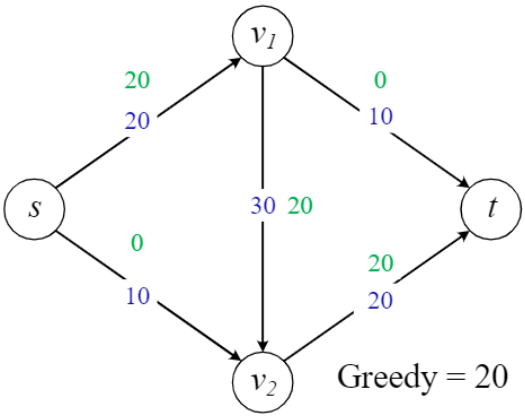
Corollary: Let f be any flow, and let (A, B) be any cut. If $v(f) = \text{cap}(A, B)$, then f is a max flow and (A, B) is a min cut.

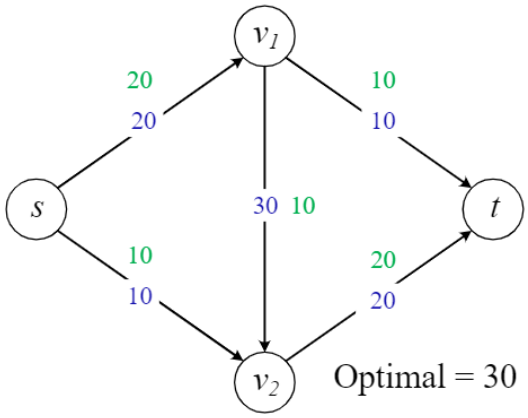
Algorithm

Idea

Greedy algorithm

- Start with $f(e) = 0$
- Find an s-t path P where each edge has $f(e) < c(e)$
- Augment flow along path P
- Repeat until you get stuck

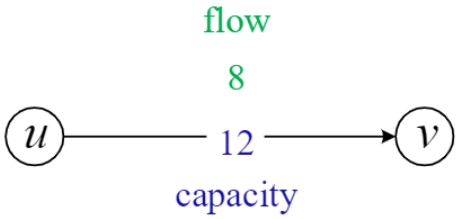




need to undo a bad decision!

Original edge: $e = (u, v) \in E$.

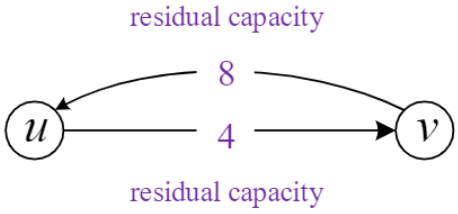
- Flow $f(e)$, capacity $c(e)$.



Residual edge: $e = (u, v) \in E$.

- "Undo" flow sent.
- $e = (u, v)$ and $e^R = (v, u)$.
- Residual capacity:

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e) & \text{if } e^R \in E \end{cases}$$



Residual Network: $G_f = (V, E_f, s, t, c_f)$. $E_f = \{e \mid f(e) < c(e)\} \cup \{e^R \mid f(e) > 0\}$

Greedy Algorithm (**Ford-Fulkerson**): Run the greedy algorithm on G_f to get a max flow f'

Key Property: f' is a flow in G_f iff f is a flow in G.

Ford-Fulkerson Algorithm

An **augmenting path** is a simple $s \rightsquigarrow t$ path in the residual network G_f.

The **bottleneck** capacity of an augmenting path P is the minimum residual capacity of any edge in P.

Key Property: Let f be a flow and let P be an augmenting path in G_f. Then, the resulting f' is a flow and $\text{val}(f') = \text{val}(f) + \text{bottleneck}(G_f, P)$.

Algorithm 1: AUGMENT(f, c, P)

$\delta \leftarrow$ bottleneck capacity of augmenting path P;
foreach $e \in P$ **do**
  if $e \in E$ **then**
    $f(e) \leftarrow f(e) + \delta$; /* forward edge */
  else
    $f(e^R) \leftarrow f(e^R) - \delta$; /* reverse edge */
return f;

Algorithm 3: Ford-Fulkerson Algorithm

Input: $G = (V, E), c, s, t$
foreach $e \in E$ **do**
  $f(e) \leftarrow 0$;
$G_f \leftarrow$ residual graph;
while *there exists augmenting path* P **do**
  $f \leftarrow$ AUGMENT(f, c, P);
  update G_f;
return f;

Augmenting Path Theorem: Flow f is a max flow iff there are no augmenting paths.

Max-flow Min-cut Theorem: The value of the max flow is equal to the value of the min cut.

Proof:

1. There exists a $cut(A, B)$ such that $v(f) = cap(A, B)$
2. Flow f is a max flow.
3. There is no augmenting path

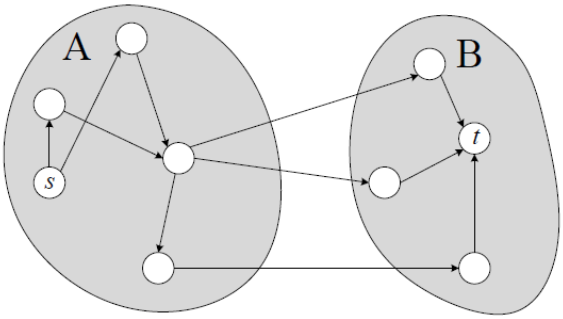
(1) $\Rightarrow$ (2): weak duality lemma

(2) $\Rightarrow$ (3): If there exists an augmenting path, then we can improve f

(3) $\Rightarrow$ (1): Let f be a flow with no augmenting paths. Let A be set of vertices reachable from s. $s \in A$ and $t \notin A$.

By definition of A, $s \in A$. By definition of f, $t \notin A$.

$$
\begin{aligned}
v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\
&= \sum_{e \text{ out of } A} c(e) - 0 \\
&= cap(A, B)
\end{aligned}
$$



Edge $e = (v, w)$ with $v \in B$, $w \in A$ must have $f(e) = 0$;
Edge $e = (v, w)$ with $v \in A$, $w \in B$ must have $f(e) = c(e)$.

note: edges with different directions have different constraints.

Invariant: Every flow value $f(e)$ and every residual capacity $c_f(e)$ remains an integer throughout the algorithm.

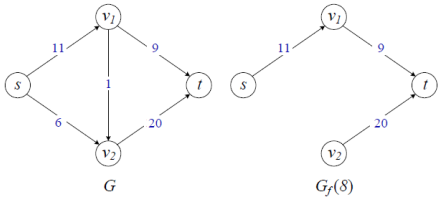
Time complexity: $v(f^*) \leq nC$, where f^* is the max flow. Each augmenting path increase f at least by 1. Each iteration takes $O(m)$ with BFS or DFS. Ford-Fulkerson Algorithms is $O(nmC)$.

Note: If the flow is real number instead of integer, then the algorithm may not terminate and the result may not converge to the max flow.

Scaling Max-Flow Algorithm

Intuition. Choosing path with highest bottleneck capacity increases flow by max possible amount.

- Don't worry about finding exact highest bottleneck path.
- Maintain scaling parameter Δ.
- Let $G_f(\Delta)$ be the subgraph of the residual graph consisting of only arcs with capacity at least Δ.



Algorithm 4: Scaling Max-Flow Algorithm

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

```
foreach e ∈ E do
    f(e) ← 0;
G_f ← residual graph;
Δ ← smallest power of 2 greater than or equal to C;
while Δ ≥ 1 do
    G_f(Δ) ← Δ-residual graph;
    while there exists augmenting path P in G_f(Δ) do
        f ← ARGUMENT(f, c, P);
        update G_f(Δ);
    Δ ← Δ/2;
return f;
```

note: $\Delta = 1 \Rightarrow G_f(\Delta) = G_f$

Lemma: Let f be the flow at the end of a Δ-scaling phase. Then the value of the maximum flow is at most $v(f) + m\Delta$

proof: We show that at the end of a Δ-phase, there exists a cut(A, B) such that $\text{cap}(A, B) \leq v(f) + m\Delta$.

Choose A to be the set of nodes reachable from s in $G_f(\Delta)$. $s \in A$ and $t \notin A$.

$$
\begin{aligned}
v(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\
&\geq \sum_{e \text{ out of } A} (c(e) - \Delta) - \sum_{e \text{ in to } A} \Delta \\
&= \sum_{e \text{ out of } A} c(e) - \sum_{e \text{ out of } A} \Delta - \sum_{e \text{ in to } A} \Delta \\
&\geq \text{cap}(A, B) - m\Delta
\end{aligned}
$$

Lemma: There are at most $2m$ augmentations per scaling phase.

Proof: Let f be the flow at the end of the previous scaling phase. We have $v(f^*) \leq v(f) + m(2\Delta)$. And each augmentation in a Δ-phase increases $v(f)$ by at least Δ

Time complexity: Scaling Max-Flow Algorithm is $O(m^2 \log C)$

Chap13: Turing Machine

Effective procedure

Basic Concepts

Intuitive Definition: An algorithm or effective procedure is a mechanical rule, or automatic method, or program for performing some mathematical operations.

counter example:
$g(n) = [\text{there is a run of exactly n consecutive 7's in the decimal expansion of } \pi]$. This procedure may never terminate. Thus, it is not a effective procedure.

other examples: Theorem Proving is in general not effective. But proof verification is effective.

Algorithm: An algorithm is a procedure that consists of a finite set of instructions which, given an input from some set of possible inputs, enables us to obtain an output through a systematic execution of the instructions that terminates in a finite number of steps.

Computable Function

When an algorithm or effective procedure is used to calculate the value of a numerical function, then the function is effectively calculable

Church-Turing Thesis: Computation Models can solve exactly the same class of problems.

Turing Machine

One-Tape Turing Machine

A Turing machine has five components.

1. A finite set $\{s_1, \ldots, s_n\} \cup \{\triangleright, \triangleleft\} \cup \{\square\}$ of symbols.
2. A tape consists of an infinite number of cells, each cell may store a symbol.
3. A reading head that scans and writes on the cells.
4. A finite set $\{q_S, q_1, \ldots, q_m, q_H\}$ of states.
5. A finite set of instructions (specification).

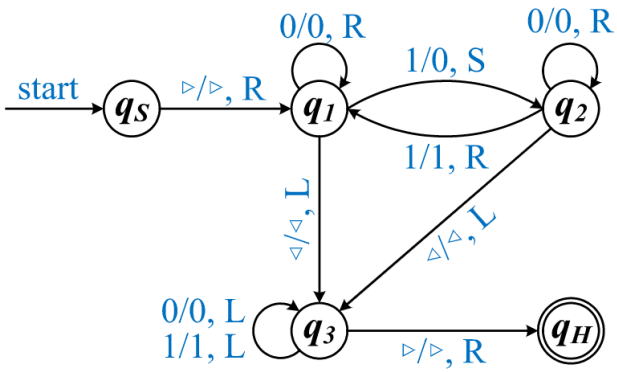
An **instruction** is of the form:

$$
\langle q_i, s_j \rangle \rightarrow \langle q_l, s_k, L/R/S \rangle
$$

which means when reads s_j with state q_i, the machine will turn to state q_1, replace s_j with s_k.

The direction can be L, R, or S, meaning move to left, right, or stay at the current position.

State Transition Diagram example:



Multi-Tape Turing Machine

A multi-tape TM is described by a tuple (Γ, Q, δ) containing

- A finite set Γ called alphabet, of symbols.
- A finite set Q of states. It contains a start state q_S and a halting state q_H.
- A transition function $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times L, S, R^k$.

note: first tape is read only.

TM Variation and TM-Computability

Let $\Sigma = \{a_1, \ldots, a_k\}$ be the set of symbols, called alphabet.

A string (word) from Σ is a sequence $a_{i_1}, \cdots, a_{i_n}$ of symbols from Σ.

Σ^* is the set of all words/strings from Σ.

example: if $\Sigma = \{a, b\}$, we have
$\Sigma^* = \{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, \ldots\}$

TM Variations

Larger Alphabets

If $f : \{0,1\}^* \to \{0,1\}^*$ is computable in time $T(n)$ by a TM M using the alphabet set Γ, then it is computable in time $4 \log |\Gamma| T(n)$ by a TM $\widetilde{M}$ using the alphabet $\{0, 1, \square, \triangleright\}$.

proof: We use $\{0,1\}$ to encode Γ with length $\log |\Gamma|$.

A state q in M is turned into a number of states in $\widetilde{M}$.

$$q, \langle q, \sigma_1^1, \ldots, \sigma_1^k \rangle, \cdots, \left\langle q, \sigma_{\log|\Gamma|}^1, \ldots, \sigma_{\log|\Gamma|}^k \right\rangle$$

To simulate one step of M, the machine $\widetilde{M}$ will

- use $\log |\Gamma|$ steps to read encoded symbol of Γ
- use its state register to store the symbols read
- use M 's transition function to compute the symbols M writes and M's new state
- store this information in state register

- use $\log |\Gamma|$ steps to write

Multi-Tape

If $f : \{0,1\}^* \to \{0,1\}^*$ is computable in time $T(n)$ by a TM M using k tapes, then it is computable in time $5kT(n)^2$ by a single-tape TM $\widetilde{M}$

proof: Interleave k tapes into one, $a_1 b_1 c_1 \cdots$. The first $n+1$ cells are reserved for the input. Every symbol a of M is turned into two symbols $a, \widehat{a}$ in $\widetilde{M}$, with $\widehat{a}$ used to indicate head position.

- The machine $\widetilde{M}$ places $\triangleright$ after the input string and then starts copying the input bits to the imaginary input tape. Whenever an input symbol is copied it is overwritten by $\triangleright$.
- $\widetilde{M}$ marks the $n+2$ cell, ..., the $n+k$ cell to indicate the initial head positions.
- $\widetilde{M}$ scans $kT(n)$ cells from the (n+1) cell to right, recording in the register the k symbols marked with the hat $\widehat{\ }$.
- use M's transition function to compute next symbols and state
- $\widetilde{M}$ scans $kT(n)$ cells from right to left to update using the transitions of M.

  For R, when it comes across current head, it moves right k cells, mark new head position, and then moves left to update.

Bidirectional Tape

bidirectional: infinite in both directions.

If $f : \{0,1\}^* \to \{0,1\}^*$ is computable in time $T(n)$ by a bidirectional TM M, then it is computable in time $4T(n)$ by a TM $\widetilde{M}$ with one-directional tape.

proof: Every state q of M is turned into $\bar{q}$ and $\underline{q}$

$\tilde{\text{M}}$ uses a larger alphabet to represent it on a standard tape:

Let H range over $\{L, S, R\}$ and let $-H$ be the reversed ones accordingly.

$\widetilde{M}$ contains the following transitions:

Reverse Direction

$\langle \bar{q}, (\triangleright, \triangleright) \rangle \to \langle \underline{q}, (\triangleright, \triangleright), R \rangle$

$\langle \underline{q}, (\triangleright, \triangleright) \rangle \to \langle \bar{q}, (\triangleright, \triangleright), R \rangle$

Respective Transition

$\langle \bar{q}, (a, b) \rangle \to \left\langle \overline{q'}, (a', b), H \right\rangle$ if $\langle q, a \rangle \to \langle q', a', H \rangle$

$\langle \underline{q}, (a, b) \rangle \to \left\langle \underline{q'}, (a, b'), -H \right\rangle$ if $\langle q, b \rangle \to \langle q', b', H \rangle$

Computable and Decidable

$M(a_1, \ldots, a_n) \downarrow b$: if $M(a_1, \ldots, a_n) \downarrow$ (halt) and $r = b$

M TM-computes f if, for all $a_1, \ldots, a_n, b \in \mathbb{N}$, $M(a_1, \ldots, a_n) \downarrow b$ iff $f(a_1, \ldots, a_n) = b$

Function f is **TM-computable** if there is a Turing Machine that TM-computes f.

Function Defined by Program: given any program, there must be a corresponding function.

$$f_P^{(n)}(a_1, \ldots, a_n) = \begin{cases} b & \text{if } P(a_1, \ldots, a_n) \downarrow b \\ \text{undefined} & \text{if } P(a_1, \ldots, a_n) \uparrow \end{cases}$$

Decidable: $P(x_1, x_2, \cdots, x_n)$ is a predicate. Let $c_p(\mathbf{x}) = [p(\mathbf{x}) \text{ holds}]$. P is decidable if c_p is computable.

On other Domains: Suppose D is an object domain. A coding of D is an explicit and effective injection $\alpha : D \to \mathbb{N}$. We say that an object $d \in D$ is coded by the natural number $\alpha(d)$.

A function $f : D \to D$ extends to a numeric function $f^* = \alpha \circ f \circ \alpha^{-1} : \mathbb{N} \to \mathbb{N}$. We say that f is computable if f^* is computable.

example: $\mathbb{Z} \to \mathbb{N}$

$$\alpha(n) = \begin{cases} 2n & \text{if } n \geq 0 \\ -2n - 1 & \text{if } n < 0 \end{cases}$$

Chap14: NP Reduction

Introduction

Definitions

Polynomial Time: $A(s)$ terminates in at most $p(|s|)$ "steps", where $p(\cdot)$ is some polynomial. ($|s|$ is length of s)

Decision Problem: X is a set of strings. Instance is string s. Algorithm A solves problem X : $A(s) =$ yes iff $s \in X$. (classification)

Search Problem: X is a set of strings. Instance is string s. Feasible solution s_x. Algorithm A searches the optimal solution for problem X: $A(s) = \min\{|s_x|\}$ or $\max\{|s_x|\}$

example: shortest path. Does there exist a path with weight $\leq k$? (whole graph is an instance s) vs Find shortest path.

P: Decision problems for which there is a poly-time algorithm.

NP(non-deterministic polynomial): Decision problems for which there exists a poly-time certifier ($C(s, t)$ is a poly-time algorithm and $|t| \leq p(|s|)$ for some polynomial $p(*)$)

Certifier doesn't determine whether $s \in X$ on its own. Rather, it checks a proposed proof t that $s \in X$.

Certifier: Algorithm $C(s, t)$ is a certifier for problem X: for every string s, $s \in X$ iff there exists a string such that $C(s, t) =$ yes.

example: composite number. $s = 437669$ and $t = 809$

EXP: Decision problems for which there is an exponential-time algorithm.

Claim: $P \subseteq NP \subseteq EXP$

proof: (1) $P \subseteq NP$, there's a poly time $A(s)$ solving X. Thus, let $t = \varepsilon$ and $C(s,t) = A(s)$. (2) $NP \subseteq EXP$, run $C(s,t)$ on all string t with $|t| \leq p(|s|)$

DTM(Deterministic Turing Machine): at most one instruction for each combination of symbol and state.

NTM: may have a set of specifications that prescribes more than one action for a given state.

P: can be solved by DTM in polynomial time

NP: Decision problems that can be solved by NTM in polynomial, or certificates of NP problem can be detected by DTM in polynomial.

Polynomial-Time Reductions

Reduction (Cook): Problem X polynomial reduces to problem Y if arbitrary instances of problem X can be solved using:

- Polynomial number of standard computational steps
- Polynomial number of calls to oracle that solves problem Y.

Remark: Y must be of polynomial size

Reduction (Karp): Problem X polynomial transforms to problem Y if given any input x to X, we can construct an input y such that x is a yes instance of X iff y is a yes instance of Y

Note: Polynomial transformation is polynomial reduction with just one call to oracle for Y.

Notation: $X \leq_p Y$. Y is more difficult than X.

- If $X \leq_P Y$ and Y can be solved in polynomial-time, then X can also be solved in polynomial time.
- If $X \leq_P Y$ and X cannot be solved in polynomial-time, then Y cannot be solved in polynomial time.
- If $X \leq_P Y$ and $Y \leq_P X$, we use notation $X \equiv_P Y$

Transitivity: If $X \leq_P Y$ and $Y \leq_P Z$, then $X \leq_P Z$

Self-Reducibility: search problem $\leq_P$ decision problem

example: To find min vertex cover

1. search cardinality of min vertex cover
2. enumerate every vertex to see if $G - \{v\}$ has a vertex cover with size $k-1$
3. if so, include v in the answer

Problems

- **SAT**: Given CNF formula Φ, does it have a satisfying truth assignment?
- **INDEPENDENT SET**: Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and no two vertices in S is adjacent?
- **VERTEX COVER**: Given a graph $G = (V, E)$ and an integer k, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge, at least one of its endpoints is in S.

- **SET COVER**: Given a set U of elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of U, and an integer k, does there exist a collection $\leq k$ of these sets whose union is equal to U.
- **HAM-CYCLE**: Given an undirected/directed graph, does there exist a simple cycle C that visits every node?

Basic Reduction Strategies

Reduction by simple equivalence

Claim: VERTEX-COVER $\equiv_P$ INDEPENDENT-SET

Proof: S is an independent set iff $V - S$ is a vertex cover.

$\Rightarrow$: Let S be an independent set, consider an arbitrary edge (u, v). S independent $\Rightarrow u \notin S$ or $v \notin S \Rightarrow u \in V - S$ or $v \in V - S$. Thus, $V - S$ covers (u, v).

$\Leftarrow$: Let S be any vertex cover. Consider two nodes $u \in V - S$ and $v \in V - S$. Note that $(u, v) \notin E$ since S is a vertex cover. Thus, no two nodes in $V - S$ adjacent $\Rightarrow V - S$ is independent set.

Reduction from special case to general case

Claim: VERTEX-COVER $\leq_P$ SET-COVER

Proof: Given a VERTEX-COVER instance $G = (V, E), k$, we construct a set cover instance with same size.

Create SET-COVER instance: $k = k, U = E, S_v = \{e \in E : e \text{ incident to } v\}$

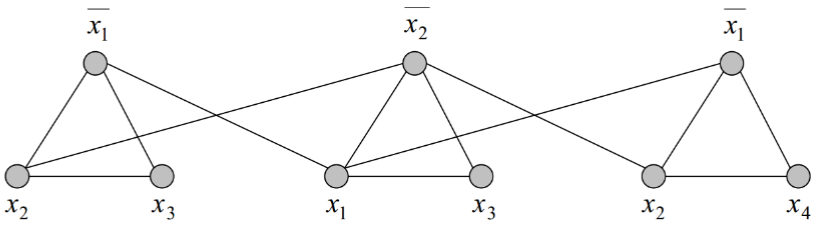
Reduction via gadgets

Claim: 3-SAT $\leq_P$ INDEPENDENT-SET.

Construction

- G contains 3 vertices for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.

G



k = 3

$$\Phi = \left(\overline{x_1} \vee x_2 \vee x_3\right) \wedge \left(x_1 \vee \overline{x_2} \vee x_3\right) \wedge \left(\overline{x_1} \vee x_2 \vee x_4\right)$$

G contains independent set of size $k = |\Phi|$ iff Φ is satisfiable.

Proof:

$\Rightarrow$: Let S be independent set of size k.

- S must contain exactly one vertex in each triangle.
- Set these literals to true and other variables in a consistent way.
- Truth assignment is consistent and all clauses are satisfied.

$\Leftarrow$: Given satisfying assignment, select one true literal from each triangle. This is an independent set of size k.

NP-Completeness

Definition

NP-Complete: A problem Y is NP-Complete if it is in NP, and for every problem X in NP, $X \leq_p Y$

Theorem: Suppose Y is an NP-complete problem. Then Y is solvable in poly-time iff $\mathrm{P} = \mathrm{NP}$.

CIRCUIT-SAT (natural NPC): Given a combinational circuit built out of AND, OR,and NOT gates, is there a way to set the circuit inputs so that the output is 1?

proof: (sketch) Any algorithm with fixed bit length of input and produces yes/no can be represented by a circuit.

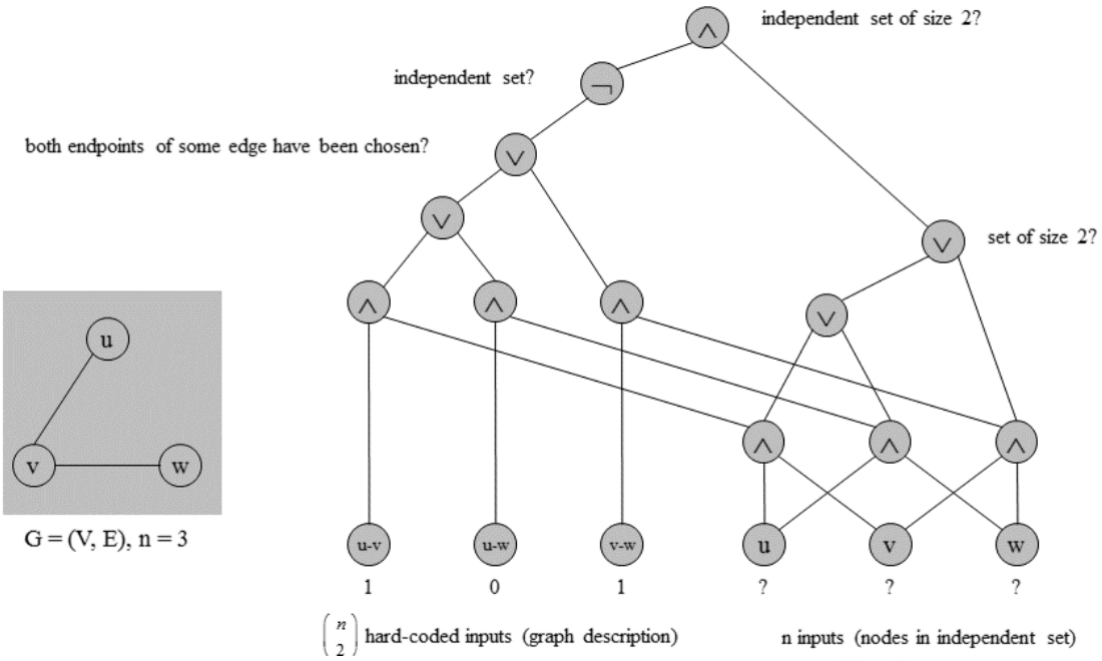
Consider some problem X in NP. It has a poly-time certifier $C(s, t)$. To determine whether s is in X is to determine whether there exists a certificate t of length $p(|s|)$ such that $C(s, t) =$ yes.

View $C(s, t)$ as an algorithm on $|s| + p(|s|)$ bits and convert it into a poly-size circuit K.

- first $|s|$ bits are hard-coded with s
- remaining $p(|s|)$ bits represent bits of t

Circuit K is satisfiable iff $C(s, t) =$ yes.

example: whether G has an independent set with size 2?



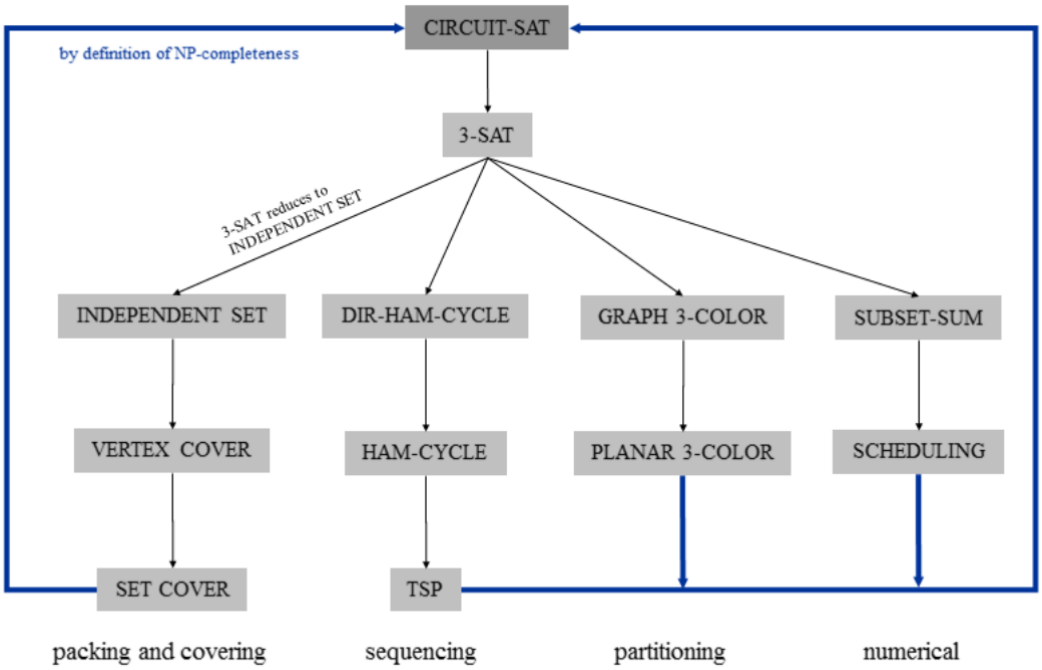
Recipe to establish NP-completeness

1. Show that Y is in NP
2. Choose an NP-complete problem X.
3. Prove that $X \leq_p Y$.

Proof: Let W be any problem in NP. Then $W \leq_p X \leq_p Y$.

Note: we need to first prove that Y is in NP!

More Examples



- Packing problems: SET-PACKING, INDEPENDENT SET.
- Covering problems: SET-COVER, VERTEX-COVER.
- Constraint satisfaction problems: SAT, 3-SAT.
- Sequencing problems: HAMILTONIAN-CYCLE, TSP.
- Partitioning problems: 3D-MATCHING, 3-COLOR.
- Numerical problems: SUBSET-SUM, KNAPSACK.

Note: select basic NPC problem similar to what we want to prove (i.e. prefer problems in the same category)

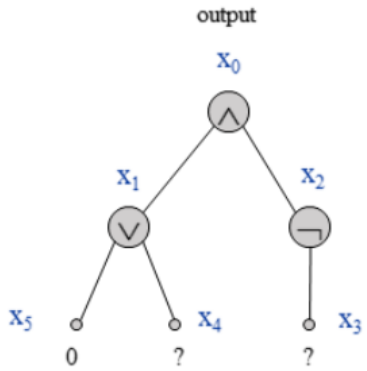
Practice: Most NP problems are either known to be in P or NP-complete.

Notable Exceptions: Factoring, Graph Isomorphism, Nash Equilibrium.

3-SAT is NPC

Claim: CIRCUIT-SAT $\leq_P$ 3-SAT, 3-SAT in NP

Proof: create a 3-SAT variable x_i for each circuit node.



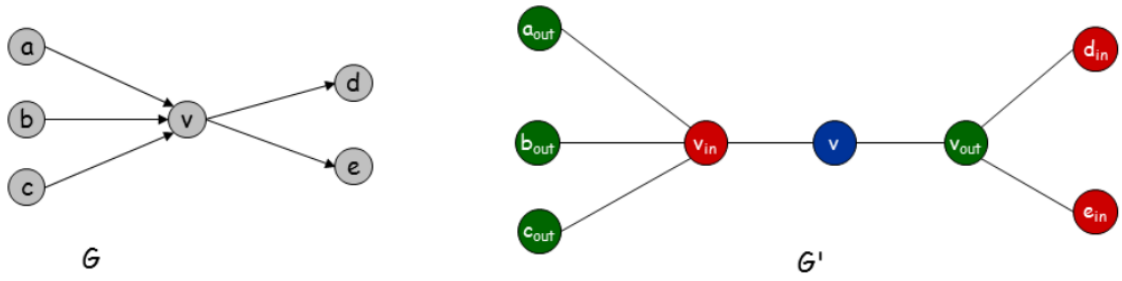
- NOT gate: $x_2 = \neg x_3 \Rightarrow$ add 2 clauses: $x_2 \vee x_3, \overline{x_2} \vee \overline{x_3}$
- Or gate: $x_1 = x_4 \vee x_5 \Rightarrow$ add 3 clauses: $x_1 \vee \overline{x_4}, x_1 \vee \overline{x_5}, \overline{x_1} \vee x_4 \vee x_5$

- AND gate: $x_0 = x_1 \wedge x_2 \Rightarrow$ add 3 clauses: $\overline{x_0} \vee x_1, \overline{x_0} \vee x_2, x_0 \vee \overline{x_1} \vee \overline{x_2}$
- Hard-coded input: $x_5 = 0 \Rightarrow$ add 1 clauses: $\overline{x_5}$
- Hard-coded output: $x_0 = 1 \Rightarrow$ add 1 clauses: x_0

Final: turn clauses with length< 3 into 3. e.g. $x \vee y \Rightarrow x \vee y \vee z, x \vee y \vee \bar{z}$

DIR-HAM-CYCLE $\leq_P$ HAM-CYCLE

proof: Given a directed G, construct undirected G' with $3n$ nodes.

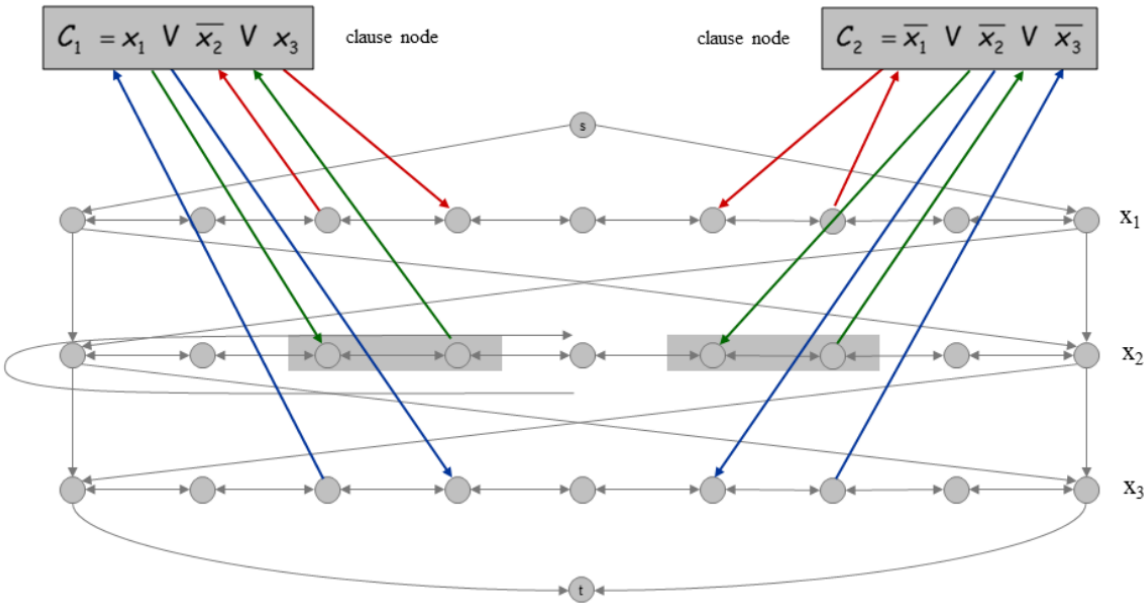


$\Leftarrow$: Suppose G has a directed Hamiltonian cycle Γ. Then G' has an undirected Hamiltonian cycle (same order).

$\Rightarrow$: Suppose G' has an undirected Hamiltonian cycle Γ'. Γ' can only be $B, G, R, B, G, R, \cdots$ or $B, R, G, B, R, C, \cdots$. Blue nodes in Γ' make up directed Hamiltonian cycle Γ in G

3-SAT $\leq_P$ DIR-HAM-CYCLE

proof: Traverse path i from left to right $\Leftrightarrow$ set variable $x_i = 1$. For each clause, add a node and 6 edges (notice the direction). Add an $t \rightarrow s$ edge to finish.



3-SAT $\leq_P$ LONGEST-PATH

LONGEST-PATH: Given a digraph $G = (V, E)$, does there exists a simple path with length at least k edges?

proof 1: same as DIR-HAM-CYCLE, but remove edge $t \rightarrow s$

proof 2: HAM-CYCLE $\leq_P$ LONGEST-PATH. Choose a vertex v and split it into v_{in} and v_{out} with same edges. Compute longest path start with v_{in}.

HAM-CYCLE $\leq_P$ TSP

TSP: Given a set of n cities and a pairwise distance function $d(u, v)$, is there a tour of length $\leq D$?

proof: (This d holds triangle-inequality)

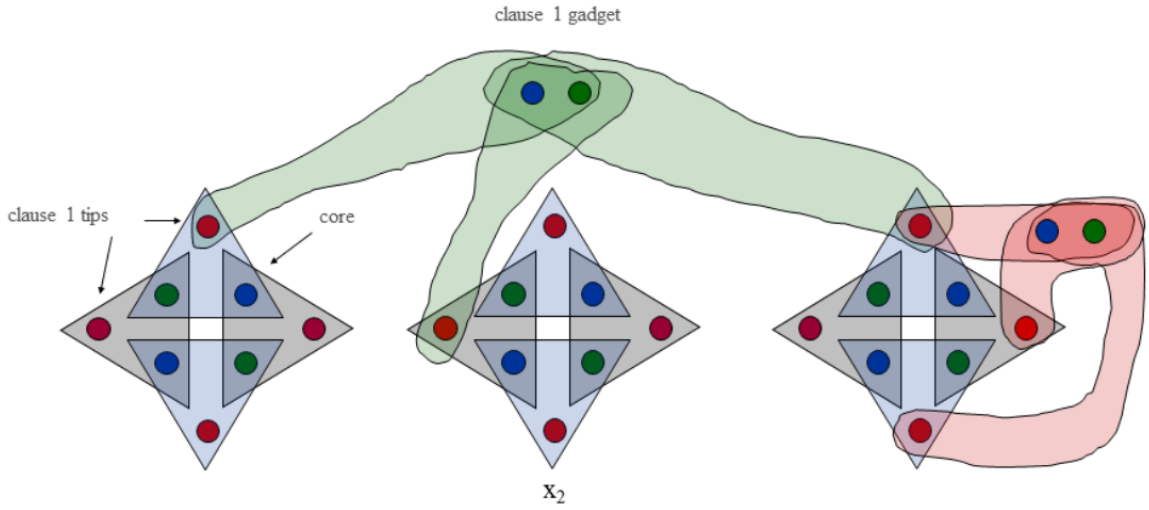
$$d(u,v) = \begin{cases} 1 & \text{if } (u,v) \in E \\ 2 & \text{if } (u,v) \notin E \end{cases}$$

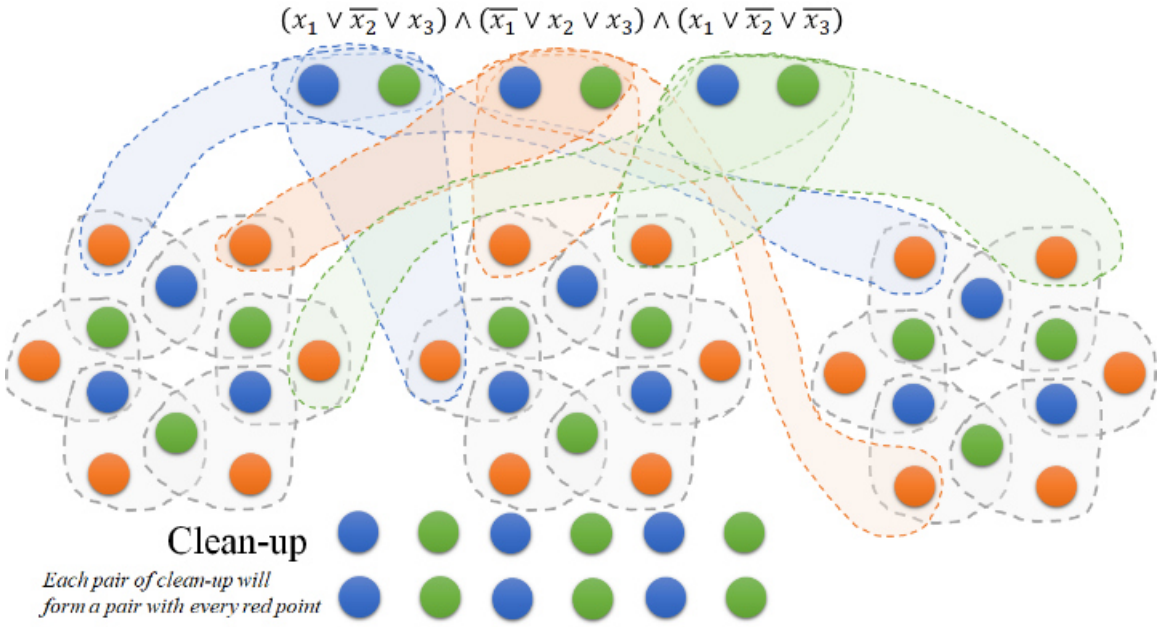
TSP instance has tour of length $\leq n$ iff G is Hamiltonian.

3-SAT $\leq_P$ 3D-MATCHING

3D-MATCHING: Given disjoint sets X, Y, Z, each of size n and a set $T = \{(x_1, y_1, z_1), \cdots\} \subseteq X \times Y \times Z$ of triples. Does there exist a subset with n triples in T such that each element of $X \cup Y \cup Z$ is in exactly one of these triples?

proof: Create gadget for each variable x_i with $2k$ core and tip (ensures at least k separate true/false assignment) elements, where k is the number of clauses. In gadget i, 3D-MATCHINIG must use either both grey triples or both blue ones. For each clause, create two elements and three triples, exactly one of these triples will be used. For each extra tip, add a cleanup gadget (clean gadget connects to every tip). $(n-1)k$ cleanup gadgets in total.

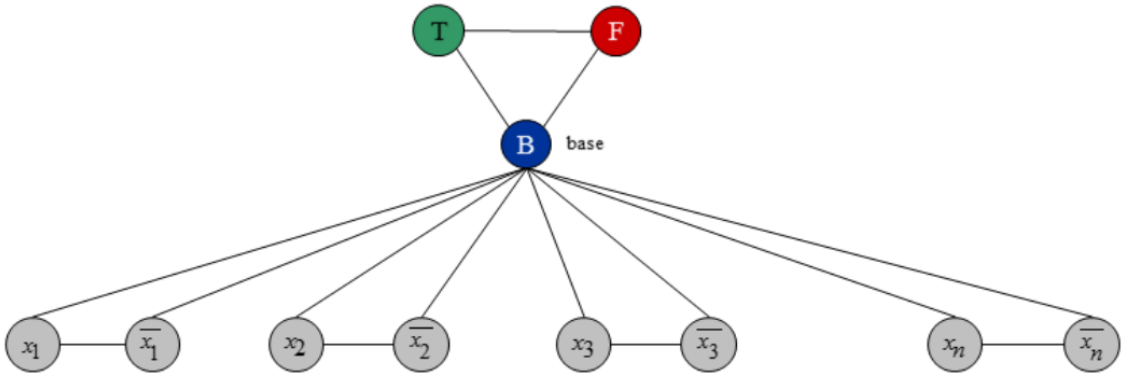




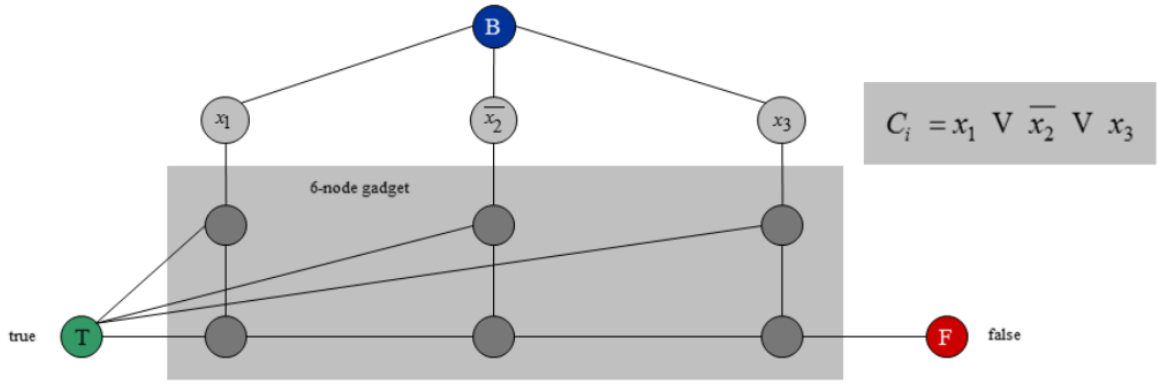
3-SAT $\leq_P$ 3-COLOR

3-COLOR: Given an undirected graph G, does there exist a way to color nodes with R, G and B, so that no adjacent nodes have same color?

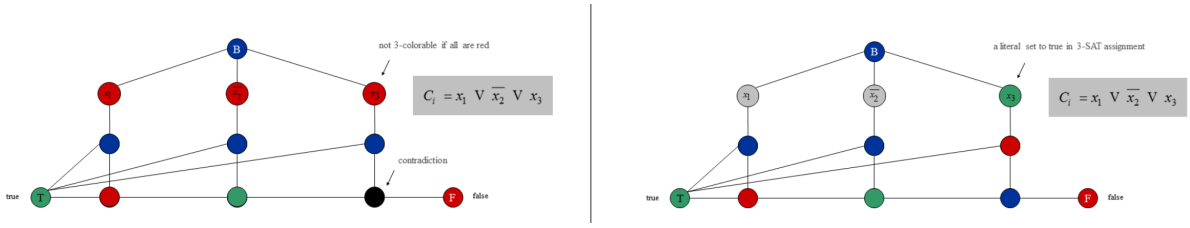
proof: Ensures each literal is T or F. Ensures a literal and its negation are opposites.



Change each clause into



Ensures at least one literal in each clause is T.



3-SAT $\leq_P$ SUBSET-SUM

SUBSET-SUM: Given natural numbers $w_1, w_2, \cdots, w_n$ and an integer W, is there a subset that adds up to exactly W?

proof: Given 3-SAT instance Φ with n variables and k clauses, form $2n + 2k$ decimal integers, each of $n + k$ digits, as illustrated below. No carries possible.

$$C_1 = \bar{x} \vee y \vee z$$
$$C_2 = x \vee \bar{y} \vee z$$
$$C_3 = \bar{x} \vee \bar{y} \vee \bar{z}$$

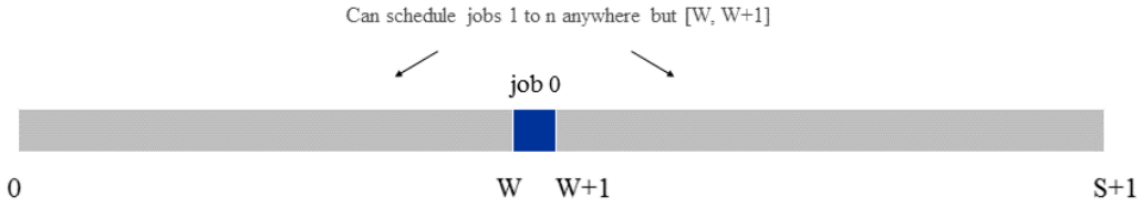
	x	y	z	C_1	C_2	C_3	
x	1	0	0	0	1	0	100,010
¬ x	1	0	0	1	0	1	100,101
y	0	1	0	1	0	0	10,100
¬ y	0	1	0	0	1	1	10,011
z	0	0	1	1	1	0	1,110
¬ z	0	0	1	0	0	1	1,001
dummies to get clause columns to sum to 4	0	0	0	1	0	0	100
	0	0	0	2	0	0	200
	0	0	0	0	1	0	10
	0	0	0	0	2	0	20
	0	0	0	0	0	1	1
	0	0	0	0	0	2	2
W	1	1	1	4	4	4	111,444

Note: In this case, 111333 is not correct, as dummies can add up to 3. Beside, it may need more than one digit to represent sum of C_i when dealing with k-SAT.

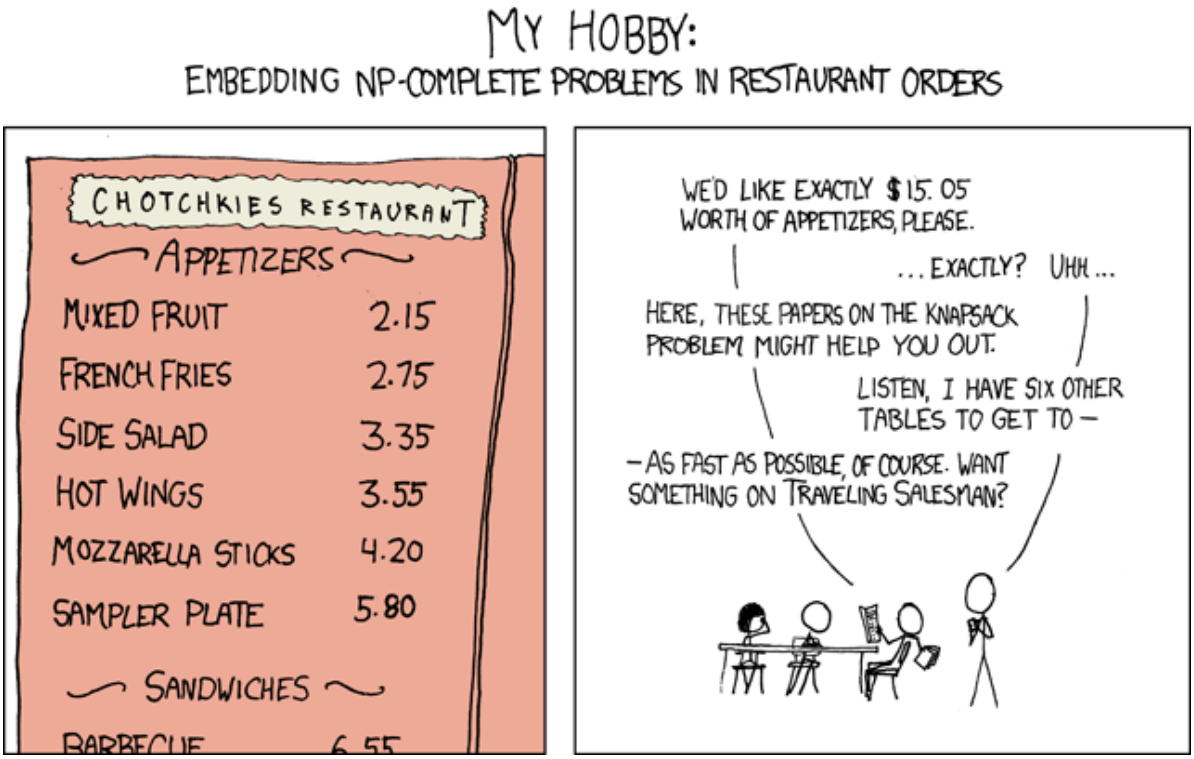
SUBSET-SUM $\leq_P$ SCHEDULE-RELEASE-TIMES

SCHEDULE-RELEASE-TIMES: Given a set of n jobs with processing time t_i, release time r_i, and deadline d_i, is it possible to schedule all jobs on a single machine such that job i is processed with a contiguous slot of t_i time units in the interval $[r_i, d_i]$?

proof: Given an instance of SUBSET-SUM $w_1, \ldots, w_n$, and target W, create n jobs with processing time $t_i = w_i$, release time $r_i = 0$, and no deadline $\left(d_i = 1 + \sum_j w_j\right)$. Create job 0 with $t_0 = 1$, release time $r_0 = W$, and deadline $d_0 = W + 1$. Job 0 can only be assign at $t = W$. Thus, it can divide jobs into two sections and the size of former one is exactly W.



The End



On seeing this comic, I suddenly recall a book named *What If*. Yes, they are from the same website.

Memories begin to come flooding on me. I remember there was a day when I stayed up to enjoy this book. It was my sophomore year at high school and I was supposed to study OI that night. Somehow, I skipped classes that night. By accident, I got this book late that night and was immersed in the fancy stories. Inspired by the fancy stories, I dreamt about my future that night, the bright but innocent one deep in the teenager's heart.

At this moment, I'm reviewing on Algorithm course. This course is not so good as some previous wonderful courses like Data Structure. And I have to struggle to survive this course, which I have never expected before.

After several years, I finally understand how the network flow algorithm works, what is NPC, and so on. It was as if he is reading the book on the balcony through dim lights from street lamps here and now. I want to tell him about what I have experienced these years. And, I have to admit what he had expected diverges a lot from what I have achieved today. I know I almost have no chance to ever go back to those delightful days with friends and without anything to worry about, instead of suffering pressure and panic from study and life. Despite this, I would also tell him to spare no effort and face bravely whatever will come. After all, it is when and where my story begins.