19/9/23

Page No.
Date

Dynamic Programming

Greedy - natural
- many problem hard can't be solved
- correctness hard.

Divide & Conquer - Improvement over existing polynomial
Dnn algo $O(n^2) \rightarrow O(n \log n)$

How to get from exponential to polynomial?

DP Borrow some idea from divide & conquer.
- Explore the space of solutions by dividing problem into smaller sub-problems.
- Find solution for smaller sub problems, build a solution for larger sub problems.
- Close to brute force (as explore the solutions space but "carefully" not explicitly (not look at each individually)

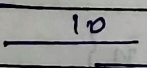
Weighted Interval Scheduling.

Input: n intervals $\{1, \ldots, n\}$ $s_i, f_i, v_i \quad \forall i \in [n]$
(s_i: start time, f_i: finish time, v_i: weight)

Task: To find $S \subseteq [n]$ s.t. $v(S) = \sum_{i \in S} v_i$ is maximised & all intervals in S are disjoint.

The solution that didn't work here (put $v = 1$ & check)

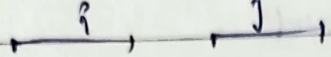
Greedy - schedule the job with earliest finish time



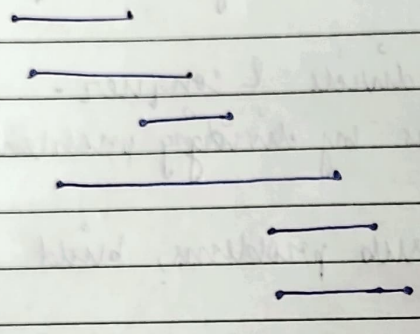
Greedy o/p = 2 optimum = 10

Assumption Intervals are sorted by finish time

$i < j \Rightarrow f_i < f_j$



parent ← $p(j) = i < j$ s.t. i is largest index not intersecting with j. (If no i exists $p(j) = 0$



$p(1) = 0$
$p(2) = 0$
$p(3) = 0$
$p(4) = 0$
$p(5) = 3$
$p(6) = 3$

Consider any optimum solution O,
look at the interval n

Case I : $n \in O$
for all i in $\in \{p(n)+1, p(n)+2, \ldots n-1\}$ $i \notin O$
Look at the subproblem $\{1, 2, \ldots p(n)\}$.
Let an optimal solution be O' for this
→ If $n \in O$ then $n \cup O'$ is an optimum solution

Case II : $n \notin O$
subproblem $\{1, \ldots, n-1\}$.
If O' optimal, it is optimal for whole problem.

For finding optimum of $\{1, \ldots n\}$
If we have solutions for $\{1, \ldots j\}$ $\forall j \in (n-1)$
we can compute solutions for $\{1, \ldots p(n)\}$ & $\{1, \ldots n-1\}$

Let O_j be the optimum solution for $\{1, \dots j\}$.
$opt(j)$ be $v(O_j) = \sum_{i \in O_j} v_i$

$O_0 = \phi$.
$OPT(0) = 0$.
we want to compute O_n & $OPT(n)$

For $j \in \{1, \dots n\}$.
either $j \in O_j$
$OPT(j) = v_j + OPT(p(j))$
or $j \notin O_j$
$OPT(j) = OPT(j-1)$

$$OPT(j) = \max(v_j + OPT(p(j)), OPT(j-1)) \} \rightarrow (1)$$

Lemma - Interval i belongs to an optimum solution O_j for the set $\{1, \dots j\}$ if and only if
$v_i + OPT(p(j)) \geq OPT(j-1)$

→ Intervals sorted by f_i
→ Calculate $p(j)$ $\forall$ $j \in [n]$
→ Recursion procedure to compute $OPT(j)$

Compute_OPT(j)
If $j=0$ {return 0}
else return max (v_j + compute_opt(p(j)), compute_opt(j-1))
End if

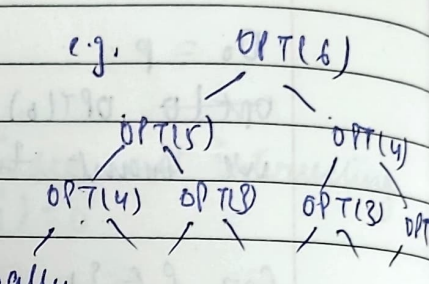
Correctness - Base case $j \neq 0$ $j=0$.
Assume that algo computes correctly for all $i < j$
For j compute_OPT(j) uses compute_OPT(p(j)) and compute_OPT(j-1)
correct by induction hypothesis

Page No.
Date

21/9/23

by ① compute_OPT(j) = OPT(j) is correct.

Running Time -
recursive call can be exponential e.g. OPT(6)
→ same value is computed again and again



Ideas Compute only once and store globally for later computation "Memoization"

Exercise - Do memoization for Fibonacci
M[j] → global array
M-Compute-opt(j)
If j=0 return 0
If M[j] is not empty
Return M[j]
Else M[j] = max(v_j + M-Compute-opt(p(j)), M-Compute-opt(j-1))
Return M[j]
End if

Correctness - same as before

Running Time - Measures approach
Measure: # non-empty entries in M[] (initial 0)
Every time a recursive call is made, one extra gets filled.

For every entry → 2 recursive calls
O(n) recursive calls ⇒ Total O(n)
we have computed OPT(n) what about On (sorted inputs)
Total recursive time if not sorted O(nlogn)
→ Compute p(j) in linear time for all j.

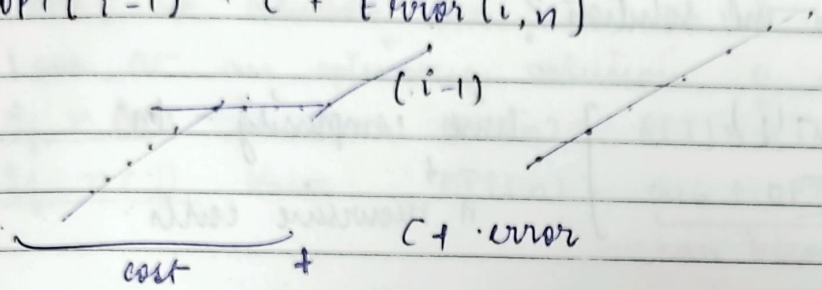
O(n)

1.
2.
3.
n.

Input

If we can identify the last segment that p_n is part of $\{p_i, p_{i+1}, \ldots p_n\}$.

cost of solution will be.
$OPT(i-1) + C + Error(i, n)$



Recurrence: $OPT(j) = \min_{1 \le i \le j} Error(i, j) + C + OPT(i-1)$ (6.7)

and the segment $p_i, p_{i+1}, \ldots p_j$ is used in an optimum solution for the subproblem for p_j. if the minimum for recurrence 6.7 is obtained for that i.

Need to compute $Error(i, j)$ for every i, j [n^2 pairs].

For each pair, $O(n)$ time
Total time $O(n^3)$ for

~~OPT~~ Algo – pseudo code (Book) → $O(n^3)$

For all pairs $i < j$
— — — — → $Error(i, j) \to O(n^3)$
End for.

$O(n^2)$ For $j = 1, 2, \ldots n$.
use the recurrences (6.7) to compute $M[j]$
end for

$OPT(j) = \min_{1 \le i \le j} Error(i, j) + C + OPT(i-1)$

Total time — $O(n^3)$

Page No.
Date

Correctness: By induction, $M[j]$ is computed correctly for all smaller subproblems.
⇒ $M[j]$ is correct using the recurrence.

How to find the solution?

Find-segment(j) — Time complexity - $O(n)$
n recursive calls.

Exercise - Find $E(i,j)$ $\forall i,j$ in $O(n^2)$ time.

23/9/23 dynamic Programming
"divide into smaller subproblems"

How to decide which subproblem?
What to do when the obvious set of subproblems is not enough.

Problem: Subset sum
n requests $\{1, \ldots, n\}$ w_i $\forall i \in [n]$
- Machine can take weight at most W (capacity).
- Want to minimise idle-time
- Put as many (by weight) jobs as possible.

Find $S \subseteq [n]$ s.t. $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} w_i$ is as large as possible.

Greedy - smallest weight first.

counter example - 2, 3, 4 W = 4
greedy 2
opt - 4

Page No.
Date

Largest wt. first
counter example : 2 3 4 with w=5
greedy → 4 opt = 5

Pf: OPT$_i$ → solution for $\{1, \ldots i\}$.
look at an optimum solution O for $\{1, \ldots n\}$.
If $n \notin O$ then $OPT(n) = OPT(n-1)$
If $n \in O$ then $OPT(n) = w_n + OPT(n-1)$
can become greater than w
∴ problem.

- Need to store some more information
- We introduce a new variable for subproblem.

The solution for $\{1, \ldots, n-1\}$ with weight at most $W - w_n$.

Subproblem - OPT(i, w) denotes optimum value of a solution for $\{1, \ldots, i\}$ and total weight at most w.
$S \subseteq [n]$, value $= \sum_{i \in S} w_i$.

Consider n^{th} request.
$n \notin O$ $OPT(n, W) = OPT(n-1, W)$
$n \in O$ $OPT(n, W) = w_n + OPT(n-1, W - w_n)$

$OPT(n, W)$ → output of the problem.

Recurrence for $OPT(i, w)$
If $w_i > w$
$OPT(i, w) = OPT(i-1, w)$
otherwise $w_i \notin OPT(i, w)$
$OPT(i, w) = OPT(i-1, w)$
~~$w_i \in OPT(i, w)$~~ $w_i \in OPT(i, w)$
$OPT(i, w) = OPT(i-1, w - w_i) + w_i$

$OPT(i,w) = \max\{OPT(i-1,w),\ OPT(i-1, w-w_i)+w_i\}$ → (1)

If we have figured out the values of $OPT(j,w)$ for all $j<i$, then we can compute $OPT(i,w)$.

Initialisation: $OPT(0,w) = 0$ for all w.

- $OPT(i,w)$ $\quad 0 \le i \le n$
  $\quad 0 \le w \le W$

W, w_i's are integers

Algorithm: Output $OPT(\ )$ → Compute using the recurrence.

→ Two dimensional table; which you

subset-sum(n,w)
Array $M[0\ldots n][0,\ldots w]$
Initialise $M[0,w]=0$ $\forall w = 0,1\ldots W$
for $i=1$ to n
  for $w=0$ to W
    use (1) to compute $M[i,w]$
  End for
End for
Return $M[n,W]$

Analysis - corrections - Induction on i
$i=0$ → Base case
We want to compute $OPT(i,w)$
Algo has already comp $OPT(i-1,w)$ $\forall w$ correctly I.H.
⇒ $OPT(i,w)$ correct due to recurrence
Show correctness of (1)

Page No.
Date

Running time - $O(n,W)$ size of the 2D array
can fill each entry in constant time
w_i & W might be given in binary
1010 → of size $\log_2 W = k$.
$W = 2^k$ $\qquad O(nW) = O(n2^k)$

→ work well for small values of W, for large W can be bad $O(nW)$ → Pseudo Polynomial (as not polynomial on the input).

Knapsack
N items $\{1, \ldots n\}$ weights $w_1, \ldots w_n$ value $v_1 \ldots v_n$
A bag with capacity C (wt)
Fill the bag w max. value
We want to find $S \subseteq [n]$ s.t. it maximise
$\sum_{i \in S} v_i$ while $\sum_{i \in S} w_i \leq C$

$OPT(i,w)$ $n \notin O$
$OPT(n,w) = OPT(n-1,w)$

~~$n \in O$~~
~~$OPT(n,w) = OPT(n-1,$~~

$n \in O$.
$OPT(n,w) = OPT(n-1, w - w_n) + v_n$

Recurrence –
If $w < w_i$, then $OPT(i,w) = OPT(i-1,w)$
otherwise $OPT(i,w) = \max(OPT(i-1,w), OPT(i-1, w-w_i) + v_i)$

Algo similar to subset sum
Running time $O(nC)$ — pseudo poly.] Exercise.
If value small, DP on values instead]

26/9/23

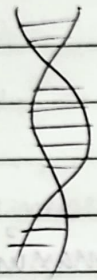
Page No.
Date

RNA Secondary Structure

Subset sum / knapsack : Needed a new variable (obvious) subproblem were not sufficient)

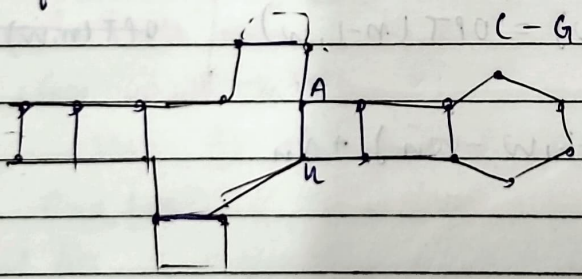
Another way of getting new subproblems :
Instead of $\{1, \ldots, i\}$ for every i, we can also have $\{i, \ldots, j\}$ for every i, j, for every i, j such that $i \leq j$. "Every interval defined a subproblem".

Problem DNA :



Double strand structure "zipped together through bonds".
Similarly building blocks are -
Bases $\{A, C, T, G\}$.
$A = T$, $C \equiv G$.

RNA : Single strand, make bonds with itself to form useful shape Bases : $\{A, C, U, G\}$.
$C - G$, $A - U$.



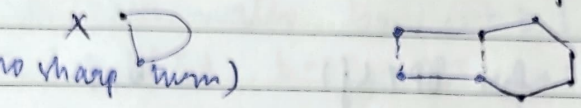
Let $B = b_1, \ldots, b_n$ be the string representing RNA molecule where $b_i \in \{A, C, U, G\}$.
- one base for one bond
- should not "cross"
- No sharp turns.

- We want to maximise number of bonds subject to above constraints.

Page No.
Date

Secondary structure - A secondary structure on a RNA molecule $B = b_1, \ldots b_n$ as a set of base pairs $S = \{(i,j)\}$ such that $i < j$ and $i, j \in [n]$ and the following conditions hold

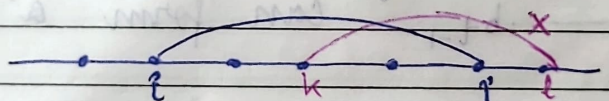
1. No sharp turns - for each pair $(i,j) \in S$, $j > i + 4$



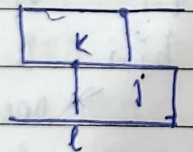
2. If $(i,j) \in S$ $\{b_i, b_j\} = \{A, U\}$.
or $\{b_i, b_j\} = \{C, G\}$.

3. S is a matching (all endpoints are distinct)
$(i_1, j_1), (i_2, j_2) \in S$ then i_1, j_1, i_2, j_2 are all distinct.
↓
not equal.

4. If (i,j) & $(k,l) \in S$ then it cannot be that $i < k < j < l$



no bond overlapping.



(unravelling will not give straight string)

Hypothesis - RNA is stronger / more stable when it has a lot of base pairs.

Aim: Given a single stranded RNA molecule $B = b_1, \ldots, b_n$, find a secondary structure with max. no. of base pairs.

Designing an algorithm.

OPT(j) → maximum # base pairs in a secondary structure of $b_1, \ldots b_j$

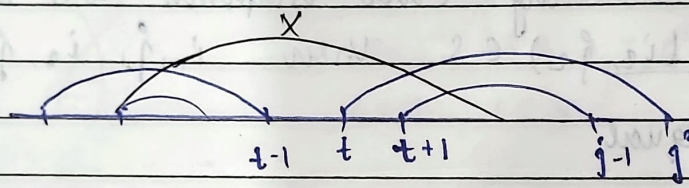
If $j \le i$, OPT(d) = 0 (condition 1)

OPT(n) → gives the answer

Recurrence for OPT(j) $b_1, \ldots b_j$

Case 1 b_j does not form a base pair in optimum.

Case 2 b_j forms a base pair with b_t with $t < j-4$.



Condition 4 says that none of the bases in $b_1, \ldots b_{t-1}$ can form a pair with $b_{t+1}, \ldots b_j$.

Two subproblem - $b_1, \ldots b_{t-1}$ OPT(t-1)

$b_{t+1}, \ldots b_{j-1}$?

→ Does this have a subproblem.

⇒ Need to have a subproblem for every $i, j \in [n]$ s.t. $i \le j$.

OPT(i, j): Maximum # base pairs in a secondary structure on $b_i, \ldots b_j$.

OPT(i, j) = 0 if $i \ge j-4$

OPT(i, j) = 0 if $0 > j$.

Page No.
Date

comming up with recurrence.
$b_i, \ldots, b_j$

Case 1 b_j does not form a base pair is optimum
$OPT(i,j) = OPT(i, j-1)$

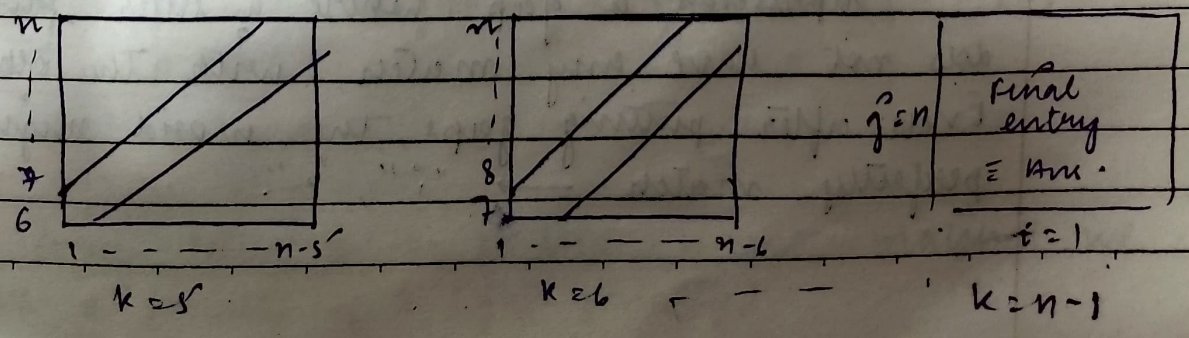
Case 2 b_j forms a pair with b_t, $t \geq 1$, $t < j-4$.
$OPT(i,j) = 1 + OPT(i, t-1) + OPT(t+1, j-1)$

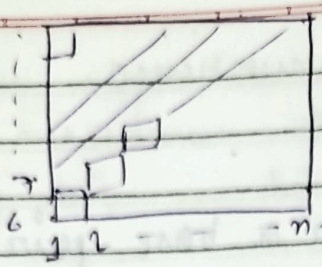
$\Rightarrow OPT(i,j) = \max_t \left(OPT(i, j-1),\ 1 + OPT(i, t-1) + OPT(t+1, j-1) \right)$

Max is taken over all t s.t. b_t & b_j are allowed to make pair by conditions ① & ②.

How to order the subproblems?
For (i,j) define $k = j - i$ (define ordering on k)
Do a DP in increasing order of $k = j - i$.

Algo:
Initialise $OPT(i,j) = 0$ where $i \geq j-4$.
for $k = 5, 6, \ldots, n-1$
  for $i = 1, 2, \ldots, n-k$ (Interval length k fixed)
    set $j = i + k$
    Compute $OPT(i,j)$ using ① → ②.
  End for
End for
Return $OPT(1, n)$

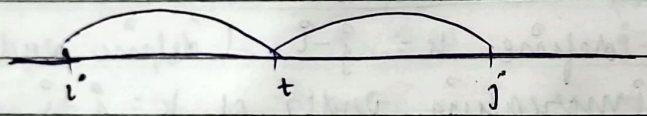




correctness - Induction on k.
Follows from the recurrence.

- we can also find the solution in addition to the value, by looking at which term maximise the recurrence.

Runtime Analysis.
subproblems : $O(n^2)$
for each subproblem how much time



t has almost n choices $t \leq k \leq n$.
Total time $O(n^3)$

28/9/23 Sequence Alignment
spelling suggestions →
"alignmint" → Did you mean "~~allow~~ "alignment"?

a l l i g n m i n t
a l = i g n m e n t.

"-" represents a gap. where a letter from a word does not have any match with the other.
Even after putting gaps, the word might not perfectly match → "i", "e".

Similarly - can define # gaps & # mismatches

a l l i g n m i _ n t] 3 gaps
a l _ i g n m _ e n t] 0 mismatch

1 0 0 0 1 1 0 0 0 0 1 1 0 & 1 0 1 0 1 1 1 0 0 0 0 0 1 0

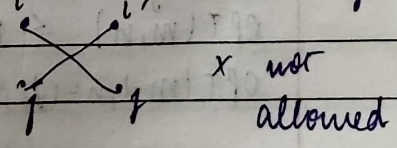
1 0 0 0 1 1 - - 0 0 0 0 1 1 0] 3 gaps.
1 0 1 0 1 1 1 0 0 0 0 0 1 - 0] 1 mismatch.

Application - DNA sequencing in molecular biology.
Question - How to define similarity - gaps, mismatches.
gaps - will have a fix penalty.
mismatches - penalty will depend on letters

Definition - Given $X = x_1 x_2 \cdots x_m$] may be of
$Y = y_1 y_2 \cdots y_n$] diff size.

$\{1, 2, \cdots m\}$ → position in X.
$\{1, 2, \cdots n\}$ → position in Y.
A matching $\{(i, j)\}$ $i \in [m]$, $j \in [n]$ is an alignment if there are no "crossing pairs".

Non-crossing pairs - (i, j), (i', j') are non crossing pairs if $i < i' \Rightarrow j < j'$



X = S t o p
Y = t o p s

	1	2	3	4
	S	t	o	p
	_	t	o	p
	1	2	3	4

alignment = $\{(2,1), (3,2), (4,3)\}$.

Cost of an alignment -
→ For every letter x or y that is not matched, we have a penalty of δ (gap penalty)

2. For each pair (p, q) in the alphabet, there is a mismatch cost $\alpha_{p,q}$ for matching p & q. $p \neq q$.
3. For each $(i, j) \in M$, we incur cost $\alpha_{x_i y_j}$
4. Usually $\alpha_{p,p} = 0$ (not necessary for algo to work)

Total cost of M — sum of gap penalties & mismatch costs. α_{pq} are external parameters which vary according to application but remain fixed for the algorithm.

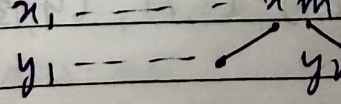
Question: Given $X = x_1 \cdots x_m$
$Y = y_1 \cdots y_n$.
find an alignment which minimises the cost.
Eg s t o p — cost $= 2\delta + \alpha_{tt} + \alpha_{oo} + \alpha_{pp}$
— t o p s

Designing an Algorithm —
$X = x_1, \cdots x_m$ $OPT(i,j) \rightarrow$ cost of an optimum solution
$Y = y_1 y_2 \cdots y_n$ for $x_1 x_2 \cdots x_i$
$y_1 y_2 \cdots y_j$.

Let M be an optimum solution for X & Y.

Case 1 $(m,n) \in M$.
$OPT(m,n) = OPT(m-1, n-1) + \alpha_{x_m y_n}$.
(To get this equality show $\leq$, $\geq$ both sides)
$OPT(m,n) \leq OPT(m-1, n-1) + \alpha_{x_m y_n}$
$OPT(m-1,n-1) \geq OPT(m,n) - \alpha_{x_m y_n}$.

Case 2 $(m,n) \notin M$ $\Rightarrow$ one of them is not going to be matched



(No partner left for y_n)
follows from "non crossing property".

Case 2.1 x_m not matched
$OPT(m,n) = OPT(m-1, n-1) + \delta$.

Page No.
Date

Case 2.2 y_m not matched $\rightarrow OPT(m,n) = OPT(m,n-1) + \delta$

similarly for $i \geq 1$, $j \geq 1$

$OPT(i,j) = \min[\alpha_{x_i y_j} + OPT(i-1, j-1), \delta + OPT(i-1, j), \delta + OPT(i, j-1)]$

(i,j) is part of optimum alignment if minimum is achieved by $\alpha_{x_i y_j} + OPT(i-1, j-1)$

No: of subproblem - $O(mn)$
cost of solution given by - $OPT(m,n)$
Initialisation - $OPT(0,i) \equiv OPT(i,0) = \delta i$ (one empty & baki i left in other so we allign w gaps - each - δ)

Alignment (X, Y)
Array $A[0 \ldots m \; 0 \ldots n]$
Initialise $A[0,j] = \delta j \quad \forall j \in [n]$
$A[i,0] = \delta i \quad \forall i \in [m]$
For $j = 1$ to n
To for $i = 1$ to m
find $A[i,j]$ from recurrence
End for
End for
Return $[m,n]$

Time complexity - Time for some $A[i,j] \rightarrow O(1)$.
Total Time - $O(mn)$
Total space - $O(mn)$

Exercise - Find the optimum ~~solution~~ cost. using $O(m+n)$ space.
$O(\min(m,n))$

Find optimum solution using $O(m+n)$ space.
(using divide & conquer)

29/9/23

Page No.
Date

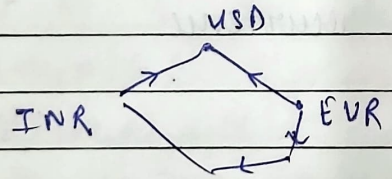
Two problems -

1. Given a directed ~~cycle~~ graph G with edge weights, decide whether G has a cycle of -ve cost.
   Does $\exists$ C (a cycle) $\sum_{(i,j) \in C} c_{ij} < 0$

2. If the graph has -ve cycle, then ~~for~~ given s & t, find a path P from s to t of minimum cost.
   (minimise $\sum_{(i,j) \in P} c_{ij}$)
   ↑ shortest path.

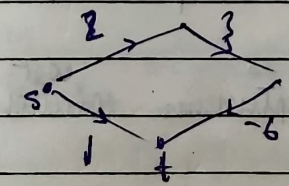
e.g. currency exchange.

INR ⟶
loose swing → -ve wt.
gain swing → +ve wt



- If there is a -ve cycle, we can go ~~without~~ around it arbitrary many times to bring down the cost of a path between two vertices.

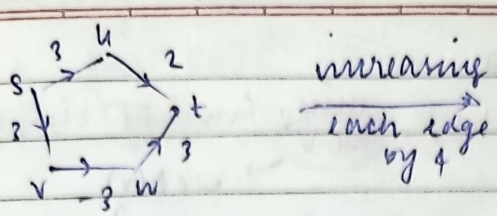
Dijkstra's -



$d(s,t) = 1$ according to Dijkstra's

$d(s,t) = 1$

Idea: Increase edge wt. uniformly



min cost path - s u w t
cost → 3

Increases by 4.
shortest path s u t, cost → 12

Dynamic programming (Bellman-Ford Algorithm)
$V = \{1, \ldots, n\}$.

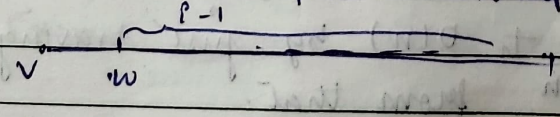
keep aside [OPT(i) = shortest path between s and t using first i vertices

subproblem - OPT(i, v) → min. cost of a v-t path having atmost i edges.
$\forall\ v \in V,\ \forall\ i \in [n-1]$

Lemma - If G has no -ve cycle, then there is a shortest path from s to t that is simple (no vertices are repeated) and hence has length $\leq n-1$

Proof - If vertex repeated → cycle → non-ve weight → remove it from path → cost cannot go up.

Final solution - OPT(n-1, s)
Express OPT(i, v) as smaller subproblems.
"Multiway choices" → like segmented least squares.
Let P be a path corresponding to OPT(i, v).



Case 1: If P has $< i$ edges.
OPT(i, v) = OPT(i-1, v)

Case 2: If P has i edges: Let w be first vertex after v on P.
suppose we know this.
OPT(i, v) = cost + OPT(i-1, w)

w → neighbours of v

Page No.
Date

Recurrence - for $i > 0$

$OPT(i,v) = \min\left(OPT(i-1,v),\ \min_{w \in V} \left(c_{vw} + OPT(i-1,w)\right)\right)$

$w \in N(v)$

shortest path (G, s, t) — $G \rightarrow$ weighted directed graph

n = # vertices in G

Array $M[0\ 1\ 2 \ldots n-1, v_1, v_2, \ldots v_n]$.

Define $M[0,t] = 0$ $\quad M[0,v] = \infty$ for all $v \neq t$.

For $i = 1$ to $n-1$

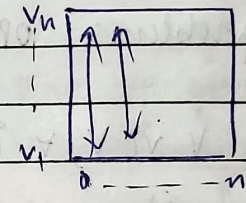
  For $v \in V$ (in any order)

    compute $M[i,v]$ using recurrence

  End for

End for

Return $M[n-1, s]$



Analysis - Correctness : follows from recurrence.

Running time - size of table $O(n^2)$

Each entry $O(n)$

Total time $O(n^3)$ } naive.

For each vertex v → check $d(v)$ entries to find min.

Running time - $O(n \cdot \sum d(v)) = O(mn)$

↓

better than $O(n^3)$ if $m << n^2$.

Space : $O(n^2)$ [$n \times n$ matrix]

↳ decrease to $O(n)$ by just having $(i-1)^{th}$ array and computing i^{th} from that.

How to find path : for each v, have something next(v)

↓

the first vertex after v in path P.

or just keep next(i,v)

i : length of path.

Keep somehow. next$[v]$ → reduces the space for finding the path.

next(v) ≡ first(v)

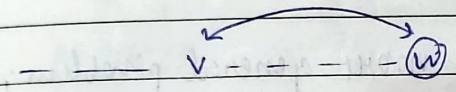
{v, first(v)} will be the edges in the shortest path from u to t

Lemma - After i iterations of the outer for loop, the path obtained by {(u, first(v))} has min. length among all the paths of length at most i from v to t.

$OPT(v) = \min(OPT(i-1), \min_{w \in V} (c_{vw} + OPT(i-1, w)))$

$\min_{w \in N(v)}$

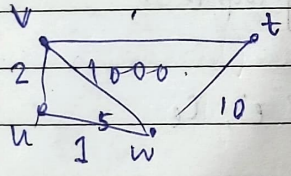
i^th



OPT(v) → length i

OPT(w) → length i+1

Invariant : After i iterations : M[v] is at most weight of any path of length at most i.



(direct edge from v-t)

M[v] → 1000 (after i=1 finishes)

M[w] = 10

i=2 ~~suppose w comes before u, v~~ u, w, v comes in order

M[u] = 11 M[w] = 10 M[v] = 13

(After i iterations main i length ka path aa chuka)

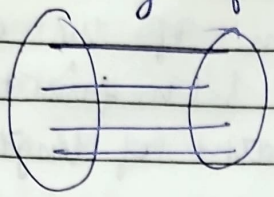
~~Exercise~~ Exercise - Floyd-Warshall APSP.

10/10/23

Page No.
Date

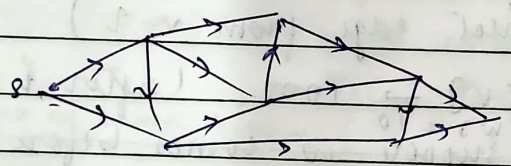
Network flows-

Bipartite Matching - Given a bipartite graph $G = (V_1, \emptyset, V_2, E)$ find a matching of maximum size in G.



- can be used to model many real world problems.
  - Job - machine, employees - employees.
- Has polynomial-time algorithm
  - ↳ different technique.
- Develop an algorithm for a more general problem; which would imply a polynomial-time algo for bipartite matching

Maximum flow problem - Model transportation networks using graphs.



$\forall e, c_e$

model road networks / pipelines.

nodes: intersection in road network or joints in pipeline.
edges: roads / pipes.
edge capacities: capacity of road / pipes.

source node (s) ⟶ sink node (t)

↓ "Generates" flows. ↓ "Absorbs" flow.

Definition - A "flow network" is a directed graph $G = (V, E)$ with following properties.

1. $\forall e \in E$, there is a capacity $c_e \in \mathbb{R} \geq 0$
2. Source node $s \in V$
3. sink node $t \in V$
4. No edges entering s, no edges containing leaving t.
5. every vertex has at least one edge incident on it.
6. c_e's are all integers

Flow - An s-t flow is a function $f : E \to \mathbb{R} \geq 0$ such that

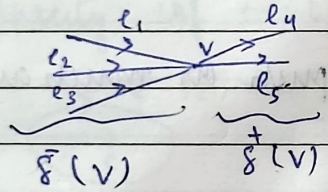
i) capacity constraint - $\forall e \in E$, $0 \leq f(e) \leq c_e$

ii) conservation constraint - $\forall u \in V \setminus \{s, t\}$.

$$\sum_{e \in \delta^-(v)} f(e) = \sum_{e \in \delta^+(v)} f(e)$$

set of incoming edges to v — set of outgoing edges from v.

value of a flow f : $v(f) = \sum_{e \in \delta^+(s)} f(e)$



$f(e_1) + f(e_2) + f(e_3) = f(e_4) + f(e_5)$

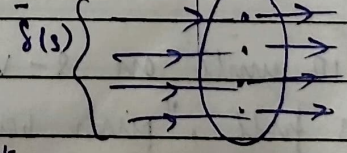
Define : $f^{in}(v) = \sum_{e \in \delta^-(v)} f(e)$ $\quad f^{out}(v) = \sum_{e \in \delta^+(v)} f(e)$

$\forall u \in V$.

~~Conservation Constraint~~

Generalise to set $S \subseteq V$

Generalise to set $S \subseteq V$



$f^{in}(S) = \sum_{e \in \delta^-(S)} f_e$ $\quad f^{out}(S) = \sum_{e \in \delta^+(S)} f_e$

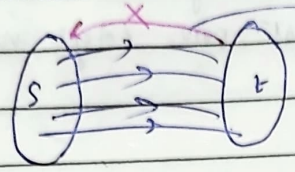
Conservation constraint - $f^{in}(v) = f^{out}(v)$

$\forall u \in V \setminus \{s, t\}$.

value of flow - $v(f) = f^{out}(s)$

Maximum flow - Given a flow network, find a flow of maximum value.

(not considering backward edges) ←



capacity of cut $\sum_{e \in cut,\ v_1 \to v_2} c_e$

Maximum flow cannot exceed capacity of all the cuts (v_1, v_2) s.t. $s \in v_1$ & $t \in v_2$

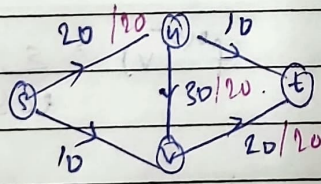
Min-cut → cut with minimum capacity.

⇒ Maximum flow ≤ minimum cut

[minimum cut ≤ Maximum flow] → we will see later

Designing an Algorithm for Maximum flow

Greedy - find any s-t path, push as much as possible on this path $\min_{e \in P} c_e$



P = s - u - v - t.

f(su) = f(uv) = f(vt) = 20.

v(f) = 20.

Maximum flow value = 30

- Push 10 units on s-v
- v-t is full, "push back" 10 units on u-v (decreasing flow by 10)
- remaining 10 units get diverted to u,t.
- Total 30.

Page No.
Date

Ideas: 1. "push forward" from s when available.
2. "push backward" on places already carrying flow.
3. "divert" flow

How to make ideas formed?

Residual Graph - Given a flow network $G = (V, E)$ & a flow f on G, define G_f as residual graph of G w.r.t f as following

$V(G_f) = V(G)$

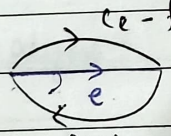
$E(G_f) = E_1 \cup E_2$.

$E_1 = \{ e \in E(G) \mid f(e) < c_e \}$.
(forward edges)

capacity of $e \in E_1 \longrightarrow c_e - f(e) > 0$

$E_2 = \{ vu \mid uv \in E(G),\ f(e) > 0 \}$.

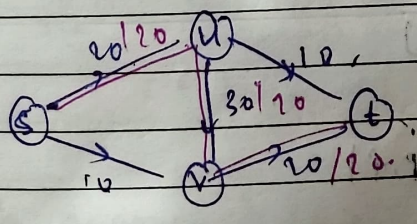
capacity of $e \in E_2 \longrightarrow f(e)$

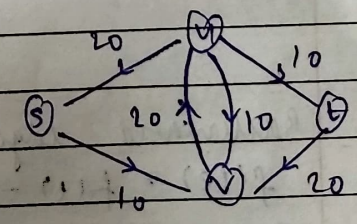


- For every edge $e = uv \in E(G)$ it can result in two edges in G_f : $0 < f(e) < c_e$ → two edges.

$E(G_f) \leq 2\,|E(G)|$

Capacity of $e \in E(G_f)$ is called "residual capacity"





Let P be a simple s-t path in G_f then we want to push some flow on P.

Bottleneck (P, t) = min. residual capacity of an edge in P.