No.

Date

CSCC73 - ALGORITHM DESIGN & ANALYSIS

GREEDY ALGORITHMS

Builds solution incrementally, at each stage it has constructed a *partial* solution that it extends to a *more complete* solution greedily and irrevocably → no back-track

↳ picks whatever is optimal at the iteration

Solves optimization problems:

- given input, find output ("solution") that

1. satisfies constraints → feasible solution
2. optimizes (min/max) → ~~satisfies~~ optimizes objective function

} gives optimal solution

INTERVAL SCHEDULING

input: A set of intervals $1, 2, \dots, n$, interval i has start time $s(i)$ finish time $f(i)$

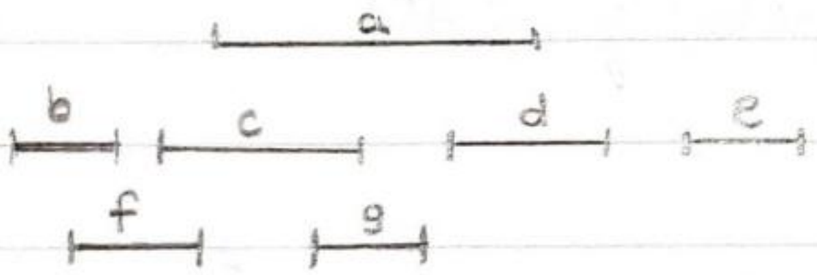
$f(i) > s(i)$

no intervals overlapping

output: A max cardinality feasible set of intervals

Intervals i,j overlap $\iff |[s(i), f(i)] \cap [s(j), f(j)]| > 1$ ← more than 1 point in common

e.g.



{b, g, e} feasible

{b, c, d, e} or {b, g, d, e} optimal

* max cardinality feasible set ⇒ feasible set that has no feasible ~~subset~~ superset

BUT ⇍ e.g. {b, a, e} not optimal (not max) but has no feasible superset

algorithm: sort intervals in increasing finish time // $O(n \log n)$

$A := \emptyset$; $F := -\infty$ // A is solution set of intervals

$O(n)$ { for each interval i in sorted order do // F is current last finishing time

if i overlaps with no interval in A then // $f(i) \geq F$

$A := A \cup \{i\}$; $F := f(i)$ // Add i to set A and // update F

return A

Running Time: $O(n \log n) + O(n) = O(n \log n)$

↑ sorting ↑ loop

Proof of Optimality ("greedy-stays-ahead" argument)

CLAIM 1: A is feasible. (algorithm only adds intervals that don't overlap)

Let $j_1, j_2, \cdots, j_k$ be intervals in A in sequential order

Let $j^*_1, j^*_2, \cdots, j^*_m$ be an optimal set A^* in sequential order

CLAIM 2: $f(j_t) \leq f(j^*_t)$ $\forall t = 1, 2, \cdots, k$ (greedy stays ahead)

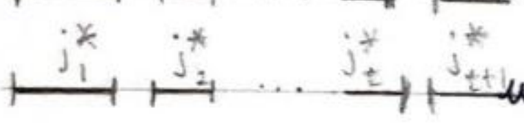
PROOF Induction on t

Basis: $t=1$ ~~$f(j_t)$ is~~ $f(j_1) = \min(f(j_i))$ $\forall i = 1, 2, \cdots n$

$\Rightarrow f(j_1) \leq f(j^*_1)$

Induction Step: Suppose for $t<k$, $f(j_t) \leq f(j^*_t)$

WTS $f(j_{t+1}) \leq f(j^*_{t+1})$



We know $f(j_t) \leq f(j^*_t) \leq s(j^*_{t+1}) < f(j^*_{t+1})$ [by IH, sequential order]

when j_t added to A by greedy algorithm

j^*_{t+1} - has not been considered yet $[f(j_t) < f(j^*_{t+1})]$

- does not overlap with any interval in A $[f(j_t) \leq s(j^*_{t+1})]$

since j_{t+1} was added to A instead of j^*_{t+1}, therefore $f(j_{t+1}) \leq f(j^*_{t+1})$

CLAIM 3: $k = m$ $|A|$ $|A^*|$

PROOF We know $k \leq m$ [claim 1 and optimality of A^*]

Suppose by contradiction $k < m$

$\Rightarrow A^*$ contains j^*_{k+1}

we have $f(j_k) \leq f(j^*_k) \leq s(j^*_{k+1}) \leq f(j^*_{k+1})$ [claim 2]

since $f(j_k) \leq f(j^*_{k+1})$ and j^*_{k+1} does not overlap with intervals in A, therefore j^*_{k+1} will be added to A by the algorithm

$\Rightarrow |A| \neq k \rightarrow$ contradiction $\Rightarrow k \geq m$

$k \leq m$ and $k \geq m \Rightarrow k = m$

Claim 1 and Claim 3 $\Rightarrow$ algorithm is optimal

Proof of Optimality ("promising set" argument)

Invariant: At the end of each iteration i, A is contained in some optimal set.

PROOF By induction on i

Basis: $i=0$

$A=\phi$ as we haven't entered the loop, $\therefore$ A is in the optimal set

Induction Step: Assume $i<k$, ~~A_i is the set~~ let A_i be the set of intervals in A and $A_i \subseteq A^*$ (IH)

WTS A_{i+1} (set of intervals in A after i+1th iteration) $\subseteq \hat{A}$ ↓ an optimal set

Case 1: no intervals added at i+1th iteration

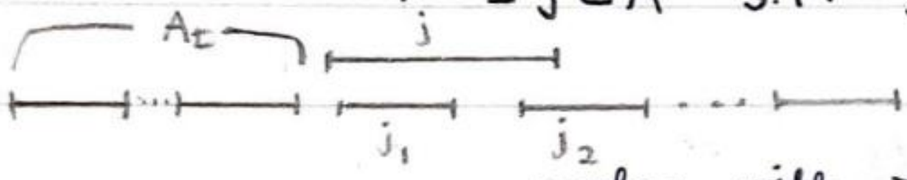
$A^* \supseteq A_i = A_{i+1}$ [IH] not necessarily A^*

Case 2: interval $j \in A^*$ is added to A on i+1th iteration

$A_{i+1} = A_i \cup \{j\} \subseteq A^*$

Case 3: interval $j \notin A^*$ is added to A on i+1th iteration

$\Rightarrow \exists j^* \in A^*$ s.t. j^* overlaps with j (o/w, j can be added to A^* to form a more optimal set)



but j^* cannot overlap with ≥ 2 intervals in $A^* - A_t$ because $j_1 \in A^*$ will be added instead $\Rightarrow$ j^* overlaps with 1 interval in $A^* - A_t$

Let $\hat{A}$ be the set obtained from A^* with j^* replaced with j

i.e. $\hat{A} = (A^* - \{j^*\}) \cup \{j\}$

i. $\hat{A}$ is feasible (only j^* and j overlap) $\Rightarrow$ $\hat{A}$ is optimal

ii. $|\hat{A}| = |A^*|$

iii. $A_{j+1} \subseteq \hat{A}$

Invariant $\Rightarrow$ Optimality

WEIGHTED INTERVAL SCHEDULING

input: A set of jobs 1, 2, ⋯, n, job i has length t(i), deadline d(i)

define Schedule S to be $S: i \longmapsto s(i)$ ← start time s(i)

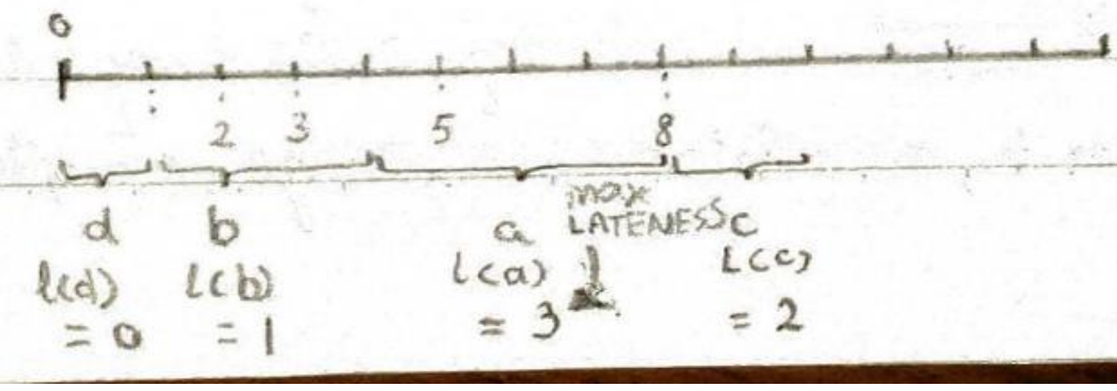
$\forall i,j \quad i \neq j \quad [s(i), s(i)+t(i)]$ and $[s(j), s(j)+t(j)]$ do not overlap

define lateness of i in S: $l(S,i) = \max(s(i)+t(i)-d(i), 0)$ and max lateness $L(S) = \max_i l(S,i)$

output: A schedule with minimum lateness

e.g.

job	t	d
a	4	5
b	3	3
c	2	8
d	1	2



algorithm: Sort jobs by increasing deadline
let d[1,…,n] be deadlines of jobs //sorted
let t[1,…,n] be the lengths of those jobs
F := (intial_end_time); ~~gs[n] = []~~ // represents max finish time
for i:=1 to n do //of all jobs scheduled
  gs(i) := F; F := gs(i)+t(i) //gs(i) is start-time of job i
return gs // gs is schedule map

Running Time: O(n log n)

* Algorithm outputs schedule with no gaps in increasing deadline order

DEFINE Inversion: for schedule s and job i,j, s(i)<s(j) but d(i)>d(j)
(i scheduled before j but deadline of i is after j)

Proof of Optimality

CLAIM 1: There is an optimal schedule with no gaps

CLAIM 2: All schedules with no gaps and inversions have same max lateness
(if swap between jobs with common deadline, lateness doesn't change)

PROOF By claim 2, as long as some schedule that has no gaps and inversions is optimal, our algorithm which produces a schedule with no gaps and inversions would produce an optimal schedule.

Let S be any optimal schedule

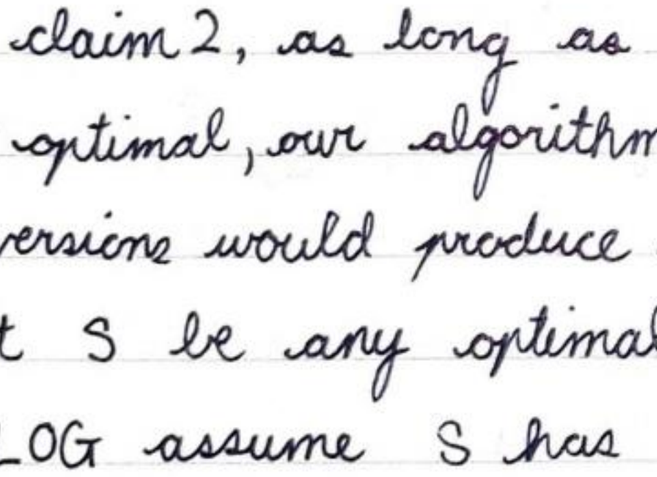
WLOG assume S has no gaps [Claim 1]

CASE 1: S has no inversion. We are done.

CASE 2: S has inversion

assume i' and j' has inversion i.e. s(i')<s(j'), d(i')>d(j')

i,j between i', j' s.t. i,j consecutive has inversion



lateness of jobs in S, S` ≠ i,j no change

lateness of j in S` better

lateness of i in S` worse, but $l(S`,i) \leq l(S,j)$

so $\max_i l(S`) \leq \max_i l(S)$

inversions ↓, max lateness no change, so optimal schedule with no

gaps and no inversions ~~produced~~ can be produced by the greedy algorithm. ∎

DIJKSTRA'S_ALGORITHM_

input: Directed graph $G=(V,E)$

A source vertex $s \in V$

A weight function $wt: E \to \mathbb{R}$ s.t. $\forall e \in E$, $wt(e) \geq 0$

output: $\forall v \in V$, $\delta(v) = \begin{cases} \min\{wt(p): p \text{ is a } s \to v \text{ path in } G\} & \text{weight of path from } s \text{ to } v \text{ that has least weight} \\ \infty, \text{ if no such path } p & v \text{ not reachable from } s \end{cases}$

$p = v_1, v_2, \dots, v_k$ A path $(wt(p) = wt(v_1,v_2) + wt(v_2,v_3) + \cdots + wt(v_{k-1}, v_k)$

↳ edge $v_1 \to v_2$

algo keeps: A set of explored nodes $R \subseteq V$

$\forall v \in V$, $d(v)$

1. $\forall v \in R$, $d(v) = \delta(v)$
2. $\forall v \in V$, $d(v) = \begin{cases} wt \text{ of } \sout{\text{path}} \text{ } \sout{\text{executing}} & s \to v \text{ path that has min wt in } R, v \notin R \\ \infty, \text{ if no } s \to v \text{ } R\text{-path} & \end{cases}$
3. $d(v) \geq \delta(v)$

$\forall v \in V$

↓

v not reachable with vertices in R

algorithm:

```
R := ∅                                  // R initially empty
d(s) := 0 ; pre(s) := NIL               // s is source with no predecessor
for all v ∈ V - {s} do d(v) := ∞        // set other vertices to have d(v) = ∞
while R != V do                         // while not fully explored
    let u be node not in R s.t. d(u) is min
    R := R ∪ {u}
    for each v s.t. (u,v) ∈ E (and v ∉ R) do   // for all neighbours of u not yet explored
        if d(v) > d(u) + wt(u,v) then          // there is a better path
            d(v) := d(u) + wt(u,v)             // update d(v) with weight of better path and set predecessor
            pre(v) := u                        // of v to u
```

Running Time:

1. "Naive" representation (array)
   - $O(n)$ initialize
   - $O(n^2)$ loop

   $\Big\}\ O(n^2)$

2. "Sophisticated" representation (heaps)
   - $O(n)$ initialize
   - $O(n \log n)$ extract min
   - $O(m \log n)$ change key

   $\Big\}\ O((n+m)\log n) \approx O(m \log n)$

   ↳ $\log n$ each time, happens n or m times total

No.

Date

if G is at least a tree (connected), $m \geq n-1$ so m dominates.

max possible $m = n^2$ => sophisticated has run-time ($O(n^2 \log n) \geq O(n^2)$)

=> naive better when G is dense

Proof of Optimality

CLAIM 1: d(v) is non-increasing. [$i \leq j \Rightarrow d_i(v) \geq d_j(v)$]

After being initialized, the value of d(v) is changed to a smaller value only.

CLAIM 2: If u is added to R st in iteration i, then d(u) doesn't change in that iteration [$d_i(u) = d_{i-1}(u)$ if u is added to R at iteration i]

PROOF Suppose for contradiction u is added to R in iteration i and d(u) changes in iteration i.

$\Rightarrow$ (u, u) is an edge and $d_{i-1}(u) + wt(u,u) < d_i(u)$. But $wt(u,u) \not< 0$, which contradicts the assumption that weight of every edge is nonnegative ■

CLAIM 3: At the end of iteration i, if $d(v) = k \neq \infty$, then there is an $s \to v$ R-path with $d_i(v)$ = wt of path = k

PROOF Induction on $i \geq 1$

Basis: $i = 1$ s is added to R, $\Rightarrow R_1 = \{s\}$, $d_1(s) = 0$

$\forall v \neq s \in V$, $d_1(v) = wt(s,v)$ if $\exists$ an edge (s,v), else $d_1(v) = \infty$. Claim holds

Induction Step: let $i > 1$ be arbitrary. Assume the claim holds for i.

WTS claim holds for i+1

Let u be node added to R in iteration i+1.

$\forall v \in V$, if $d_{i+1}(v) = d_i(v)$, claim holds by IH.

else, by claim 2 $d_{i+1}(u) = d_i(u)$ and $d_{i+1}(v) = d_{i+1}(u) + wt(u,v)$

and $d_i(u) \neq \infty$ ($d_{i+1}(v) \neq \infty$ so $d_i(u) \neq \infty$)

By IH there is a path p $s \to u$ of wt $d_{i-1}(u)$. Since $R_i = R_{i-1} \cup \{u\}$, $s \to u \to v$ is an R_i-path to v with wt $wt(p) + wt(u,v) = d_i(u) + wt(u,v) = d_{i+1}(u) + wt(u,v)$

So the claim holds after iteration i

By PSI, the claim holds, $\forall i \geq 1$ ■

CLAIM 4: If $d(u) = \infty$ when u is added to R, then there is no $s \to u$ path

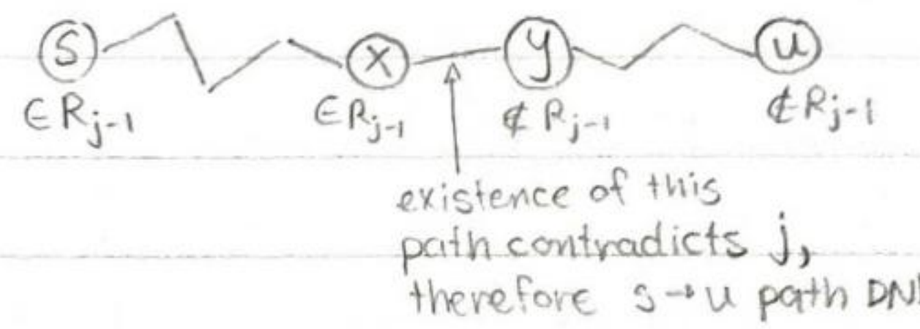
PROOF Suppose u is added to R at iteration i and $d_i(u) = \infty$.

Let j be min iteration # s.t. a node with d-value $= \infty$ is added to R

Suppose for contradiction $\exists (x,y) \in E$ s.t. $x \in R_{j-1}$ and $y \notin R_{j-1}$ (*)

for any $j' \leq j-1$, iteration $j' \Rightarrow d_{j'}(x) \neq \infty \to d_{j'}(y) \neq \infty \Rightarrow d_{j-1}(y) \neq \infty$ [claim 1]

Since $y \notin R_{j-1} \Rightarrow y$ can be added to R_j with $d_j(y) \neq \infty$, which contradicts our assumption of j.



■

CLAIM 5: If u is added to R in iteration i, then $d_i(u) = \delta(u)$

PROOF Induction on i

Basis: $i = 1 \Rightarrow u = s$ $\quad d(s) = 0 = \delta(s)$ $\quad \therefore$ claim holds

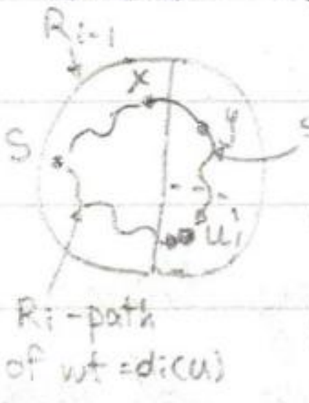
Induction Step: Suppose $i > 1$ and claim holds $\forall k \in \mathbb{N}, 1 \leq k < i$

WTS $d_i(u) = \delta(u)$

If $d_i(u) = \infty$ claim holds by claim 4 and 3

If $d_i(u) = k \neq \infty$, by claim 3 $\exists$ $s \to u$ R_i-path of $wt = d_i(u) \Rightarrow d_i(u) \geq \delta(u)$

Show $d_i(u) \leq \delta(u)$



$d_i(u) = d_{i-1}(u)$ [claim 2]

$\leq d_{i-1}(y)$ [y not added to R] ~~part of shortest path~~

$\leq d_j(y)$ [claim 1] ~~path~~ (iteration when x added to R)

$\leq d_j(x) + wt(x,y)$ $\quad j \leq i-1$ [claim 2 and algo]

$= \delta(x) + wt(x,y)$ [by IH]

$= wt(\text{shortest path } s \to x) + wt(x,y) \leq wt(p)$ [$wt(s) \geq 0$]

$= \delta(u)$

$d_i(u) \geq \delta(u)$ and $d_i(u) \leq \delta(u)$

$\Rightarrow d_i(u) = \delta(u)$

By PCI the claim holds ■

THEOREM: When the algorithm terminates, $\forall v \in V$, $d(v) = \delta(v)$

PROOF: by claim 5, $d(v) = \delta(v)$ when v was added to R.

by claim 1 it cannot increase, by claim 3 it cannot decrease

Therefore theorem holds ■

HUFFMAN'S_ALGORITHM_

Alphabet: Γ = finite set of symbols $|\Gamma| = n$

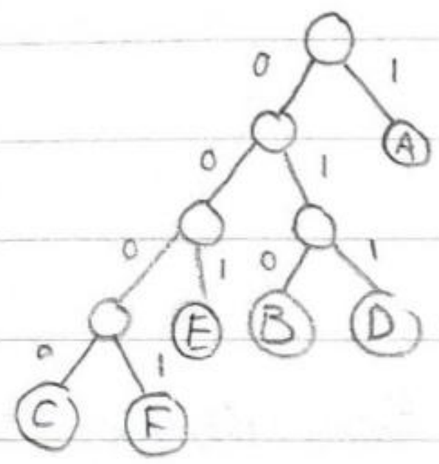
Code: map $a \in \Gamma \rightarrow$ binary string (codeword for symbol a)

- either fixed length ($\lceil \log_2 n \rceil$)
- or shorter lengths for more frequent symbols

  e.g. $\Gamma = \{A, B, C, D, E, F\}$

X	f(x)
A	0.38
B	0.15
C	0.1
D	0.2
E	0.12
F	0.05

$\Big\} = 1$



every symbol is a leaf, so no codeword is prefix of ~~so~~ another

can cause ambiguity

↳ Prefix Code: no codeword is prefix of another

⇒ unambiguous decoding

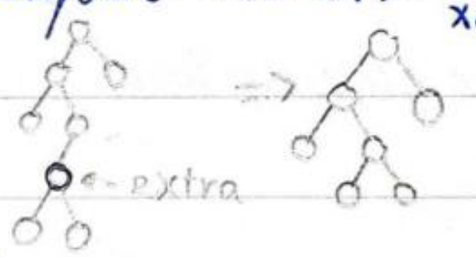
input: An ~~set of~~ alphabet set Γ, $n = |\Gamma|$

$\forall x \in \Gamma$, $f(x)$ = frequency of x s.t. $\sum_{x \in \Gamma} f(x) = 1$

output: A binary tree representing optimal ~~decoding~~ encoding of (Γ, f)

↳ tree that minimizes average depth $AD(T) = \sum_{x \in \Gamma} f(x) \cdot depth_T(x)$

note: 1. an optimal tree must be full



2. in optimal tree T, $f(x) > f(y)$

$\Rightarrow depth_T(x) \leq depth_T(y)$

1 and 2 => If x, y are symbols with min frequency then

$\exists$ optimal tree s.t. x and y are siblings at max depth

algorithm: Huffman (n, f)

```
for i := 1 to n do
    H[i] := (i, f(i))
    create a leaf node labeled i (children NIL)
BUILDHEAP(H)                    // create a heap from H
for i := n+1 to 2n-1 do         // for n symbols, there are n-1 internal
    x := EXTRACTMIN(H); y := EXTRACTMIN(H)   // nodes in the binary tree
```

create a node labeled i with children the nodes labeled $x.label$ and $y.label$

INSERT$(H, (i, x.freq+y.freq))$ //insert i with frequency $x.freq + y.freq$ into H

running time: $O(n \log n)$ (when using heaps)

Proof of Optimality

PROOF induction on $|\Gamma| = n$

Basis: $n=2$

optimal tree is where both codewords are 1 bit long, which is what the algorithm produces.

Induction Step: Let $n \geq 2$ be arbitrary. Suppose algorithm produces optimal tree for a Γ s.t. $|\Gamma| = n \geq 2$.

WTS algorithm produces optimal tree for a $|\Gamma'| = n+1$

Let Γ be an alphabet with $|\Gamma| = n+1$, $\forall a \in \Gamma$, $f(a)$ is the frequency.

Let H be the tree produced by Huffman's algorithm for Γ, f.

By algorithm $\exists x, y \in \Gamma$ s.t. x, y are symbols with min frequencies and are siblings of max depth

Let $z \notin \Gamma$ be a new symbol and let $\Gamma' = (\Gamma - \{x, y\}) \cup \{z\}$

and f' be frequencies in Γ'

$$\forall a \in \Gamma', f'(a) = \begin{cases} f(a), & a \neq z \\ f(x)+f(y), & a = z \end{cases}$$

since $AD(H) = \sum_{\substack{a \in \Gamma \\ a \neq x \text{ or } a \neq y}} f(a) \cdot depth_H(a) + f(x) \cdot d + f(y) \cdot d$ and

let H' be tree replacing (x)(y) with (z)

$$AD(H') = \sum_{\substack{a \in \Gamma' \\ a \neq z}} f'(a) \cdot depth_{H'}(a) + f(z) \cdot d-1 = \sum_{\substack{a \in \Gamma \\ a \neq x \text{ or } a \neq y}} f(a) \cdot depth_H(a) \quad \cancel{+ f(z) \cdot d +} + (f(x)+f(y)) \cdot (d-1)$$

$$= AD(H) - (f(x)+f(y))$$

therefore $AD(H) = AD(H') + (f(x)+f(y))$ (*)

by IH, H' is optimal for Γ', f'

Let T be an optimal tree for Γ, f. WLOG, assume x, y are siblings at max depth.

Let T' be obtained from T by replacing (x)(y) with (z)

$AD(T) = AD(T') + (f(x) + f(y))$ [by definition of ad]

$\geq AD(H') + (f(x) + f(y))$ [by ☒ + optimality of H']

$= AD(H)$ [by *]

Since T is optimal, therefore H is also optimal for Γ, f.

☒ Since Base case and induction step hold, by PSI, the algorithm produces optimal trees for alphabets with n symbols and their associated frequencies: ∎

DIVIDE AND CONQUER

Used to solve an instance of size n of some problem

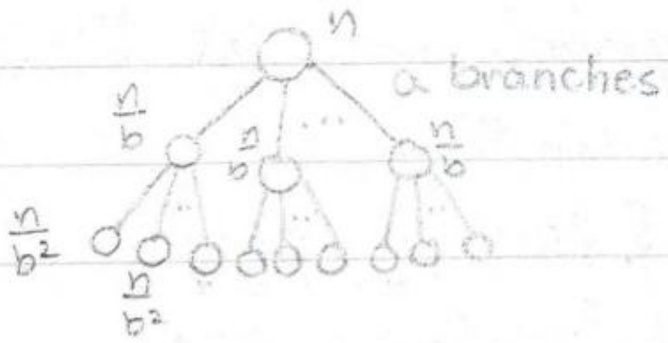
if n is small, solve directly

else 1. split input into a subproblems, each of size $\frac{n}{b}$ ← reduce input by constant factor

2. recursively solve every smaller instance (where $a \geq 1, b > 1$ are constant integers)

3. combine results of smaller instances into results for given instance

recursive tree:



assume $n = b^k$, for some k

running time: $T(n) = \underbrace{a \cdot T(\frac{n}{b})}_{\text{time for recusive calls}} + \underbrace{cn^d}_{\text{time for divide and conquer}}$ c, d constants $d \geq 0$

$T(1) = c$

MASTER THEOREM

If T(n) satisfies running time, then

a. $a < b^d \Rightarrow T(n) = \Theta(n^d)$ more work on divide and conquer step

b. $a = b^d \Rightarrow T(n) = \Theta(n^d \log n)$ equal time on recursive calls and divide and conquer

c. $a > b^d \Rightarrow T(n) = \Theta(n^{\log_b a})$ more work on recursive calls
↑ leafs

level	# subproblems	input size	total time
0	1	n	cn^d
1	a	$\frac{n}{b}$	$ac(\frac{n}{b})^d$
2	a^2	$\frac{n}{b^2}$	$a^2c(\frac{n}{b^2})^d$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
i	a^i	$\frac{n}{b^i}$	$a^i \cdot c(\frac{n}{b^i})^d$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\log_b n$	$a^{\log_b n} = n^{\log_b a}$	1	$cn^{\log_b a}(1)^d$

time for each subproblem

$$T(n) = \sum_{i=0}^{\log_b n} a^i \left(\frac{n}{b^i}\right)^d \cdot c = cn^d \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i$$

a. $a < b^d \Rightarrow \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i < \sum_{i=0}^{\infty} \left(\frac{a}{b^d}\right)^i = \frac{1}{1-\frac{a}{b^d}} \Theta(1)$

$\therefore T(n) = cn^d \cdot \Theta(1) = \Theta(n^d) \quad = \Theta(1)$

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

$\therefore T(n) = cn^d \cdot \Theta(\log n) = \Theta(n^d \log n)$

c. $a > b^d \Rightarrow \sum_{i=0}^{\log_b n} \left(\frac{a}{b^d}\right)^i = \Theta\left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right)$

$= \Theta\left(\frac{n^{\log_b a}}{n^d}\right)$

$\therefore T(n) = cn^d \Theta\left(\frac{n^{\log_b a}}{n^d}\right) = \Theta(n^{\log_b a})$

S_n Series

if $0 \le x < 1$ $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$

$S_n = \sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$

e.g. MERGE_SORT

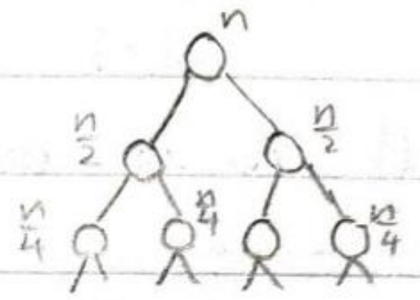
algorithm: if $n = 1$ then done

else

split array into two halves of size $\frac{n}{2}$

recursively sort each half

merge two sorted halves



running time: $T(n) = 2 \cdot T(\frac{n}{2}) + cn$ $a = 2$ $b = 2$ $d = 1$ $\Rightarrow a = b^d$

$\Rightarrow T(n) = \Theta(n \log n)$

e.g. ~~BINA~~ BINARY_SEARCH

algorithm: if $n = 1$ then return $A[1] == Search$

else if $A[\lfloor \frac{n}{2} \rfloor] \ge$ element being sought then // maybe in first half

recursively search $A[1, \cdots, \lfloor \frac{n}{2} \rfloor]$

else

recursively search $A[\lfloor \frac{n}{2} \rfloor + 1, \cdots, n]$

running time: $T(n) = T(\frac{n}{2}) + c$ $a = 1$ $b = 2$ $d = 0$ $\Rightarrow a = b^d$

$\Rightarrow T(n) = \Theta(\log n)$

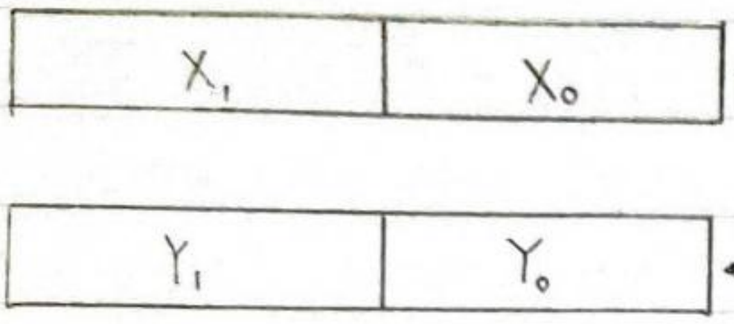
MULTIPLICATION_ALGORITHM

input: 2 n-bit binary numbers $b_{n-1}, b_{n-2} \cdots b_1, b_0$ $b_i = \{0, 1\}$ $i \in \mathbb{N}$ $i \le n-1$

output: product of the two numbers

- addition takes $\Theta(n)$ - multiplication usually $\Theta(n^2)$

Karatsuba's multiplication algorithm



$X = X_1 \cdot 2^{\frac{n}{2}} + X_0$ (n-bits; $\frac{n}{2}$-bits; $\frac{n}{2}$-bits)

$Y = Y_1 \cdot 2^{\frac{n}{2}} + Y_0$ (n-bits; $\frac{n}{2}$-bits; $\frac{n}{2}$-bits)

$$(X_1 \cdot 2^{\frac{n}{2}} + X_0)(Y_1 \cdot 2^{\frac{n}{2}} + Y_0)$$
$$= 2^n X_1Y_1 + 2^{\frac{n}{2}} X_1Y_0 + 2^{\frac{n}{2}} X_0Y_1 + X_0Y_0$$

4 $\frac{n}{2}$ bits multiplication

$$X \cdot Y = X_1Y_1 \cdot 2^n + ((X_1+X_0)\cdot(Y_1+Y_0) - X_1Y_1 - X_0Y_0) \cdot 2^{\frac{n}{2}} + X_0Y_0$$

requires only 3 $\frac{n}{2}$ bits multiplication

running time: $3T(\frac{n}{2}) + cn$ ← addition, subtraction, left shift takes linear time

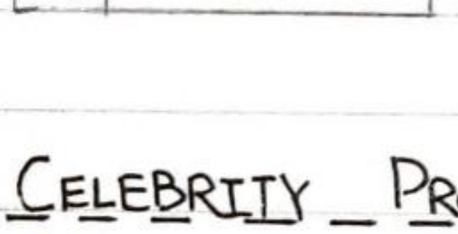
$a=3 \Rightarrow a > b^d \Rightarrow T(n) = \Theta(n^{\log_2 3})$

$b=2$

$d=1$

$\log_2 3 = 1.585\ldots \approx 1.6 < 2$

what if n is odd? see $(X_1+X_0)\cdot(Y_1+Y_0)$ ↓ chance of 1 extra bit in result



$X_1 + X_0 = u_1 \cdot 2^{\frac{n}{2}} + u_0$

$Y_1 + Y_0 = V_1 \cdot 2^{\frac{n}{2}} + V_0$

$$(X_1+X_0)\cdot(Y_1+Y_0) = U_1V_1 \cdot 2^n + (U_1V_0 + U_0V_1) \cdot 2^{\frac{n}{2}} + U_0V_0$$

U_1V_1: linear; $(U_1V_0 + U_0V_1)$: linear; U_0V_0: linear

CELEBRITY_PROBLEM_

You and n people are in a room. 1 of them is a celebrity (knows no one), everyone knows celebrity except you. Find celebrity

approach: $A \to B$ A knows B? yes ⇒ A not celebrity

no ⇒ B not celebrity

Pair people to groups of 2 i.e. $\frac{n}{2}$ groups

Eliminate $\frac{n}{2}$ people by above question

running time: $T(n) = T(\frac{n}{2}) + \frac{n}{2}$ ($\frac{n}{2}$ questions, $c = \frac{1}{2}$) $\Rightarrow$ $a=1$, $b=2$, $d=1$ $a < b^d$

$\Rightarrow T(n) = \Theta(n)$

No.

Date

CLOSEST_SETS_OF_POINTS

input: set $P = \{$ points on x-y plane$\}$ $|P| = n$ $\quad p=(x,y) \in P$

distance : $p=(x,y)$, $q=(x',y') \Rightarrow d(p,q) = \sqrt{(x-x')^2+(y-y')^2}$

output: closest pair of points in P

i.e. $p, q \in P$, $p \neq q$ s.t. $\forall\, p', q' \in P$ $p' \neq q'$ $d(p,q) \leq d(p',q')$

algorithm: ClosestPair(P)

P_x := list of points in P sorted by x-coordinate → $O(n \log n)$

P_y := list of points in P sorted by y-coordinate → $O(n \log n)$

return RCP(P_x, P_y)

RCP(P_x, P_y)

if $|P_x| \leq 3$ then

  calculate all pairwise distances and return closest pair

else

  L_x := 1st half of P_x ; R_x := 2nd half of R_x ← $O(n)$

  m := (max x-coordinate in L_x + min x-coordinate in R_x) / 2 ← $O(1)$ (bisector)

  L_y := sublist of P_y (of points) in L_x ; R_y := sublist of P_y of points in R_x ← $O(n)$ (no need to resort)

  (p_L, q_L) := RCP(L_x, L_y) ; (p_R, q_R) := RCP(R_x, R_y)

  δ := min($d(p_L, q_L)$, $d(p_R, q_R)$)

  if $d(p_L, q_L) = \delta$ then

    $(p^*, q^*) := (p_L, q_L)$ // let (p^*, q^*) be pair that has // shorter distance

  else

    $(p^*, q^*) := (p_R, q_R)$

  } $O(1)$

  B := sublist of P_y of points whose x-coordinates are within δ of m ← $O(n)$

  for each $p \in B$ in order of appearance on B do

    for each of the (up to) next seven points q after p on B do ← $O(1)$

      if $d(p,q) < d(p^*, q^*)$ then $(p^*, q^*) := (p, q)$ // check d for points with $m-\delta \leq x \leq m+\delta$

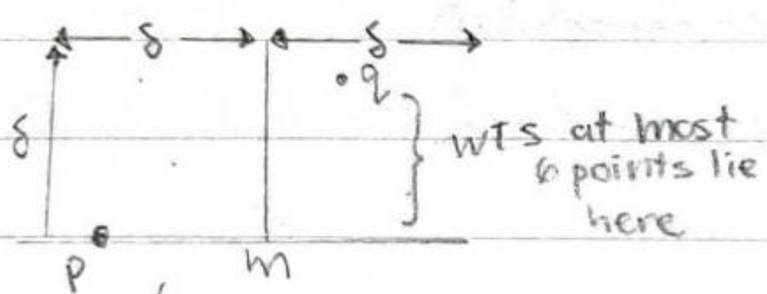
  } $O(n)$

return (p^*, q^*)

Why check up to 7?

CLAIM: if $p,q \in B$ and $d(p,q) < \delta$ then there are at most 6 points in B other than p and q whose y-coordinates are between the y-coordinates of p and q $\forall p \in B$, there are at most 7 points $\in B$ that will have pairwise distance $< \delta$ (first 7 ordered by y-coordinates)

PROOF Assume for contradiction, $p,q \in B$ s.t. $d(p,q) < \delta$ and there are ≥ 7 points in B with y-coordinates between p,q's y-coordinates



Consider the two $\delta \times \delta$ squares shown. They contain ≥ 9 points (including p and q) whose y-coordinates are between p's and q's (includes)

$\Rightarrow$ 1 of the squares has ≥ 5 points in B by (PHP)

This contradicts that points on the same side of the bisector are $\geq \delta$ apart

$\Rightarrow$ there are at most 6 points between p and q ∎

running time: $2T(\frac{n}{2}) + cn \Rightarrow a=2, b=2, d=1$

$\Rightarrow T(n) = \Theta(n \log n)$

FAST FOURIER TRANSFORM (FFT)

- Multiplication of 2 degree m polynomials
- normal way: $\Theta(m^2)$:

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m$$

$$B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_m x^m$$

$$C(x) = A(x) \cdot B(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{2m} x^{2m}$$

where $c_0 = a_0 b_0$

$c_1 = a_0 b_1 + a_1 b_0$

$c_2 = a_0 b_2 + a_1 b_1 + a_2 b_0$

$\vdots$

$c_{2m} = a_m b_m$

i.e. $c_k = \sum_{0 \leq i,j \leq m}^{m} a_i b_j$

$0 \leq k \leq 2m$ s.t. $i+j=k$

running time: $\Theta(1+2+\cdots+m+(m+1)+m+m-1+\cdots+1)$

$= \frac{m(m+1)}{2} + \frac{m(m+1)}{2} + m$

$= m(m+2) \geq m^2$

$= \Theta(m^2)$

better way : $O(m \log m)$:

CONVOLUTION OF $\vec{a} = (a_0, a_1, \cdots, a_m)$ and $\vec{b} = (b_0, b_1, \cdots, b_m)$

$\vec{a} * \vec{b} = (C_0, C_1, \cdots, C_{2m})$ where $C_k = \sum_{\substack{0 \le i,j \le m \\ \text{s.t. } i+j=k}} a_i b_j$ $\quad 0 \le k \le 2m$

COEFFICIENT REPRESENTATION OF $A(x)$

$(a_0, a_1, \ldots, a_m)$

VALUE ~~RERES~~ REPRESENTATION OF $A(x)$

$(A(x_0), A(x_1), \cdots, A(x_n))$ for distinct $x_0, x_1, \cdots, x_n$

Interpolation Theorem: A ~~polynomial~~ polynomial of degree m is uniquely determined by its value in $m+1$ distinct points

input: 2 degree m polynomials $A(x)$, $B(x)$

output: $A(x) \cdot B(x) = C(x)$

method: 1. Select n points $x_0, \cdots, x_{n-1}$, $n \ge 2m+1$ and $n = 2^k$ for some $k \in \mathbb{N}$

2. Evaluate $A(x_0), \cdots, A(x_{n-1})$; $B(x_0), \cdots, B(x_{n-1})$

3. Compute $\underbrace{A(x_0)B(x_0)}_{C(x_0)}, \cdots, \underbrace{A(x_{n-1})B(x_{n-1})}_{C(x_{n-1})}$

4. Interpolate $c_0, \cdots, c_{2m}$ from above value representation of C

STEP 2 (evaluation) by DIVIDE & CONQUER

Let $A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$, $n = 2^k$ for some k

$= a_0 + a_2 x^2 + a_4 x^4 + \cdots + a_{n-2} x^{n-2}$ $\leftarrow A_e(x^2)$ both degree $\frac{n}{2} - 1$

$+ x(a_1 + a_3 x^2 + a_5 x^4 + \cdots + a_{n-1} x^{n-2})$ $A_o(x^2)$

$\Rightarrow A(x) = A_e(x^2) + x A_o(x^2)$

note: $A(x) = A_e(x^2) + x A_o(x^2)$

$A(-x) = A_e(x^2) - x A_o(x^2)$

we can find evaluate ~~two~~ polynomials at two points by evaluating 2 polynomials of degree $\frac{n}{2} - 1$ at $\frac{n}{2}$ points

PRIMITIVE ROOTS

z is an "primitive" n-th root of 1 iff

1. z is an n-th root of 1, 2. $z^i \neq 1$ for $i = 1, 2, \cdots, n-1$

note: if z is a primitive n^{th} root of 1
then z^2 is a primitive $\frac{n}{2}^{th}$ root of 1

input: coefficients $a_0, a_1, \cdots, a_{n-1}$ of polynomial $A(x)$ of degree $n-1$
n is a power of 2

output: $A(\omega^0), A(\omega^1), \ldots, A(\omega^{n-1})$ where ω is a primitive n^{th} root of 1.

algorithm: FFT$((a_0, a_1, \cdots, a_{n-1}), \omega)$
if $n=1$ then return a_0 // $A(x) = a_0$
else
$(s_0, s_1, \cdots, s_{\frac{n}{2}-1}) := FFT((a_0, a_2, \cdots, a_{n-2}), \omega^2)$ // $S_j = A_e(\omega^{2j})$
$(t_0, t_1, \cdots, t_{\frac{n}{2}-1}) := FFT((a_1, a_3, \cdots, a_{n-1}), \omega^2)$ // $t_j = A_o(\omega^{2j})$
for $j := 0 \cdots \frac{n}{2}-1$ do
$r_j := s_j + \omega^j t_j$ // $r_j := A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_o(\omega^{2j})$
$r_{j+\frac{n}{2}} := s_j - \omega^j t_j$ // $r_{j+\frac{n}{2}} := A(-\omega^j) = A_e(\omega^{2j}) - \omega^j A_o(\omega^{2j})$
return $(r_0, r_1, \cdots, r_{n-1})$

running time: $T(n) = \underset{\text{split odd and even}}{2} \cdot T(\frac{n}{2}) + \underset{\text{for loop}}{cn}$ $a=2, b=2, d=1$ $\therefore a = b^d$
$\Rightarrow T(n) = \Theta(n \log n)$

STEP 4 (interpolation)
from FFT we have: $FFT((a_0, a_1, \cdots, a_{n-1}), \omega) \rightarrow (A(\omega^0), A(\omega^1), \cdots, A(\omega^{n-1}))$
CLAIM $\frac{1}{n} FFT((A(\omega^0), \cdots, A(\omega^{n-1})), \omega^{-1}) \rightarrow (a_0, a_1, \cdots, a_{n-1})$ inverse FFT

$$\begin{pmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix}}_{M} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \Rightarrow M \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} A(x_0) \\ \vdots \\ A(x_{n-1}) \end{pmatrix}$$

$$M^{-1}M \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix} = M^{-1} \begin{pmatrix} A(x_0) \\ \vdots \\ A(x_{n-1}) \end{pmatrix} \Rightarrow M^{-1} \begin{pmatrix} A(x_0) \\ \vdots \\ A(x_{n-1}) \end{pmatrix} = \begin{pmatrix} a_0 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

let $M_\omega = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{n-1} \\ \vdots & & & \\ 1 & \omega^{n-1} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix}$ $M_\omega[j,k] = \omega^{jk}$ $0 \le j, k < n$

$(j,k)^{th}$ entry

CLAIM: $M_\omega^{-1} = \frac{1}{n} M_{\omega^{-1}}$ i.e. interpolation $= \frac{1}{n} \cdot FFT(A(\omega^0), A(\omega^1), \cdots, A(\omega^{n-1}), \omega^{-1})$

PROOF

let $M_{\omega^{-1}} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \cdots & \omega^{-(n-1)} \\ \vdots & & & \\ 1 & \omega^{-(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{pmatrix}$

$M_\omega \cdot M_{\omega^{-1}} = X = \begin{pmatrix} n & & & \\ & n & & 0 \\ & & \ddots & \\ 0 & & & n \end{pmatrix} = nI$

$X_{jk} = \sum_{l=0}^{n-1} \omega^{jl} \cdot \omega^{-lk} = \sum_{l=0}^{n-1} (\omega^{j-k})^l$ ← geometric series

$= \begin{cases} n, & \text{if } \omega^{j-k} = 1 \iff j = k \\ \frac{(\omega^{j-k})^n - 1}{\omega^{j-k} - 1} = 0 \iff j \neq k & \text{if } \omega^n = 1 \end{cases}$

$M_\omega^{-1} M_\omega M_{\omega^{-1}} = M_\omega^{-1} nI$

$M_{\omega^{-1}} = n M_\omega^{-1} \quad \therefore M_\omega^{-1} = \frac{1}{n} M_{\omega^{-1}}$ ■

algorithm: $n :=$ smallest power of 2, $\geq 2m+1$ ← select n points

$\omega := e^{i\frac{2\pi}{n}}$ // primitive n^{th} root of 1

$a_{m+1}, \cdots, a_{n-1} := 0$; $b_{m+1}, \cdots, b_{n-1} := 0$ // fill rest of polynomial with // coefficient 0

$(A_0, \cdots, A_{n-1}) := FFT((a_0, \cdots, a_{n-1}), \omega)$] evaluate $A(x_0), \cdots, A(x_{n-1})$;

$(B_0, \cdots, B_{n-1}) := FFT((b_0, \cdots, b_{n-1}), \omega)$] $B(x_0), \cdots, B(x_{n-1})$

for $j := 0, \cdots, n-1$ do $c_j := A_j * B_j$ ← compute $A(x_j)B(x_j) = C(x_j)$

$(c_0, c_1, \cdots, c_{n-1}) := \frac{1}{n} * FFT((c_0, c_1, \cdots, c_{n-1}), \omega^{-1})$ ← interpolate

return $(c_0, c_1, \cdots, c_{2m})$ $c_0, c_1, \cdots, c_{2m}$

running time: $O(n \log n)$

ORDER STATISTICS

$A = a_1, a_2, \cdots, a_n$, WLOG, assume the numbers a_i are distinct

input: $A = a_1, \cdots, a_n$; $k \in \mathbb{N}$, $1 \le k < n$

output: k-th smallest element in A

No.

Date

algorithm1: SELECT(A, k) // $k=1 \Rightarrow \min(A)$, $k=n \Rightarrow \max(A)$, $k=\lceil \frac{n}{2} \rceil \Rightarrow$ median(A)

```
if k=1 or |A|=1 then return A[1]
else
    S := arbitrary element of A
    A⁻ := list of elements in A that are <S
    A⁺ := list of elements in A that are >S
    if k ≤ |A⁻| then return SELECT(A⁻, k)
    else if k == |A⁻|+1 then return S
    else return SELECT(A⁺, k-(|A⁻|+1))      # elements in A⁻ and S we skipped
```

running time: assume S is ith smallest element of A

$T(n) = T(\max(i-1, n-i)) + cn$ ← splitting array into A^- and A^+

($i-1$: SELECT(A^-); $n-i$: SELECT(A^+))

↳ Best case: S = median $\Rightarrow T(n) = T(\frac{n}{2}) + cn$ $a=1, b=2, d=1$

$a < b^d \Rightarrow T(n) = \Theta(n)$

↳ Worst case: S = Smallest or largest $\Rightarrow T(n) = T(n-1) + cn = T(n-2) + cn + cn$

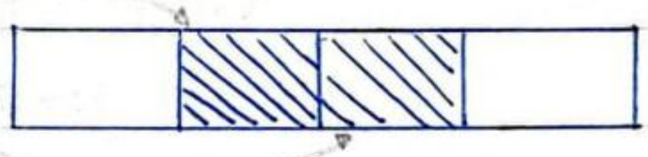
$= \cdots = n \cdot cn = \Theta(n^2)$

BUT...

To achieve $O(n)$ S not required to be median, as long as able to split A by constant factor $b>1$

"Good" splitter S: $\geq \frac{1}{4}$ of the elements in A are $<S$

$\geq \frac{1}{4}$ of the elements in A are $>S$



S in shaded region reduces input size from n to $\leq \frac{3}{4}n \Rightarrow b = \frac{4}{3} > 1$

can split around middle only sometimes and still have $O(n)$

Let Random Variable Z be # of trials before chosen S is good splitter

$E(Z) = \sum_v P(Z=v) \cdot v = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \cdots + \frac{1}{2^i} \cdot i = \sum_{i=1}^{\infty} i(\frac{1}{2})^i = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$

($\frac{1}{2}$: $P(Z=1)$; $Z \sim Geo(\frac{1}{2})$)

Define: Algorithm is in phase j if size of input n_i, $(\frac{3}{4})^{j+1} n < n_i \leq (\frac{3}{4})^j n$

$n = n_0 \geq n_1 \geq n_2 \geq n_3 \geq n_4 \geq n_5 \geq n_6 \geq \cdots$ n_i = size of input on ith recursive call

e.g. $n \geq n_i > \frac{3}{4}n$ (phase 0); $\frac{3}{4}n \geq n_i > (\frac{3}{4})^2 n$; $(\frac{3}{4})^2 n \geq n_i > (\frac{3}{4})^3 n$

Let RV X = # steps taken by algorithm

RV X_j = # steps taken by algorithm while in phase j

↳ $X = X_0 + X_1 + X_2 + \cdots$

$X_j \leq$ max # of steps for a recursive call in phase j $\times$ # of recursive calls in phase j

$\leq (\frac{3}{4})^j n$ ← largest possible input size in phase j ; RV z_j

$E(X_j) \leq c(\frac{3}{4})^j n \cdot E(z_j) \Rightarrow E(X_j) \leq 2cn(\frac{3}{4})^j$

↳ $E(X) = E(\sum_j X_j) = \sum_j E(X_j) \leq \sum_j 2cn(\frac{3}{4})^j = 2cn \sum_j (\frac{3}{4})^j \leq 8cn \Rightarrow \underline{\Theta(n)}$

algorithm 2: D-SELECT (A, k)

pick S instead of random:

if $|A| \leq 5$ then sort A and return kth element

Partition A into $\frac{n}{5}$ groups of 5 elements each

Sort each group to find median // sort 5 elements is constant,

M := list of medians // all groups = $\frac{n}{5} \Rightarrow \Theta(n)$

S := D-SELECT $(M, \lceil \frac{|M|}{2} \rceil)$ // this choice always splits A to // size of $\frac{3}{4}n$

Same before:

A^- := list of elements in A $< S$

A^+ := list of elements in A $> S$

if $k \leq |A^-|$ then return D-SELECT (A^-, k)

else if $k = |A^-| + 1$ then return S

else return D-SELECT $(A^+, k - (|A^-| + 1))$

running time: $T(n) \leq T(\lceil \frac{n}{5} \rceil) + T(\lfloor \frac{3n}{4} \rfloor) + cn$ ← sort $\frac{n}{5}$ groups + partition A or A into A^-, A^+

pick S ; finding k-th element

$\leq d\lceil \frac{n}{5} \rceil + d\lfloor \frac{3n}{4} \rfloor + cn$ [by IH]

$\leq d(\frac{n}{5} + 1) + d\frac{3n}{4} + cn$

$= dn(\frac{19}{20}) + (c+1)n$

$\leq dn \Rightarrow T(n) = \Theta(n)$

WTS $T(n) \leq dn$

for $n \geq d$, pick $d \geq 20(c+1)$

because $dn \geq \frac{19}{20}dn + (c+1)n$

$\frac{1}{20}d \geq c+1$

$d \geq 20(c+1)$

DYNAMIC PROGRAMMING

LARGEST INCREASING SUBSEQUENCE (LIS)

input: A sequence of numbers $A = a_1, a_2, \dots, a_n$

output: (length of) A LIS of A

Optimal substructure of Problem

⇔ optimal solution to a problem is an extension of an optimal solution to a subproblem

Suppose we have a LIS of A = S , S ends at position i of A

Let $S = S' a_i$ where S' - longest increasing subsequence of $A - a_i$

- ends at position j, $j < i$ and $a_j < a_i$

Suppose for contradiction S' is not !

⇒ ∃ S'' that is an - increasing subsequence of A^{-a_i} ends in some position k

- $k < i$ and $a_k' < a_i$

- $|S''| > |S'|$ ⇒ $S'' a_i$ is an increasing subsequence of A ending at i longer than S

↳ contradiction

we have: length of LIS ending at i = 1 + length of LIS ending at j

define $L(i)$:= length of LIS ending at i (*)

the recursive formula we get from definition of (*)

$$L(i) := \begin{cases} 1 + \max\{L(j) \text{ s.t. } j < i \text{ and } a_j < a_i\} & \text{if } \exists j < i \text{ s.t. } a_j < a_i \\ 1 & \text{otherwise} \end{cases} \quad (**)$$

algorithm: LIS(A)

```
for i := 1 to n do
    L(i) := 1 ; pre(i) := 0
    for j := 1 to i-1 do
        if A[j] < A[i] and L(i) < L(j)+1 then
            L(i) := L(j)+1
```

pre(i) := j

m := maxarg(L)

return L[m]

running time: $O(n^2)$

Description Of Dynamic Programming Algorithm To Solve Problems

1. Define a (polynomial) # of subproblems to solve [from which we can "easily" solve P]
2. Give a recursive formula to compute solution to subproblem
3. Derive solution to P from solutions to subproblems

Weighted Interval Scheduling

input: A set of intervals $1, 2, \cdots, n$. For interval i: start time $s(i)$, finish time $f(i)$, $f(i) \geq s(i)$

output: A feasible subset S of intervals with value $v(i)$

max value: $V(S) = \sum_{i \in S} v(i)$

Assume intervals sorted in increasing finish time

i.e. $\forall i = 1, 2, \cdots, n$ define $P(i) = \begin{cases} \max j < i \text{ s.t } f(j) \leq s(i) \\ 0 \text{ if no such } j \end{cases}$

↑ a j that goes before i

Let S be optimal set of intervals for $1, 2, \cdots, n$

1. $n \notin S$: S is optimal for $1, 2, \cdots, n-1$
2. $n \in S$: $S = S' \cup \{n\}$, S' is optimal for $1, 2, \cdots, p(n)$

Subproblems: Find optimal set of intervals for $1, \cdots, i$ $\forall i = 1, 2, \cdots, n$

Let $V(i)$ = value of an optimal set for intervals $1, \cdots, i$ (*)

Recursive Formula: $V(i) = \begin{cases} \max(\underbrace{V(i-1)}_{①}, \underbrace{V(p(i)) + v(i)}_{②}), & \text{if } i > 0 \quad (+) \\ 0, & \text{if } i = 0 \end{cases}$

algorithm: sort intervals by finish time — $O(n \log n)$

```
for i := 1 to n do
    if there exists j s.t. f(j) ≤ s(i) do
        p(i) := max(all those js)
    else
        p(i) := 0
```

$O(n \log n)$

```
V(0) := 0
for i := 1 to n do
    if V(i-1) ≥ V(p(i)) + v(i) then V(i) := V(i-1)   // S(i) := S(i-1)
    else V(i) := V(P(i)) + V(i)                      // S(i) := S(p(i)) ∪ {i}
```

$O(n)$

```
S := φ ; i := n
while i ≠ 0 do
    if V(i) = V(i-1) then i := i-1
    else S := S ∪ {i} ; i := p(i)
return V(n)
```

for finding the intervals

running time: $O(n \log n)$

0/1 KNAPSACK

input: A set of items $1, 2, \cdots, n$. For item i: weight $w(i) > 0$, value $v(i) > 0$

knapsack capacity $C > 0$

$w(i), C \in \mathbb{Z}$

output: Knapsack S of max value ~~$v(S)$~~ $V(S) = \sum_{i \in S} v(i)$

$S \subseteq \{1, 2, \cdots, n\}$ is a knapsack if $\sum_{i \in S} w(i) \leq C$

Let S be optimal knapsack for $1, 2, \cdots, n$

1. $n \notin S$: S is optimal knapsack for $1, 2, \cdots, n-1$
2. $n \in S$: $S = S' \cup \{n\}$, S' is optimal knapsack for $\{1, 2, \cdots, n-1\}$ with remaining capacity $C - w(n)$

SUBPROBLEMS: Find optimal knapsack for items $1, 2, \cdots, i$ with remaining capacity c, $\forall i = 0, 1, 2, \cdots, n$ and $\forall c = 0, 1, 2, \cdots, C$

Let $K(i, c)$ = value of an optimal knapsack for item $1, \cdots, i$ with available capacity $c \in \mathbb{Z}$ (*)

Recursive Formula:

$$K(i,c) = \begin{cases} \max(K(i-1,c),\ v(i) + K(i-1, c-w(i))) & \text{, if } i>0 \text{ and } c \geq w(i) \\ K(i-1, c) & \text{, if } i>0,\ c>0 \text{ and } c<w(i) \\ 0 & \text{, if } i=0 \text{ or } c=0 \end{cases} \quad (\dagger)$$

algorithm: KNAPSACK($w[1 \cdots n]$, $v[1 \cdots n]$, C) — weights, values

```
for c := 0 to C do k(0,c) := 0          ← O(C)
for i := 0 to n do k(i,0) := 0          ← O(n)
for i := 1 to n do                      } O(nC)
    for c := 1 to C do
        if c < w(i) then k(i,C) := k(i-1,c)
        else if k(i-1,c) ≥ k(i-1, c-w(i)) then
            k(i,c) := k(i-1,c)
        else
            k(i,c) := k(i-1, c-w(i)) + v(i)
return K(n,C)

S := ∅ ; i := n ; c := C                // if want set of items
while i>0 and c>0 do
    if K(i,c) = K(i-1,c) then i = i-1
    else S := S ∪ {i} ; i := i-1 ; c := c-w(i)
```

running time: $O(nC)$

"pseudo polynomial-time algo"

- polynomial time: polynomial in size of input

assumption: weights and values have size around size representation of C.

(using) size of input: $n \log_2 C + \log_2 C = O(n \log C)$ ← C is EXPONENTIAL in $\log C$

(C: value size; $\log C$: size input)

⇒ Algorithm has running time

- exponential in input size, polynomial in value of input, i.e. pseudo polynomial time

NP-COMPLETE PROBLEMS

- none has poly time algo
- if one of them has a poly time algo then they all do

OPTIMAL ALIGNMENT (EDIT DISTANCE)

Consider $X[1 \cdots m]$ two strings. How different are they?

$y[1 \cdots n]$

Edit Distance: The min number of edits (single character change) needed to transform $x \to y$.

Edits: insert / delete / change

e.g. x = boarder y = barbers

```
boarder-        or      boarder-
-barbers                b-arbers
 ↑↑ ↑ ↑                  ↑ ↑ ↑
 d c c i                 d c i
```

= 4 edits = 3 edits

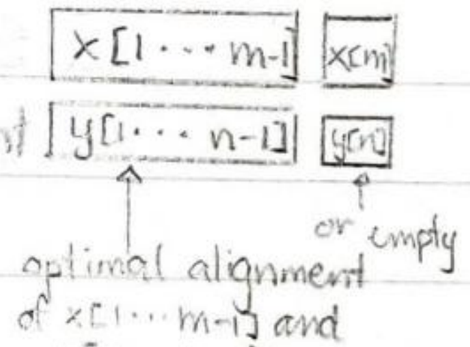
input: $X[1 \ldots m], y[1 \ldots n]$

output: Edit distance between x and y

Look at end of optimal alignment

1. $\begin{matrix} x[m] \\ y[m] \end{matrix}$ (match last symbols)
2. $\begin{matrix} x[m] \\ - \end{matrix}$ (last symbol of x and empty => delete)
3. $\begin{matrix} - \\ y[n] \end{matrix}$ (last symbol of y and empty => insert)

} optimal alignment for



subproblem: Let $ED(i,j)$ = edit distance between $x[1 \cdots i], y[1 \cdots j]$.

~~what~~ Want to find $E(i,j)$ $\forall\, 0 \le i \le m$ and $0 \le j \le n$ $(*)$

recursive formula:

$$ED(i,j) = \begin{cases} \min(ED(i-1,j-1) + diff(i,j),\ ED(i-1,j)+1,\ ED(i,j-1)+1), & \text{if } i>0,\ j>0 \\ j, & \text{if } i=0 \text{ (j inserts)} \\ i, & \text{if } j=0 \text{ (i deletions)} \end{cases} \quad (\dagger)$$

$$diff(i,j) = \begin{cases} 0 & \text{if } x[i] = y[j] \\ 1 & \text{o/w} \end{cases}$$

algorithm: EDITDISTANCE(X[1...m], y[1...n])

```
for i := 0 to m do ED(i,0) := i
for j := 1 to n do ED(0,j) := j
for i := 1 to m do
    for j := 1 to n do
        ED(i,j) := min(ED(i-1,j-1)+diff(i,j), ED(i-1,j)+1, ED(i,j-1)+1)
return ED(m,n)
```

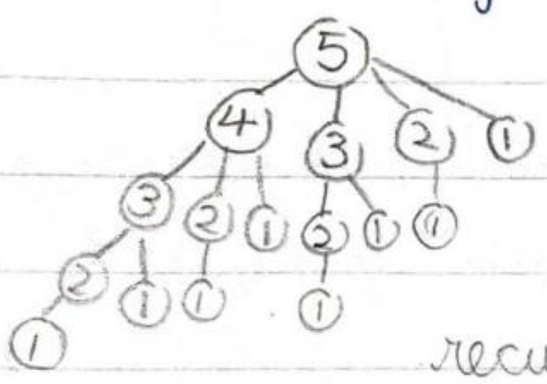
running time: $\Theta(mn)$

RECURSION AND DYNAMIC PROGRAMMING

DP solves overlapping subproblems

LIS: $L(i) = 1 + \max\{L(j) \cdot 1 \le j < i \text{ and } a_j < a_i\}$

e.g.



← many duplicate subproblems!

recursion tree for solving LIS $L(n) = 2^{n-1}$

MEMOIZATION

```
ED(m,n)
if m = 0 then return n
else if n = 0 then return m
else return min(ED(m-1,n-1)+diff(m,n), ED(m,n-1)+1, E(m-1,n)+1)
```

exponential!

↓

```
Initialize T(i,j) := nil  ∀i,j  1 ≤ i ≤ m, 1 ≤ j ≤ n
MED(m,n)                                   //memoization
if m = 0 then return n
else if n = 0 then return m
else
    if T(m,n) = nil then T(m,n) := min(MED(m-1,n-1)+diff(m,n), MED(m-1,n)+1,
                                       MED(m,n-1)+1)
return T(m,n)
```

$\Theta(mn)$

↓

No

Date

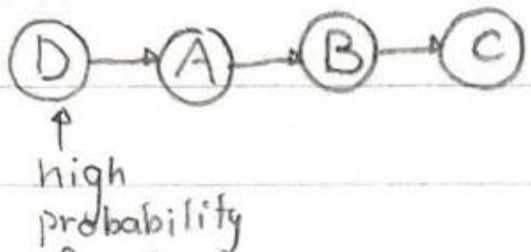
OPTIMAL BINARY SEARCH TREE

We have n records with keys 1, 2, ⋯, n (in a static dictionary)
with P(i) = probability of searching for key i
Linked List L

$C(L) = \sum_{i=1}^{n} P(i) \cdot \text{position}(i, L)$ ← want an L that minimizes C(L)

e.g.

	P( )
A	0.25
B	0.23
C	0.22
D	0.30

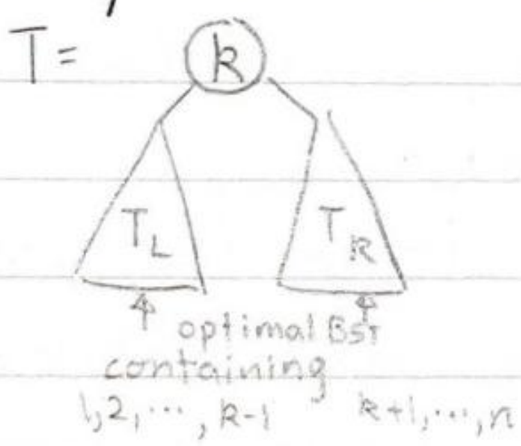


Binary ~~tree~~search tree T $\frac{\binom{2n}{n}}{n+1} = 4^n$ possible trees

$C(T) = \sum_{i=1}^{n} p(i) \cdot (\text{depth}(i, T) + 1)$

input: A set of keys 1, 2, ⋯, n with P(i) = probability of searching key i
output: An optimal BST that minimizes $C(T) = \sum_{i=1}^{n} p(i)(\text{depth}(i, T) + 1)$

Let T be optimal BST for P( )



CLAIM: $C(T) = C(T_L) + C(T_R) + \sum_{u \in T} P(u)$

PROOF $C(T) = \sum_{i \in T} p(i) \cdot (\text{depth}(i, T) + 1)$ ($\text{depth}(i, T_L)+1$, $\text{depth}(i, T_R)+1$)

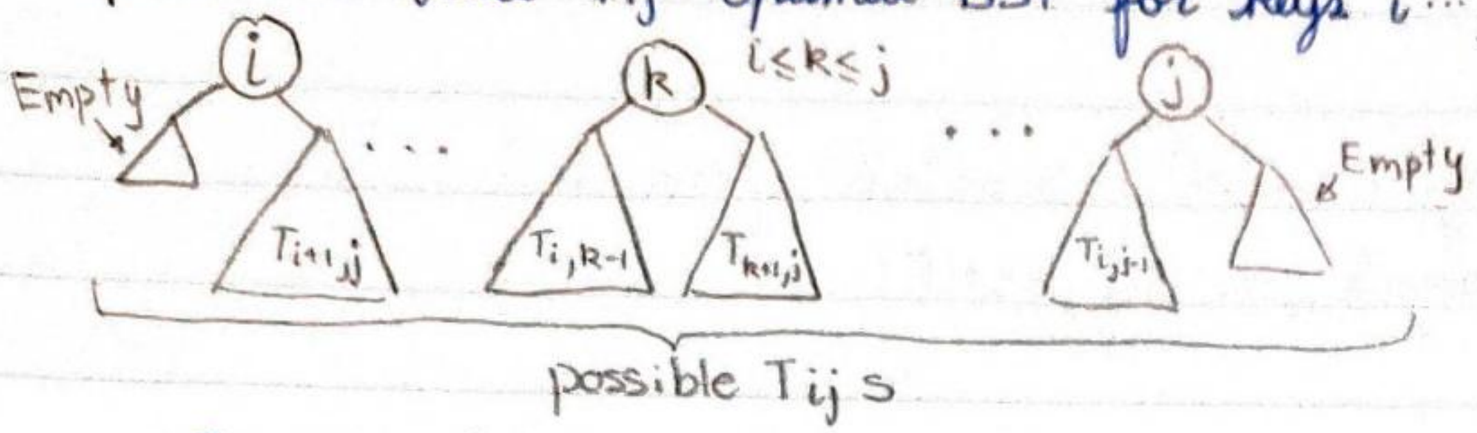
$= p(k) + \sum_{i \in T_L} p(i)(\text{depth}(i, T) + 1) + \sum_{i \in T_R} p(i)(\text{depth}(i, T) + 1)$

($\text{depth} + 1 = 1$)

$= p(k) + \sum_{i \in T_L} p(i) + \sum_{i \in T_R} p(i) + \sum_{i \in T_L} p(i)(\text{depth}(i, T_L) + 1) + \sum_{i \in T_R} p(i)(\text{depth}(i, T_R) + 1)$

$= \sum_{u \in T} p(u) + C(T_L) + C(T_R)$ ∎

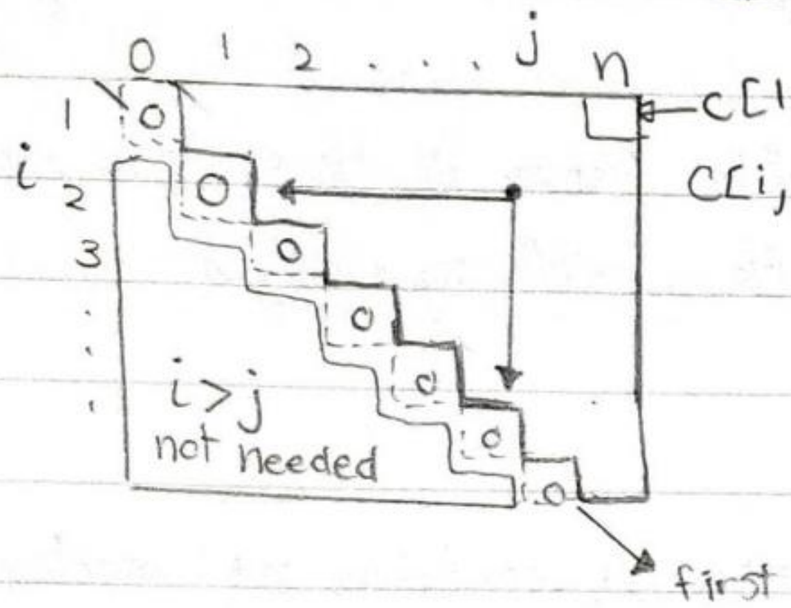
Subproblems: find T_{ij} = optimal BST for keys $i \cdots j$ $1 \le i \le j \le n$



$C[i,j] = c(T_{ij})$ (*) $1 \le i \le j \le n$

~~$C[i,j]$~~

recursive formula: $C[i,j] = \begin{cases} \min\limits_{i \le k \le j} \{C(i,k-1) + C(k+1,j) + \sum\limits_{u=i}^{j} P(u)\} & i < j+1 \quad (+) \\ 0 & i = j+1 \end{cases}$



algorithm: OPT BST (P[1 ... n])

```
S[0] := 0
for i := 1 to n do S[i] := S[i-1] + P(i)
for j := 0 to n do C[j+1, j] := 0
for d := 0 to n-1 do

    for i := 1 to n-d do
        j := i+d
        C[i,j] := ∞
        for k := i to j do
            C[i,j] := min(C[i,j], C[i,k-1] + C[k+1,j] + S[i] - S[i-1])
return C[1,n]
```

running time : $O(n^3)$

Catalan Number $C(n)$ = # of BSTs with n nodes

$$C(n) = \begin{cases} 1 & , n=0 \\ \sum\limits_{k=1}^{n} C(k-1) \cdot C(n-k) \end{cases} \qquad C(n) = \frac{\binom{2n}{n}}{n+1}$$

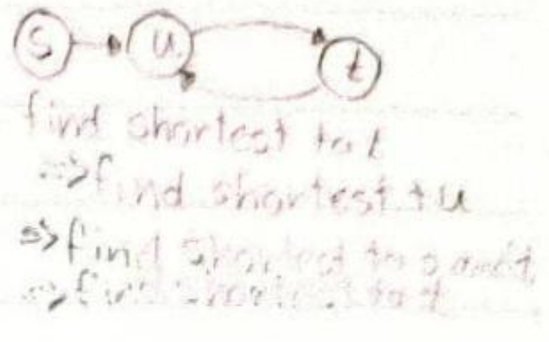
(4)

Date

DP_AND_SHORTEST_PATHS_

BELLMAN-FORD_

Assume no negative weight cycles reachable from s



input: $G=(V,E)$ directed graph, $n=|V|$, $m=|E|$

s start node

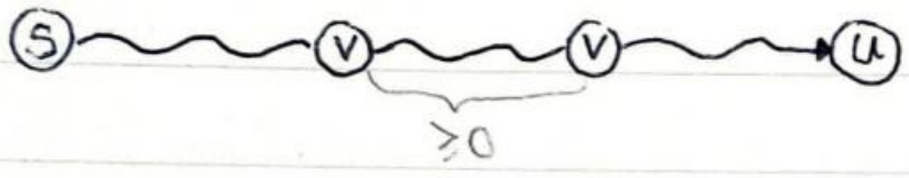
$wt: E \to \mathbb{R}$ weight functions (can be negative)

output: compute shortest path from s to predecessor of t

Subproblems: Find (weight of) shortest $s \to u$ paths that uses $\leq k$ edges

(k-path)

CLAIM 1: If G has no neg-wt cycles reachable from s then $\forall u$ reachable from s there is a shortest $s \to u$ path with $\leq n-1$ edges

PROOF $P =$ 

Suppose by contradiction n edges $\Rightarrow$ path contains $n+1$ nodes

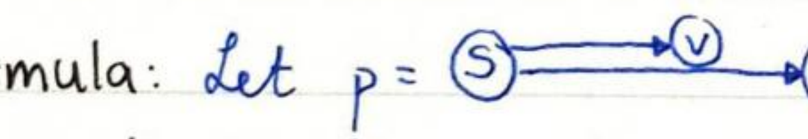
$\Rightarrow \exists$ a cycle with node v repeated

Since we assumed there are no negative weight cycles

↳ path $v \to v \geq 0 \Rightarrow$ can remove the cycle and it's shortest with $\leq n-1$ edges.

Define $L(u,k) = \begin{cases} \min wt(s \overset{k}{\to} u) & (s \to u \text{ path using} \leq k \text{ edges}) \\ \infty & \text{if no such path} \end{cases}$

$\forall u \in V, \forall k = 0, 1, \cdots, n-1$ (*)

recursive formula: Let $p =$ 

shortest k-path

1. p is a $(k-1)$ path $\Rightarrow L[u,k) = L(u,k-1)$
2. p has exactly k edges $\Rightarrow L(u,k) = \min\{L(v,k-1) + wt(v,u) : (v,u) \in E\}$

$$L(u,k) = \begin{cases} \min(\{L[u,k-1]\} \cup \{L[v,k-1] + wt(v,u) : (v,u) \in E\}) & \text{if } k>0 \\ 0 & \text{if } k=0 \text{ and } u=s \\ \infty & \text{if } k=0 \text{ and } u \neq s \end{cases} \quad (\dagger)$$

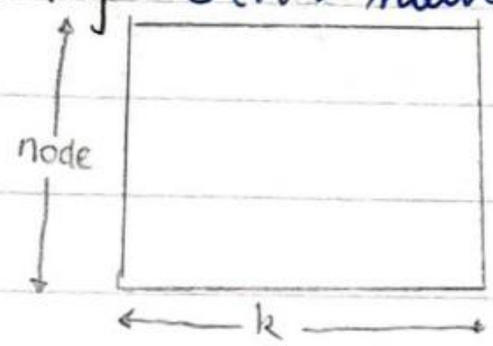
algorithm: BF(G, s, wt)

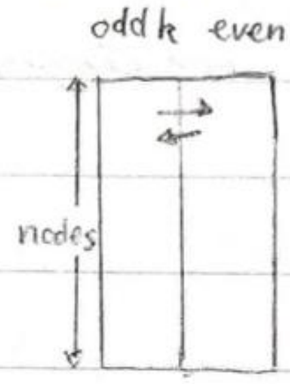
```
L[s,0] := 0
foreach u≠s do L[u,0] := ∞
for k := 1 to n-1
    for each u∈V do
        L[u,k] := L[u,k-1]
        foreach v∈V s.t. (v,u)∈E do
            if L[u,k] > L[v,k-1] + wt(v,u) then
                L[u,k] := L[v,k-1] + wt(v,u)
```

running time: Adjacency matrix ⇒ $O(n^3)$

Adjacency list ⇒ ~~$O(n^3)$~~ $O(nm)$

Space complexity: $O(n^2)$ naively





each compare only need the previous k, so only 2 columns required.

DETECTING NEG-WT CYCLES

CLAIM 2: G has no neg-weight cycle reachable from s $\Leftrightarrow \forall u\ L[u,n] = L[u,n-1]$

PROOF "⇒" immediately from claim 1.

"⇐" follows from

① $\forall u, L[u,k] = L[u,k-1] \Rightarrow \forall u, L[u,k+1] = L[u,k]$

$L[u,k+1] = \min(\{L[u,k]\} \cup \{L[v,k] + wt(v,u) : (v,u)\in E\})$

$= \min(\{L[u,k-1]\} \cup \{L[v,k-1] + wt(v,u) : (v,u)\in E\})$

(↓ from assumption; ↓ from assumption)

$= L[u,k]$

Not $\forall u\ (L[u,k] = L[u,k-1] \Rightarrow L[u,k+1] = L[u,k])$

↳ so even if you allow more edges, the min weight $s \to u$ k-path stays the same. That means no negative weight cycles

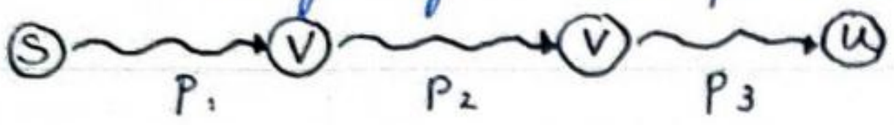
■

No.

Date:

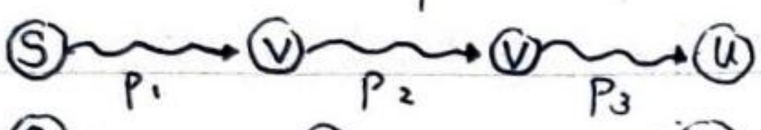
CLAIM 3: For any $u \in V$ s.t. $L(u,n) \neq L(u,n-1)$
if p is a shortest $s \xrightarrow{n} u$ n-path then
a. p contains a cycle and
b. every cycle on p has wt < 0

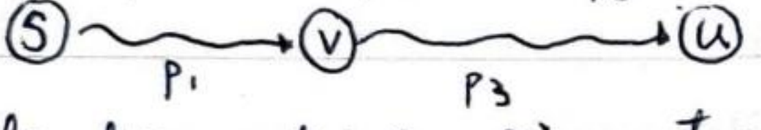
PROOF



Since $L[u,n] \neq L[u,n-1]$, p has exactly n edges

a) holds because p_2 is a $V \to V$ path

if $p =$ 

$p' =$ 

← shorter than p

if cycle has wt ≥ 0, p' contradicts that min-wt n-path has at most n edges, as from ~~so~~ claim one no negative cycles $\Rightarrow$ path has $n-1$ edges

Finding neg-wt cycle
- Run BF for n iterations
  - if no L-label changes in n-th iteration, then there is no neg-wt cycle reachable from s (CLAIM 1)
  - if u's L-label changes then follow backwards min-wt n-path from u back to s. We will discover a neg-wt cycle (CLAIM 3)

FLOYD-WARSHALL'S ALGO (ALL PAIRS SHORTEST PATH)

input: Digraph $G = (V, E)$, $wt : E \to \mathbb{R}$

output: (weights of) shortest $u \to v$ paths $\forall u, v \in V$

Assume WLOG that $V = \{1, 2, \cdots, n\}$ no nodes $> k$

Define $i \xrightarrow{k} j$ path : $i \to j$ path with no intermediate nodes $> k$ 

Subproblems: Define $C[i,j,k] = \begin{cases} \min wt(i \xrightarrow{k} j \text{ paths}) & (*) \\ \infty, \text{ if no such path} \end{cases}$

$\forall i \in V, j \in V, 1 \leq k \leq n$ ($k \in V$)

recursive formula: Let p be shortest $i \xrightarrow{k} j$ path

1. k not intermediate node in P : p is an $i \xrightarrow{k-1} j$ path $\Rightarrow C[i,j,k] = C[i,j,k-1]$
2. k is an intermediate node in P: p_1 is a shortest $i \xrightarrow{k-1} k$ path

p_2 is a shortest $k \xrightarrow{k-1} j$ path

$\Rightarrow C[i,j,k] = C[i,k,k-1] + C[k,j,k-1]$

$$C[i,j,k] = \begin{cases} \min(C[i,j,k-1], C[i,k,k-1] + C[k,j,k-1]) & \text{if } k>0 \\ wt(i,j) & \text{if } k=0 \text{ and } (i,j) \in E \\ \infty & \text{if } k=0 \text{ and } (i,j) \notin E \\ 0 & \text{if } i=j \text{ and } k=0 \end{cases} \quad (\dagger)$$

algorithm: FW(G, wt)

```
for i := 1 to n do
    for j := 1 to n do
        if i = j then C[i,j,0] := 0
        else if (i,j) ∈ E then C[i,j,0] := wt(i,j)
        else C[i,j,0] := ∞
for k := 1 to n do
    for i := 1 to n do
        for j := 1 to n do
            C[i,j,k] := min(C[i,j,k-1], C[i,k,k-1] + C[k,j,k-1])
```

running time: $\Theta(n^3)$

space complexity: naively $\Theta(n^3)$, $\Theta(n^2)$ suffice

TRANSITIVE CLOSURE OF DIGRAPH $G = (V, E)$

$G^* = (V, E^*)$ $E^* = \{(u,v) : \exists\, u \to v \text{ path in } G\}$

Use Floyd-Warshall algo to compute transitive closure if there is a shortest path from i to j $\Leftrightarrow$ in $G \Rightarrow$ there is a path from i to j in G^*

subproblems: $C[i,j,k] = \begin{cases} 1 & \text{if } \exists\, i \xrightarrow{k} j \text{ path} \\ 0 & \text{otw} \end{cases}$

No.

Date

recursive formula: $C[i,j,k] = \begin{cases} C[i,j,k-1] \vee (C[i,k,k-1] \wedge C[k,j,k-1]) \\ 1 \quad \text{if } k=0 \text{ and } (i=j \text{ or } (i,j) \in E) \\ 0 \quad \text{o/w} \end{cases}$

DETECTING NEG-WT CYCLES

CLAIM 4: G has a negative weight cycle $\Leftrightarrow \exists\, u$ s.t. $C[u,u,n] < 0$

PROOF from assignment 5+6

JOHNSON ALL PAIRS

Floyd Warshall : $O(n^3)$

Dijkstra's n times : $O(nm\log n)$ is better than $O(n^3)$ if G (few edges) is sparse, but assumes edges have nonnegative weights

Reweigh edges s.t.

a. All edges have $wt \geq 0$

b. New weight preserves shortest paths (no long paths being penalised)

METHOD: Assign weight x_u to node u $\forall u \in V$

Define $wt' : E \to \mathbb{R}$, $wt'(u,v) = wt(u,v) + x_u - x_v$, $\forall (u,v) \in E$ $x_v - x_u \leq wt(u,v)$ (**) ▷ so $wt'(u,v) \geq 0$

For a path $p = (u_1, u_2, \cdots, u_k)$

$wt(p) = wt(u_1,u_2) + wt(u_2,u_3) + \cdots + wt(u_{k-1},u_k)$

$wt'(p) = wt(u_1,u_2) + x_{u_1} - x_{u_2} + wt(u_2,u_3) + x_{u_2} - x_{u_3} + wt(u_3,u_4) + x_{u_3} - x_{u_4} + \cdots + wt(u_{k-1},u_k) + x_{u_{k-1}} - x_{u_k}$

$= wt(p) + x_{u_1} - x_{u_k}$ (*)

(*) implies all $u \to v$ paths change by same amount

$\Rightarrow$ p is shortest $u \to v$ path under wt $\Leftrightarrow$ p is shortest $u \to v$ path under wt'

CLAIM 5: (**) are satisfiable $\Leftrightarrow$ G has no neg-wt cycle under wt'

PROOF "$\Rightarrow$" Assume for contradiction that (**) are satisfied by setting $x_u = \hat{x}_u$, but G has negative weight cycle $u_1, u_2, \cdots, u_k, u_1$

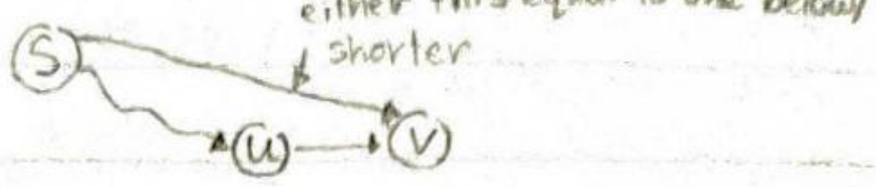
by (**), $\hat{x}_{u_2} - \hat{x}_{u_1} \leq wt(u_1,u_2)$

$\Rightarrow \hat{x}_{u_2} - \hat{x}_{u_1} + \hat{x}_{u_3} - \hat{x}_{u_2} + \cdots + \hat{x}_{u_k} - \hat{x}_{u_{k-1}} + \hat{x}_{u_1} - \hat{x}_{u_k} \leq wt(u_1,u_2) + wt(u_2,u_3) + \cdots + wt(u_k,u_1)$

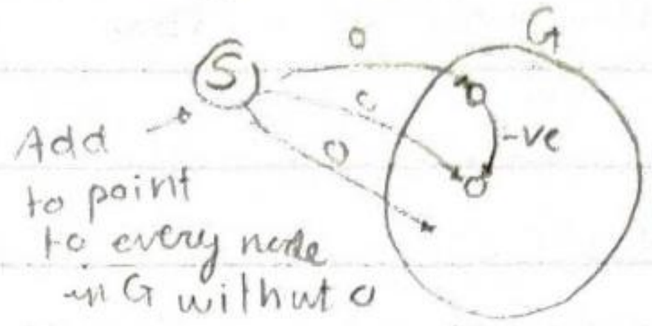
$0 \leq wt(\text{cycle}) < 0 \Rightarrow$ contradiction!

"⇐" $x_v \leq x_u + wt(u,v)$ ← either a shorter $s \to v$ path exist or $s \to u \to v$ path is shortest

x_v: wt of shortest $s \to v$ path; x_u: wt of shortest $s \to u$ path



$x_v - x_u \leq wt(u,v)$, as wanted



algorithm: Johnson (G, wt)

- Define $\hat{G} = (\hat{V}, \hat{E})$ $\hat{V} = V \cup \{s\}$, $\hat{E} = E \cup \{(s,u) : u \in V\}$
- Define $\hat{wt} : \hat{E} \to \mathbb{R}$ $\hat{wt}(s,u) = 0$, $\hat{wt}(u,v) = wt(u,v)$ $\forall u, v \in V$
- Run BF on $\hat{G}, s, \hat{wt}$ to compute x_u = wt of shortest $s \to u$ path in $\hat{G}$, or to determine that $\hat{G}$ has a negative weight cycle
- If $\hat{G}$ has a negative weight cycle then return "impossible"
- For each $(u,v) \in E$ do $wt'(u,v) = wt(u,v) + x_u - x_v$
- For each node $u \in V$ do
  - run Dijkstra's (G, w, wt')
  - $D`[u,v]$:= wt of shortest $u \to v$ path under wt'
- For each $u, v \in V$ do $D[u,v] := D`[u,v] + x_v - x_u$

running time: $O(n\, m \log n)$

MAXIMUM FLOW AND APPLICATIONS

Define Flow network $\mathcal{F} = (G, s, t, C)$: $G = (V, E)$ is a digraph, $s, t \in V$ s.t.
s has no incoming edges, $C : E \to \mathbb{R}^+$ maps edges to positive reals (capacity)
t has no outgoing edges

Define Flow in $\mathcal{F}$: $f : E \to \mathbb{R}$ s.t.

a. capacity $\forall e \in E$ $0 \leq f(e) \leq C(e)$ (flow is positive, lower than capacity)

b. conservation $\forall u \neq s, t$, $\sum_{e \in in(u)} f(e) = \sum_{e \in out(u)} f(e)$ (how much goes in, how much goes out)

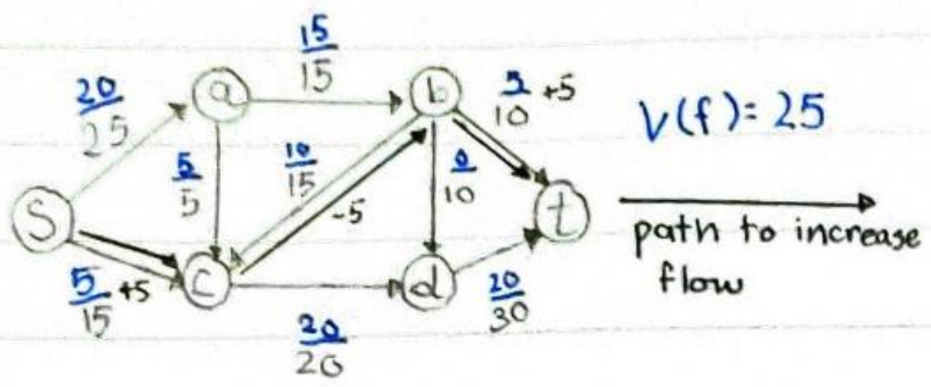
$= \{(v,u) : (v,w) \in E\}$ $= \{(u,v) : (u,v) \in E\}$

Define Value of flow f $V(f) = \sum_{e \in out(s)} f(e)$

No.

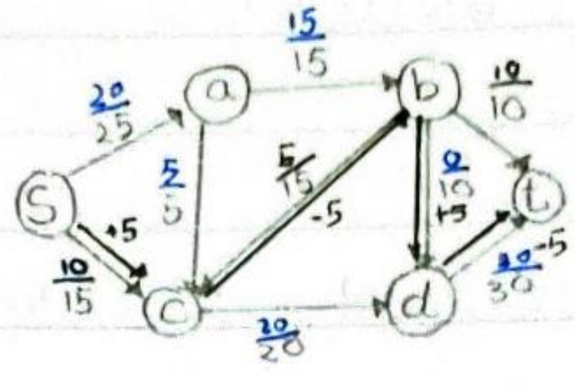
Date

e.g.



$V(f) = 25$

path to increase flow



$V(f') = 30$

$V(f'') = 35$

flow / capacity

u → v

MAX FLOW PROBLEM — EQUIVALENTLY — MIN CUT PROBLEM

input: Flow network $\mathcal{F} = (G, s, t, c)$

output: Flow f of max value $V(f) \geq V(f')$ $\forall$ flows f' in $\mathcal{F}$

input: Flow network $\mathcal{F}$

output: A cut S, T of min capacity

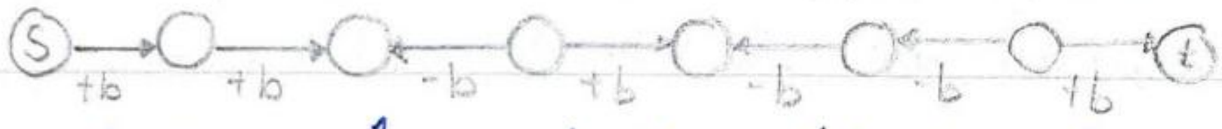
i.e. $\forall$ cuts (S', T') $c(S, T) \leq c(S', T')$

Define cut of $\mathcal{F}$ (S, T): $S, T \subseteq V$ s.t. $S \cup T = V$ $S \cap T = \phi$

~~for $s \in S, t \in T$~~

Define capacity of cut (S, T): $c(S, T) = \sum_{\substack{e \in out(s) \\ \cap in(T)}} c(e)$ $\left(out(S) = \bigcup_{u \in S} out(u)\right)$

FIND SIMPLE "PATH" (assume no direction)



- Increase flow by b on forward edges ($b \leq c(e) - f(e)$, $\forall$ forward edge e)
- Decrease flow by b on backward edges ($b \leq f(e)$)

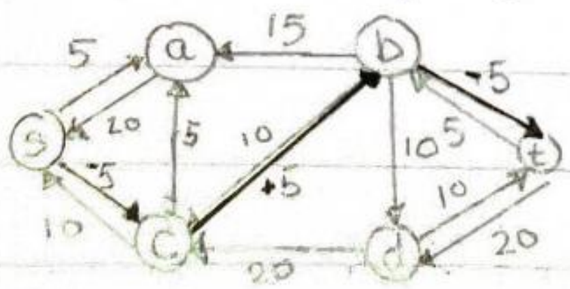
FLOW AUGMENTATION STEP

Define Residual graph G_f of $\mathcal{F} = (G, s, t, c)$ wrt flow $\mathcal{F}$

: $G_f = (V, E_f)$ $c_f : E_f \to \mathbb{R}^+$ ← residual capacities (V, E)

- For every $e = (u, v)$ of G s.t. $f(e) < c(e)$, add (u, v) to E_f with $c_f(e) = c(e) - f(e)$ ← how much more flow the edge can take
- For every $e = (u, v)$ of G s.t. $f(e) > 0$, add (v, u) to E_f with ~~$c_f(e)$~~ $c_f(v, u) = f(e)$ ← how much flow can be decreased

e.g.



AUGMENTATION STEP FOR f

algorithm: augment(f, p) // p is simple s→t path in G_f

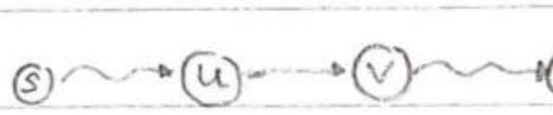
```
b := min residual capacity of any edge on p
for each edge (u,v) of p
    if (u,v) is foward edge then f(u,v) := f(u,v)+b
    else f(v,u) := f(v,u)-b          // backward edge
return f                             // all edges not in p remain same, other edges in p either ↑ or ↓ by b
```

running time: $O(m)$

LEMMA1: If f is a flow then f` = augment(f,p) is a better flow

PROOF 1. f` is a flow

a. capacity

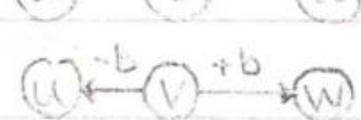


by def of b

i. (u,v) forward: $f'(u,v) = f(u,v) + b \leq f(u,v) + c(u,v) - f(u,v) \leq c(u,v)$

ii. (u,v) backward: $f'(v,u) = f(v,u) - b \geq f(v,u) - f(v,u) \geq 0$

b. conservation

i.	F	F		v gets more, gives more
ii.	F	B		u gives more, w gives less
iii.	B	F		u gets less, w gets more
iv.	B	B		v gets less, gives less

2. f` is better i.e. ~~f`>f~~ V(f`) > V(f)

$V(f') = V(f) + b_{>0} > V(f)$  always forward => +b

MAX FLOW (FORD-FULKERSON MAX FLOW ALGORITHM)

algorithm: FF(J) // J=(G,s,t,c), n=|V|, m=|E|

```
for each edge (u,v) ∈ G do f(u,v) := 0        // O(m)
construct residual graph Gf                   // O(n+m) = O(m)
while Gf has an s→t path do                   // repeat O(C)
    p := any simple s→t path                  // O(m)
    f := augment(f,p)                         // O(m)
    update Gf                                 // O(n)
return f
```

} O(m)

No

Date

running time: $O(mC)$, $C = \sum_{e \in out(s)} c(e)$, for any flow f: $V(f) \leq C$

↑ pseudo polynomial

assumes C is an integer (or rational), then FF terminates with max flow

Better Chance Of Augmenting Paths

① "Fastest Path" (always find path with max b) → terminates in $O(m \log C)$ iterations

$\Rightarrow O(m^2 \log C)$ time

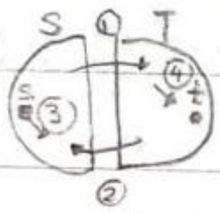
② "fewest edge path" → terminates in $O(mn)$ iterations $\Rightarrow O(m^2 n)$ time

can actually be done in $O(mn)$ but very complicated

Proof of Correctness

LEMMA 2: For any flow f, any cut (S,T): $v(f) = \sum_{\substack{e \in out(s) \\ \cap in(T)}} f(e) - \sum_{\substack{e \in in(s) \\ \cap out(T)}} f(e) = \sum_{e \in out(s)} f(e)$

$S \to T$ $T \to S$ source



① $s \to t$ ③ stay in S

② $t \to s$ ④ stay in t

PROOF $V(f) = \sum_{u \in S} \left(\sum_{e \in out(u)} f(e) - \sum_{e \in in(u)} f(e) \right)$ for $u = s$: $V(f) - 0$

$= \sum_{u \in S} \sum_{e \in out(u)} f(e) - \sum_{u \in S} \sum_{e \in in(u)} f(e)$ $u \neq s$: $\sum_{e \in out(u)} f(e) = \sum_{e \in in(u)} f(e) \Rightarrow 0$

by def no edges into s

simplify simplify

$= \sum_{e \in out(S)} f(e) - \sum_{e \in in(S)} f(e) = \sum_{\substack{e \in out(S) \\ \cap in(S)}} f(e) + \sum_{\substack{e \in out(S) \\ \cap in(T)}} f(e) - \sum_{\substack{e \in in(S) \\ \cap out(S)}} f(e) - \sum_{\substack{e \in out(T) \\ \cap in(S)}} f(e)$

④ ① ③ ②

$= \sum_{\substack{e \in out(S) \\ \cap in(T)}} f(e) - \sum_{\substack{e \in in(S) \\ \cap out(T)}} f(e)$

COROLLARY 3: $\forall$ flow f, $\forall$ cut (S,T) : $V(f) \leq C(S,T)$

PROOF $V(f) = \sum_{\substack{e \in out(S) \\ \cap in(T)}} f(e) - \sum_{\substack{e \in out(T) \\ \cap in(S)}} f(e) \leq C(S,T) - \sum_{\substack{e \in out(T) \\ \cap in(S)}} f(e) \leq C(S,T)$

$\leq \sum_{\substack{e \in out(S) \\ \cap in(T)}} c(e)$ $x \geq 0$

$\leq C(S,T)$

COROLLARY 4: ∀ flow f, ∀ cut (S,T) if V(f)=C(S,T), then f is max flow, (S,T) is min cut.

PROOF Let f be flow returned by FF algorithm.

G_f = residual graph with respect to f

S = set of nodes reachable from s in G_f

T = rest of nodes of G_f

$s \in S$, $t \notin S$ by termination condition of FF $\Rightarrow$ (S,T) is a cut

(no s→t path $\Rightarrow$ t not reachable)

WTS 1. $\forall (u,v) \in E$ s.t. $u \in S$ and $v \in T$, $f(u,v) = c(u,v)$ (all flow from S to T at capacity)

2. $\forall (u',v') \in E$ s.t. $u' \in T$ and $v' \in S$, $f(u',v') = 0$ (no flow from T to S)

1. Suppose for ~~cond~~ contradiction $f(u,v) < c(u,v)$

$\Rightarrow$ G_f has forward edge (u,v) (can only be the case $u \in S, v \in S$ or $u \in T, v \in T$)

contradicts assumption of $u \in S, v \in T$

2. Suppose for contradiction $f(u',v') > 0$

$\Rightarrow$ G_f has backward edge (v',u') (the flow could have been shifted)

$v' \in S \Rightarrow u' \in S$ contradicts $u' \in T$ (this would contradict no s→t path)

■

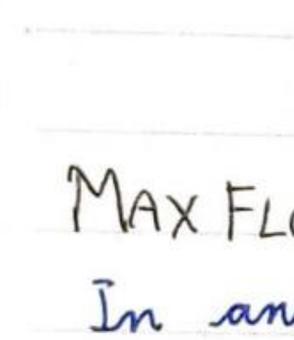
MAX FLOW MIN CUT THEOREM (↑ from above)

In any flow network ∃ the value of a max flow = capacity of a min cut.

INTEGRALITY THEOREM (FF finds one)

If capacities are integers then there is some max flow where all edges have integer flows. $f(e) \in \mathbb{Z}_0^+$, $\forall e$

Define Bipartite Graph: undirected graph G=(V, E) where V can be partitioned into X, Y s.t. every ~~node~~ edge in E connects an node in X to a node in Y. i.e. G=((X,Y), E)



Define Matching in G=((X,Y),E) A set of edges $M \subseteq E$ s.t. no two edges in M are incident on the same node

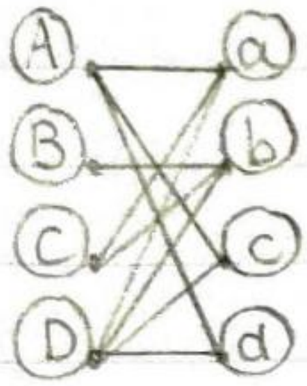
{{A,a}, {B,b}} ✓ {{A,a}, {A,c}} ✗

Define Max Matching: matching of max cardinality

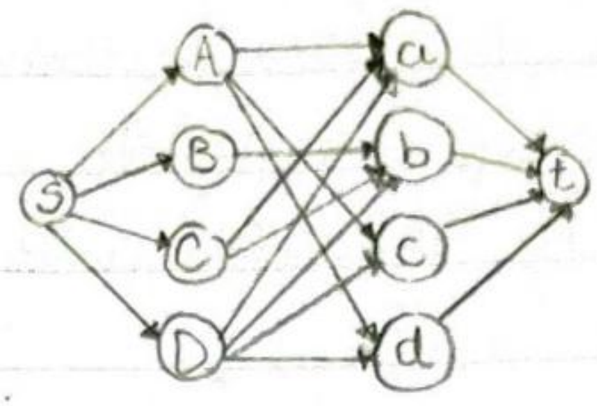
BIPARTITE MATCHING

input: Bipartite graph $G=((X,Y),E)$

output: A max matching M of G

G: 

$\mathcal{F}=(G',s,t,c)$ all edge capacities = 1



algorithm: BipartiteMatch(G) // $n=|V|$, $m=|E|$

construct flow network $\mathcal{F}$ as indicated (add s, s have edges point to X, add t, t have edges point from Y, change original edges to X→Y, each with capacity=1) ↑ O(m+n)

$f := FF(\mathcal{F})$ // $O(mC)=O(mn)$

$M := \{\{x,y\} : x\in X, y\in Y \text{ and } f(x,y)=1\}$ // $O(m)$

return M

running time: $O(mn)$

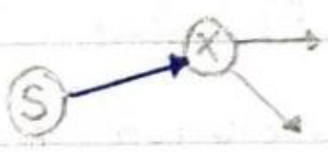
CLAIM 1: Matching of size k in G $\rightleftarrows$ Integral flow of value k in $\mathcal{F}$

PROOF "⇒" Given M, define f_M

$$f_M(e)=\begin{cases}1 & \text{if } e \text{ is on path } s\to x\to y\to t \text{ s.t. } \{x,y\}\in M\\ 0 & \text{o/w}\end{cases}$$

Verify f_M is a flow ← 1. capacity, 2. conservation

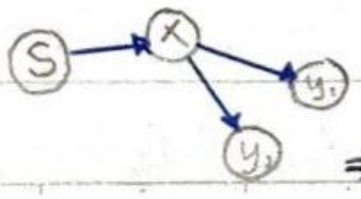
1. since all edges have capacity 1 and $\max(f_M(e))=1$, capacity constraint holds
2. Three cases that violates

① 

no flow out: no point in doing this if there is no matching

② 

no in flow: no s→x path, contradicts $f_M(e)$

③ 

more than 1 out: not a match so not possible

⇒ conservation constraints hold.

$$V(f_M) = \sum_{\substack{e \in out(X) \\ \cap in(Y)}} f(e) - \sum_{\substack{e \in out(Y) \\ \cap in(X)}} f(e) \quad \text{(lemma 2 last class)}$$

$= 0$ because no edges from Y to X

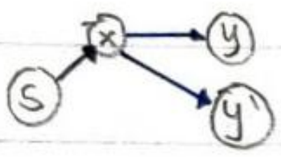
$= |M|$

"⇐" Given integral flow f in $\mathcal{F}$. define M_f

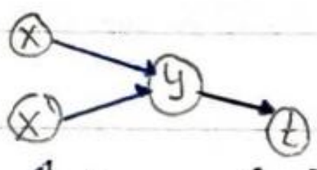
$M_f = \{\{x,y\} : x \in X, y \in Y \text{ and } f(x,y) = 1\}$

Verify M_f is a matching

Two cases M_f is not matching

① 

there is flow from x to y and x to y'. Not possible because $s \to x$ only has flow of 1

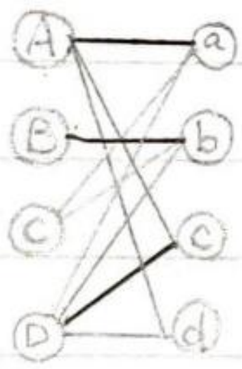
② 

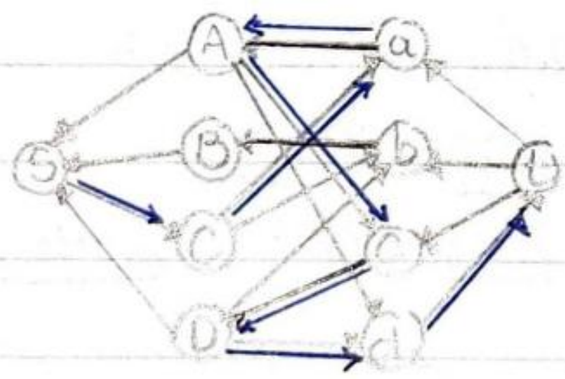
there is flow ~~to~~ from x to y and x' to y. Not possible because $V = 1$ flow allowed from $y \to t$

$$V(f) = \sum_{\substack{e \in out(X) \\ \cap in(Y)}} f(e) - \sum_{\substack{e \in out(Y) \\ \cap in(X)}} f(e) = |M_f|$$

$= 0$ because no edges from Y to X ∎

How Is Flow Improved?





residual graph

forward forward

$s \to$ forward $\to$ back $\to f \to b \to f \to t$

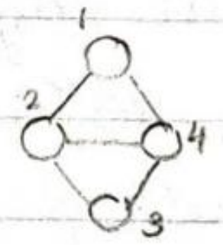
start and end are forward, alternating

$\Rightarrow$ new matching

Min Vertex Cover In Bipartite Graphs

Define vertex cover of $G: V' \subseteq V$ $G=(V,E)$ undirected

s.t. $\forall$ edges $\{u,v\} \in E$ $u \in V'$ or $v \in V'$

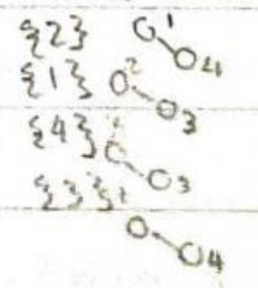
e.g. 

{1,2,3} vertex cover

{1,3} not ($2-4$ not covered)

{2,4} min vertex cover

any one node can't



Vertex Cover Problem

input: Undirected graph $G = (V, E)$

output: Min cardinality vertex cover of G

NP-complete for general graphs.

but polynomial time for <u>bipartite graphs</u>

No.

Date

CLAIM 1: ∀ graph G, ∀ matching M of G, and any vertex cover R of G, $|M| \leq |R|$

PROOF By pigeon hole principle

(Each match is a pigeon and the vertex covers are holes. There are not enough pigeons to fill in the holes. or each pigeon fills up the hole.) ∎

CLAIM 2: If graph G has a matching M and a vertex cover ~~VC~~ R s.t. $|M| = |R|$ then

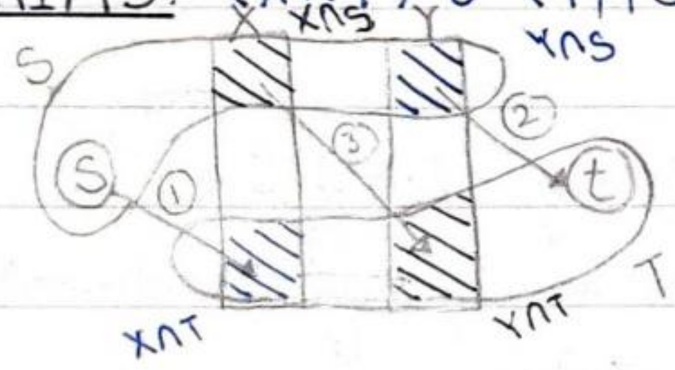
a. M is max matching, and

b. R is a min vertex cover

PROOF From claim 1 $|M| \leq |R|$ ∎

~~CLAIM 3:~~ THM If $G = ((X,Y), E)$ is bipartite then size of max M in G = size of min R in G (not true for arbitrary graphs)

CLAIM 3: $(X \cap T) \cup (Y \cap S)$ is a vertex cover.



PROOF
1. $X \cap S \rightarrow \underline{Y \cap S}$
2. $X \cap S \rightarrow Y \cap T$ DNE such edge
3. $\underline{X \cap T} \rightarrow \underline{Y \cap S}$
4. $\underline{X \cap T} \rightarrow Y \cap T$

underline => cover

$G \rightarrow \mathcal{F}$ f = max flow in $\mathcal{F}$

(S, T) corresponding min cut

Why $X \cap S \rightarrow Y \cap T$ edge PNE? Suppose for contradiction ~~there exists an~~ such edge (x, y)

$x \in X \cap S$, $y \in Y \cap T$

$\Rightarrow \exists$ path p $s \rightarrow x$ in G'_f ← G' is the directed graph obtained from bipartite graph G

and $f(x, y) = 1$ (if not, there will be forward edge from x to y in G'_f, which means $y \in Y \cap S$ instead)

CASE 1: $p = s \rightarrow x \Rightarrow (s, x) \in G'_f$ (forward edge) $\Rightarrow f(s, x) = 0$

contradicts $f(x, y) = 1$ (no flow $s \rightarrow x$ but flow $x \rightarrow y$)

CASE 2: $p = s \xrightarrow{FW} x_1 \xrightarrow{FW} y_1 \xrightarrow{BW} x_2 \xrightarrow{FW} y_2 \rightarrow \cdots \rightarrow x_k \xrightarrow{BW} y_k \xrightarrow{FW} x \xrightarrow{BW}$

FW $\Rightarrow$ flow = 0, BW $\Rightarrow$ flow = 1 $\Rightarrow$ ~~$f(y_k, x)$~~ $f(x, y_k) = 1$ but $y_k \in S$, $y \notin S \Rightarrow y_k \neq y$

so X has flow to at least y_R and y, but there is only one flow to x
⇒ contradiction that F is a flow
∴ $(X \cap T) \cup (Y \cap S)$ is a vertex cover.
as there ∄ an edge $x \to y$ where $x \in X \cap S$ and $y \in Y \cap T$

CLAIM 4: $|(X \cap T) \cup (Y \cap S)| = |\text{matching}|$

PROOF $|(X \cap T) \cup (Y \cap S)| = \#\text{type ① edges} + \#\text{type ② edges}$
$= C(S,T) \leftarrow \sum_{e \in \text{out}(S) \cap \text{in}(T)} c(e)$ and since no type 3
$= V(f)$ by max flow min cut
$=$ size of max matching in G

Algorithm: ① Construct $\mathcal{F}$ from G // $O(m+n)$
② Find max flow in $\mathcal{F}$ // $O(mn)$
③ Find min cut(S,T) in $\mathcal{F}$ // $O(m+n)$
④ Return $(X \cap T) \cup (Y \cap S)$ // $O(m)$

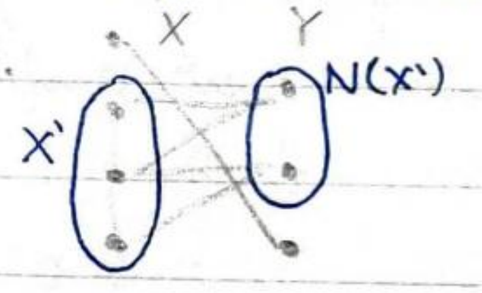
running time: $O(mn)$

HALL'S THM

happenes when $|X| = |Y|$ matching size $|X| = |Y|$

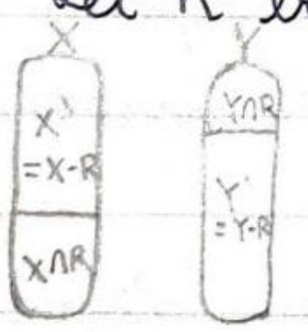
Bipartite graph G has no perfect matching $\Leftrightarrow \exists X' \subseteq X$ s.t. $|X'| > |N(X')|$.
where $N(X') = \{y \in Y \mid \exists x \in X' \text{ s.t. } \{x,y\} \in E\}$ (neighbour)

e.g.



PROOF "⇐" # X > # neighbours of X so cannot match all $x \in X'$ to a y in $N(X')$

"⇒" Assume $G = ((X,Y),E)$ has no perfect matching
Let R be a min vertex cover of G



$N(X') \subseteq Y \cap R$ (otherwise R is not a vertex cover)
$|N(X')| \leq |Y \cap R|$
$= |R| - |X \cap R| = |R| - (|X| - |X'|)$
$= |R| - |X| + |X'|$
$= |\text{max matching in } G| - |X| + |X'|$
< 0 because no perfect matching
$< |X'|$

FINDING EDGE-DISJOINT PATHS IN GRAPHS

input: A digraph $G=(V,E)$ s.t. $s,t \in V$ $s \neq t$

output: A set of edge-disjoint paths (no edge used in two paths) of max cardinality

Given G, s, t. construct a flow network $\mathcal{J}=(G', s, t, c)$

where G' = same nodes as G except delete edges into s and out of t

~~all~~ $c(e)=1 \quad \forall e \in E'$

algorithm: MaxPaths (G,s,t)

① Construct flow network $\mathcal{J}$ as indicated // $O(m+n)$

② $f := FF(\mathcal{J})$ // $O(mn)$

③ ~~d~~ Decompose f into a set ~~of~~ P of $V(f)$ $s \to t$ edge-disjoint simple paths i.e. $p := PathDecomp(f)$ // $O(mn)$

④ Return p

recurse up to n times; each recursive call $O(n)$

CLAIM: Set of k edge-disjoint $s \to t$ simple path in $G \iff$ Integral flow of value k in $\mathcal{J}$

PROOF "$\Rightarrow$" Given set P of edge-disjoint $s \to t$ simple paths, construct f_P:

$$f_P(e) = \begin{cases} 1 & \text{if } e \text{ is on a path in } P \\ 0 & \text{otherwise} \end{cases}$$

Verify f_P is a flow — 1. capacity

— 2. conservation

1. Since $f_P(e)$ is either 1 or 0 and $c(e)=1$, flow $\leq$ capacity $\forall e$.
2. Since each path uses different edges, therefore amount in V = amount out V $\forall V \in V$, $V \neq s$ and $V \neq t$

Since there are k paths, $\sum_{e \in out(s)} f(e) = k \Rightarrow \mathcal{J}$ has integral flow of val k.

"$\Leftarrow$" Given integral flow f of value k in $\mathcal{J}$. $\exists$ set P_f of k edge disjoint $s \to t$ simple paths where $f(e)=1$ $\forall e \in$ a path $\in P_f$

1. $s \to u_1 \to u_2 \to \cdots \to t$ is an edge disjoint $s \to t$ simple path
2. $s \to u_1 \to \cdots \to u_i \to \cdots u_{\ell-1}$ can remove loop and relocate path

Prove by complete induction on # edges used by f

Assume ⇐ holds for integral flows using $< m$ edges [IH]

Consider integral flow f of value k that uses m edges.

WTS ∃ set $\hat{P}_f$ of k edge-disjoint $s \to t$ simple paths where every edge e on a path in P_f is used by f.

CASE 1. $V(f) = 0 \Rightarrow P_f = \phi$

CASE 2: $V(f) > 0 \Rightarrow \exists\, e \in out(s)$ s.t. $f(e) = 1$

Let $u_0, u_1, \cdots, u_\ell$ be a path p in G s.t. $u_0 = s$, $(u_i, u_{i+1}) \in E$, $f(u_i, u_{i+1}) = 1$ $\forall i$ $0 \le i < \ell$ and either $u_\ell = t$ or $u_\ell = u_i$, $1 \le i < \ell$

a. $u_\ell = t$: Define $f'(e) = \begin{cases} 0 & \text{if } e \text{ on path } p \text{ + being used up} \\ f(e) & \text{o/w} \end{cases}$

verify f' is a flow = 1. capacity 2. conservation

$V(f') = V(f) - 1 = k - 1$

By IH, ∃ set $P_{f'}$ of $k-1$ edge-disjoint $s \to t$ simple paths with edges used by f' $\Rightarrow P_f = P_{f'} \cup \{\text{path } p\}$ remove flow from cycle

b. $u_\ell = u_i$ $i < \ell$ Define $f'(e) = \begin{cases} 0 & \text{if } e = (u_j, u_{j+1}) \text{ “} i \le j < \ell \\ f(e) & \text{o/w} \end{cases}$

verify f' is a flow − 1. capacity − 2. conservation

Note: $V(f') = V(f)$ ← no change in $\sum_{e \in out(s)} f(e)$

By IH, $P_{f'}$ of k edge-disjoint $s \to t$ simple paths with edges used by f' $P_f = P_{f'}$

algorithm of ③: PathDecomp (f)

```
if V(f) = 0 then return ∅
else
    u := s ; p := s
    mark all nodes as unvisited
    while u is unvisited and u ≠ t do
        mark u as visited
        v := any node s.t. f(u,v) = 1
        u := v ; p := p · v        // p concatenate v
```

```
if u=t then
    for each edge e on p do f(e) := 0
    return PathDecomp(f) ∪ {p}
else
    for each edge e on suffix of P from u to u do f(e) := 0
    return PathDecomp(f)
```

LINEAR PROGRAMMING

GENERAL FORM OF LP

1. variables $x_1, x_2, \cdots, x_n \in \mathbb{R}$ (not discrete)
2. objective functions min/max $\sum_{j=1}^{n} c_j x_j$
3. constraints $\sum_{j=1}^{n} a_{ij} x_j \begin{bmatrix} \leq \\ \geq \\ = \end{bmatrix} b_i$ for $1 \leq i \leq m$

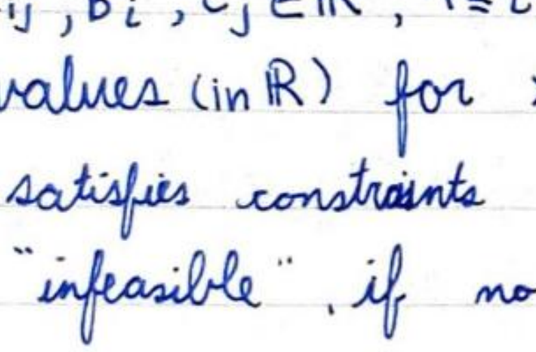
input: $a_{ij}, b_i, c_j \in \mathbb{R}$, $1 \leq i \leq m$, $1 \leq j \leq n$

output: values (in $\mathbb{R}$) for $x_1, \cdots, x_n$ where $\sum_{j=1}^{n} c_j x_j$ is max/min and satisfies constraints $\sum_{j=1}^{n} a_{ij} x_j \begin{matrix} \leq \\ \geq \end{matrix} b_i$ if such values exists

"infeasible", if no values for $x_1, \cdots, x_n$ that satisfy all constraints (over constraint)

"unbounded", if values satisfying constraint exist, but min/max does not (infinite solutions) (under constraint)

Since constraints are linear, they split $\mathbb{R}^n$ into half planes.

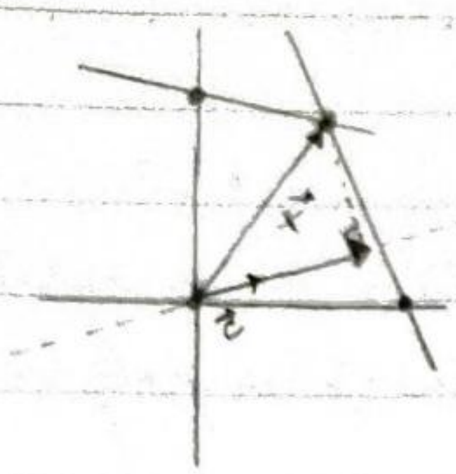


The feasible region is convex (not 5)

optimal solutions exists in corners of the feasible region

FACT 1: feasible region is a convex polygon.

FACT 2: if an optimal solution exists then an optimal solution that is a vertex of the feasible region exists



$c_1x_1 + \cdots + c_nx_n = \vec{c} \cdot \vec{x}$ where $\vec{c} = (c_1, \cdots, c_n)$

$\vec{x} = (x_1, \cdots, x_n)$

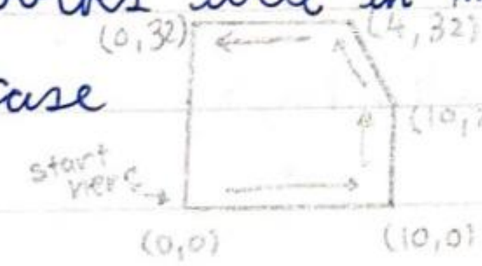
$\frac{\vec{c} \cdot \vec{x}}{\|\vec{c}\|} \vec{c} = \text{proj}_{\vec{c}} \vec{x}$

$\frac{\vec{c} \cdot \vec{x}}{\|\vec{c}\|}$ ← if max, we want to max this

Definitions:

objective function: function to be optimized
solution: assignment of values in $\mathbb{R}$ to variables
feasible solution: solution satisfying all constraints
feasible region: set of feasible solutions
optimal solution: feasible solution optimizing objective function
value of solution: value of objective function at solution
optimal value of LP: value of optimal solution
feasible LP: a feasible solution exists
infeasible LP: no feasible solution exists
bounded LP: an optimal solution exists
unbounded LP: feasible LP with no optimal solution

algorithms: - Simplex algorithm (Dantzig 1947) Kantorovitch 1939 had another algo
works well in most practical settings but is exponential in worst case



$3x_1 + 7x_2$

(0,0) → (10,0) → (10,20) → (4,32) → (0,32)

value: 0 30 70 76 64

↑ local optimum, which is also global optimum because region convex

- Ellipsoid method (Khachyian 1979)
polynomial but slow in practice
- Interior point method (Karmarker 1984)
polynomial and good in practice
↳ # of vars: n size of numbers: a_{ij}, b_i, c_j
# of constraints: m

No.

Date

REDUCING PROBLEMS TO LP PROBLEMS

given instance I of problem (e.g. max flow)

- construct an LP problem P_I whose optimal solution is a solution to I
- size of P_I (# variables, constraints, size of coefficients) is polynomical in size of I

MAX FLOW AS LP PROBLEM

variables: $x_e, e \in E$, where x_e = flow on e

objective function: $\max \sum_{e \in out(s)} x_e \Rightarrow$

constraints: $x_e \geq 0$, $x \leq c(e) \quad \forall e \in E$

$\forall$ node $v \neq s,t \quad \sum_{e \in out(v)} x_e - \sum_{e \in in(v)} x_e = 0$

TRANSPORTATION PROBLEM

k plants $P_1, \ldots, P_k$ P_i makes s_i amount of product

l outlets $O_1, \ldots, O_l$ O_j demands d_j amount of product

c_{ij} = cost of shipping a unit of product from P_i to O_j

variables: x_{ij} = amount of product shipped from P_i to O_j $\quad \forall 1 \leq i \leq k, 1 \leq j \leq l$

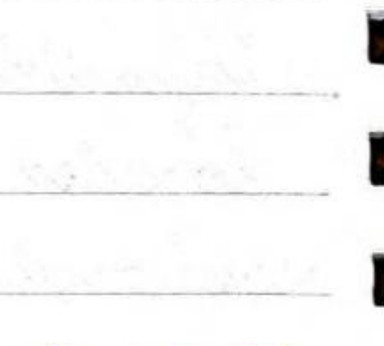
objective function: $\min \sum_{\substack{1 \leq i \leq k \\ 1 \leq j \leq l}} c_{ij} x_{ij}$

constraints $\sum_{j=1}^{l} x_{ij} \leq s_i \quad \forall 1 \leq i \leq k \qquad x_{ij} \geq 0 \ \forall 1 \leq i \leq k, 1 \leq j \leq l$

$\sum_{i=1}^{k} x_{ij} \geq d_j \quad \forall 1 \leq j \leq l$

LINE FITTING (LINEAR REGRESSION)

Determine how "attributes" affect "outcome" in a "population"



Definitions:	Stats	Machine Learning	
attributes :	independent var	features	} real valued numbers
outcomes :	dependent var	labels	}

input: m points $P_1, \cdots, P_m \in \mathbb{R}^d$, where d = # attributes
m labels $l_1, \cdots, l_m \in \mathbb{R}$ → $P_i = (p_{i1}, p_{i2}, \cdots p_{id})$ = values for each attribute

output: A linear function h of attributes that "best fits" data

i.e. $h: \mathbb{R}^d \to \mathbb{R}$ s.t. $h(P_i)$ "is close to" l_i $\forall i$, $1 \le i \le m$

$h(x_1, \cdots, x_d) = a_1 x_1 + \cdots + a_d x_d + b$

where a_k and b are the coefficients we are looking for

Define the Error of function with respect to point P_i:

Given linear function $h(\vec{a}, b)$ ($a_1, a_2, \cdots, a_d$) $E_i(\vec{a}, b) = |$vertical distance of $h(P_i)$ from $l_i|$

$= \left| \underbrace{b + \sum_{j=1}^{d} a_j p_{ij}}_{h(P_i)} - l_i \right|$

$\Rightarrow E(\vec{a}, b) = \sum_{i=1}^{m} E_i(\vec{a}, b)$ is error of function

FORMING AN LP

variables: $a_1, \cdots, a_d, b$ → $a_1, \cdots, a_d, b, y_1, \cdots, y_m$

objective function: $\min \sum_{i=1}^{m} |(b + \sum_{j=1}^{d} a_j p_{ij}) - l_i|$ → $\min \sum_{i=1}^{m} y_i$

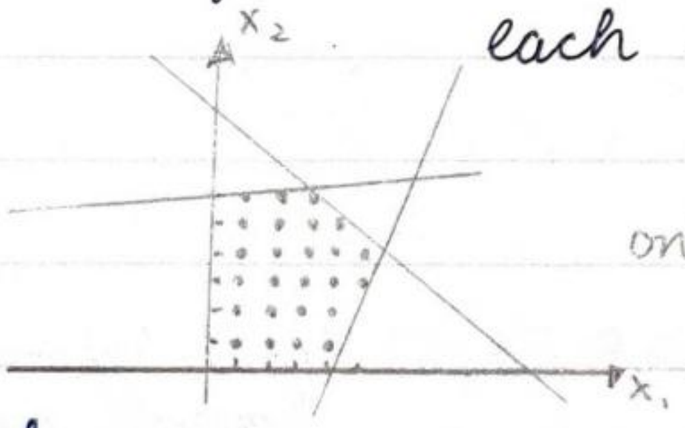
$= \max (b + \sum_{j=1}^{d} a_j p_{ij} - l_i)$

constraints: none → $y_i \ge (b + \sum_{j=1}^{d} a_j p_{ij}) - l_i$ $\forall i$, $1 \le i \le m$

$y_i \ge -(b + \sum_{j=1}^{d} a_j p_{ij}) - l_i$

INTEGER_LP_PROBLEM

Define IntInteger LP: a LP problem with additional constraint $x \in \mathbb{Z}$ for each variable x



only • are feasible (unlike the whole region for normal LP)

NP-hard.

0/1 LP (BOOLEAN_LP)

Define 0/1 LP: LP problem with additional constraint $x \in \{0, 1\}$ for each variable x (non linear)

NP-hard.

No.

Date

FACILITY LOCATION

n possible "facility" locations $i = 1, \cdots, n$

m "clients" $j = 1, \cdots, m$

f_i cost to "open" facility at location i

c_{ji} cost to assign client j to facility at location i

Objective: choose subset of locations, assign clients to one location s.t minimize cost opening facility + cost client-facility assignment

variables: $x_i \quad 1 \le i \le n$ $\quad x_i = \begin{cases} 1, \text{ if facility } i \text{ is chosen} \\ 0, \text{ otherwise} \end{cases}$

$y_{ji} \quad 1 \le i \le n$

$1 \le j \le m$ $\quad y_{ji} = \begin{cases} 1, \text{ if client } j \text{ is assigned to facility } i \\ 0, \text{ otherwise} \end{cases}$

objective function: $\min \sum_{i=1}^{n} f_i x_i + \sum_{j=1}^{m} \sum_{i=1}^{n} c_{ji} y_{ji}$ (facility cost + service cost)

constraints $\sum_{i=1}^{n} y_{ji} = 1 \quad \forall\, 1 \le j \le m$ (client assign to one location)

$y_{ji} \le x_i \quad \forall\, 1 \le j \le m, \forall\, 1 \le i \le n$ (if client assigned, facility location must be chosen)

non-linear $\begin{cases} x_i \in \{0,1\} & \forall\, 1 \le i \le n \\ y_{ji} \in \{0,1\} & \forall\, 1 \le j \le m, \forall\, 1 \le i \le n \end{cases}$

0/1 FORMULATION OF KNAPSACK

variables: $x_i \quad i = 1, \cdots, n$ $\quad x_i = \begin{cases} 1 & \text{if item } i \text{ is chosen} \\ 0 & \text{otherwise} \end{cases}$

objective function: $\max \sum_{i=1}^{n} v_i x_i$

constraints $\sum_{i=1}^{n} w_i x_i \le C$

$x_i \in \{0,1\} \quad 1 \le i \le n \xrightarrow{\text{replace by}} x_i \ge 0, x_i \le 1$ (which becomes fractional knapsack problem)

0/1 FORMULATION OF VERTEX COVER

Variables: $x_v \quad v \in V$ $\quad x_v = \begin{cases} 1 & \text{if } v \text{ is chosen to be in vertex cover} \\ 0 & \text{otherwise} \end{cases}$

objective function: $\min \sum_{v \in V} x_v$

constraints $\forall \{u,v\} \in E \quad x_u + x_v \ge 1$ (either node chosen for all edges)

APPROXIMATION ALGORITHMS

Runtime is polynomial

Answer is always within constant factor $c>1$ of optimal solution
↑ approximation ratio

MIN objective function ⇒ output of algo $\leq c \cdot$ value of optimal solution

MAX objective function ⇒ output of algo $\geq \frac{1}{c} \cdot$ value of optimal solution

VERTEX COVER LP RELAXATION

$\vec{x} = (x_v : v \in V)$ is feasible solution to 0/1 LP formulation $\Leftrightarrow \{v : x_v = 1\}$ is a V.C. of G

↳ optimal solution to 0/1 LP gives min vertex cover of G

variables: x_v $\forall v \in V$

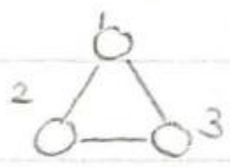
~~$x_v = \begin{cases} 1 & \text{if } v \text{ is chosen to be in vertex cover} \\ 0 & \text{otherwise} \end{cases}$~~

objective function: $\min \sum_{v \in V} x_v$

constraints $\forall \{u,v\} \in E \quad x_u + x_v \geq 1$

$x_v \geq 0 \quad , x_v \leq 1 \quad \forall v \in V$

running time: polynomial using LP solver

e.g. 

$\min \; x_1 + x_2 + x_3$ s.t. $x_1 + x_2 \geq 1$ $\quad x_1 \geq 0, \; x_1 \leq 1$

$x_2 + x_3 \geq 1$ $\quad x_2 \geq 0, \; x_2 \leq 1$

optimal solution: $x_1^* = x_2^* = x_3^* = \frac{1}{2}$ $\quad x_1 + x_3 \geq 1$ $\quad x_3 \geq 0, \; x_3 \leq 1$

Round (possibly fractional) optimal solution

$$\hat{x}_v = \begin{cases} 1 & \text{if } x_v^* \geq \frac{1}{2} \\ 0 & \text{otherwise} \end{cases} \Rightarrow \text{a solution to 0/1 LP problem}$$

CLAIM 1: ~~{~~$(\hat{x}_v : v \in V)$ is a feasible solution to original 0/1 LP problem

PROOF if $\{u,v\} \in E \Rightarrow$ Relaxation problem has $x_u + x_v \geq 1$

$\Rightarrow x_u^* + x_v^* \geq 1$

$\Rightarrow x_u^* \geq \frac{1}{2}$ or $x_v^* \geq \frac{1}{2}$ (or both) ∴ otherwise constraint not meet

$\Rightarrow \hat{x}_u = 1$ or $\hat{x}_v = 1$

$\Rightarrow \hat{x}_u + \hat{x}_v \geq 1$ ∎

No.

Date

CLAIM 2: The value is close to the optimal solution with an approximation ratio of 2.

PROOF Let $\bar{x}_v$, $v \in V$ be optimal solutions to the 0/1 LP problem

a. $\sum_{v \in V} x_v^* \leq \sum_{v \in V} \bar{x}_v$

since the LP that gives x_v^* has less constraint (more relaxed), the solution is more optimal (smaller value for objective function)

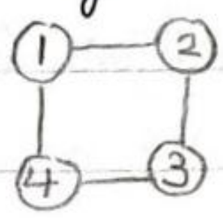
b. $\hat{x}_v \leq 2x_v^*$ by definition of rounding x^*_v to obtain $\hat{x}_v$

$\sum_{v \in V} \hat{x}_v \leq \sum_{v \in V} 2x^*_v$ from b

$= 2 \sum_{v \in V} x^*_v$

$\leq 2 \sum_{v \in V} \bar{x}_v$ from a

$=$ 2 · size of optimal vertex cover, as wanted. ■



optimal solution to relaxed:
$x_1 = x_2 = x_3 = x_4 = \frac{1}{2}$
$\Rightarrow \hat{V} = \{1, 2, 3, 4\}$
$\bar{V} = \{1, 3\}$

algorithm: Approx Min Vertex Cover (G) $n = |V|$ $m = |E|$

① write 0/1 LP problem P for vertex cover of G } // $O(m+n)$
② construct LP relaxation P` of P
③ solve P` to find optimal solution x^* // polynomial (using solver)
④ round x^* to obtain $\hat{x}$ // $O(n)$
⑤ $\hat{V} := \{v \mid \hat{x}_v = 1\}$; return $\hat{V}$ // $O(n)$

running time: polynomial

Unless P=NP, there is no poly time algorithm for vertex cover with approx ratio < 1.361 ← theoretically but no algo does this

VERTEX COVER - GREEDY

finds a maximal matching M of $G = (V, E)$ ($n = |V|$, $m = |E|$) maximal: matching cannot be extended / superset of output is not a matching

algorithm: find maximal matching greedily // $O(m+n)$ ← go through edges, if neither nodes used, add edge to M
U := set of endpoints of edges in M // $O(m)$
return U

running time: $O(m+n)$

No.

Date

CLAIM 1: U is a vertex cover

PROOF Suppose U is not a vertex cover, i.e. $\exists \{u,v\} \in E$ not covered by U

$\Rightarrow M \cup \{u,v\}$ is a matching, contradicting M is maximal.

$\therefore U$ is a vertex cover. ■

CLAIM 2: Let $\bar{U}$ be the optimal vertex cover. Then $|U| \leq 2|\bar{U}|$

PROOF size of every matching $\leq$ size of every vertex cover

$\Rightarrow |M| \leq |\bar{U}|$

$|U| = 2 \cdot |M| \leq 2 \cdot |\bar{U}|$

M has edges with distinct nodes.
U has those distinct nodes

■

WEIGHTED VERTEX COVER

Now node v has weight w_v

Conditions are same as 0/1 vertex cover LP, with objective function

$$\min \sum_{v \in V} w_v x_v$$

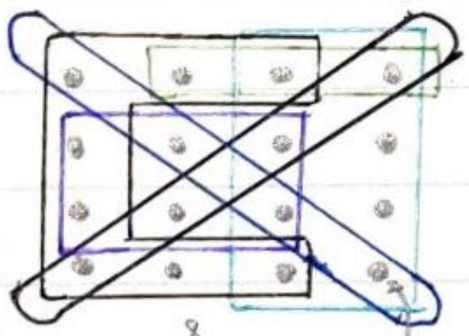
SET COVER

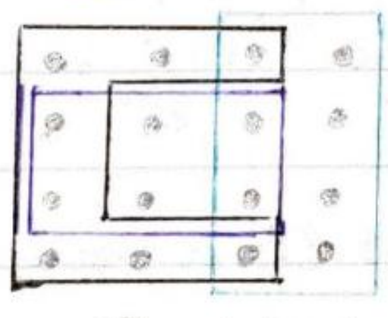
input: $U = \{a_1, a_2, \cdots, a_n\}$ (universe), $\mathcal{S} = \{A_1, A_2, \cdots, A_m\}$ $A_j \subseteq U$

output: minimum size set cover $\mathcal{C}$ of $\mathcal{S}$

i.e. $\mathcal{C} \subseteq \mathcal{S}$ s.t. $\bigcup_{A \in \mathcal{C}} A = U$

e.g.





algorithm: $C := \emptyset$; $R := U$ $n = |U|$ ~~$k = |\mathcal{C}|$ $t = |\mathcal{C}^*|$~~ $m = |\mathcal{S}|$

```
while R ≠ ∅:
    let A ∈ S - C be s.t. |A ∩ R| is max   // part of U that has not been covered
    C = C ∪ {A} ; R := R - A
return C
```

running time: $O(mn^2)$

Let $n = |U|$, $k = |\mathcal{C}|$, $\ell = |\mathcal{C}^*|$, where $\mathcal{C}^*$ = min set cover for U & $\mathcal{S}$

FACT: $k \leq \overbrace{\ln n}^{\text{approximation ratio}} \cdot \ell$

R_0 ⟵ n_0 ⟶ $n_0 = n$ → R_1 ⟵ n_1 ⟶ ⊣ - - - ⊣ $\frac{n_0}{\ell}$ (greedy ensures at least this many is covered)

$n_1 \leq n_0 - \frac{n_0}{\ell} = n(1-\frac{1}{\ell})$

R_2 ⟵ n_2 ⟶ ⊣ $\frac{n_1}{\ell}$ ⊣ - - ⊣ $\frac{n_0}{\ell}$ - - ⊣

$n_2 \leq n_1(1-\frac{1}{\ell}) = n(1-\frac{1}{\ell})^2$

⋮

at iteration i:

$\mathcal{C}_{i-1}$ ⟶ $\}\cup\{A\}$ ⟶ $\mathcal{C}_i = \mathcal{C}_{i-1} \cup \{A\}$

R_{i-1} ⟶ $\}-A$ ⟶ $R_i = R_{i-1} - A$

$n_{i-1} = |R_{i-1}|$ ⟶ ⟶

$n_i = |R_i|$ ⟵ ⟶ ⊣ - - ⊣ $\geq \frac{n_{i-1}}{\ell}$ elements in R_{i-1} now covered by A

CLAIM: $\exists A^* \in \mathcal{C}^*$ s.t. a. $A^* \notin \mathcal{C}_{i-1}$ and

b. A^* covers $\geq \frac{n_{i-1}}{\ell}$ elements in R_{i-1}

PROOF There are $\leq \ell$ sets in $\mathcal{C}^*$ not already in $\mathcal{C}_{i-1}$

If all ℓ sets have $< \frac{n_{i-1}}{\ell}$ elements

then the union of those sets cover $< \ell \frac{n_{i-1}}{\ell}$ elements $= n_{i-1}$ elements

$\Rightarrow \mathcal{C}^*$ is not a set cover (contradiction) $\Rightarrow A^*$ exists ∎

PROOF FACT $n_i \leq n_{i-1}(1-\frac{1}{\ell}) \leq n_{i-2}(1-\frac{1}{\ell})^2 \leq \cdots \leq n(1-\frac{1}{\ell})^i$

$\Rightarrow n_i \leq n(1-\frac{1}{\ell})^i < ne^{-\frac{i}{\ell}}$ $\quad (\forall x \in \mathbb{R},\ 1-x \leq e^{-x},\ x \neq 0 \Rightarrow 1-x < e^{-x})$

let $i = \ell \cdot \ln n$

$ne^{-\frac{i}{\ell}} = ne^{-\frac{\ell \cdot \ln n}{\ell}} = ne^{-\ln n} = \frac{n}{e^{\ln n}} = 1 \quad \Rightarrow n_i < 1$

$\Rightarrow$ algorithm terminates after $\leq \ell \cdot \ln n$ iterations

$\Rightarrow k = |\mathcal{C}| \leq \ln n \cdot \ell$ ∎

Unless P=NP, every polytime approximation algo for set cover has approx ratio $\Omega(\log n)$

MIN MAKESPAN

input: "machines" $1, 2, \cdots, m$ $\quad m \geq 1$ $\quad$ "jobs" $1, 2, \cdots, n$ $\quad n \geq 1$

t_j = length of job j

output: Assignment A with min makespan

Define assignment $A(i) \subseteq \{1, 2, \cdots, n\}$ jobs assigned to machine i

$\forall j \ \exists i$ s.t $j \in A(i)$ and $A(i) \cap A(i') = \emptyset \quad i \neq i'$

Define load of machine i (under A) $= \sum_{j \in A(i)} t_j = l_i$

Define makespan of $A = \max_{1 \leq i \leq m} l_i$

NP hard.

algorithm (greedy1): Consider the jobs in arbitrary order

assign job to currently least loaded machine

useful for online (on demand) jobs come in arbitrary order

Let L^g = makespan of assignment produced by greedy algorithm

L^* = optimal makespan

THM 1: $L^g \leq L^* (2 - \frac{1}{m})$ approx ratio

PROOF Let i be the most loaded machine in assignment of greedy algorithm

$\Rightarrow L^g = l_i$

Let l be the last job ~~assigned~~ assigned to i

$L_g = [\text{min load of any machine before } l \text{ added to } i] + t_l \leq \frac{1}{m}(\sum_{j=1}^{n} t_j - t_l) + t_l$

$\leq \frac{1}{m}((\sum_{j=1}^{n} t_j) - t_l)$ ← evenly split time to machine excluding t_l

$\leq m \cdot L^*$ ↑ above is absolute best case (cutting up jobs)

$\leq \frac{1}{m}(mL^* - t_l) + t_l$

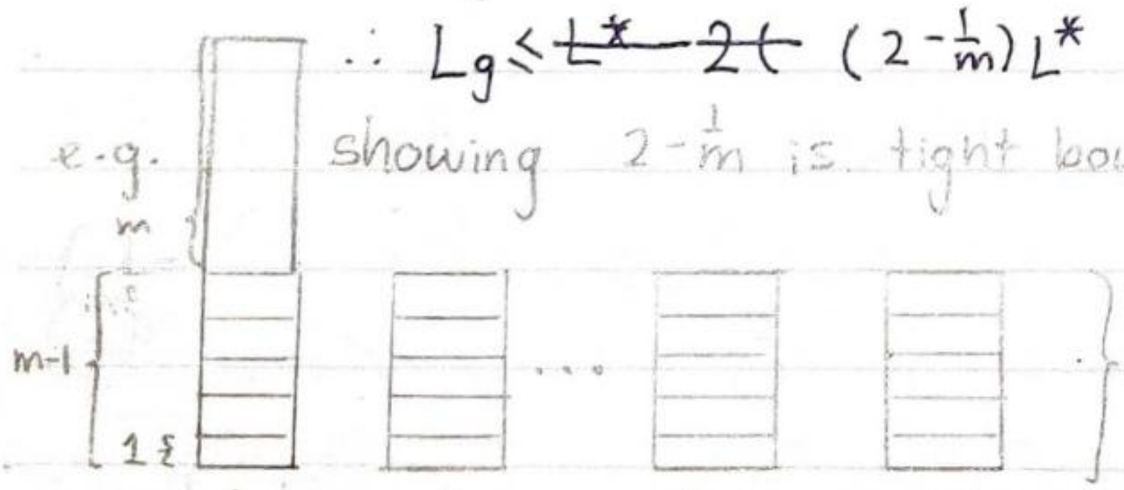
$= L^* + (1 - \frac{1}{m}) t_l$

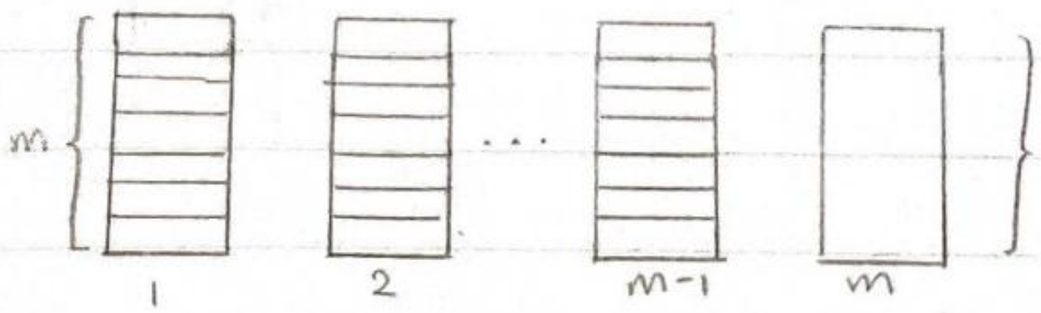
$\Rightarrow L_g \leq L^* + (1 - \frac{1}{m}) t_l$ ($t_l \leq L^*$)

$L_g \leq L^* + (1 - \frac{1}{m}) L^*$

$\therefore L_g \leq \cancel{L^* \quad 2t} \ (2 - \frac{1}{m}) L^*$ ■

e.g. showing $2 - \frac{1}{m}$ is tight bound





$\frac{\text{greedy}}{\text{optimal}} = \frac{2m-1}{m} = 2 - \frac{1}{m}$

algorithm (greedy 2): Arrange jobs in decreasing length order // $t_1 \geq t_2 \geq \cdots \geq t_n$

Assign job to currently least loaded machine

Let L^{lf} = makespan of assignment produced by longest first greedy algorithm (longest first)

L^* = optimal makespan

THM 2: $L^{lf} \leq L^*(\frac{3}{2} - \frac{1}{2m})$

PROOF Let i be most loaded machine in assignment of longest first greedy algo $L^{lf} = l_i$

Let l be last job assigned to i

CASE 1: l is only job assigned to i

$\Rightarrow L^{lf} = L^* \quad L^{lf} = t_l \leq L^*$

CASE 2: l is not the only job assigned to i

$\Rightarrow l \geq m+1$ (m jobs have been assigned to the m machines)

Note: $t_{m+1} \leq \frac{L^*}{2}$

Assume for contradiction $t_{m+1} > \frac{L^*}{2}$

$\Rightarrow$ for any assignment (particularly optimal one), some machine has two jobs $j, j' \leq m+1$ assigned to it

$\Rightarrow$ load on that machine $\geq \overbrace{t_j + t_{j'}}^{\text{at least } j, j' \text{ assigned}} > \frac{L^*}{2} + \frac{L^*}{2} = L^*$

$L^{lf} \leq L^* + (1 - \frac{1}{m}) t_l$ (some from last page) $t_j + t_{j'} > L^* \Rightarrow$ contradiction!

$\leq L^* + (1 - \frac{1}{m}) t_{m+1}$ ($t_l \leq t_{m+1}$)

$\leq L^* + (1 - \frac{1}{m}) \frac{L^*}{2}$ ($t_{m+1} \leq \frac{L^*}{2}$)

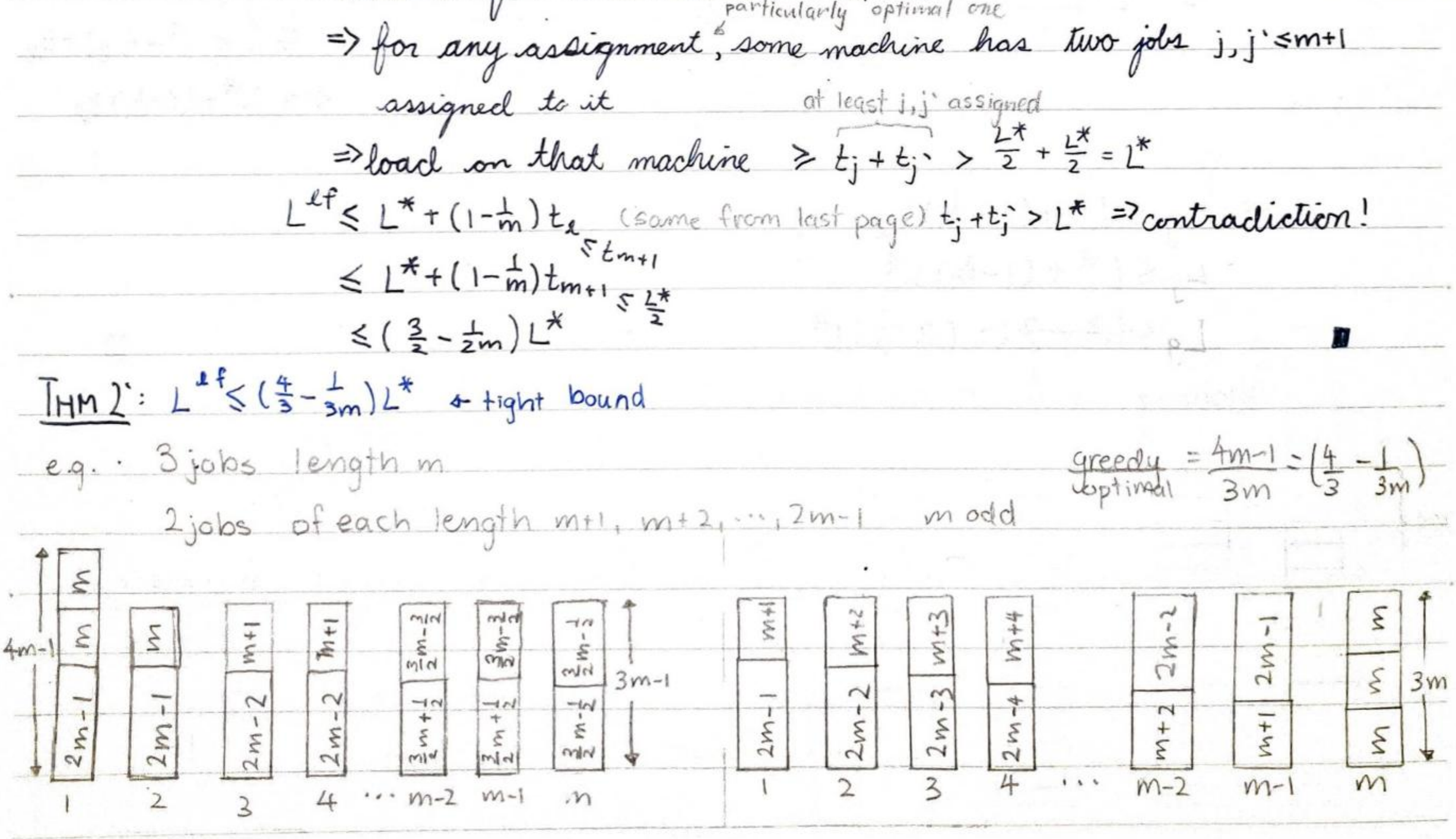
$\leq (\frac{3}{2} - \frac{1}{2m}) L^*$

THM 2': $L^{lf} \leq (\frac{4}{3} - \frac{1}{3m}) L^*$ ← tight bound

e.g. · 3 jobs length m

2 jobs of each length $m+1, m+2, \cdots, 2m-1$ m odd

$\frac{\text{greedy}}{\text{optimal}} = \frac{4m-1}{3m} = (\frac{4}{3} - \frac{1}{3m})$

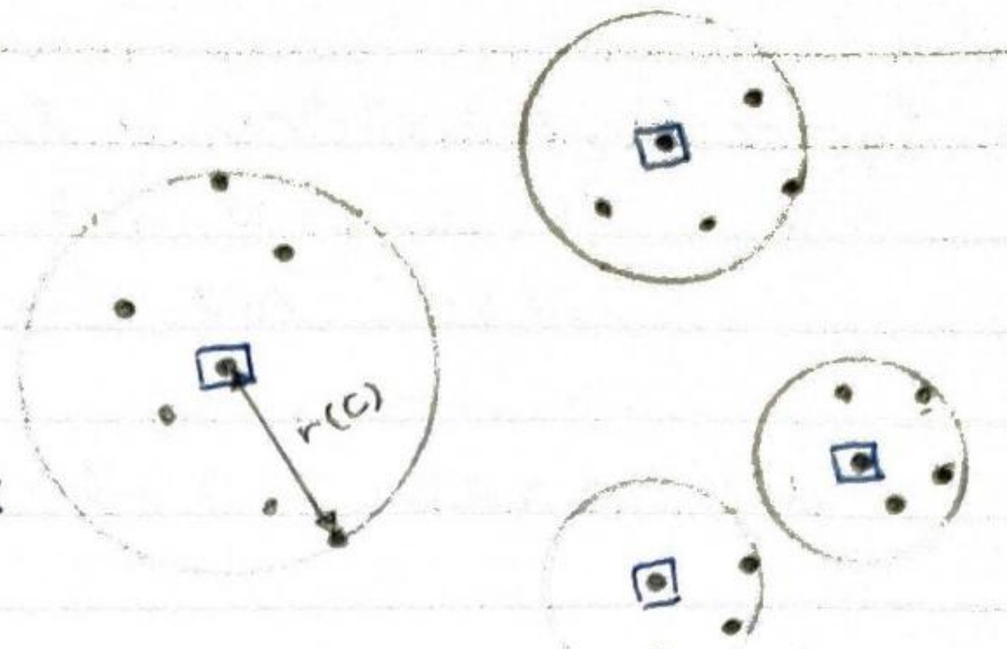


make span (longest first) = 4m-1 make span (optimal) = 3m

K-CENTRE PROBLEM

input: set S of $n \geq 1$ "sites", $k \in \mathbb{Z}^+$

output: $C \subseteq M$ s.t. $|C| = k$ with min $r(C)$



depending on size of k, the location of □ (facilities) can change drastically

Metric space M with distance function d

$d: M \times M \to \mathbb{R}^+$ (sites $\in M$)

d satisfies: a. $d(x,y) = 0 \iff x = y$

b. $d(x,y) = d(y,x)$

c. $d(x,y) \leq d(x,z) + d(z,y)$ (triangle inequality)

$d(S,C) = \min_{s \in C} d(S,s)$ (~~distance of~~ minimum distance from S to a centre in C)

$r(C) = \max_{s \in S} d(s,C)$ (max distance from any site to its closest centre)

algorithm: $S_1 :=$ any site

$C_1 := \{S_1\}$

for $i := 2$ to k do

$S_i :=$ a site $s \in S$ that maximizes $d(s, C_{i-1})$

$C_i := C_{i-1} \cup \{S_i\}$

return C_k

Let C^g = set of centres (located in chosen site) computed by greedy algorithm

C^* = optimal set of centres (any one point in space)

CLAIM: $r(C^g) \leq 2\, r(C^*)$

PROOF Suppose we run for loop until $k+1$ (1 more than we need, but still return C_k)

CLAIM: For each iteration $i \geq 2$ $\forall i \in \{2, 3, \cdots, k+1\}$

a. $d(S_i, C_{i-1}) = r(C_{i-1})$

b. $\forall s, s' \in C_i$, $s \neq s'$, $d(s,s') \geq d(S_i; C_{i-1})$

PROOF ~~imm~~ immediate from choice of S_i and definition of $r(C_{i-1})$

No.

Date

b. Suppose for contradiction we have $s, s' \in C_i$. $s \neq s'$
with $d(s_i, C_{i-1}) > d(s, s')$

1. $s, s' \neq s_i$ (not new centre)

so, $s = s_j$, $s' = s_{j'}$ $j, j' < i$ WLOG suppose $j > j'$

$$d(s_i, C_j) \geq d(s_i, C_{i-1}) > d(s, s') = d(s_j, s_{j'}) \geq d(s_j, C_{j-1})$$

$C_j \subseteq C_{i-1}$; assumption ; assumed $j > j'$ must have sth at least as close as $s_{j'}$ in C_{j-1} from s_j as $s_{j'} \in C_{j-1}$

But then the algorithm would have added s_i, not s_j to C_{j-1} in iteration j

$\Rightarrow$ contradiction $\therefore d(s_i, C_{i-1}) \leq d(s, s')$ ∎

CLAIM $\Rightarrow$ $r(C_k) \leq d(s, s')$ $\forall$ $s, s' \in C_{k+1}$ s.t. $s \neq s'$ (*)

C_{k+1} has $k+1$ centres, C^* has k centres $\overset{\text{PHP}}{\Rightarrow}$ $\exists$ distinct $s, s' \in C_{k+1}$ that are within $r(C^*)$ of one of the centres of C^* (call it c)

$r(C_k) \leq d(s, s')$ [by (*)]

$\leq d(s, c) + d(c, s')$ [by triangle inequality]

$\leq r(C^*) + r(C^*)$

$= 2r(C^*)$ ∎

Unless P=NP, there is no polytime algorithm for k centre problem with approx ratio < 2

0/1 KNAPSACK PTAS

Polynomial Time Approximation Scheme $\rightarrow (1-\varepsilon)$·optimal value

running time: $O(\frac{n^3}{\varepsilon})$ (smaller ε is, larger runtime is)

input: items $1, 2, \cdots, n$, item i has value v_i, ε

C = knapsack capacity weight w_i

output: ~~max~~ value of knapsack $\geq (1-\varepsilon)$·optimal value

algorithm (original DP): $K(0,0) := 0$; $V(0) := 0$

for $i := 1$ to n do $K(i,0) := 0$; $V(i) := V(i-1) + v_i$

$K(i,V)$ = min weight of items needed for a set of value $\geq V$

for $i := 1$ to n do

for $v := 1$ to $V[i]$ do

if $v > V[i-1]$ then $K(i,v) := w_i + K(i-1, \max(0, v-v_i))$

else $K(i,v) := \min(K(i-1,v), w_i + K(i-1, \max(0, v-v_i))$

return $\max \{ v : K(n,v) \leq C \}$

running time: $O(nV)$ $\quad V = \sum_{i=1}^{n} v_i$

algorithm (approximation): Discard all items of $wt > C$

$V_{max} := \max_{1 \leq i \leq n} v_i$

obtain new scaled values $\hat{v}_i := \left\lfloor \frac{v_i}{v_{max}} \cdot \frac{n}{\varepsilon} \right\rfloor \quad \forall i = 1, 2, \cdots, n$ (scale down to $(0,1]$)

run DP algo with new scaled values

Return set output from above

running time: $O(n^2 \hat{V}_i)$ $\quad \hat{v}_i = \left\lfloor \frac{v_i}{v_{max}} \cdot \frac{n}{\varepsilon} \right\rfloor \leq \frac{n}{\varepsilon} \Rightarrow O\left(\frac{n^3}{\varepsilon}\right)$

$\hat{V}_i = \sum_{1 \leq i \leq n} \hat{v}_i \leq \frac{n^2}{\varepsilon}$

WHY $\frac{n}{\varepsilon}$?

Let $\hat{v}_i = \left\lfloor \frac{v_i}{v_{max}} \cdot \sigma \right\rfloor$ $\quad$ Let $\hat{S}$ = optimal knapsack for scaled values (approximation)

S^* = optimal knapsack for original values v_i

$\Downarrow$

$\hat{v}_i \leq \frac{v_i}{v_{max}} \sigma \Rightarrow v_i \geq \hat{v}_i \frac{v_{max}}{\sigma}$

we want $\sum_{i \in \hat{S}} v_i \geq (1-\varepsilon) \sum_{i \in S^*} v_i$

$\sum_{i \in \hat{S}} v_i \geq \sum_{i \in \hat{S}} \hat{v}_i \cdot \frac{v_{max}}{\sigma} = \frac{v_{max}}{\sigma} \sum_{i \in \hat{S}} \hat{v}_i \geq \sum_{i \in S^*} \hat{v}_i$ because $\hat{S}$ optimal for scaled $\hat{v}_i$

$\geq \frac{v_{max}}{\sigma} \sum_{i \in S^*} \hat{v}_i$

$= \frac{v_{max}}{\sigma} \cdot \sum_{i \in S^*} \left\lfloor \frac{v_i}{v_{max}} \sigma \right\rfloor$ [by def of $\hat{v}_i$]

$\geq \frac{v_{max}}{\sigma} \sum_{i \in S^*} \frac{v_i}{v_{max}} (\sigma - 1)$ $\quad [\lfloor x \rfloor \geq x - 1]$ $\quad$ let k^* = optimal val for original knapsack

$= \sum_{i \in S^*} \frac{v_{max}}{\sigma} \cdot \frac{v_i \sigma}{v_{max}} - \frac{v_{max}}{\sigma} |S^*|$ ($|S^*| \leq n$, $k^* \geq$) $\quad$ let $\frac{n}{\sigma} = \varepsilon$

$\geq \sum_{i \in S^*} v_i - \frac{n k^*}{\sigma} = k^* \left(1 - \frac{n}{\sigma}\right)$ $\Rightarrow$ $\Rightarrow \frac{n}{\varepsilon} = \sigma$ ($\frac{n}{\sigma} = \varepsilon$)