Part I

Lecture 1 - 28/09

Contents

1 Organisation

Mail Tatiana : `starikovskaya@di.ens.fr` Homeworks are 30% of the final grade, final (theory from lecture) Textbooks :

- *Introduction to Algorithms* - Cormen, Leiserson, Riverst, Stein
- *Algorithms on strings, trees, and sequences* - Gusfield
- *Approximation Algorithms* - Vazirani
- *Parametrized Algorithms* - Cygan, Fomin, Kowalik, Lokshtanov, Marx, Pilipczuk, Saurabh

2 Introduction

Algorithm take Inputs and give an output.

Open Problem 1 (Mersenne Prime)**.** *Find a new prime of form* $2^n - 1$

Algorithms do not depend on the language. Algorithms should be simple, fast to write and efficient. Word RAM model : Two Parts : one with a constant number of registers of w bits with direct access, and one with any number of registers, only with indirect access (pointers). Allows for elementary operations: basic arithmetic and bitwise operations on registers, conditionals, goto, copying registers, halt and malloc. To index the memory storing input of size n with n words, we need register length to verify $w \geq \log n$ Algorithms can always be rewritten using only elementary operations. Complexity :

- $Space(n)$ is the maximum number of memory words used for input of size n
- $Time(n)$ is the maximum number of *elementary* operations used for input of size n

Complexity Notations :

- $f \in \mathcal{O}(g)$ if $\exists n_0 \in \mathbb{N}, c \in \mathbb{R}_+,\ f(n) \leq c \cdot g(n),\ \forall n \geq n_0$
- $f \in \Omega(g)$ if $\exists n_0 \in \mathbb{N}, c \in \mathbb{R}_+,\ f(n) \geq c \cdot g(n),\ \forall n \geq n_0$
- $f \in \Theta(g)$ if $\exists n_0 \in \mathbb{N}, c_1, c_2 \in \mathbb{R}_+,\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n),\ \forall n \geq n_0$

3 Data Structures

3.1 Introduction

Way to store elements of a data base that is created to answer frequently asked queries using pre-processing. We care about space used, construction, query and update time. Can be viewed as an algorithm, which analysed on basics. Containers are basic Data Structures, maintaining the following operations :

1. Random Access : given i, access e_i
2. Access first/last element
3. Insert an element anywher
4. Delete any element

3.2 Array

An array is a pre-allocated contiguous memory area of a *fixed* size. It has random access in $\mathcal{O}(1)$, but doesn't allow insertion nor deletion.

Linear Search : given an integer x return 1 if $e_i = x$ else 0.

Algorithm 1 Linear Search in an Array.
Complexity : Time = $\mathcal{O}(n)$ | Space = $\mathcal{O}(n)$

3.3 Doubly Linked List

Memory area that does not have to be contiguous and consists of registers containing a value and two pointers to the previous and next elements. It has random access in $\mathcal{O}(n)$, access/insertion/deletion at head/tail in $\mathcal{O}(1)$.

Algorithm 2 Insertion in a Doubly Linked List
Complexity : $\mathcal{O}(1)$

```
**Input** $L, x$
$x.next \leftarrow L.head$
**if** $L.head \neq NIL$ **then**
    $L.head.prev \leftarrow x$
**end if**
$L.head \leftarrow x$
$x.prev = Nil$
```

3.4 Stack and Queue

Stack : Last-In-First-Out data structure, abstract data structure. Access/insertion/deletion to top in $\mathcal{O}(1)$.

Open Problem 2 (Optimum Stack Generation)**.** *Given a finite alphabet Σ and $X \in \Sigma^n$. Find a shortest sequence of stack operations push, pop, emit that prints out X. You must start and finish with an empty stack. Current best solution is in $\tilde{\mathcal{O}}(n^{2.8603})$.*

Queue : First-In-First-Out abstract data structure. Access to front, back in $\mathcal{O}(1)$, deletion and insertion at front and back in $\mathcal{O}(1)$.

4 Approaches to algorithm design

Solve small sub-problems to solve a large one.

4.1 Dynamic Programming

Break the problem into many closely related sub-problems, memorize the result of the sub-problems to avoid repeated computation

Examples :

Algorithm 3 Recursive Fibonacci Numbers
Complexity: Exponential

```
RFibo($n$) :
**Input** $n$
**if** $n \leq 1$ **then**
    **return** $n$
**end if**
**return** RFibo($n-1$) + RFibo($n-2$)
```

Levenshtein Distance between two strings can be computed in $\mathcal{O}(mn)$ instead of exponential time. Based on `https://arxiv.org/pdf/1412.0348.pdf`, this is the best one can do. RNA folding : retrieving the 3D shape of RNA based on

Algorithm 4 Dynamic Programming Fibonacci Numbers
Time = $\mathcal{O}(n)$ | Space = $\mathcal{O}(n)$

```
**Input**$n$
$Tab \leftarrow zeros(n)$                    ▷ $zeros(n)$ returns a $n$-array of zeros.
$Tab[0] \leftarrow 0$
$Tab[1] \leftarrow 1$
for $i \leftarrow 2$ to $n$ do
    $Tab[i] = Tab[i-1] + Tab[i-2]$
end for
return Tab[n]
```

their representation as strings. Currently, we know it is possible to find $\mathcal{O}(n^3)$, in $\tilde{\mathcal{O}}(n^{2.8606})$ and if *SETH* is true, there is no $\mathcal{O}(n^{\omega-\varepsilon})$. We know $\omega \in [2, 2.3703)$

Open Problem 3. *Is there a better Complexity for RNA folding ? What is the true value of* ω *?*

Knapsack problem : An optimization problem with bruteforce complexity $\mathcal{O}(2^n)$.

Algorithm 5 Knapsack : Dynamic Programming
Time = $\mathcal{O}(nW)$ | Space = $\mathcal{O}(nW)$

```
Input $W, w, v$                    ▷ Capacity, weight and values vectors.
$KP = zeros(n, W)$
for $i \leftarrow 0$ to $n$ do
    $KP[i, 0] = 0$
end for
for $w \leftarrow 0$ to $W$ do
    $KP[0, w] = 0$
end for
for $i \leftarrow 0$ to $n$ do
    for $w \leftarrow 0$ to $W$ do
        if $w < w_i$ then
            $KP[i, w] \leftarrow KP[i-1, w]$
        else
            $KP[i, w] = \max \begin{cases} KP[i-1, w] \\ KP[i-1, w-w_i] + v_i \end{cases}.$
        end if
    end for
end for
return $KP[n, W]$
```

4.2 Greedy Techniques

Problems solvable with the greedy technique form a subset of those solvable with DP. Problems must have the optimal substrcture property. Principle : choosing the best at the moment.

Example : The Fractional Knapsack Problem
Algorithm : Iteratively select the greatest value-per-weight ratio.

Theorem 4.2.1. *This algorithm returns the best solution, in time $\mathcal{O}(n \log n)$*

By contradiction. Suppose we have $\frac{v_1}{w_1} \geq \ldots \geq \frac{v_n}{w_n}$. Let $ALG = p = (p_1, \ldots, p_n)$ be the output by the algorithm and $OPT = q = (q_1, \ldots, q_n)$ be optimal.
Assume that $OPT \neq ALG$, let i be the smallesst index such $p_i \neq q_i$. There is $p_i > q_i$ by construct.
Thus, there exists $j > i$ such that $p_j < q_j$. We set $q' = (q_1^{'}, \ldots, q_n^{'}) = (q_1, \ldots, q_{i-1}, q_i + \varepsilon, q_{i+1}, \ldots, q_j - \varepsilon \frac{w_i}{w_j}, \ldots, q_n)$
$q^{'}$ is a feasible solution : $\sum_{i=1}^{n} q_i^{'} \cdot w_i = \sum_{i=1}^{n} q_i w_i \leq W$
Yet, $\sum_{i=1}^{n} q_i^{'} \cdot v_i > \sum_{i=1}^{n} q_i \cdot v_i$, ce qui contredit la ■

Part II

Lecture 2 - 5/10

Contents

5 Divide and Conquer

Divide a problem into smaller ones to solve those, then combine the solutions to get a solution of the bigger problem.
Example : *Merge Sort* : Its complexity verifies $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \mathcal{O}(n)$.
From that we will derive that $T(n) = \mathcal{O}(n \log n)$

6 Analysis of Recursive Algorithms

We have recurrences we want to solve. We have three methods :

6.1 Substitution Method

The method :

1. Guess the asymptotic of $T(n)$
2. Show the answer via induction

For *Merge Sort*: we guess $T(n) \leq c \cdot n \log_2 n, \forall n \geq 2$. We choose c that verifies that property until $n = 6$.
Substituting in the recurrence equation :

$$T(n) \leq cn \log_2 \frac{n}{2} + c \log_2 \frac{n}{2} + c\frac{n+2}{2} + a \cdot n = c\dot{n} \log_2 n + a \cdot n + c \cdot \log_2 n - c\frac{n}{2}$$

If we then choose c so that it is bigger than $20a$ we get :

$$T(n) \leq cn \log_2 n + a \cdot n - c \cdot n/20 \leq cn \log_2 n$$

6.2 Recursion-tree Method

1. Simplify the equation :
   - Delete floors and ceils
   - Suppose n is of a good form
2. Draw a tree, rooted with the added term and the recursive calls
3. Start again, and recursively fill the tree

We get a tree of depth $\log_k n$ if n is divided by k in recursive calls. We can now sum the values of the nodes, to get an approximation, and start verifying.

7 Master Theorem

7.1 The Theorem

Theorem 7.1.1 (Master Theorem). *If we have recurrence equation $T(n) = aT(n/b) + f(n)$ where $a \geq 1, b > 1$ are integers, $f(n)$ is asymptotically positive. Let $r = \log_b a$, we have :*

1. *If $f(n) = \mathcal{O}(n^{r-\varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^r)$*
2. *If $f(n) = \Theta(n^r)$ then $T(n) = \Theta(n^r \log n)$*
3. *If $f(n) = \Omega(n^{r+\varepsilon})$ for some $\varepsilon > 0$, and $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n (regularity condition) then $T(n) = \Theta(f(n))$.*

Remark 7.1.1.1. *The Master Theorem 7.1.1 does not cover all possible cases for $f(n)$. Example : $f(h) = h^r / \log h$*

Remark 7.1.1.2. *The Master Theorem 7.1.1 is sometimes called* Théorème sur les Récurrences de Partition

8 The Proof

Plan :

- Analyse the recurrence as if T is defined over reals (continuous version)
- Prove the discrete version

8.1 Continuous Master Theorem

Proof.

Lemma 8.1.1. *Define $T(n) = \begin{cases} \Theta(1) & \text{if } n \leq \hat{n} \\ aT(n/b) + f(n) & \text{if } n > \hat{n} \end{cases}$ Then*

$$T(n) = \Theta\left(n^r\right) + \sum_{k=0}^{\lceil \log_b(n/\hat{n}) \rceil - 1} a^k f(n/b^k)$$

Proof. In the Recursion-Tree, stopped when the argument of T is smaller than $\hat{n}$ which is when depth is $\lceil \log_b(n/\hat{n}) \rceil - 1$, we get :

$$\begin{aligned} T(n) &\leq \sum_{k=0}^{\lceil \log_b(n/\hat{n}) \rceil - 1} a^k f(n/b^k) + \Theta(a^{log_b(n/\hat{n})}) \\ &= \sum_{k=0}^{\lceil \log_b(n/\hat{n}) \rceil - 1} a^k f(n/b^k) + \Theta(a^{\log_b(n)}) \\ &= \sum_{k=0}^{\lceil \log_b(n/\hat{n}) \rceil - 1} a^k f(n/b^k) + \Theta(n^{\log_b(a)}) \end{aligned}$$

■

Back to the proof :

Lemma 8.1.2. *Define* $g(n) = \Theta(n^r) + \sum_{k=0}^{q} a^k f(n/b^k)$ *Then :*

1. *If* $f(n) = \mathcal{O}(n^{r-\varepsilon})$ *then* $g(n) = \Theta(n^r)$
2. *If* $f(n) = \Theta(n^r)$ *then* $g(n) = \Theta(n^r \log n)$
3. *If* $f(n) = \Omega(n^{r+\varepsilon})$ *and we have the regularity condition then* $g = \Theta(f)$

Proof.

1. Case 1 :

$$g(n) \begin{aligned} &= \Theta(n^r) + \sum_{k=0}^{q} a^k f(n/b^k) \\ &= \Theta(n^r) + \mathcal{O}\left(\sum_{k=0}^{q} a^k (n/b^k)^{r-\varepsilon}\right) \end{aligned}$$

   However :

$$\begin{aligned} \sum_{k=0}^{q} a^k (n/b^k)^{r-\varepsilon} &= n^{r-\varepsilon} \sum_{k=0}^{q} (ab^{\varepsilon}/b^r)^k \\ &= n^{r-\varepsilon} \sum_{k=0}^{\lceil \log_b(n/\hat{n}) \rceil - 1} (b^{\varepsilon})^k = \mathcal{O}(n^{r-\varepsilon}(n/\hat{n})^{epsilon}) \end{aligned}$$

   Thus : $g(n) = \Theta(n^r)$

2. Case 2 : We have :

$$\begin{aligned} g(n) &= \Theta(n^r) + \sum_{k=0}^{q} a^k f(n/b^k) \\ &= \Theta(n^r) + \Theta\left(\sum_{k=0}^{q} a^k (n/b^k)^r\right) \end{aligned}$$

   However :

$$\begin{aligned} \sum_{k=0}^{q} a^k (n/b^k)^r &= n^r \sum_{k=0}^{q} (a/b^r)^k \\ &= n^r \sum_{k=0}^{\lceil \log_b(n/\hat{n}) \rceil - 1} 1 = n^r \Theta(\log_b n/\hat{n}) \end{aligned}$$

3. Case 3 : By induction on k : $a^k f(n/b^k) \leq c^k f(n)$. Thus :

$$\sum_{k=0}^{q} a^k f(n/b^k) \leq \sum_{k=0}^{q} c^k f(n) = f(n) \sum_{k=0}^{q} c^k = \Theta(f(n))$$

∎

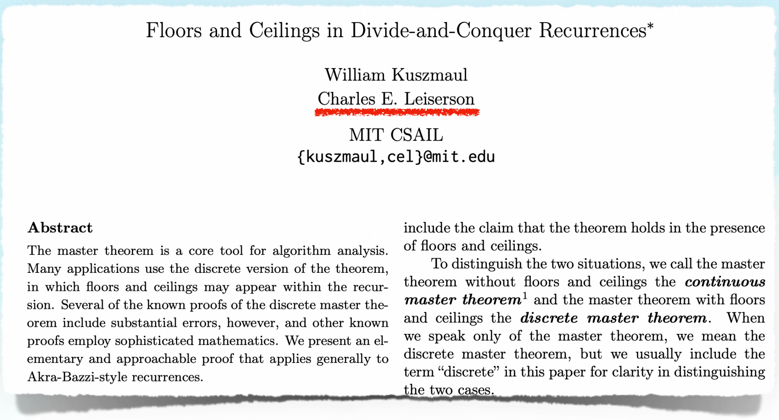
We thus have proved the continuous Master Theorem. ∎

8.2 Discrete Master Theorem

We have now showed the continuous Master Theorem, following William Kuszmaul, Charles E. Leiserson, *Floors and Ceilings in Divide-and-Conquer Recurrences*, Symposium on Simplicity in Algorithms 2021.

Proof. See slides below

Why not to follow CLRS textbook?


Floors and Ceilings in Divide-and-Conquer Recurrences*

William Kuszmaul
Charles E. Leiserson

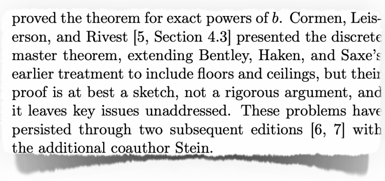
MIT CSAIL
{kuszmaul,cel}@mit.edu

Abstract

The master theorem is a core tool for algorithm analysis. Many applications use the discrete version of the theorem, in which floors and ceilings may appear within the recursion. Several of the known proofs of the discrete master theorem include substantial errors, however, and other known proofs employ sophisticated mathematics. We present an elementary and approachable proof that applies generally to Akra-Bazzi-style recurrences.

include the claim that the theorem holds in the presence of floors and ceilings.

To distinguish the two situations, we call the master theorem without floors and ceilings the ***continuous master theorem***[1] and the master theorem with floors and ceilings the ***discrete master theorem***. When we speak only of the master theorem, we mean the discrete master theorem, but we usually include the term "discrete" in this paper for clarity in distinguishing the two cases.


proved the theorem for exact powers of b. Cormen, Leiserson, and Rivest [5, Section 4.3] presented the discrete master theorem, extending Bentley, Haken, and Saxe's earlier treatment to include floors and ceilings, but their proof is at best a sketch, not a rigorous argument, and it leaves key issues unaddressed. These problems have persisted through two subsequent editions [6, 7] with the additional coauthor Stein.

Why not to follow CLRS textbook?

- Aho, Hopcroft, Ullman offered one of the first treatments of divide-and-conquer recurrences, giving three cases for recurrences of the form T(n) = aT (n/b) + cn (1974)
- Bentley, Haken, and Saxe introduced the master theorem in modern form, but proved it for $n = b^k$ only (1980)
- CLRS extended the proof to the discrete version, but gave only a sketch of the proof (1990)
- Akra and Bazzi considered $T(n) = \sum_{i=1}^{t} a_i T(n/b_i) + f(n)$ (1998)
- Leighton simplifies the proof of Akra and Bazzi and extends is to the discrete version (1996)
- Campbell spots several flows in the proof of Leighton and devotes more than 300 pages to carefully correct the issues (2020)
- More generalizations by Drmota and Szpankowski(2013), Roura (2001), Yap (2011)

Definitions

Discrete recurrences

$$T(n) = f(n) + \sum_{i \in S} a_i T(\lfloor n/b_i \rfloor) + \sum_{i \notin S} a_i T(\lceil n/b_i \rceil)$$

$a_i \in \mathbb{R}^+, b_i \in \mathbb{R}^+, n \geq \hat{n}$

For $1 \leq n < \hat{n}$, there exist c_1, c_2: $c_1 \leq T(n) \leq c_2$

Polynomial-growth condition

$\exists \hat{n} > 0$ such that $\forall \Phi \geq 1 \, \exists d > 1 : d^{-1} f(n) \leq f(\varphi n) \leq d f(n)$

for all $1 \leq \varphi \leq \Phi$ and $n \geq \hat{n}$

6 technical slides ahead!

KEEP CALM AND CARRY ON

Lemma 1. For $\beta > 1, n \in \mathbb{N}$ let $L = \Pi_{i=1}^{n}(1 - \frac{1}{\beta^i + 1})^2$, $U = \Pi_{i=1}^{n}(1 + \frac{1}{\beta^i - 1})^2$

We have $L = \Omega(1)$ and $U = O(1)$.

Proof.

$$\beta > 1 \Rightarrow \frac{1}{\beta^i} < \frac{1}{\beta^i - 1} \Rightarrow 1/L = \Pi_{1=1}^{n}(1 + \frac{1}{\beta^i})^2 < \Pi_{i=1}^{n}(1 + \frac{1}{\beta^i - 1})^2 = U$$

$$U = \Pi_{i=1}^{n}(1 + \frac{1}{\beta^i - 1})^2 \le \Pi_{i=1}^{\infty}(1 + \frac{1}{\beta^i - 1})^2 \le \Pi_{i=1}^{\infty}(e^{1/(\beta^i - 1)})^2 =$$

(Here we use $1 + 1/x \le e^{1/x}$ for $x \neq 0$)

$$= \exp(\sum_{i=1}^{\infty} \frac{2}{\beta^i - 1}) \le \exp(\sum_{i=1}^{\infty} \frac{4}{\beta^i}) + O(1) = O(1)$$

Lemma 2. Let $\beta > 1; \beta_i \ge \beta, 1 \le i \le k; B := \Pi_{i=1}^{k}\beta_i$

There exists $c = c(\beta) > 0$ such that for all $n_1, n_2, \ldots, n_k$ where $n_i > \max(\beta, 1 + 1/(\sqrt{\beta} - 1))$ and $\lfloor n_{i-1}/\beta_i \rfloor \le n_i \le \lceil n_{i-1}/\beta_i \rceil$, we have $c^{-1/4}(n_0/B) \le n_k \le c^{1/4}(n_0/B)$.

Proof. Let $\rho_i := \frac{n_i}{n_{i-1}/\beta_i}$.

$$(n_0/B)\Pi_{i=1}^{k}\rho_i = \frac{n_0\Pi_{i=1}^{k}\rho_i}{\Pi_{i=1}^{k}\beta_i} = n_0\Pi_{i=1}^{k}\frac{\rho_i}{\beta_i} = n_0\Pi_{i=1}^{k}\frac{n_i}{n_{i-1}} = n_k$$

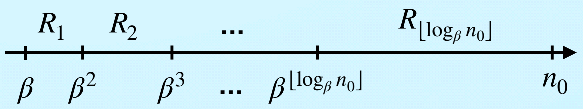
It is enough to show that $c^{-1/4} \le \Pi_{i=1}^{k}\rho_i \le c^{1/4}$ for some $c = c(\beta)$

$$n_{i-1}/\beta_i - 1 \le \lfloor n_{i-1}/\beta_i \rfloor \le n_i \le \lceil n_{i-1}/\beta_i \rceil \le n_{i-1}/\beta_i + 1 \Rightarrow$$

$$n_i - 1 \le n_{i-1}/\beta_i \le n_i + 1 \Rightarrow \boxed{\underbrace{\frac{n_i}{n_i + 1}}_{1 - \frac{1}{n_i + 1}} \le \rho_i \le \underbrace{\frac{n_i}{n_i - 1}}_{1 + \frac{1}{n_i - 1}} \ (*)}$$

Proof of Lemma 2 (continued).

$$\boxed{\underbrace{\frac{n_i}{n_i+1}}_{1-\frac{1}{n_i+1}} \le \rho_i \le \underbrace{\frac{n_i}{n_i-1}}_{1+\frac{1}{n_i-1}} \; (*)}$$



R_1
R_2
...
$R_{\lfloor \log_\beta n_0 \rfloor}$
β
β^2
β^3
...
$\beta^{\lfloor \log_\beta n_0 \rfloor}$
n_0

From $(*)$: $\rho_i \le 1 + \dfrac{1}{n_i - 1} \le 1 + \dfrac{1}{1/(\sqrt{\beta}-1)} = \sqrt{\beta}$

$n_{i+2} = \dfrac{n_i \rho_{i+1} \rho_{i+2}}{\beta_{i+1}\beta_{i+2}} \le n_i/\beta \Rightarrow$ every range R_j contains at most two n_i's

From $(*)$ again: $n_i \in R_j \Rightarrow 1 - \dfrac{1}{\beta^j + 1} \le \rho_i \le 1 + \dfrac{1}{\beta^j - 1} (n_i > \beta^j)$

Therefore, $\Pi_{i=1}^k \rho_i = \Pi_{j=1}^{\lfloor \log_\beta n_0 \rfloor} (\Pi_{n_i \in R_j} \rho_i) \le \Pi_{j=1}^{\lfloor \log_\beta n_0 \rfloor} (1 + \dfrac{1}{\beta^j - 1})^2 \le c^{1/4}$ **(Lemma 1)**

$$\Pi_{i=1}^k \rho_i \ge \Pi_{j=1}^{\lfloor \log_\beta n_0 \rfloor} (1 - \frac{1}{\beta^j + 1})^2 \ge c^{-1/4} \textbf{ (Lemma 1)}$$

Lemma 3. $\beta_{\min}, \beta_{\max} > 1$. Assume that for all $1 \le i \le k$, $\beta_{\min} \le \beta_i \le \beta_{\max}$, and let $B = \Pi_i \beta_i$.

There exists $c = c(\beta_{\min}, \beta_{\max})$ such that for any $n_1, n_2, \ldots, n_k$ with $n_0 \ge cB$ and $\lfloor n_{i-1}/\beta_i \rfloor \le n_i \le \lceil n_{i-1}/\beta_i \rceil$, we have $c^{-1}(n_0/B) \le n_k \le c(n_0/B)$.

Proof.

Let $c = c(\beta_{\min})$ be the constant from Lemma 3. W.l.o.g. $\sqrt{c} > \max\{\dfrac{1}{\sqrt{\beta_{\min}} - 1} + 1, \beta_{\min}\}$ $(*)$ and $c^{1/4} > 2\beta_i$

If $n_j \ge \sqrt{c}$ for all j, then **Lemma 3** follows from **Lemma 2** and $(*)$. Let j be the smallest value such that $n_j \le \sqrt{c}$. We have $j \ge 1$ as $n_0 \ge cB \ge \sqrt{c}$.

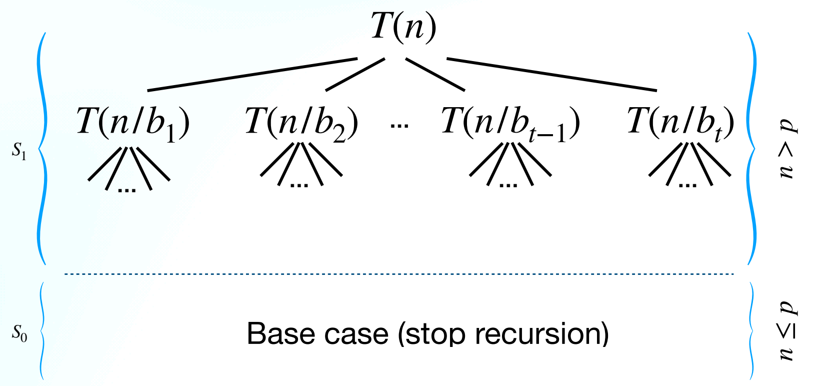
- If $j = 1$, then $n_{j-1} = n_0 \ge c^{-1/4}(n_0/B)$ (trivial).
- If $j > 1$, we apply **Lemma 2** to $\beta_1, \beta_2, \ldots, \beta_{j-1}$ and $n_0, n_1, \ldots, n_{j-1}$ and $\beta = \beta_{\min}$ (all conditions are satisfied) to obtain that $n_{j-1} \ge c^{-1/4}(n_0/B)$

In both cases, $n_{j-1} \ge c^{-1/4}(n_0/B) \ge c^{3/4}$. Therefore, $n_j \ge \lfloor \underbrace{n_{j-1}}_{\ge c^{1/4} > 2\beta_j} / \beta_j \rfloor \ge n_{j-1}/(2\beta_j) > n_{j-1}/c^{1/4} \ge \sqrt{c}$

Lemma 4. Let $a_1, a_2, \ldots, a_t > 0$ and $b_1, b_2, \ldots, b_t > 1$ be constants, $f(n) : \mathbb{R}^+ \to \mathbb{R}^+$ which satisfies the polynomial-growth condition. Consider $T(n) = f(n) + \sum_{i=1}^{t} a_i T(n/b_i)$ defined for $n \in \mathbb{R}^+$ ($*$). Assume that $T'(n)$ defined on $\mathbb{N}$ also satisfies ($*$), but each n/b_i is replaced with $\lfloor n/b_i \rfloor$ or $\lceil n/b_i \rceil$. Then $T'(n) = \Theta(T(n))$.

Proof.

Let c be the constant from Lemma 3 for $\beta_{\min} = \min_i b_i$ and $\beta_{\max} = \max_i b_i$. Let $\hat{n}$ be a sufficiently large constant. Define $p := \max\{\hat{n}, c \cdot \max_i b_i\}$. For $T(n)$, the base case is $n \le p$.



Subproblem q at level j:
$q = < q(1), q(2), \ldots, q(j) >$
$q(i) \in \{1,2,\ldots,t\}, |q| = j$
$q(i)$ - choice of the recursion brunch at level i
$n_q = n/\Pi_{i=1}^{|q|} b_{q(i)}$

Proof.

$$T(n) = \sum_{q \in S_1} f(n_q)\Pi_{i=1}^{|q|} a_{q(i)} + \sum_{q \in S_0} T(n_q)\Pi_{i=1}^{|q|} a_{q(i)} + f(n) = \sum_{q \in S_1} f(n_q)\Pi_{i=1}^{|q|} a_{q(i)} + \Theta(\sum_{q \in S_0} \Pi_{i=1}^{|q|} a_{q(i)}) + f(n)$$

When computing $T'(n)$ for a subproblem q:

$$\lfloor \frac{n'_{<q(1),q(2),\ldots,q(j-1)>}}{q(j)} \rfloor \le n'_q \le \lceil \frac{n'_{<q(1),q(2),\ldots,q(j-1)>}}{q(j)} \rceil$$

$$T'(n) = \sum_{q \in S_1} f(n'_q)\Pi_{i=1}^{|q|} a_{q(i)} + \sum_{q \in S_0} T'(n'_q)\Pi_{i=1}^{|q|} a_{q(i)} + f(n) \ (*)$$

As $n_q > p$ for $q \in S_1$, $n_q > p/\max_i b_i \ge c$ for all $q \in S$. By Lemma 3 with $\beta_i = b_{q(i)}$, for all q we have $n'_q = \Theta(n_q)$. It follows that $\exists \Phi > 1$ such that $n'_q \in [\Phi^{-1} n_q, \Phi n_q]$. Therefore, $n'_q \ge n_q/\Phi \ge \hat{n}/\Phi$ and we can choose $\hat{n}$ so that ($*$) is defined correctly.

By the polynomial-growth condition, $f(n'_q) = \Theta(f(n_q))$ for all $q \in S$. For $q \in S_0$, $n'_q = \Theta(1)$ and therefore $T'(n'_q) = \Theta(1)$. It follows:

$$T'(n') = \sum_{q \in S_1} \Theta(f(n_q))\Pi_{i=1}^{|q|} a_{q(i)} + \Theta(\sum_{q \in S_0} \Pi_{i=1}^{|q|} a_{q(i)}) + f(n) = \Theta(T(n))$$

Proof.

$$T(n) = \sum_{q \in S_1} f(n_q)\Pi_{i=1}^{|q|} a_{q(i)} + \sum_{q \in S_0} T(n_q)\Pi_{i=1}^{|q|} a_{q(i)} + f(n) = \sum_{q \in S_1} f(n_q)\Pi_{i=1}^{|q|} a_{q(i)} + \Theta(\sum_{q \in S_0} \Pi_{i=1}^{|q|} a_{q(i)}) + f(n)$$

When computing $T'(n)$ for a subproblem q:

$$\lfloor \frac{n'_{<q(1),q(2),\ldots,q(j-1)>}}{q(j)} \rfloor \le n'_q \le \lceil \frac{n'_{<q(1),q(2),\ldots,q(j-1)>}}{q(j)} \rceil$$

$$T'(n) = \sum_{q \in S_1} f(n'_q)\Pi_{i=1}^{|q|} a_{q(i)} + \sum_{q \in S_0} T'(n'_q)\Pi_{i=1}^{|q|} a_{q(i)} + f(n) \; (*)$$

As $n_q > p$ for $q \in S_1$, $n_q > p/\max_i b_i \ge c$ for all $q \in S$. By Lemma 3 with $\beta_i = b_{q(i)}$, for all q we have $n'_q = \Theta(n_q)$, and hence $\exists \Phi > 1$ such that $n'_q \in [\Phi^{-1} n_q, \Phi n_q]$. Therefore, $n'_q \ge n_q/\Phi \ge \hat{n}/\Phi$ and we can choose $\hat{n}$ so that $(*)$ is defined correctly.

By the polynomial-growth condition, $f(n'_q) = \Theta(f(n_q))$ for all $q \in S$. For $q \in S_0$, $n'_q = \Theta(1)$ and therefore $T'(n'_q) = \Theta(1)$. It follows:

$$T'(n') = \sum_{q \in S_1} \Theta(f(n_q))\Pi_{i=1}^{|q|} a_{q(i)} + \Theta(\sum_{q \in S_0} \Pi_{i=1}^{|q|} a_{q(i)}) + f(n) = \Theta(T(n))$$

Discrete Master theorem

$T(n) = a_1 T(\lfloor n/b \rfloor) + a_2 T(\lceil n/b \rceil) + f(n)$, where
$a := a_1 + a_2 \ge 1$, $b > 1$, $f(n)$ - asymptotically positive.

Define $r := \log_b a$.

Case 1. If $f(n) = O(n^{r-\varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^r)$.

Case 2. If $f(n) = \Theta(n^r)$, then $T(n) = \Theta(n^r \log n)$.

Case 3. If $f(n) = \Omega(n^{r+\varepsilon})$ for some $\varepsilon > 0$, and if $a_1 f(\lfloor n/b \rfloor) + a_2 f(\lceil n/b \rceil) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Discrete Master theorem

Case 1.

Fact. Replacing $f(n)$ with a function $f'(n)$ satisfying $f'(n) \leq f(n)$ (resp. $f'(n) \geq f(n)$) for all n in the domain of f does not increase (resp. decrease) $T(n)$.

Let $f(n) = O(n^c)$ for $c < \log_b a$. Then as a "bigger" function consider $f'(n) = r(n^c + 1)$ for r big enough. By Lemma 4 and the continuos Master theorem, $T(n) = O(n^{\log_b a})$.

As a "smaller" function, consider $f'(n) = 0$. By Lemma 4 and the continuos Master theorem, $T(n) = \Omega(n^{\log_b a})$.

Exercise. Both bigger and smaller functions satisfy the polynomial growth condition.

Case 2. Analogous.

Discrete Master theorem

Case 3.

$T(n) \geq f(n)$ and hence $T(n) = \Omega(f(n))$. It remains to show that $T(n) = O(f(n))$.

Regularity condition: $a_1 f(\lfloor n/b \rfloor) + a_2 f(\lceil n/b \rceil) \leq cf(n)$ for some $c < 1$ and all $n \geq p$.

For all $n < p$, there exists $s \geq 1$: $T(n) \leq sf(n)$. We show by induction that for all $n \in \mathbb{N}$, $T(n) \leq qf(n)$ for $q = s/(1-c)$.

- Base case: $n < p$ - by the choice of s
- Suppose that $n \geq p$ and the claim holds for all smaller n

$$T(n) = a_1 T(\lfloor n/b \rfloor) + a_2 T(\lceil n/b \rceil) + f(n) \leq a_1 q f(\lfloor n/b \rfloor) + a_2 q f(\lceil n/b \rceil) + f(n) \leq$$

$$\leq qcf(n) + f(n) = (\frac{sc}{1-c} + 1) f(n) = \frac{s - \overbrace{(1-c)s + 1 - c}^{\leq 0}}{1-c} f(n) \leq qf(n)$$

■

Remark 8.2.0.1 (Remarks on the Proof).

- *Lemmas 1 to 3 serve to show that the argument does not go too far when it is rounded up or down.*
- *Slide 36 Last Line :* $\frac{2}{\beta^i - 1} < \frac{4}{\beta^i}$ *for* $i \geq i_0$. *Thus :* $\sum_{i=1}^{\infty} \frac{2}{\beta^i - 1} < \sum_{i=1}^{\infty} \frac{4}{\beta^i} + \sum_{i=0}^{i_0} \frac{2}{\beta^i - 1}$ *and that last sum is* $\mathcal{O}(1)$
- *Slide 37 Line 3 : The first inequalities comes from the Recursion-Tree, so that we can ensure the argument does not deviate to much, by the second inequalities.*

8.3 Use Cases

Using the Master Theorem we can show the complexity of many algorithms :

1. Merge Sort Complexity : $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \mathcal{O}(n) = \Theta(n \log n)$
2. Strassen's Algorithm for Matrix Multiplication : $T(n) = 7T(n/2) + \Theta(n^2) \Rightarrow T(n) = \mathcal{O}(n^{\log_2 7}) = \mathcal{O}(n^{2.8074})$

9 Fast Multiplication of Polynomials

The sum of two degree n polynomials can be done in $\mathcal{O}(n)$, Horner's rule for evaluation produces $\mathcal{O}(n)$ complexity. The naïve product can be done in $\mathcal{O}(n^2)$
Remembering Lagrange's Theorem on Polynomials (or Vandermonde's Determinant, or anything really), degree n polynomials are entirely represented by their point-value reprensentation over n distinct points (x_i, y_i). Then, by converting the coefficient reprensentation to point-value representation, then by point-wise multiplicating the polynomials, then by going back to the coefficient representation, we can have a better algorithm.

9.1 Point-Value Multiplication

It is easily done in $\mathcal{O}(n)$ if both polynomials are represented over the same axis.

9.2 Coefficient to Point-Value Conversion - Fast Fourier Transform

For $P = \sum_{i=0}^{n-1} a_i x^i$, we define :

$$\begin{aligned} P_{odd}(x) &= a_{n-1}x^{n/2-1} + a_{n-3}x^{n/2-3} + \ldots + a_1 x \\ P_{even}(x) &= a_{n-2}x^{n/2-2} + a_{n-4}x^{n/2-4} + \ldots + a_2 x^{2/2} + a_0 \end{aligned}$$

1. We have : $P = xP_{odd}(x^2) + P_{even}(x^2)$
2. We evaluate P_{odd}, P_{even} at $(\omega_n^i)^2$ recursively by the halving property.
3. We combine the result.

9.3 Point-Value to Coefficient Conversion - Inverse Fast Fourier Transform

Theorem 9.3.1. $V_n^{-1}[i,j] = \omega_n^{-ij}/n$

Part III
Lecture 3 - 12/1

Contents

Answering queries of appartenance on a set, maintaining a dictionary. Python dictionaries do not have upper bound guarantees, we shouldn't use them.
Suppose we have a set S over a universe U. We dnote by n the number of keys, m the size of the hash table

10 Naïve Array-based implementation

Here we assume that no two objects have equal keys. We store 1 in a bitvector of length $|U|$ at each position i such that i is in S.
This takes $\mathcal{O}(|U|)$ space, for $\mathcal{O}(1)$ search time, and $\mathcal{O}(1)$ modification time.

11 Chained Hash Tables

We give ourselves a function $h : U \to [1, m]$. We send the keys to their image by h in a table. However, since we have no guarantee that $|U| \leq m$, there might be collisions.
To deal with that, instead of storing in the table a boolean, we store a list of all the keys corresponding to $h(k)$.
Then we insert a key in constant time $\mathcal{O}(1)$, search a key in time $\mathcal{O}(|h[key]|)$ and delete a key in time $\mathcal{O}(|h[key]|)$ since in the worst case the key is at the end of the list.

Further analysis leads to the following theorem.

Theorem 11.0.1 (Simple Uniform Hashing Assumption). *Assuming SUHA : "h equally distributes the keys into the table slots", and assuming $h(x)$ can be computed in $\mathcal{O}(1)$, $E\left[T_{search}(n)\right] = \mathcal{O}\left(1 + \frac{n}{m}\right)$, and same for deletion time.*
Formally, SUHA is :

$$\forall y \in [1, |T|] \mathbb{P}\left(h(x) = y\right) = \frac{1}{|T|}$$

$$\forall y_1, y_2 \in [1, |T|]^2 \mathbb{P}\left(h(x_1) = y_1,\ h(x_2) = y_2\right) = \frac{1}{|t|^2}$$

Proof.

1. Unsuccessful Search : Suppose that $k_0, \dots, k_{n-1}$ are keys in the dictionary and we perform an unsuccessful search for a key k. The number of comparisons is : $\sum_{i=0}^{n-1} \mathbb{1}_{h(k)=h(k_i)}$. Then, the expected time is : $\mathbb{E}[T_{search}] = \frac{n}{|T|}$ by SUHA.

2. Successful Search : Suppose keys were introduced in order $k_0, \dots, k_{n-1}$. k_i appears before any of $k_0, \dots, k_{i-1}$ and after any of $k_{i+1}, \dots, k_{n-1}$ that are in the same linked list. Then, to search for k_i, we need $\sum_{j=i+1}^{n-1} \mathbb{1}_{h(k_j)=h(k_i)}$. Under SUHA, the expectation of each of these variables is $\frac{1}{|T|}$. Then, the average expected search time is : $\frac{1}{n}\sum_{i=0}^{n-1} = 1 + \frac{1}{n}\sum_{i=0}^{n-1} \frac{(n-1-i)}{|T|} = \mathcal{O}(1 + \frac{n}{|T|})$

This concludes the proof of the theorem. ■

Good hash functions are functions that distribute the keys evenly. Yet, we do not know what the keys are, and thus will need various heuristics to answer this question. At least, without loss of generality, we can assume that keys are integers.

11.1 Heuristic Hash Functions

- Division Method : $h(k) = k \mod m$. It is better to choose m to be a prime number, and avoid $m = 2^p$.
- Multiplication Method : $h(k) = \lfloor m \left\{k \cdot A\right\} \rfloor$. $A \in (0, 1)$ and $m = 2^p$.

Yet, fixing the function can allow anyone to construct a probability distribution for which the function will be "bad".

11.2 Universal Family of Hash Functions

$H = \{h : U \to [0, |T| - 1]\}$ is Universal if :

$$\forall k_1 \neq k_2 \in U,\ \left|\{h \in H \mid h(k_1) = h(k_2)\}\right| \leq \frac{|H|}{m}$$

Theorem 11.2.1. *If h is a hash function chosen uniformly at random from a universal family of hash functions. Suppose that $h(k)$ can be computed in constant time and there are at most n keys. Then the expected search time is $\mathcal{O}(1 + \frac{n}{|T|})$*

Proof. Same as the case when h satisfies SUHA. Observe that the probability comes this time from choosing the function. ■

Theorem 11.2.2. *Let $p \in \mathcal{P}$ such that $U \subseteq [0, p-1]$. Then*

$$H = \left\{ h_{a,b}(k) = ((ak+b) \mod p) \mod |T| \mid a \in \mathbb{Z}_p^* b \in \mathbb{Z}_p \right\}$$

is a universal family.

Proof. Let $k_1 \neq k_2 \in U$. Let $l_i = (ak_i + b) \mod p$. We have $l_1 \neq l_2$ and $a = ((l_1 - l_2)((k_1 - k_2)^{-1} \mod p) \mod p)$ and $b = (l_1 - ak_1) \mod p$. There is then one-to-one mapping between (a, b) and (l_1, l_2). The number of $h \in H$ such that $h(k_1) = h(k_2)$ is :

$$|\{(l_1, l_2) \mid l_1 \neq l_2 \in \mathbb{Z}_p,\ l_1 = l_2 \mod m\}| \leq \frac{p(p-1)}{|T|} \leq \frac{|H|}{|T|}$$

■

12 Open Addressing

Elements are stored in the table. To insert, we probe the hash table until we find x or an empty slot. If we find an empty slot, insert x here. To define which slots to probe, we use a hash function that depends on the key and the probe number. To search, we probe the hash table until we either find x (return YES) or an empty slot (return NO). In the analysis, we will assume h to be uniform.

Theorem 12.0.1 (Analysis)**.** *Given an open-address hash-table with load factor $\alpha = \frac{n}{|T|} < 1$, the expected number of probes in an unsuccessfulsearch is at most $\frac{1}{1-\alpha}$, assuming uniform hashing.*

Proof. An unsuccessful search on x means that every probed slot except the last one is occupied and does not contain x, and the last one is empty. We define A_i the event " The i-th probe occurs and is occupied.". By Bayes's Theorem, we must estimate :

$$\mathbb{P}[\# \text{ of probes} \geq i] = \prod_{k=1}^{i-1} \mathbb{P}[A_k \mid \bigcap_{j=1}^{k-1} A_j]$$

But we have : $\mathbb{P}[A_j \mid A_1 \cap \ldots \cap A_{j-1}] = \frac{n-j+2}{|T|-j+2}$ So we have : $\mathbb{P}[\# \text{ of probes} \geq i] \leq \frac{n}{|T|}^{i-1} = \alpha^{i-1}$ Then, the expected number of probes is :

$$\sum_{i=1}^{+\infty} \mathbb{P}[\# \text{ of probes} \geq i] \leq \sum_{i=1}^{+\infty} \alpha^{i-1} = \frac{1}{1-\alpha}$$

■

Corollary 12.0.1.1. *The expected number of probes during insertion is at most $\frac{1}{1-\alpha}$.*

Proof. If we insert x we first ran an unsuccessful search for it ■

Theorem 12.0.2. *The expected number of probes during a Successful search is at most* $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$.

Proof. A successful search for x probes the same sequence of slots as insertion. If x is the i-th element inserted into the table, insertion probes less than $\frac{1}{1-\frac{i}{|T|}}$ slots in expectation. Therefore, the expected time of a successful search is at most :

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{|T|}{|T|-i} = \frac{|T|}{n}\sum_{i=0}^{n-1}\frac{1}{|T|-i} = \sum_{i=0}^{|T|}\frac{1}{i} - \sum_{i=0}^{|T|-n}\frac{1}{i} \leq \frac{|T|}{n}\ln\frac{|T|}{|T|-n} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

■

This is hard to implement, so we will use heuristics.

12.1 Heuristic hash functions

Let $h^{'}, h^{''}$ be two auxiliary hash functions.

- Linear Probing : $h(k,i) = (h^{'}(k)+i) \mod |T|$ This is easy to implement but it suffers from clustering.
- Quadratic Probing : $h(k,i) = (h^{'}(k)+c_1 i + c_2 i^2) \mod |T|$ We must choose the constants c_1 and c_2 carefully, and this still suffers from clustering.
- Double Hashing : $h(k,i) = (h^{'}(k)+ih^{''}(k)) \mod |T|$ To use the whole table, $h^{''}(k)$ must be relatively prime to m, e.g. $h^{''}(k)$ is always odd, $m = 2^i$.

13 Cuckoo Hashing

Hashing scheme with search time constant in the worst case, as it maintains a hash function without collisions to achieve perfect hashing. This is possible if the set of keys is static. Assume that we have two hash functions h_1, h_2 that satisfy SUHA. We store x in either $T[h_1(x)]$ or $T[h_2(x)]$. Search for x is done in constant time.

```
function INSERT(x)
    if $x = T[h_1(x)]$ or $x = T[h_2(x)]$ then return
    end if
    $pos \leftarrow h_1(x)$
    for $i \leftarrow 0$ to $n$ do
        if then$T[pos] = Null$
            $T[pos] = x$; return
        end if
        $x \longleftrightarrow T[pos]$
        if $pos = h_1(x)$ then
            $pos \leftarrow h_2(x)$
        else
```

```
            $pos \leftarrow h_1(x)$
        end if
    end for
    REHASH
    INSERT(x)
end function
```

Theorem 13.0.1. *This insertion is done in constant time.*

Lemma 13.0.2. *Suppose that $|T| \geq c{\cdot}n$ for some $c > 1$. For any i, j, the probability that there exists a path from i to j of length $l \geq 1$ which is a shortest path from i to j is at most $\frac{1}{c^l \cdot |T|}$*

Proof. By induction on l :

- Initialization : By SUHA, $\mathbb{P}[h_{1,2}(x) = y] = \frac{1}{|T|}$. Thus $\mathbb{P}[\text{there is an edge from } i \text{ to } j] = \frac{n}{|T|^2} \leq \frac{1}{c|T|}$
- Heredity : For $l \geq 1$ there must exist k such that there is a path of length $l - 1$ from i to k and an edge from k to j.

■

Proof. We define the bucket of x as all the cells that can be reached either from $h_1(x)$ or $h_2(x)$. x, y are in the same bucket if and only if there is a path from $\{h_1(x), h_2(x)\}$ to $\{h_1(y), h_2(y)\}$. Then :

$$\mathbb{P}[x, y \text{ are in the same bucket}] \leq 4 \sum_{l=1}^{\infty} \frac{1}{c^l\,|T|} = \frac{4}{(c-1)\,|T|}$$

So :

$$\mathbb{E}[|\text{bucket of } x|] = \mathbb{E} \sum X_{x,y} = \sum \mathbb{E}[X_{x,y}] = \sum \mathbb{P}[x, y \text{ are in the same bucket}] \leq \frac{4}{c-1}$$

Hence, in the absence of rehash, expected insertion time in constant.

The probability that we need a rehash is at most probability that there is a cycle, i.e. a path from i to i : $\frac{4}{c-1} \leq \frac{1}{2}$. The probability that we will need d rehashes is then at most $\frac{1}{2^d}$. Thus, the expected time per insertion is :

$$\frac{1}{n} \cdot \mathcal{O}(n) \sum_{d=1}^{+\infty} \frac{1}{2^d} = \mathcal{O}(1)$$

■

13.1 Rolling Hash Functions

Definition 13.1.1. *The Karp-Rabin fingerprint of a string $S = s_1 \dots s_m$ is :*

$$\varphi(s_1 \dots s_m) = \sum_{i=1}^{m} s_i r^{m-1} \mod p$$

where p is a big enough prime and $r \in \mathbb{F}_p$.

Proposition 13.1.1.
- *If $S = T$, then $\varphi(S) = \varphi(T)$*
- *Else, $\varphi(S) \neq \varphi(T)$ with high probability.*

Proof. Let σ be the size of the alphabet, $p \geq max\,\{\sigma, n^c\}$ where $c > 1$ is a constant. We have:

$$\varphi(S) = \varphi(T) \Leftrightarrow \sum_{i=1}^{m} (s_i - t_i) \cdot r^{m-i} \mod p = 0$$

Hence, r is a root of $P(x) = \sum_{i=1}^{m} (s_i - t_i) \cdot x^{m-i}$ a polynomial over $\mathbb{F}_p$. The probability of such event is at most $\frac{m}{p} \leq \frac{1}{n^{c-1}}$. ■

The Karp-Rabin algorithm is as follows :

- Compute the fingerprint of the pattern
- Compare it with the fingerprint of each m-length substring of the text. If the fingerprint is equal to the fingerprint of a substring, report it as an occurrence

Proposition 13.1.2. *We have :*

$$\varphi(s_1 \dots s_{j+1}) = \varphi(s_1 \dots s_j) \cdot r + s_{j+1} \mod p$$

Proof. Observe that :

$$\varphi(s_1 \dots s_{j+1}) = \sum_{i=1}^{j+1} s_i r^{j+1-i} \mod p$$

■

Part IV

Lecture 4 : 26/10

Contents

14 Binary Search Trees

Definition 14.0.1. *A Binary Tree is a rooted tree with every node having at most two children. Binary search trees are binary trees verifying the following :*

If a node contains element l, the left subtree rooted at the left child contains only elements $\leq l$ and the right subtree elements $> l$.

Proposition 14.0.1. *To access a given element, the time needed is $\mathcal{O}(h)$ time where h is the height of the tree.*

h is not necessarily small, there are many implementations to bound h.

14.1 Red-Black Trees

Definition 14.1.1. *A red-black tree is a binary tree that satisfies the following red-black properties :*

1. *Every node has a color : red or black*
2. *The root is black*
3. *Every leaf (NIL) is black*
4. *If a node is red, both its children are black*
5. *For each node, all paths from the node to descendant leaves contain the same number of black nodes.*

Lemma 14.1.1. *The height of a red-black tree with n nodes is at most $2\log(n+1)$.*

Proof. We introduce $bh(x)$ the number of black nodes in a path from x to a leaf. By induction on $bh(x)$, the subtree rooted at x contains at least $2^{bh(x)} - 1$:

- Initialization : $bh(x) = 0$ implies the tree has more than 0 nodes which is true.
- Heredity :
  - If x is black, let y_1, y_2 denote its children. We have $bh(y_1) = bh(x) - 1 = bh(y_2) - 1$. By summing, we get the result.

- If x is red, both of its children are black by rule 4. Let us denote the grand children of x by $z_1, \dots, z_4$. We have : $bh(z_i) = bh(x) - 1$. By sum, we get the result.

Then if h the black height of the tree, $n \geq 2^h - 1$ and $h \leq \log n + 1$. Since in any root-to-leaf path, the number of nodes is at most twice the number of black nodes per rule 4, the lemma follows. ■

Proposition 14.1.1 (Insertion). *We insert a node into a n-node red-black tree by :*

- *Inserting the node into T as if it were an ordinary binary search tree*
- *We color it red*
- *We perform a number of rotations and node recolouring.*

This is done in $\mathcal{O}(\log n)$.

We define the left rotation on a node x whose right child is not null like this :

$$Tree(x, \alpha, Tree(y, \beta, \gamma)) \mapsto Tree(y, Tree(x, \alpha, \beta), \gamma)$$

We define right rotations to be the reverse operation.

Proposition 14.1.2. *Rotations do not break the BST properties.*

Only on right rotations. Elements in γ are greater than y, in β are greater than x and lower or equal than y, and elements in α are lower or equal than x. After the rotation, we still have those properties. ■

Proposition 14.1.3 (Restoring red-black properties). *Rule* 1 *is never violated. Rules* 2 *and* 3 *can be fixed in $\mathcal{O}(1)$ time. Rule* 5 *is never violated as we colour the new node red.*
We will try and maintain the invariant that the properties are broken in at most one node. There are three cases in which Rule 4 *can be violated : Let z be the node that violates the red-black properties.*

1. *z's uncle y is red $\Rightarrow$ We colour z.p black, colour y black and z.p.p red. We moved the problematic node up by two layers.*
2. *z's uncle y is black and z is a right child $\Rightarrow$ We left-rotate around the edge z.p-z, we then have z's uncle y to be black and z to be a left child, i.e. case 3.*
3. *z's uncle y is black and z is a left child $\Rightarrow$ We color z.p black, colour z.p.p red and right-rotate.*

Proposition 14.1.4 (Complexity). *Red black tree use $\mathcal{O}(n)$ space, $\mathcal{O}(\log n)$ in time for insertion and deletion in the worst case.*

14.2 Treaps

Definition 14.2.1. *Treaps are binary trees with every node having a key and a priority verifying :*

- *a binary search tree for the keys*
- *a min-heap for the priorities*

Proposition 14.2.1 (Search). *The usual algorithm yields time in $\mathcal{O}(depthofthenode)$ for a successful search and $\mathcal{O}(\max(\texttt{depth}(v^-), \texttt{depth}(v^+)))$ where $\texttt{key}(v^-)$ is the predecessor of the searched key.*

Proposition 14.2.2 (Insertion). *The standard BST algorithm with added rotations to fix the heap properties (if $p(z) \leq p(z.p)$, rotate auround $(z, z.p)$) yields time complexity in $\mathcal{O}(\max(\texttt{depth}(v^-), \texttt{depth}(v^+)))$.*

Proposition 14.2.3 (Deletion). *We run the insertion algorithm backwards : let $\texttt{light}(z)$ be the child of z with smaller priority. The choice of this edge preserves the heap property everywhere except at z. As long as z is not a leaft, rotate around $(z, \texttt{ligth}(z))$, then chop it off.*

Proposition 14.2.4 (Split). *If we want to split the tree along a pivot π, we can insert a new node z with key π and priority $-\infty$. After the insertion, this new node is the root since it has the smallest priority, and all keys in the left subtree are smaller than the pivot and this subtree is a treap.*

Proposition 14.2.5 (Merge). *Suppose that we want to merge two treaps T_- and T_+, where keys in the first tree are smaller than the keys in the second tree. We create a dummy root with priority $-\infty$ and then delete it.*

Proposition 14.2.6 (Complexity).

- *Time for a successful search : $\mathcal{O}(\texttt{depth(v)})$ where $\texttt{key}(v) = k$.*
- *Time for an unsuccessful search : $\mathcal{O}(\max(\texttt{depth}(v^-), \texttt{depth}(v^+)))$ where $\texttt{key}(v^-)$ is the predecessor of the searched key.*
- *Insertion Time : $\mathcal{O}(\max(\texttt{depth}(v^-), \texttt{depth}(v^+)))$ where $\texttt{key}(v^-)$ is the predecessor of the searched key.*
- *Deletion Time : $\mathcal{O}(tree\ depth)$*
- *Split/Merge Time : same as insertion / deletion*

We will choose the priorities to be independently and uniformly distributed continuous random variables. We denote by $x_1, \dots, x_n$ the nodes of the tree in the increasing order of the keys. We denote by $i \uparrow k$, the event x_i if a proper ancestor of x_k : $\texttt{depth}(x_k) = \sum_{i=1}^{i=n} [i \uparrow k]$

Lemma 14.2.1. *Define*

$$X(i,k) = \begin{cases} x_i, \dots, x_k & \text{if } i < k \\ x_k, \dots, x_i & \text{otherwise.} \end{cases}$$

For all $i \neq k$ we have $i \uparrow k$ iff x_i has the smallest priority in $X(i,k)$.

Proof.

1. If x_i is the root, then x_i has the smallest priority
2. If x_k is the root, then x_i is not an ancestor of x_k and does not have the smallest priority
3. If x_j is the root with $i < j < k$, then x_i is not an ancestor of x_k and does not have the smallest priority.
4. If x_j is the root and either $i < k < j$ or $i < j < k$ then x_i and x_j are in the same subtree and the claim follow by induction.

■

Corollary 14.2.1.1.

$$\mathbb{P}\left[i \uparrow k\right] = \begin{cases} \dfrac{1}{k-i+1} & \textit{if } i < k \\ 0 & \textit{if } i = k \\ \dfrac{1}{i-k+1} & \textit{if } i > k \end{cases}$$

Then :

$$\mathbb{E}\left[\texttt{depth}(x_k)\right] = \sum_{i=1}^{i=n} \mathbb{P}\left[i \uparrow k\right] = \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{k+1}^{n} \frac{1}{i-k+1} = H_k - 1 + H_{n-k+1} - 1 < 2\ln n - 2$$

Thus, all treap operations take $\mathcal{O}(\log n)$ time in expectation

Remark 14.2.1.1. *The fastest sorting algorithm is Quicksort, and it can be seen as inputing all the values in the list in a treap then taking the infix order of the treap.*

Remark 14.2.1.2. *Tango trees are $\mathcal{O}(\log \log n)$-competitive and Splay trees are conjecture to be fastest in $\mathcal{O}(1)$-competitive, but we do not know if this is true.*

15 Lower Bound for Sorting

The question here is : can we sort in $\mathcal{O}(n \log n)$ or better ? If comparisons only are allowed then no : indeed, any comparison-based sorting program can be thought of as defining a decision tree of possible executions. Running the same program twice on the same permutation causes it to do the exact same thing bu running it on different permutations of the same data causes a different sequence of comparisons to be made on each.

15.1 Decision Tree

Definition 15.1.1. *A decision tree for a sorting algorithm is a binary tree that shows the possible executions of an algorithm on a set.*

Proposition 15.1.1. *The minimum height of a decision tree is the worst-case complexity of comparison-based sorting.*

Lemma 15.1.1. *The height of any decision tree is $\Omega(n \log n)$*

Proof. Since any two different permutations of n elements require a different sequence of steps to sort, there must be at least $n!$ different paths from the root to leaves in the decision tree. Thus there must be at least $n!$ different leaves in this binary tree. Since a binary tree of height h has at most 2^h leaves, we know that : $n! \leq 2^h$ i.e. $h \geq \log(n!)$. Finally, we have $\log n! = \Omega(n \log n)$. ■

Theorem 15.1.2. *Any comparison-based sorting algorithm must use $\Omega(n \log n)$ time.*

16 Predecessor Problem

We want to maintain a set S of integers from a universe $U = [1, u]$ subject to insertions, deletions, predecessor and succesor queries. This problem is harder than dictionaries and hashing. BTSs give a solution in $\mathcal{O}(n)$ space and $\Theta(\log n)$ time.

16.1 van Emde Boas Trees

If the time per operation satisfies : $T(u) = T(\sqrt{u}) + \mathcal{O}(1)$, by Substitution, $T(u) = \mathcal{O}(\log \log u)$. We will split U into $\sqrt{u}$ chunks of size $\sqrt{u}$ size, and the recurse.

Definition 16.1.1 (Recursive van Emde Boas Trees).

- *`T.summary` is a vEB-tree of size $\sqrt{u}$ containing all i such that the i-th chunk is not empty.*
- *For each $1 \leq i \leq \sqrt{u}$, `T.chunk[i]` is a vEB-tree of size $\sqrt{u}$ containing x`mod`$\sqrt{u}$ for each x in the i-th tree.*
- *`T.min` is the smallest element in T, not stored recursively.*

We represent each integer $x = \langle c, i \rangle$ where

- *c is the chunk coordinates : $c = x // \sqrt{u}$*
- *i is the position of x in the chunk : $i = x$`mod`$\sqrt{u}$*

Proposition 16.1.1 (Successor Operation). *With the following algorithm we get the wanted complexity :*

Proposition 16.1.2 (Insertion). *The algorithm below gets the correct complexity.*

Proposition 16.1.3 (Deletion). *The algorithm below gets the correct complextiy.*

Proposition 16.1.4 (Complexity).

- *All of these operations' complexities verify $T(u) = T(\sqrt{u}) + \mathcal{O}(1)$ and thus have time complexity $\mathcal{O}(\log \log u)$*
- *Space complexity satifies : $S(u) = (\sqrt{u} + 1)\,\dot{S}(\sqrt{u}) + \mathcal{O}(1)$ and therefore $S(u) = \mathcal{O}(u)$.*

Proposition 16.1.5 (Original van Emde Boad trees). *For `Successor`(x), as the path from the root to x is monotone, binary searching the path to find the lowest 1 gives us either the predecessor or the successor of x. By storing all nodes in an array of size $\mathcal{O}(u)$ to allow efficient binary search; a pointer from each node to the*

Algorithm 6 Successor Complexity Verifies : $T(u) = T(\sqrt{u}) + \mathcal{O}(1)$

```
function (Successor)
    Input  $(T, x = \langle c, i \rangle)$
    if $x < T.min$ then
        return $T.min$
    end if
    if $i < T.chunk[c].max$ then
        return $\langle c, \texttt{Successor}\,(T.chunk[c], i)\rangle$
    else
        $c^{'} = \texttt{Successor}(T.summary, c)$
        return $\langle c^{'}, T.chunk[c^{'}].min\rangle$
    end if
end function
```

Algorithm 7 Insertion Complexity Verifies : $T(u) = T(\sqrt{u}) + \mathcal{O}(1)$

```
function (Insert)
    Input  $(T, x = \langle c, i \rangle)$
    if $T.min = None$ then
        $T.min = T.max = x$
        return
    end if
    if $x < T.min$ then
        $\texttt{swap}(x, T.min)$
    end if
    if $x > T.max$ then
        $T.max = x$
    end if
    if $T.chunk[c].min = None$ then
        $\texttt{Insert}(T.summary, c)$
    end if
    $\texttt{Insert}(T.chunk[c], i)$
end function
```

Algorithm 8 Deletion Complexity Verifies : $T(u) = T(\sqrt{u}) + \mathcal{O}(1)$

```
function (Delete)
    Input  $(T, x = \langle c, i \rangle)$
    if $x = T.min$ then
        $c = T.summary.min$
        if c = None then
            $T.min = None.$ return
        end if
        $x = T.min = \langle c, i = T.chunk[c].min \rangle$
    end if
    Delete($T.chunk[c], i$)
    if $T.chunk[c].min = None$ then
        Delete($T.summary, c$)
    end if
    if $T.summary.min = None$ then
        $T.max = T.min$
    else
        $c^{'} = T.summary.max$
        $T.max = \langle c^{'}, T.chunk[c^{'}].max \rangle$
    end if
end function
```

maximum and minimum of their subtree; all the elements as a doubly-linked list, we find the successor and the predecessor of x in $\mathcal{O}(\log \log u)$ time.
Update is done in $\mathcal{O}(\log u)$ time, since we only need to update the element-to-root path, in $\Theta(u)$ space

16.2 Improvements

16.2.1 x-fast trees

Definition 16.2.1. *In an x-fast tree, we store every root-to-green node (nodes representing an element from the set) path, viewed in binary (left = 0, right = 1), via Cuckoo Hashing.*

Proposition 16.2.1. *Predecessor queries are done in $\mathcal{O}(\log \log u)$ time, updates in $\mathcal{O}(\log u)$ expected amortised time, but this tree only takes $\mathcal{O}(n \log u)$ space.*

Proof. To maintain successor and predecessor, we use the vEB-tree algorithm, giving us the same complexity, and same for updates. Yet, since we only store the root-to-element paths which are of $\log u$ length, we need $\mathcal{O}(n \log u)$ space. ∎

16.2.2 y-fast trees

Definition 16.2.2. *We maintain elements in groups of size in $\left[\frac{\log u}{4}, 2 \log u\right]$. For each group, we build a BST, and we store representatives of the group using an x-fast tree :*

- *If there are fewer than $\frac{\log u}{2}$ elements, we store them in a single BST.*
- *otherwise, suppose we add/delete an element. If a group becomes too large, we split it in two. If a group becomes too small, we merge it with its neighbour, then split if needed.*

Proposition 16.2.2. *Predecessor queries are in $\mathcal{O}(\log \log u)$ time, updates in $\mathcal{O}(\log \log u)$ expected amortised time, since insertion into the x-fast trie happens only once per $\Theta(\log u)$ new elements.*

Proof. This comes directly from the definition and the definition of the x-fast trees. ■

Part V
Lecture 5 : 9/11

Contents

17 General Definitions

Definition 17.0.1. *An alphabet Σ is a finite set. Elements of Σ are letters, characters or symbols, denoted by small letters. A string (or word), is a finite sequence of letters, denoted by a capital letter.*
The length of a string $S = s_1 s_2 \dots s_n$ where $s_i \in \Sigma$ is denoted by $|S|$ and defined to be equal to n.
For $1 \leq i \leq j \leq S$, we say that $S[i,j] = s_i \dots s_j$ is a substring of S. If $i = 1$, $S[i,j]$ is a prefix of s. If $j = |S|$, $S[i,j]$ is a suffix of S. $S[i,j]$ is an occurence of a string X in S if $S[i,j] = X$

18 Pattern Matching

Definition 18.0.1. *We define the pattern matching problem as such : Given a string P of length m (pattern), a string T of lenth $n \geq m$ (text), return all occurences of P in T specified by their starting positions.*

18.1 Naïve Algorithm

The naïve algorithm consists of looking, for each possible starting position in the text, check if the m following characters form a string equal to the pattern. This algorithm has time complexity $\mathcal{O}(nm)$ and space complexity $\mathcal{O}(n+m)$.

Remark 18.1.0.1. *There are more than 80 algorithms for this problem. See S.Faro, T.Lecroq https://arxiv.org/pdf/1012.2547.pdf*

18.2 Knuth-Morris-Pratt Algorithms

Definition 18.2.1. *The border of a string P is a proper prefix of P that is equal to a suffix of P.*
The border array of length $|P|$ is such that $B[i]$ is the maximal length of a border of $P[1,i]$, 0 if undefined.

Proposition 18.2.1. *B can be computed in $\mathcal{O}(|P|)$ time.*

Lemma 18.2.1. *First, note that :*

1. $B[1] = 0$
2. *If X is a border of $P[1,k]$ of length x, then $X' = X[1, x-1]$ is a border of $P[1, k-1]$.*
3. *$B[k] - 1 = B[k-1]$ if and only if $P[k] = P[B[k-1]+1]$.*
4. *If X is a border of X and Y is a border of X, Y is a border of S.*

Suppose that we have computed $B[1], \dots, B[k-1]$. We will now compute $B[k]$.
By property 3, if $P[k] = P[B[k-1]+1]$, then $B[k] = B[k-1]+1$.
Else, if $P[k] \neq P[B[k-1]+1]$, consider $B^2[k-1] = B[B[k-1]]$. If $P[k] = P[B^2[k-1]+1]$, set $B[k] = B^2[k-1]+1$, else consider $B^3[k-1]$, and so on.

This algorithm is correct and in $\mathcal{O}(m)$.

Proof.
- Correctness : In fact, $B[1,k] \in \{B[k-1]+1, B^2[k-1]+1, \dots, 0\}$.
  From property 2, any border of $P[1,k]$ can be obtained by appending $P[k]$ to some border of $P[1, k-1]$.
  From property 4, $B^j[k-1]$ is the j-th longest border of $P[1, k-1]$.

- Time : $\texttt{Time}(B[k]) \leq |B[k] - B[k-1]|$, indeed, if $B[k] = B^j[k-1]+1$, we use j steps. Observe that :

  $$-1 \leq \sum_k (B[k] - B[k-1]) = \overset{-}{\sum} (B[k] - B[k-1]) + \overset{+}{\sum} (B[k] - B[k-1]) \leq m$$

  However, by property 2, the sum on the positive differences if $\mathcal{O}(m)$. Hence, the total time is in $\mathcal{O}\left(\sum |B[k] - B[k-1]|\right) = \mathcal{O}(m)$

■

Algorithm 9 Knuth-Morris-Pratt Algorithm : KMP

```
Input  $T, P, n, m$
$B \leftarrow$ BorderArray($P$)
$L \leftarrow []$
$i \leftarrow 1,\ j \leftarrow 1$
while $i \leq n - m + j$ do
    $(i, j) \leftarrow$ match($i, j, m$)
    if $j = m + 1$ then
        $L \leftarrow (i - m) :: L$
    else
        if $j = 1$ then
            $i \leftarrow i + 1$
        else
            $j \leftarrow B[j-1] + 1$
        end if
    end if
end whilereturn  $L$
```

Proposition 18.2.2. *This algorithm is Correct*

Lemma 18.2.2. *No occurence of P can start in the shift interval.*

Proof. If $T[x, x+m-1] = P$, $P[1, j-1]$ has a border of length $> B[j-1]$, contradiction. ■

Lemma 18.2.3. *We have $P[1, B[j-1]] = T[i + (j - 1 - B[j-1]), i + j - 2]$. Hence, we do not need to compare the first $B[j-1]$ letters of the patter with the letters of the text after the shift.*

Of the Correctness. This stands from 18.2.2 and 18.2.3. ■

Proposition 18.2.3. *This is done using $\mathcal{O}(m)$ extra space, in $\mathcal{O}(n+m)$ time.*

Proof.
- Space : $\mathcal{O}(m)$ to store B + constant
- Time : From **??**, $\mathcal{O}(m)$ to build B. To process T we need $\mathcal{O}(n+m)$ time, since $1 \leq i \leq n$ and $1 \leq j \leq n$ and their difference increases after any step.

■

Definition 18.2.2. *An algorithm is called online if can process its input item-by-item in a serial fashion, without having the entire input available from the start.*
An online algorithm is called real-time if it processes each data item in constant time.
KMP is an online algorithm, but it is not real-time.

Proposition 18.2.4. *We can construct a real-time version of KMP, by using $B^{'}[j, T[i]]$ to decide how to shift the pattern, where :*

$$B^{'}[j, a] = \max\{k \mid k < j,\ P[1, k] = P[j - k + 1, j],\ P[k + 1] = a\}$$

$B^{'}$ occupies $\mathcal{O}(m)$ space and is computed in $\mathcal{O}(m \cdot |\Sigma|)$ time. The modified KMP is real-time and correct.

19 Dictionaries, Multiple Pattern Matching

19.1 Tries

Definition 19.1.1. *A dictionary D is a set of string. A trie of D is a tree with every node being labeled by a letter such that :*

- *For every node, outgoing edges are labeled with different letters*
- *For every string $S \in D$, there is a root-to-node path that spells out S. The end of the path is labeled with the id of S.*
- *Every root-to-leaf path spells out a string in D.*

Proposition 19.1.1. *A trie for D uses $\mathcal{O}(\sum_{S \in D} |S|)$ space.*

Proof. By laying out each root-to-leaf path. ■

We will consider two applications :

1. Dictionary Look-Up : Given a string P, decide if it belongs to the dictionary.
2. Multiple Pattern Matching : Given a dictionary of patterns D, find all their occurences in a text T.

19.2 Dictionary Look-Up

Proposition 19.2.1. *For P a pattern, $P \in D \Leftrightarrow$ there is a root-to-node path that spells out P labeled by an id of a string from D. By starting at the root, if there is an from the root to its child u labeled by $P[1]$, the algorithm moves to u and continues recursively. This takes time in $\mathcal{O}(|P|)$.*

19.3 Aho-Corasick Algorithm

We will first assume that no pattern in D is a substring of another.

Definition 19.3.1. *The failure link from a node u labeled by a string S points to the deepest node v labeled by a proper suffix of S.*

Lemma 19.3.1. *The trie occupies $\mathcal{O}(m)$ space and can be constructed in $\mathcal{O}(m)$ time.*

Proof.

- Space : There are $\mathcal{O}(m)$ nodes, each node has exactly one failure link.
- Time : Failure links are built top-to-down. Links from nodes of depth 1 point to the root. Suppose that we have built links for all nodes to depth $\leq d-1$. The failure link from a node labeled with Sa must point to a node labeled with $S^{'}a$, where $S^{'}$ is a suffix of S. In other words, the parent of the end of the failure link from the node must belong to the failure link path from the node $p(u)$, and it must be the deepest node that has an outgoing edge labeled with a. The algorithm works as follows: we follow the failure link path from $p(u)$. The first node with an outgoing edge labeled with a is the end of the failure link for u. Consider the time needed to build the failure links for one root-to-leaf path. Let $root, u_1, \ldots, u_k$ be the nodes in this path, and denote by $f(u_i)$ the depth of the end of the failure link for u_i. The time to find the link for u_i is $\leq c \times (2 + f(u_{i-1} - f(u_i)))$. Thus the total time for the trie is $\mathcal{O}(m)$.

■

Algorithm 10 Aho-Corasick Algorithm

```
Input $T, D, n$
$curr\_node \leftarrow root, i \leftarrow 1, L \leftarrow []$
while $i \leq n$ do
    if $\exists e = (curr\_node, u)$ labeled with $T[i]$ then
        $curr\_node \leftarrow u$
        if $curr\_node$ corresponds to a pattern then
            $L \leftarrow i :: L$
        end if
        $i \leftarrow i + 1$
    else
        if $curr\_node = root$ then
            $i \leftarrow i + 1$
        else
            $curr\_node$ = failure_link($curr\_node$)
        end if
    end if
end while
```

Proposition 19.3.1. *This algorithm has space complexity $\mathcal{O}(m)$ and time complexity $\mathcal{O}(m+n)$.*

Proof.
- Space : m is the total length of the patterns
- Time : We need $\mathcal{O}(m)$ time to build the trie. Consider how the depth of *curr_node* changes during the algorithm.
  - Every time we do down an edge (happens $\leq n$ times), it increases by 1. Every time we follow a failure link, it decreases by ≥ 1.
  - Therefore, as the depth is always positive, we follow a failure link at most n times.

■

Theorem 19.3.2. *If no pattern is a substring of another, the multiple pattern matching problem can be solved in $\mathcal{O}(m)$ space and $\mathcal{O}(n+m)$ time.*

Definition 19.3.2. *When patterns are substrings of others, we add output links : an output link from a node u goes to the nearest node on the failure link path outgoing from u that corresponds to a pattern of the dictionary.*

Proposition 19.3.2. *Output links can be built in $\mathcal{O}(m)$ time by one top-down traversal of the failure link tree.*

We modify the algorithm : at each step it follows the path from *curr_node* and outputs the patterns in the output link path.

Theorem 19.3.3. *The multiple pattern matching problem can ve solved in $\mathcal{O}(m)$ space and $\mathcal{O}(n+m+occ)$ where occ is the total number of occurences of the patterns in the text.*

20 Suffix Trees and Applications

Definition 20.0.1. *A text index is a data structure that represents a text and supports pattern matching queries. The suffix tree is one of them.*

We define the compact trie for D as the trie for D where nodes with only one child have been replaced by an edge. The suffix tree of a string is the compact trie for the set of the suffixes of that string with \$ appended (\$ must not be in Σ).

20.1 Suffix Trees

Proposition 20.1.1. *A suffix tree for a string of length n has n leaves, and at most $2n-1$ nodes and $2n-2$ edges. Storing the labels on the edges can take $\Theta\left(|T|^2\right)$. To save space we represent each label as two numbers, the left and the right endpoints in T.*

Proof. By induction. ■

To implement algorithms on the suffix tree efficiently, we need to be able to identify, given a node u and a letter a, an edge u, v such that its label starts with a. Then, for each node u, we store an array A_u of size $|\Sigma|$, with $A_u[a]$ being a pointer to the child v of u such that the label of (u, v) starts with a.

Proposition 20.1.2. *This structure takes $\mathcal{O}(|Sigma| \cdot |T|)$ space.*

20.2 Pattern Matching

Remark 20.2.0.1. *A suffix of T starts with P if and only if it corresponds to a leaf of the suffix tree that belongs to a subtree rooted at the end of the path labeled with P.*

To answer pattern matching queries, we start at the root. Then, we follow the path labeled by the letters of P, and use arrays A_u to find the next edge to follow. If there is no path labeled by P, there are no occurences of P in T. Otherwise, the starting positions of the occurences of P in T are the starting positions of the suffixes that are in the subtree rooted at the end of the path labeled by P. We retrieve the occurences by traversing the subtree depth-first.

Proposition 20.2.1. *This algorithm has total time $\mathcal{O}(m + occ)$.*

Proof. The arrays allow to decide which edge to follow next in $\mathcal{O}(1)$ time. In total, finding the path requires $\mathcal{O}(m)$ time. If there is a path labeled by P, the subtree rooted at its end has $\mathcal{O}(occ)$ leaves, where occ is the number of occurences of P in T. As the subtree does not have nodes of degree one (except possibly for the root), its size is $\mathcal{O}(occ)$. ■

20.3 Longest Common Substring

Given two strings, find the longest substring that occurs both in T_1 and T_2. We will denote $n = |T_1| + |T_2|$.

To find the longest common substring of T_1 and T_2, we need the suffix tree containing the suffixes of both string (sometimes called generalised suffix tree).

Proposition 20.3.1. *This suffix tree can be built in $\mathcal{O}(|T_1| + |T_2|)$.*

Remark 20.3.0.1. *If a string X occurs at position i in T_1, and at position j in T_2, then the subtree rooted at the end of the path labeled by X contains the leaf corresponding to $T_1[i \ldots]$ and $T_2[j \ldots]$.*

By bottom-up traversal, we find the deepest node in the tree such that its subtree contains both leaves corresponding to suffixes of T_1 and suffixes of T_2. It is labeled by the longest common substring of T_1 and T_2.

Proposition 20.3.2. *This algorithm has time complexity $\mathcal{O}(|T_1| + |T_2|)$.*

We can generalise this to any m strings in time $\mathcal{O}(n)$ where $n = \sum_{k=1}^{m} |T_k|$. We will use the following fact : we can preprocess a tree of size $\mathcal{O}(n)$ in time $\mathcal{O}(n)$ to support lowest common ancestor (LCA) queries in constant time. (see) $LCA(u, v)$ must return the lowest node that is an ancestor of both u and v.

Open Problem 4. *We showed that the longest common substring problem for two string can be solved in $\mathcal{O}(n)$ space and $\mathcal{O}(n)$ time. It is also known that the problem can be solved in $\mathcal{O}(s)$ extra space and time*

$$\mathcal{O}\left(\frac{n^2 \log(n) \log^*(n)}{L \cdot s} + n \log n\right)$$

where L is answer. What is the optimal trade-off.

Part VI
Lecture 6 : 23/11

Contents

21 Disjoint-set Data Structure

Definition 21.0.1. *A disjoint-set data structure is a way to maintain a collection of disjoint sets with 3 operations :*

- *`make_set(x)` : create a new set containing one element, x*
- *`union_set(x, y)` : union the sets containing x and y*
- *`find_set(x)` : return a pointer to the representative of the set containing x*

We will parametrize by n the number of elements and m the number of operations.

It can be used to preprocess the connected components of a graph, so as to maintain the queries of the paths between two vertices.

It can also be used to represent the ways player play in the HEX game.

22 Linked-list Representation

Here, each set is a linked list, and the representative is the head of the list. For every element, we store a pointer to the node containing it. In that node, we store a pointer to the representative.

Proposition 22.0.1. *In this situation : `make_set` and `find_set` are done in $\mathcal{O}(1)$.*

For `union_set`, it requires more work.
Indeed, simply appending the set of y to the set of x takes constant time, but updating the pointers to the representatives is done in $\Theta\left(|\text{set of } y|\right)$. By doing the following sequence of operation :

$$\texttt{make\_set}(x_1), \texttt{make\_set}(x_2), \ldots, \texttt{make\_set}(x_n); \texttt{union\_set}(x_2, x_1), \texttt{union\_set}(x_3, x_2), \ldots, \texttt{union\_se}$$

we see that even the amortised time per operation can be large, as it takes a total time of $\Theta\left(n^2\right)$ and the amortised being $\Theta(n)$.

Then, we propose a weighted-union strategy : if the set of x is larger than the set of y, we append the latter to the former, else we append the set of x to the set of y.

Theorem 22.0.1. *Using linked-lists and the weighted-union strategy, doing m `make_set`, `union_set` and `find_set`, n of which are `union_set` takes $\mathcal{O}\left(m + n\log n\right)$*

Proof. At any moment, the largest size of a set is bounded by n. Therefore, an element can change its set representative at most $\log_2 n$, as every time it happends the size of the set at least doubles in size. Therefore, the total time for updating the representatives is $\mathcal{O}(n \log n)$. All other operations take $\mathcal{O}(m)$ time. ■

23 Disjoint-set forests

Here, each set is a treee whose root is the representative. Each node stores a pointer to its parent. `make_set`(x) creates a tree containing only one node, `find_set`(x) follows the path from x to the root and `union_set`(x, y) makes the root of the tree of y become a child of the root of the tree of x.
We will use two strategies :

- Path Compression : During `find_set`(x), we make each node on the path to the root be a child of the root.
- Union by Rank : During `union_set`, the root of the tree with smaller rank becomes a child of the root of the tree with larger rank.

Definition 23.0.1. *We define ranks as upper bounds on the height of the node :*

- *When we add a new node x, we initialise the rank of x with 0.*
- *If we union two trees (i.e.,two sets) of different ranks, ranks do not change. If we union two trees of equal ranks, the rank of the root of the resulting tree increases by one.*

24 Time Analysis

We defined n as the number of elements, i.e. of `make_set` and m the total number of operations.

Algorithm 11 Union-Find

```
function (make_set)
    Input $x$
    $x.p \leftarrow x$
    $x.rank \leftarrow 0$
end function
function (union_set)
    Input $x, y$
    link(find_set($x$), find_set($y$))
end function
function (find_set)
    Input $x$
    if $x \neq x.p$ then
        $x.p \leftarrow find\_set(x.p)$
    end ifreturn $x.p$
end function
function (link)
    Input $x, y$
    if $x \neq y$ then
        if $x.rank > y.rank$ then
            $y.p \leftarrow x$
        else
            $x.p \leftarrow y$
            if $x.rank = y.rank$ then
                $y.rank \leftarrow y.rank + 1$
            end if
        end if
    end if
end function
```

Theorem 24.0.1 (Tarjan)**.** *Using both Union by Rank and Path Compression, the time complexity for any sequence of Find and Union is $\mathcal{O}(m + n\alpha(n))$ where $\alpha(4)$ is the inverse of the Ackermann function. In any conceivable application $\alpha(n) \leq 4$. This time complexity is tight.*

24.1 A simpler Bound

We will first start with a simpler analysis showing $\mathcal{O}(m \log^\star n)$.

Proposition 24.1.1. *Suppose we convert a sequence of $m^{'}$ operations* ***make_set, union_set, find_set*** *into m operations* **make_set, link, find_set**. *If we can execute the latter in $\mathcal{O}(m \cdot \alpha(n))$, we can execute the former in $\mathcal{O}(m^{'} \cdot \alpha(n))$.*

Proof. $m^{'} \leq m \leq 3m^{'}$ ■

We will thus assume we have only `make_set, link, find_set` operations.

Proposition 24.1.2.
- *The rank of a node can only increase with time. Rank of non-root nodes are frozen forever.*
- *For every node x, $x.rank \leq x.p.rank$ and the inequality is strict unless x is a root node.*
- *The ranks of nodes in any path strictly increase*
- *The rank of any node is bounded by $n - 1$.*

Proposition 24.1.3 (Rank Lemma)**.** *At any moment, for every $r \in \mathbb{N}$, there are at most $\frac{n}{2^r}$.*

Proof. By induction, the number of nodes in a tree with a root of rank r is at least 2^r :

- Base Case : $r = 0$ is true
- Induction Step : After linking two trees with roots of rank r_1, r_2 :
  - If $r_1 < r_2$ the claim is true by induction for r_2.
  - If $r = r_1 = r_2$, then the root of the resulting tree has rank $r + 1$ and the tree has at least $2^r + 2^r = 2^{r+1}$ nodes.

Consider all nodes of rank r. None of them is not and was not an ancestor of another, so there are 2^r nodes corresponding to each of them in the forest, and these sets are independent. As we have n nodes in total, the bound follows. ■

24.2 Buckets

Definition 24.2.1. *We define buckets so they form a partition of* $\mathbb{N}$ *with :* $B_0 = [0,0], B_1 = [1,1], B_2 = [2, 2^2 - 1], B_3 = \left[2^2, 2^{(2^2)} - 1\right]$ *and so on. They are not implemented, bu only used for the analysis. The total number of buckets is at most* $\log^\star n$.

Proposition 24.2.1. *Put a node with rank* r *into the bucket* B *containing* r*. The number of nodes in a bucket* $B = \left[b, 2^b - 1\right]$ *is at most :*

$$\sum_{r \in B} \frac{n}{2^r} \leq \frac{2n}{2^b}$$

Definition 24.2.2. *At a given time a node is good if :*

- *It is a root*
- *It is a child of a root*
- *It is in a smaller bucket than its parent*

It is bad otherwise.

A `find_set` visits $\mathcal{O}(\log^\star n)$ good nodes. Therefore, the total time complexity is $\mathcal{O}(m \log^\star n + |\{\text{bad nodes visited}\}|)$.

Proposition 24.2.2. *The total number of visits of bad nodes is* $\mathcal{O}(n \log^\star n)$

Proof. Bad nodes have frozen rank. When a bad node x is visited by a `find_set`, the new $x.p$ has rank strictly larger than the previous one. Hence, for each bad node x in a bucket $B = \left[b, 2^b - 1\right]$ the number of times x is visited while x is bad is at most $2^b - 1$, so by 24.2.1, it is at mode $\mathcal{O}\left(\frac{2n}{2^b}\right)$. So the number of times we visit a bad node in B is $\mathcal{O}(n)$. ■

Thus, the total complexity is $\mathcal{O}\left(m \log^\star n\right)$
In the following sections, we will try to improve this complexity.

24.3 Ackermann's function

Definition 24.3.1. *For* $k \geq 0$, $j \geq 1$ *two integers, the Ackermann function of* k *and* j *is defined as :*

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{j+1}(j) & \text{if } k \geq 1 \end{cases}$$

Proposition 24.3.1 (Properties).

- *Lemma 1 :* $\forall j \geq 1, A_1(j) = 2j + 1$
- *Lemma 2 :* $\forall j \geq 1, A_2(j) = 2^{j+1}(j+1) - 1$

Proof. Both are done by induction. ■

Definition 24.3.2. *We define reverse Ackermann's function by :*

$$\alpha(n) = \min\{k \mid A_k(1) \geq n\}$$

We have :

$$\alpha(n) = \begin{cases} 0 & \text{if } 0 \leq n \leq 2 \\ 1 & n = 3 \\ 2 & 4 \leq n \leq 7 \\ 3 & 8 \leq n \leq 2047 \\ 4 & 2048 \leq n \leq A_4(1) \end{cases}$$

24.4 Potential Function

We define a potential function :

Definition 24.4.1.

- $\texttt{level}(x) = \max\{k \mid x.p.rank \geq A_k(x.rank)\}$
- $\texttt{iter}(x) = \max\left\{i \mid x.p.rank \geq A^{(i)}_{\texttt{level}(x)}(x.rank)\right\}$

For a node x, we define :

$$\Phi_q(x) = \begin{cases} \alpha(n) \cdot x.rank & \text{if } x \text{ if root or } x.rank = 0 \\ (\alpha(n) - \texttt{level}(x)) \cdot x.rank - \texttt{iter}(x) & \text{otherwise} \end{cases}$$

For a forest, define :

$$\Phi_0(F) = 0 \text{ and } \Phi_q(F) = \sum_{x \in F} \Phi_q(x)$$

We will show that $\Phi_q(F) \geq 0$: Then, if we denote by t_q the time for the q-th operation :

$$\sum_q t_q \leq \sum_q t_q + \Phi_q(F) - \Phi_0(F) = \sum_q t_q + \sum_q (\Phi_{q+1}(F) - \Phi_q(F)) = \sum_q (t_q + \Phi_{q+1}(F) - \Phi_q(F))$$

Lemma 24.4.1. *If $x \neq x.p$ and $x.rank \geq 1$: $0 \leq \texttt{level}(x) \leq \alpha(n) - 1$*

Proof. We have $x.p.rank \geq x.rank + 1 = A_0(x.rank)$, hence $\texttt{level}(x) \geq 0$.
As $A_{\alpha(n)}(x.rank) \geq A_{\alpha(n)}(1) \geq n \geq x.p.rank$, we get the other inequality. ■

Lemma 24.4.2. *If $x.rank \geq 1$, $1 \leq \texttt{iter}(x) \leq x.rank$.*

Proof. We have $x.p.rank \geq A_{\texttt{level}(x)}(x.rank) = A^1_{\texttt{level}(x)}$ thus $\texttt{iter}(x) \geq 1$.
Since : $A^{x.rank+1}_{\texttt{level}(x)}(x.rank) = A_{\texttt{level}(x)+1}(x.rank) > x.p.rank$ and thus $\texttt{iter}(x) \leq x.rank$. ■

Lemma 24.4.3. $\forall x, q, 0 \leq \Phi_q(x) \leq \alpha(n)x.rank$

Proof. If x is the root or $x.rank = 0$, there is equality and we are done. Otherwise :

$\Phi_q(x) = (\alpha(n) - \texttt{level}(x)) \cdot x.rank - \texttt{iter}(x) \geq (\alpha(n) - (\alpha(n) - 1)) \cdot x.rank - \texttt{iter}(x) \geq x.rank - x.ra$

Also :

$$\Phi_q(x) \leq \alpha(n) \cdot x.rank - 1 < \alpha(n)x.rank$$

■

Corollary 24.4.3.1. *If x is not a root and $x.rank > 0$, then $\Phi_q(x) < \alpha(n)x.rank$*

Lemma 24.4.4. *If x is not a root, the q-th operation is `link` or `find_set` then $\Phi_q(x) \leq \Phi_{q-1}(x)$. Moreover, if $x.rank \geq 1$ and either `level`(x) or `iter`(x) changes, then $\Phi_q(x) \leq \Phi_{q-1}(x) - 1$.*

Proof. Since x is not a root, the rank of x is frozen.

1. If $x.rank = 0$ then $\Phi_q(x) = \Phi_{q-1}(x) = 0$
2. Assume $x.rank \geq 1$.
   - If `level`(x) increases, $(\alpha(n) - \texttt{level}(x))\,x.rank$ drops by at least $x.rank$ but `iter`(x) can drop by at most $x.rank - 1$ therefore, $\Phi_q(x) \leq \Phi_{q-1}(x) - 1$.
   - If `level`(x) is unchanged, `iter`(x) remains unchanged or increases. If it does not change $\Phi_q(x) = \Phi_{q-1}(x)$, otherwise, $\Phi_q(x) \leq \Phi_{q-1}(x) - 1$

■

24.5 Time Bound

More lemmas :

Lemma 24.5.1. *The amortised cost of each `make_set` is $\mathcal{O}(1)$*

Proof. `make_set` creates a node x with $\Phi_q(x) = 0$. Hence, $\Phi_q(F) = \Phi_{q-1}(F)$ and the amortised time is 1. ■

Lemma 24.5.2. *The amortised time of `link`(x, y) is $\mathcal{O}(\alpha(n))$.*

Proof. WLOG, y becomes the parent of x. By 24.4.4, the potentials of any node but x and y cannot increase. Before `link`, the node x is a root and $\Phi_{q-1}(x) = \alpha(n)x.rank$. If $x.rank = 0$, we have $\Phi_{q-1}(x) = \Phi_q(x) = 0$. Otherwise by 24.4.3.1, $\Phi_q(x) < \alpha(n)x.rank = \Phi_{q-1}(x)$.
For y, either $\Phi_{q-1}(y) = \Phi_q(y)$ or $\Phi_q(y) = \Phi_{q-1}(y) + \alpha(n)$.

Hence, $\Phi_q(F) - \Phi_{q-1}(F) \leq \alpha(n)$. The actual time of `link`$(x, y)$ is 1 and thus the lemma follows. ■

Lemma 24.5.3. *The amortised time of `find_set` is $\mathcal{O}(\alpha(n))$.*

Proof. Assume the find path contains s nodes :

1. There are no potential increases from 24.4.4.
2. The potential of at least $\max\{0, s - \alpha(n) - 2\}$ decreases by at least one. Consider all nodes x on the find path with rank at least 1 and such that there exists y an ancestor of x with the same `level`. There are at least $s - (\alpha(n) + 2)$ such nodes.

   Consider such a node x, let $k = \texttt{level}(x)$:

Prior to path compression, $x.p.rank \geq A_k^{\texttt{iter}(x)}(x.rank)$ and $y.p.rank \geq A_k(y.rank)$. Then, by monotony of $rank$ along a path, $y.rank \geq x.p.rank$. Let $i = \texttt{iter}(x)$. We have $y.p.rank \geq A_k\left(A_k^i(x.rank)\right)$

After path compression, x and y have the same parent, the root so $x.p.rank = y.p.rank \geq A_k^{i+1}(x.rank)$. since $x.rank$ does not change. Therefore, either $\texttt{iter}(x)$ or $\texttt{level}(x)$ increases, and hence, $\Phi_q(x) \leq \Phi_{q-1}(x) - 1$ from 24.4.4.

As a corollary, we obtain that $\Phi_q(F) \leq \Phi_{q-1}(F) - \max\{0, s - (\alpha(n) + 2)\}$. Since the actual time of `find_path` is s, the amortised time is $\mathcal{O}(\alpha(n) + 2)$. ■

Finally :

Theorem 24.5.4. *A sequence of m `make_set`, `union_set`, `find_set`, out of which n are `make_set` takes $\mathcal{O}(m\alpha(n))$ time. In other words, one operation takes $\mathcal{O}(\alpha(n))$ amortised time.*

Part VII

Lecture 7 : 16/11

Contents

25 Definitions

All along the course :

Definition 25.0.1.
- *n is the number of vertices*
- *m is the number of edges*
- *An algorithm in $\mathcal{O}(n+m)$ is said to be linear*

Table 1: Some Graph Parameters

$\delta(G)$	minimum degree
$\Delta(G)$	maximum degree
$\omega(G)$	clique number
$\alpha(G)$	size of a maximum independent set
$\chi(G)$	chromatic number
$\tau(G)$	vertex cover
$\kappa(G)$	vertex connectivity
$\lambda(G)$	edge connectivit
$\mu(G)$	size of a maximum matching
Girth	length of a shortest cycle

- *K_n is the complete graph on n vertices*
- *$G = (U, V, E)$ is a bipartite graph means U and V is the partition.*
- *$K_{a,b}$ denotes the complete bipartite graph with parts of size a and b.*
- *H is a subgraph of G if H can be obtained from G by deleting vertices and edges.*
- *H is a induced subgraph of G if H can be obtained from G by deleting vertices.*
- *H is a subdivision of G if H can be obtained from G by subdividing some edges.*
- *H is a minor of G if H can be obtained from G by deleting vertices, edges and contracting edges.*
- *A sink is a vertex of outdegree 0*

Definition 25.0.2 (Generic Graph Search)**.** *The problem is to visit the vertices following the edges of the graph. A search also outputs a search tree, and an ordering on the vertices.*

26 BFS, DFS and Graph Encoding

26.1 Properties

Definition 26.1.1. *We will denote by $v.d$ the time at which a vertex is discovered, and $v.f$ the time at which a vertex is finished, i.e. all of its neighbour are discovered.*

Theorem 26.1.1 (Parenthesis Theorem)**.** *In any DFS of a graph, for any two vertices u and v either :*

- *$[u.d, u.f] \cap [v.d, v.f] = \varnothing$ and u is not a descendant nor an ancestor of v.*
- *$[u.d, u.f] \subset [v.d, v.f]$ and u is a descendant of v*

- $[v.d, v.f] \subset [u.d, u.f]$ *and* v *is a descendant of* u

Corollary 26.1.1.1. *A vertex* v *is a descendant of* u *in the DFS forest if and only if* $u.d < v.d < v.f < u.f$.

Theorem 26.1.2 (White-path Theorem)**.** *In a DFS forest of a digraph, a vertex* v *is a descendant of a vertex* u *if and only if at time* $u.d$*, there is a* (u, v)*-path made of undiscovered vertices.*

Definition 26.1.2. *We define arc types in terms of the DFS forest* G_π *:*

- *Tree arcs are arcs of* G_π
- *Forward arcs are arcs* uv *such that* u *in an ancestor of* v
- *Back arcs are arcs* uv *such that* u *is a descendant of* v
- *Cross arc are arcs* uv *such that* u *is not an ancestor of* v *and* v *is not an ancestor of* u.

Proposition 26.1.1. *A digraph has a directed cycle if and only (any) DFS produces a back arc.*

26.2 Topological Sort

Definition 26.2.1. *A topological ordering of the vertices of a digraph is a labeling* f *such that* $uv \in Eonlyiff(u) < f(v)$.

Theorem 26.2.1. *G has a topological ordering if and only if it is a DAG.*

Theorem 26.2.2. *The problem of finding a topological is solved in linear time using a DFS.*

Lemma 26.2.3. *A digraph is acyclic if and only if a DFS of the graph yields no back edges.*

26.3 Strongly Connected Components

Definition 26.3.1. *A strong connected component (scc) of a directed graph G is a maximum set of vertices such that every pair of vertices as a directed path from the first to the second and from the second to the first.*
We define G^{SCC} *the quotient graph for* $u\mathcal{R}v \Leftrightarrow u \rightsquigarrow v \wedge v \rightsquigarrow u$.

Proposition 26.3.1. G^{SCC} *is a DAG.*

Lemma 26.3.1. *Let* C, $C^{'}$ *be two scc of a digraph. Let* $u, v \in C$ *and* $u^{'}, v^{'} \in C^{'}$. *If* $u \rightsquigarrow u^{'}$ *then there is no path from* $v^{'}$ *to* v.

Definition 26.3.2. *We extend the discovery and finishing times to sets of vertices :*

- $d(S) = \min \{u.d \mid u \in S\}$

- $f(S) = \max\{u.f \mid u \in S\}$

Lemma 26.3.2. *Let C and C' be two scc and let $u \in C$, $v \in C'$ such that $uv \in E$. Then $f(C) > f(C')$.*

Definition 26.3.3. *We define $G^R = \left(V, E^R\right)$ where $E^R = \{uv \mid vu \in E\}$.*

Proposition 26.3.2. *G^R has the same scc as G*

Algorithm 12 Kosaraju's Two Pass Algorithm

```
Call DFS($G$)
Compute $G^R$
Call DFS($G^R$) considering vertices in order of decreasing $u.f$ return Output the
vertices of each tree in the DFS forest computed by DFS($G^R$)
```

Theorem 26.3.3. *Kosaraju's two pass algorithm computes the scc in linear-time.*

27 Minimum Weighted Spanning Tree - MWST

27.1 Trees

Definition 27.1.1. *A tree is a connected acyclic graph. A forest is an acyclic graph. A leaf of a tree is a vertex of degree* 1.

Proposition 27.1.1.
- *Every tree contains at least two leaves*
- *Every tree T satifies : $|E(T)| = |V(T)| - 1$.*
- *Every subgraph of a tree is a forest.*

Proposition 27.1.2. *Let $G = (V, E)$ be a graph. The following are equivalent :*

- *G is a tree*
- *Any two vertices in G are linked by a unique path*
- *G is connected and $|E| = |V| - 1$*
- *G is acyclic and $|E| = |V| - 1$*
- *G is connected but for every edge uv, $G - \{uv\}$ has exactly two connected components, one containg u and the other v.*
- *G is acyclic but if any edge is added to G, the resulting graph contains a unique cycle going through this added edge.*

Definition 27.1.2. *$\omega : E \to \mathbb{R}$ is called a weight function. A spanning tree is a subgraph of G that is a tree and contains all vertices of G. The weight of a subgraph is the sum of the weights of its edges.*

Proposition 27.1.3 (Cut and Paste Technic)**.** *Let $G = (V, E)$ and let T be a spanning tree of G. Let $uv \in E(G) - T$ and let T_{uv} be the unique path linking u and v in T. Then for every edge xy of T_{uv} $T \setminus \{xy\} \cup \{uv\}$ is a spanning tree of T.*

Proof. Since T_{uv} is in the unique path of T linking u and v, removing any edge $xy \in T_{uv}$ from T breaks T into two connected components, one containing u and the other v. Adding uv reconnects the two parts and form a new spanning tree $T \setminus \{xy\} \cup \{uv\}$. ■

Theorem 27.1.1. *Let $G = (V, E, \omega)$. A spanning tree T of G is a minimum spanning tree if and only if for every edge $e \in E \setminus T$:*

$$\omega(e) \geq \omega(f) \forall f \in \textit{ the unique cycle of } T \cup \{e\}$$

Definition 27.1.3. *A set of edges A is said to be promising if it can be completed into a minimum spanning tree. At each step, we determine an edge uv that we can add to A without violating this invariant, that is $A \cup \{uv\}$ is still a promising set. Such an edge uv is said to be safe with respect to A.*

The algorithm works as follows : We start with an empty set, and while our set is not a spanning tree, we find a safe edge and add it to the set. How to find safe edges ?

Definition 27.1.4. *A cut $(S, V - S)$ is a partition of V. An edge of the cut or crossing edge is an edge with one end in S and the other in $V - S$. A cut respects a set of edges if no edge of the set is a crossing edge of the cut.*

Proposition 27.1.4. *Let A be a promising set of edges. Let $(S, V - S)$ be a cut respecting A and uv be a lightest crossing edge. Then uv is safe.*

Proof. Let T be a MST containing A. If T contains uv we are done, so assume it does not. Let T_{uv} be the unique path linking u and v in T. Since uv is a crossing edge of $(S, V - S)$, T_{uv} contains an edge xy of $(S, V - S)$. By the cut and paste technique, $T' = (T \setminus \{xy\}) \cup \{uv\}$ is a spanning tree of G. By hypothesis, uv is a lightest edge of $(S, V - S)$, so $\omega(uv) \leq \omega(xy)$ and thus:

$$\omega(T') = \omega(T) + \omega(uv) - \omega(xy) \leq \omega(T)$$

So T' is a MST containing uv ■

27.2 KRUSKAAAAAAAAL

Kruskal algorithm grows a promising forest A :

- Sort the edges by non-decreasing order of weight
- Start with the empty set
- Add a minimum weighted edge that does not create a cycle which is equivalent to add a minimum weighted edge that connects two connected components of the growing forest.

Algorithm 13 Kruskal

```
Input  $G = (V, E, \omega)$
$A \leftarrow \varnothing$
for $u \in V$ do
    Make-Set($u$)
end for
Sort($E$)                              ▷ Sort by non-decreasing order of weight
for $uv \in E$ do
    if Find($u$) $\neq$ Find($v$) then
        $A \leftarrow A \cup \{uv\}$
        Union($u, v$)
    end if
end forreturn   A
```

Theorem 27.2.1. *The Kruskal algorithm answers the problem in $\mathcal{O}(m\log(n))$ time*

Proof.
- Complexity : from union-find, we do $\mathcal{O}(m)$ `Find-Set` and $\mathcal{O}(n)$ `Union` and `Make-Set` in $\mathcal{O}(m+n)\alpha(n)^4$. As we do one sort, the total complexity is $\mathcal{O}(m\log(n))$.
- Correctness : Comes from the safe property, by considering the loop invariant : Prior to each iteration, A is promising.

■

27.3 Prim

This algorithm works like Kruskal's, but instead of growing a forest, we grow a tree A. At each iteration, we add the lightest edge with exactly one extremity in $V(A)$.

Definition 27.3.1. *A Priority Queue is a data structure that maintains a set S of elements, each with an associated value called a key. TIt supports the following :*

- *`Insert`(S, x)*
- *`Minimum`(S)*
- *`Extract-Min`(S)*
- *`Decrease-Key`(S, x, k) does $x.key \leftarrow k$.*

Remark 27.3.0.1. *This is for a min-queue, we can also do a max-queue.*

We implement priority queues using a heap.

Definition 27.3.2. *A max-heap is a complete binary tree where the key of a node is larger than the keys of its children. It can be implemented in an array : `Parent`$(i) = \lfloor \frac{i}{2} \rfloor$, `Left`$(i) = 2i+1$, `Right`$(i) = 2i+2$. It supports the same operations as a max-queue.*

Theorem 27.3.1. *Using a binary heap, insertion is done in* $\log N$*, max-retrieval is done in constant time and max-deletion is done in* $\log N$*.*

Proof. See implementation. ■

Algorithm 14 PRIM

```
**Input** $G = (V, E, \omega), r$
**for** $v \in V$ **do**
    $v.\pi \leftarrow$ `NIL` and $v.key \leftarrow +\infty$
**end for**
$r.key \leftarrow 0$
$Q \leftarrow V$
**while** $Q \neq \varnothing$ **do**
    $u \leftarrow$ `Extract-Min`$(Q)$
    **for do**$v \in$ `Adj`$[u]$
        **if then**$v \in Q \wedge v.key > \omega(uv)$
            $v.key \leftarrow \omega(uv)$
            $v.\pi \leftarrow u$
        **end if**
    **end for**
**end while**
```

Theorem 27.3.2. *This algorithm solves the problem in* $\mathcal{O}(m + n\log n)$*.*

Proof.
- Correctness comes from the safe property
- Complexity is in $\mathcal{O}(m \log n)$ with a min-heap and $\mathcal{O}(m + n\log n)$ with a Fibonacci Heap

■

28 Matroids and Greedy Algorithms

28.1 Definitions

Definition 28.1.1. *Let* E *be a set of elements and* $\mathcal{I} \subseteq 2^E$*.* $(E, \mathcal{I})$ *is a hereditary set system if it satisfies*

- *M1 :* $\varnothing \in \mathcal{I}$
- *M2 : If* $X \subseteq Y \in \mathcal{I}$*,* $X \in \mathcal{I}$*.*
- E *is called the ground set.*
- *Sets in* $\mathcal{I}$ *are called independent sets.*
- *Maximal independent sets are called bases.*

- *For $X \subseteq E$, the rank of X denoted by $rk(X)$ is the size of a maximum independent set included in X.*
- *Sets not in $\mathcal{I}$ are called dependent sets.*
- *Minimal dependent sets are called circuit.*

Given a positive weight function, we want to find a maximum independent set such that its weight is maximum. For example :

- MST : $\mathcal{I} = \{I \subseteq E \mid I \text{ is a forest } \}$
- TSP, SPP, Maximum Matching Problem, Maximum Stable Set, Knapsack are also of the sort.

Theorem 28.1.1 (Rado - Edmonds)**.** *The greedy algorithm for this problem is optimal for any weight function if and only if $(E, \mathcal{I})$ is a matroid.*

Definition 28.1.2. *A matroid is a hereditary set system $E, \mathcal{I}$ such that : if $X, Y \in \mathcal{I}$ and $|X| > |Y|$, there is $x \in X \setminus Y$ such that $Y \cup \{x\} \in \mathcal{I}$*

Part VIII

Lecture 9 : 21/12

29 Complexity

29.1 Definitions

We usually aim for poly-time algorithm in worst case. Since many problems of interest are NP-hard, we want to find better complexity than $\mathcal{O}(c^n)$ for them, as poly-time in unlikely.
For NP-hard problems, there is no algorithm that solves all instances optimally in poly-time. We will now look to measure the complexity in term of the input size and a parameter of the input.

For example, Vertex Cover is NP-complete, and looks for a graph G and an integer k, is it possible to cover the edges with k vertices ? In Brute-Force, there are $\mathcal{O}(n^k)$ possibilities but a $\mathcal{O}(2^k n^2)$ algorithm exists. Yet, for k-Independent Set, there are no $n^{o(k)}$ algorithm known.

Definition 29.1.1. *A parametrized algorithmic problem is a problem where a certain parameter k is given in addition to the input. The complexity is studied as a function of n and k. k can be the size of the solution, or an implicit parameter of the input graph.*

We then have three main possibilities :

1. Either the problem is already hard for fixed k : $\chi(G) \leq k$.

2. Or the problem is NP-hard for k in the input but polynomial for fixed k : $\alpha(G) \leq k$. We say the problem is XP

3. Or it is FPT for k, there is an algorithm in $\mathcal{O}(f(k) \times n^{\mathcal{O}(1)})$

Formally :

Definition 29.1.2. *Let Σ be a finite alphabet used to encode the input. A parametrized problem is a set $\mathcal{P} \subseteq \Sigma^\star \times \mathbb{N}$. The set $\mathcal{P}$ contains (x, k) if the answer to the question encoded by x, k is* `true`*. k is the parameter.*

Definition 29.1.3.

- *A parametrized problem is Fixed-Parameter Tractable if there is an algorithm deciding $\mathcal{P}$ in time $f(k)n^c$ where f is computable and c is constant.*
- *A parametrize problem $\mathcal{P}$ is XP if there is an algorithm deciding $\mathcal{P}$ in time $n^{f(k)}$ for some computable f and constanc c.*

The W hierarchy is a hierarchy inside of XP that looks at the parametrized problems in the same way as P hierarchy.

Open Problem 5. *n-variable 3-SAT cannot be solved in time $2^{o(n)}$*

29.2 Clique by Degree

We define :

Definition 29.2.1. *Given a graph G with maximum degree Δ and an integer k, does G have a clique of size at least k ?*

This is FPT since looking at the neighbourhoods of each vertex gives us $\mathcal{O}(2^\Delta n)$ time complexity. There might be no FPT solution for Clique parametrized by k.

30 Algorithmic Techniques

30.1 Branching Method

30.1.1 k-Vertex Cover.

Definition 30.1.1. *The main idea of the branching method is to reduce the problem to solving a bounded number of problems with parameter $k' < k$. Since the parameter decreases in every recursive call, the depth of the search tree is at most k. The size of the search tree branching into c directions in c^k.*
We denote $\mathcal{O}(\alpha^k n^{\mathcal{O}(1)})$ by $\mathcal{O}^\star(\alpha^k)$

Then, for Vertex Cover we branch on vertices of degree at least 3 :

- For each vertex u of degree at least 3, either u is in the solution, or all the neighbours of u are in the solution.

- If every vertex has degree at most 2, we can solve Vertex Cover in poly-time because the graph is the disjoint union of paths and cycles. These graphs will be the leaves of our search-tree.

Proposition 30.1.1. *The above algorithm gives us a solution to Vertex Cover in $\mathcal{O}^\star(1.4656^k)$*

Proof. Let $T(k)$ be the number of leaves in the search tree and $T(k) = 0$ if $k \leq 1$. We have $T(k) \leq T(k-1) + T(k-3)$. We want to find $c \geq 1$ such that $T(k) \leq c^k$ for all k.
We see that c such that $c^k \geq c^{k-1} + c^{k-3}$. In particular, $c^3 - c^2 - 1 \geq 0$. Then $c = 1.4656$ is valid. ■

Remark 30.1.0.1. *There exists an algorithm in $\mathcal{O}^\star(1.2738^k)$.*

Definition 30.1.2. *We define the branching vector of an algorithm as the vector with the decrease parameters.*

Here it was $(1, 3)$. If we find an algorithm that branches for $(2, 5, 6, 6, 7, 7)$, we will get $\mathcal{O}^\star(1.4483^k)$

30.1.2 k-Feedback Vertex Set

Definition 30.1.3. *Given a multigraph G and an integer k, find a set S of at most k vertices such that $G \setminus S$ is a forest.*

A feedback vertex set is a set of vertices that hits every cycle of the graph.
Here, at least one vertex of each cycle must be in the solution, but the size of a cycle can be arbitrarily large. We are going to identify a set of $\mathcal{O}(k)$ vertices such that any size k-FVS has to contain one of these vertices and branch on it. To do so, we need reduction rules.

Proposition 30.1.2 (Reduction Rules for FVS).
1. *If there is a loop at v, then delete v and decrease k by one.*
2. *If there is an edge of multiplicity large than 2, reduce its multiplicity to 2*
3. *If there is a vertex v of degree ≤ 1, delete it.*
4. *If there is a vertex v of degree 2, delete v and add an edge between the neighbours of v.*

Applying the reduction rules returns a graph G', k' with no loops, edge multiplicty $>$ 2, minimum degree 3.
We have G, k a YES instance if and only if G', k' is a YES instance.

Lemma 30.1.1. *Let G be a graph with minimum degree 3, let V_{3k} be the $3k$ largest degree vertices. Then every FVS of size at least k contains at least a vertex of V_{3k}.*

Proof. Suppose there is S a solution disjoint frome V_{3k} and let $d = \min d(V_{3k})$. Let $X = V(G) \setminus (S \cup V_{3k})$.

$$\sum_{v \in X \sqcup V_{3k}} d(v) \geq 3\,|X| + 3kd$$

But $G[X \cup V_{3k}]$ is a forest so its number of edges is $\leq |X| + 3k - 1$. So the sum of degrees in it is at most $2\,|X| + 6k - 2$.
Thus we get the number of edges to be :

- $\geq 3kd + 3\,|X| - (2\,|X| + -k - 2) > 3kd - 6k$
- $\leq dk$ because S has k vertices of degree at most d.

So we would have $2kd - 6k < 0$ which is impossible since $d \geq 3$. ■

We then get an algorithm to compute a solution by applying the reduction rules and branching (in polynomial time) on each vertex in V_{3k}. Since we branch into $3k$ directions, we get a time complexity in $\mathcal{O}^\star\left((3k)^k\right)$

30.2 Kernelization

30.2.1 Definitions

Kernelization is a method for parametrized preprocessing. We want to reduce the size of the instance (x, k) to an equivalent instance with size bounded by $f(k)$.

Definition 30.2.1. *Let $\mathcal{P}$ be a parametrized problem and f a computable function. A kernel of size $f(k)$ is an algorithm that, given (x, k), runs in polynomial-time in $|x| + k$ and outputs an instance x', k' such that :*

- $x, k \in \mathcal{P} \Leftrightarrow x', k' \in \mathcal{P}$
- $\left|x'\right| \leq f(k)$ *and* $k' \leq k$.

We say the kernel is polynomial is f is polynomial.

Theorem 30.2.1. *A parametrized problem is FPT if and only if it is decidable and has a kernel.*

Proof.
- If the problem has a kernel, we reduce the size of the instance in poly-time and bruteforce on it.
- If the problem is solved in $f(k) \cdot |x|^c$.
  If $|x| \leq f(k)$ then we have our kernel. Else, if $|x| \geq f(k)$ then we can solve the problem in time $f(k) \cdot |x|^c \leq |x|^{c+1}$

■

The real issue is to know which problems have polynomial kernels.

30.2.2 Vertex Cover

We define two reduction rules for k-Vertex Cover :

Proposition 30.2.1.
- *If v has degree 0, then reduce to $(G-v,k)$*
- *If v has degree $k+1$, then reduce to $(G-v,k-1)$.*

Lemma 30.2.2. *If (G,k) is a YES instance for k-Vertex Cover on which reduction rules cannot be applied, G has at most k^2 edges and k^2+k vertices.*

Proof. Let S be a vertex cover of G of size at most k. Each vertex hits at most k edges because of the second rule. So there is at most k^2 edges. Each vertex is either in S or is one of the at most k neighbours of a vertex in S so there are at most k^2+k vertices. ∎

Proposition 30.2.2 (Kernelization for Vertex Cover).
- *Apply reduction rules exhaustively.*
- *We get an equivalent instance $(G^{'},k')$ with $k' \leq k$.*
- *If $|E(g)| > k'^2$ $|V|(G) > k'^2 + k'$ return NO*
- *Otherwise we have a kernel of size $\mathcal{O}(2k^2+k)$.*

Theorem 30.2.3. *Vertex Cover has a kernel with at most $2k$ vertices.*

This is done using Linear Programming.

30.3 Color Coding

ENVIE DE MOURIR NIQUE L'ALEATOIRE

30.3.1 k-Path

Definition 30.3.1. *Given G,k, decide if G contains a path on k vertices.*

We will use a randomize[1] algorithm that can be determinized.
We transform the problem into the following :

- Assume the vertices are colored randomly with $\{1,\dots,k\}$.
- Find a path colored $1-2-\cdots-k$.

The algorithm is the following :

- Assign color from $[k]$ to the vertices uniformly and independently at random
- Output YES if there is a path colored $1-2-\cdots-k$ else NO

[1]BOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO

Proposition 30.3.1. *If G has a k-path, the probability that this k-path is colored $1 - 2 - \cdots - k$ is $1/k^k$. So if G is a YES instance the algo outputs YES with probability at least $1/(k^k)$, and if it is a NO instance, the algorithm outputs NO. Then, after k repetitions, error probability is at most $1/e$.*

To find a colored path, we delete edges in the graph from vertices colored i to vertices colored j for $j \neq i-1, i+1$ and orient edges towards the larger class. We can find a path in linear time with BFS.

Proposition 30.3.2. *The final complexity of this algorithm is $\mathcal{O}(c \cdot k^k \cdot (n+m))$ and it has probability of success $1/e^c$.*

We can still improve color coding by solving the existence of a colorful path instead of the existence of an ordered coloured path. Indeed, if there is k-path, the probability that is it colorful is $> e^{-k}$. We now only need to solve the problem e^k times instead of k^k.
We can solve this problem using Dynamic Programming : we solve for each vertex v and set of color $C \subseteq [\![1, k]\!]$, the value of $D(v, C)$ that is YES is there is path ending at v using each colour of C.
If we denote by $\chi : V \to [k]$ the random coloring, $D(v, C)$ is YES if and only if $\chi(v) \in C$ and there is an edge uv for which $D(u, C \setminus \chi(v))$ is YES. We thus can solve this in time $2^k \times |E|$.

Finally we get the following algorithm :

Proposition 30.3.3. *Repeat e^k times :*

1. *Sample a coloring c*
2. *Check if G contains a colorful k-path in time $\mathcal{O}(2^k \cdot |E|)$ and return YES if it does.*

If no colorful k-path was found, return NO.

FINALLY :

30.3.2 Derandomization

Definition 30.3.2. *A family $\mathcal{H}$ of function $[\![1, n]\!] \to [\![1, k]\!]$ is a k-perfect family of hash functions if for every $S \subseteq [\![1, n]\!]$ with $|S| = k$, there is an $h \in \mathcal{H}$ such that $h(x) \neq h(y)$ for any $x \neq y \in S$.*

Theorem 30.3.1. *There is a k-perfect family of functions $[\![1, n]\!] \to [\![1, k]\!]$ with size $2^{\mathcal{O}(k)} \log n$ and can be constructed in poly-time in the size of the family.*

Then, instead of trying $\mathcal{O}(e^k)$ random colorings, we go through a k-perfect family $\mathcal{H}$ of functions $[\![1, n]\!] \to [\![1, k]\!]$. Then, k-Path can be solved in deterministic time $2^{\mathcal{O}(k)} \cdot n^{\mathcal{O}(1)}$.

Part IX

Lecture 10 : 11/01

31 Definitions

Suppose we want to solve an optimization problem $\mathcal{P}$. Say that a solution has a certain cost we want to minimize. Denote by $OPT(P)$ the cost of an optimal solution of $\mathcal{P}$. Suppose we have an algorithm that outputs a non-optimal solution of cost $ALG(P)$. ALG has approximation ratio $\rho(n)$ if for any instance P of size n we have :

$$ALG(P) \leq \rho(n) \cdot OPT(P)$$

Definition 31.0.1. *An algorithm has an approximation ratio of $\rho(n)$ if for any input of size n the cost ALG of the solution produced by the algorithm is with a factor of $\rho(n)$ of the cost OPT of an optimal solution :*

$$1 \leq \max\left(\frac{OPT}{ALG}, \frac{ALG}{OPT}\right) \leq \rho(n)$$

Many NP-Hard problems have small constant approximation ratio. Others have approximation ratio that grows with n. When the best approximation ration grows linearly, there is trouble.

An approximation scheme is an algorithm scheme is an algorithm that outputs a $(1+\varepsilon)$-approximation where ε is part of the input.

Definition 31.0.2. *An algorithm is a Polynomial Time Approximation Scheme if it satifies*

- *Input : An instance P and $\varepsilon > 0$*
- *Output : A $(1+\varepsilon)$-approximation*
- *Polynomial in n for every fixed ε*

We say it is a Fully Polynomial Time Approximation Scheme if it is polynomial in n and $1/\varepsilon$.

31.1 Min Vertex-Cover

Algorithm 15 Naïve Algorithm

```
For each vertex $v$
if $d(v) \geq 1$ then
    Put $v$ in the solution
    Delete $v$
end if
```

Algorithm 16 Greedy Algorithm

```
while there are edges do
    Choose Vertex with Maximum Degree
    Delete It
end while
```

Yet, for a star graph, the algorithm returns a set of size $n-1$ while there is a solution of size 1.

We need to find a lower bound of an optimal solution :
Assume there is a certain quantity Q such that $Q \leq OPT$. If we can prove that $ALG \leq c \times Q$ then ALG is a c-approximation. Here, a Vertex Cover is at least as large as a maximal matching. We get :

Algorithm 17 Based on Matching

```
$M \leftarrow$ Maximal Matching
while there are edges do
    Add $u$ and $v$ to the solution
    Delete $u$ and $v$
end while
```

Here, we have a 2-approximation algorithm. We need to see if we can improve the approximation ratio. It is tight from looking at $K_{n,n}$.
State of the art, the best approx algorithm known has approximation factor $\left(2 - \Theta\left(\frac{1}{\sqrt{\log n}}\right)\right)$.
If Unique Game Conjecture is true, we cannot find better that 2 for a constant factor.

31.2 Travelling-Salesman Problem

Definition 31.2.1. *TSP asks the question of given a list of cities and distances between each pair of cities, find a hamiltonian cycle (that goes every vertex exactly once) of minimum cost ?*

We can assume WLOG that the graph is complete. We denote by $c(A)$ the weight of the path A.

Theorem 31.2.1. *For every constant $\rho \geq 0$, there is no polynomial-time algorithm for TSP with approximation ration ρ, unless $P = NP$*

Proof. Suppose there is an algorithm ALG that computes a ρ-approximation of TSP in polynomial-time on every input G.
Then, let G be an instance for Hamiltonian Path. We transform G intro an instance for TSP by taking a complete graph on $V(G)$ with cost

$$c(uv) = \begin{cases} 1 & \text{if } uv \in E(G) \\ \rho\,|V| & \text{otherwise} \end{cases}$$

Let OPT be the cost of an optimal tour of G', c. Observe that, if G has an hamiltonian cycle, then $OPT = |V(G)|$, else $OPT > \rho \times |V(G)|$. ∎

Definition 31.2.2. *Metric Travelling-Salesman Problem is TSP where the weight function c of the graph is a metric.*

Theorem 31.2.2. *MTSP has a 2-approximation algorithm.*

Proof. We have $cTSP > cMST$. Indeed, removing an edge TSP gives a spanning tree. This is a two approx : If T is a ■

Algorithm 18 Metric TSP using MSTs

```
  $M \leftarrow MST$ (using Prim or Kruskal)
  return Preorder Walk of $M$
```

31.3 Set-Covering

Definition 31.3.1. *Given a hypergraph $(V, \mathcal{F}, c)$ where V is finite a set of elements, $\mathcal{F}$ is a set of subsets of V and c is a cost function, output a cost minimum subset of $\mathcal{F}$ that covers all elements of V.*

First, a greedy algorithm. Here, the criteria is basing on the number of uncovered vertices.
Yet, if there is a set in $\mathcal{F}$ of weight 10^{10} that is covered by two sets of weight 1,

Algorithm 19 Greedy Algorithm

```
  $U \leftarrow V$
  $\mathcal{C} \leftarrow \varnothing$
  while $U \neq \varnothing$ do
      Select $S \in \mathcal{F}$ that maximizes $|S \cap U|$
      $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$
      $U \leftarrow U \setminus S$
  end while
```

the algorithm will choose the first option. So, we modify the greedy criteria to act on the average cost of newly covered vertices : $c(S)/|S \cap U|$

Algorithm 20 Greedy Set-Cover

```
  $U \leftarrow V$
  $\mathcal{C} \leftarrow \varnothing$
  while $U \neq \varnothing$ do
      Select $S \in \mathcal{F}$ that minimizes $\frac{c(S)}{|S \cap U|}$
      $\mathcal{C} \leftarrow \mathcal{C} \cup \{S\}$
      $U \leftarrow U \setminus S$
  end while
```

Theorem 31.3.1. *Greedy-Set-Cover is a $\log(n)$-approximation.*

Proof. Define $price(v) = c(S)/|S \cap U|$ where S is the hyperedge that covers v for the first time.
The total cost of a solution is $\sum_{v \in V} price(v)$. Number the elements of V by order of choose, with v_n being the first chose one. We have :

Lemma 31.3.2. *For each $k \in [\![1, n]\!], price(v_k) \leq OPT/k$*

Proof. Assume the algorithm is going to choose a set S that is going to cover v_k for the first time. Let U be the set of uncovered elements at this moment. So $\{v_k, \dots, v_1\} = U$. Since an optimal solution covers U with at most OPT, there must be a set S with cost-effectiveness at most $OPT/|U|$. Hence, by the greedy price criteria

$$price(v_k) \leq \frac{OPT}{|U|} \leq \frac{OPT}{k}$$

■

Hence, the total cost is :

$$\sum_{k=1}^{n} price(v_k) \leq OPT \sum_{k=1}^{n} \frac{1}{k} \leq OPT \cdot (\log n + 1)$$

■

Theorem 31.3.3. *Greedy-Set-Cover does not have an approximation ration of $(1 - o(1)) \ln n$.*

31.4 Layering and Weighted Vertex-Cover

Definition 31.4.1. *Given a graph and a weight function, find a minimum weight vertex cover.*

Definition 31.4.2. *A degree weighted function of a graph G is a function $\omega : V \to \mathbb{N}$ such that*

$$\exists c, \forall v, \omega(v) \leq c \cdot \deg v$$

Lemma 31.4.1. *Let G be a graph with weight function ω. If ω is a degree weighted function, then $\omega(V) \leq 2 \cdot OPT$*

Proof. We have :

$$\omega(V) = \sum_{v \in V} \omega(v) \leq \sum_{v \in V} c \deg(v) \leq 2c |E|$$

If $S^\star$ is an optimal solution,

$$OPT = \omega(S^\star) = \sum_{v \in S^\star} \omega(v) = c \cdot \sum_{v \in S^\star} \deg v$$

Since $S^\star$ covers all edges we have

$$\sum_{v \in S^\star} \deg v \geq |E|$$

■

Algorithm 21 Layering Algorithm(G, ω)

```
while $G \neq \varnothing$ do
    Remove $D = \{v \mid \deg v = 0\}$
    Compute $c = \min\{\omega / \deg(v)\}$
    Compute degree-weighted function $t = c \cdot \deg(v)$
    Compute the residual weight function $\omega_1 = \omega - t = \omega - c \cdot \deg$
    Pick the set $W = \{v \mid \omega_1(v) = 0\}$
    $G_1 = G - (D \cup W)$
    $G \leftarrow G_1, \omega \leftarrow \omega_1$.
end while
```

Theorem 31.4.2. *The layering algorithm is a 2-approximation*

Proof. It is clearly correct.
Let $S = W_0 \cup \ldots \cup W_k$ be the solution output by the algorithm. Observe that $\omega_l(v) = \omega_0(v) - \sum_{i=0}^{l-1} t_i(v)$. Let $v \in S$. If $v \in W_l$, $w_0(v) = \sum_{i=0}^{l-1} t_i(v)$.
Let $S^\star$ be an optimal vertex cover of G. For each layer i, $S^\star \cap V(G_i)$ is a vertex cover of G_i. Therefore by the degree weighted function Lemma :

$$t_i\left(S \cap V(G_i)\right) \leq 2t_i\left(S^\star \cap V(G_i)\right)$$

Hence :

$$\omega(S) = \sum_{i=0}^{k} t_i\left(S \cap V(G_i)\right) \leq 2\sum_{i=0}^{k} t_i(S^\star \cap V(G_i)) \leq 2\omega(S^\star)$$

■

Part X

Lecture 11 : 18/01

32 Definition

32.1 Linear Programming

Definition 32.1.1. *A system of linear equation is of the form :*

$$\begin{aligned} a_{11}x_1 + \ldots + a_{1n}x_n &= b_1 \\ \vdots + \ldots + \vdots &= \vdots \\ a_{n1}x_1 + \ldots + a_{nn}x_n &= b_n \end{aligned}$$

We write it as $Ax = b$ where A is a matrix.

Definition 32.1.2. *A linear program is made of n decision variables $x_1, \ldots, x_n \in \mathbb{R}$, m linear constraints*

$$\sum_{j=1}^{n} a_{ij} \star b_i$$

where $\star \in \{\leq, \geq, =\}$; and a function we want to maximise.

Proposition 32.1.1. *Linear Programming and Linear Algebra work in a similar way :*

	Basic Problem	*Algorithm*	*Solution Set*
Linear Algebra	*System of Linear Equations*	*Gaussian Elimination*	*Affine Subspace*
Linear Programming	*System of Linear inequalities*	*Simplex Method*	*Convex Polyhedron*

Linear Programming allows to model most of the problems we have seen in this class. Despite this generality, Linear Programs can be solved efficiently, both in theory (weak polynomial time) and in practice.

Definition 32.1.3. *If we allow the variable to be only in* $\mathbb{N}$ *(instead of* $\mathbb{R}$*), we call it Integer Linear Programming and it becomes NP-hard.*

32.2 First Example

Suppose a factory makes tables and chairs. Each table returns a profit of 200$ and each chair a profit of $100. Each table takes 1 unit of metal and 3 units of wood and each chair takes 2 units of metal and 1 unit of wood. The factory has $6k$ units of metal and $9k$ units of wood. We want to maximize profit.
We can model this problem using linear Programming, let x_1 be the number of tables and x_2 be the number of chairs

$$\begin{aligned} 3x_1 + x_2 &\leq 9 \\ x_1 + 2x_2 &\leq 6 \\ \text{Maximize } 2x_1 + x_2 & \end{aligned}$$

32.3 Flow

Proposition 32.3.1. *We are given* $G = (V, E, s, t, c)$ *with:*

- *A directed Graph* $G = V, E$
- *A source vertex* $s \in V$
- *A tap vertex* $t \in V$
- *A non-negative integral capacity function* c*.*

Introduce one decision variable for each edge $\{f_e\}_{e \in E(G)}$

$$f_e \leq 0$$

33 Integer Linear Programming and Relaxation

33.1 Definition

Definition 33.1.1. *The relaxation of an integer linear program is the problem that arises by removing the integrality constraint of each variable.*

33.2 Vertex Cover

Proposition 33.2.1. *Introduce $x_v \in \{0,1\}$ for each $v \in V(G)$. Setting $x_v = 0$ means that x_v is not in the solution.*
We want to minimize $\sum_{v\in V} x_v$ subject to

$$\begin{aligned} x_u + x_v &\geq 1 \textit{ for all } uv \in E(G) \\ x_v &\in \{0,1\} \textit{ for all } v \in V(G) \end{aligned}$$

Its relaxation is called Fractional Vertex Cover: We want to minimize $\sum_{v\in V} x_v$ subject to

$$\begin{aligned} x_u + x_v &\geq 1 \textit{ for all } uv \in E(G) \\ x_v &\in [0,1] \textit{ for all } v \in V(G) \end{aligned}$$

34 Solving LP

34.1 Definition

Definition 34.1.1 (Standard form)**.** *We want to maximize $\sum_{j=1}^{n} c_j x_j$ subject to*

$$\begin{aligned} \sum_{j=1}^{n} a_{ij}x_j &\leq b_i,\ \forall 1 \leq i \leq m \\ x_j &\geq 0\ \forall 1 \leq j \leq m \end{aligned}$$

Standard form is to maximize ${}^{\mathsf{t}}Cx$ subject to $Ax \leq b$ and $x \geq 0$.

Proposition 34.1.1. *A Linear Program can always be set in standard form.*

34.2 Geometric Interpretation

The set of points $x \in \mathbb{R}^n$ at which a constraint holds with equality is a hyperplane. Thus, each constraint is satisfied by a closed half-space of $\mathbb{R}^n$, and the set of feasible solution is the intersection of m closed half-spaces, that is, a convex polyhedron P.
A linear program can have no optimal solution :

- if the set of feasible solution is empty
- if for every integer M, there exists a feasible point x such that $c \cdot x \geq M$. In this case the set of feasible solution is unbounded.

Proposition 34.2.1. *Because the objective function is linear, the set $\{x \mid {}^{\mathsf{t}}Cx = t\}$ is a hyperplan for any $t \in \mathbb{R}$. For each t, this is called a level set of the objective function.*
Thus, if the maximum is $z^\star$, it is a supporting hyperplane of the P. So $\{x \mid {}^{\mathsf{t}}Cx = z^\star\}$ contains an extreme point (or vertex) of P.

Definition 34.2.1. *A supporting hyperplane of a convex set S is a hyperplane such that $S \cap H \neq \varnothing$*

34.3 Algorithm

We compute the coordinates of all the vertices of the polytope :

- There are n variables and m constraints.
- A vertex of P is a solution that satifies at least n inequality constraints with equality.

Algorithm 22 Naive Algorithm for Linear Programs

```
for each set of $n$ linearly independent constraints do
    Check if the points that satifies all of them is feasible and compute its value
end for
Get the maximum value
```

This is a finite algorithm.

34.4 Simplex Method

Algorithm 23 Simplex Method

```
Start at a vertex of feasible polytope
while There is a vertex of better objective value do
    Choose a pivot
    Go to next vertex
end while
```

34.5 Equational Form

Definition 34.5.1. *We want to maximize $\sum_{j=1}^{n} c_j x_j$ subject to*

$$\sum_{j=1}^{n} a_{ij}x_j = b_i,\ \forall 1 \le i \le m$$

$$x_j \ge 0\ \forall 1 \le j \le m$$

Proposition 34.5.1. *All linear programs can be put in equational form*

Proof. We introduce slack variables that compute the difference between the right side and the left side of the inequalities in standard form ■

35 Analysis

35.1 Complexity

Proposition 35.1.1. *The simplex is very fast in practice and routinely solves linear programs. However, it is a bizarre mathematical fact that the worst-case running time of the simplex method is exponential in the input size.*

Proof. The number of vertices of an hypercube is exponential in the dimension. A 'squashed' version of the hypercube can be used to force the simplex method to have exponential time by going through all the vertices. ■