Greedy approach.

→ every soln. is irrevocable.

⇒ soln optimal at that moment is chosen

→ all the soln put together gives final soln.

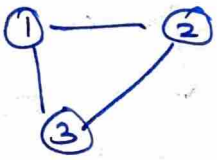
---

suggests constructing soln thro sequence of steps, each expanding a partially constructed soln. obtained so far, until a complete soln. to the problem is reached.

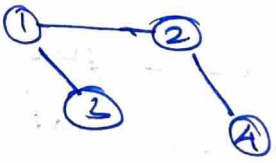
⇒ choice is feasible, locally optimal & irrevocable.

Minimum spanning Tree:

Given a connected graph, a spanning tree is a connected acyclic sub graph that contains all vertices of a graph



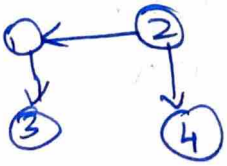
connected graph



Vertices connected by edges.

Edges may be undirected lines.

or directed.



all edges directed → directed graph

otherwise

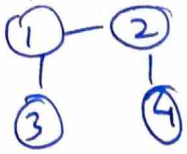
undirected graph

every edge

⇒ weighted graph

every node has a degree.

how many edges are incident on a node



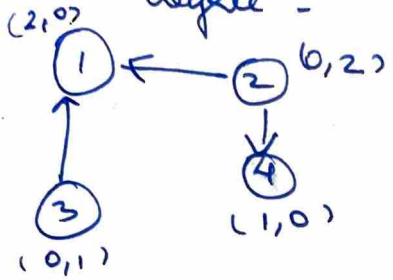
① - 2 edges are incident

degree of ① is 2

degree of ② is 1.

in degree - how many coming into node

Out degree - " " " out of node



in degree of ① ⇒ 2

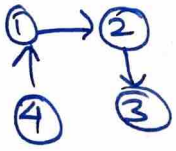
out degree of ① ⇒ 0

out degree of ② ⇒ 2.

edge from ② to ① but no edge from ① to ② so we say ② & ① are adjacent.

⇒ Direct path btw 2 nodes without intermediate, then they are said to be adjacent.

(edge connecting 2 vertices)

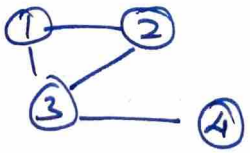


eg: 1 & 2 are adjacent

1 & 3 not adjacent.

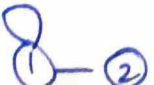
---

cycle:

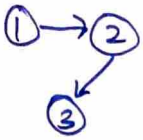


From a node if you are able to return to that node, it is cycle.

we can have self loops

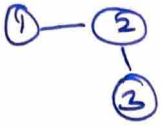


directed acyclic graph (DAG)



how many edges it traverses gives length of the graph.

If you can travel from every vertice to every other vertex, it is called connected graph.



1 to 3 through 2

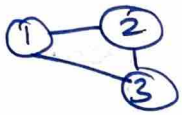
---

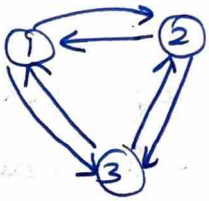
a directed graph. two nodes can be connected in both dirn.

---

complete graph.

Separate edge btw every vertex to every other vertex.



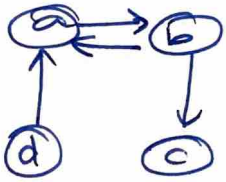


complete graph → 'n' nodes

$\frac{n(n-1)}{2}$ → directed edges

$\frac{n(n-1)}{2}$ → undirected edges

Representation of graphs : adjacency matrix, list.



acyclic graph

4 vertices.

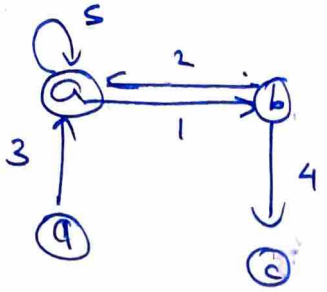
adjacency matrix 4x4.

	a	b	c	d
a	0	1	0	0
b	1	0	1	0
c	0	0	0	0
d	1	0	0	0

unweighted → binary matrix

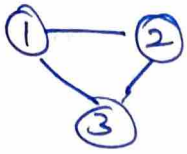
weighted → weights will be present in graph

self loop then that value will (x,x) - 1.



	a	b	c	d
a	5	1	0	0
b	2	0	4	0
c	0	0	0	0
d	3	0	0	0

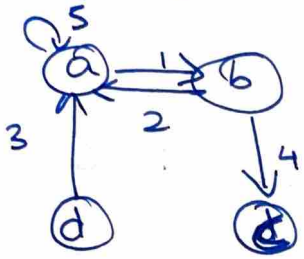
weighted matrix

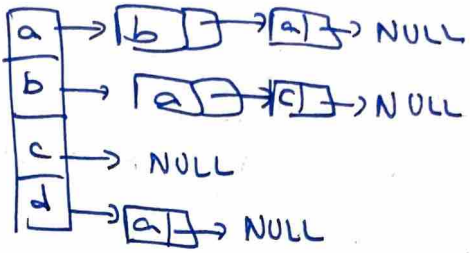


When you remove cycles from a graph, it becomes a tree.

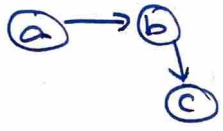
Tree → acyclic graph.

Adjacency list:





only a set of vertices & edges from a given graph is called subgraph.

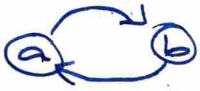
eg: 

Btw. a pair of vertices if we have more than one edge connecting them

↙

Multigraph

→ this is multigraph



→ not a multigraph

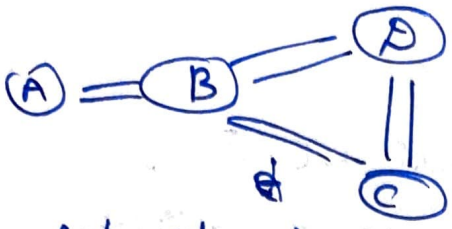
graph w. only one vertex → trivial graph.

Isomorphic graph:

2 graphs are isomorphic - same no. of vertices, edges & same no. of vertices of the given degree.
same

Euler circuit

every edge exactly once & return back



Not euler circuit as A-B traversed more than once.

Hamiltonian circuit

traverse every vertex exactly once & return back

Directed → symmetric

Undirected → asymmetric.

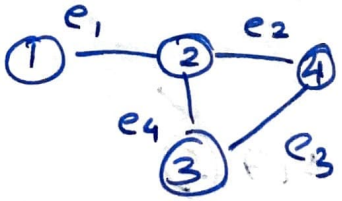
Incidence matrix

'n' no. of vertices, 'e' edges

	e1	...	...	ee
1	1			
2	1			
n				

⇒ Mij = 1 if jth edge is incident on ith vertex.

eg:

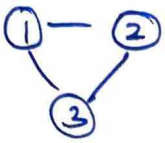


	e1	e2	e3	e4
1	1	0	0	0
2	1	1	0	1
3	0	0	1	1
4	0	1	1	0

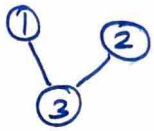
Minimum spanning tree:

Given a connected graph, a ST is a connected acyclic subgraph that contains all vertices of the graph.

In a weighted graph, a minimal ST is a ST of graph with the smallest sum of weights of edges in ST.



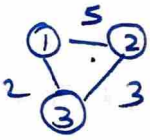
acyclic subgraphs.

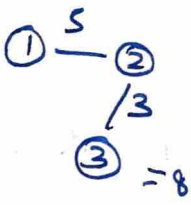


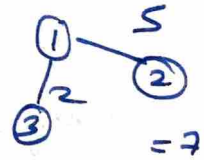
Spanning tree - weighted / unweighted

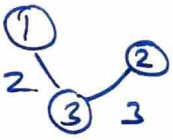
Minimum S.T - weighted.

Minimum ST



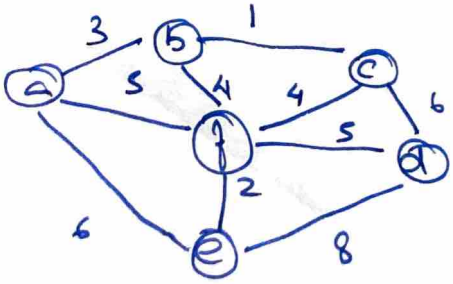






sum of weights of all edges = 8, 7, 5

Kruskal's algorithm.

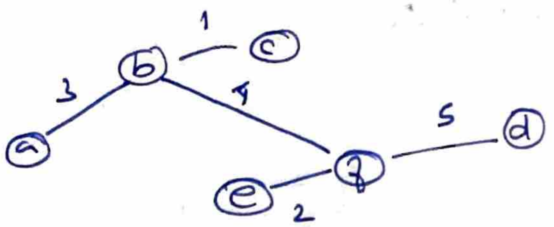


1 bc 2 ef 3 ab

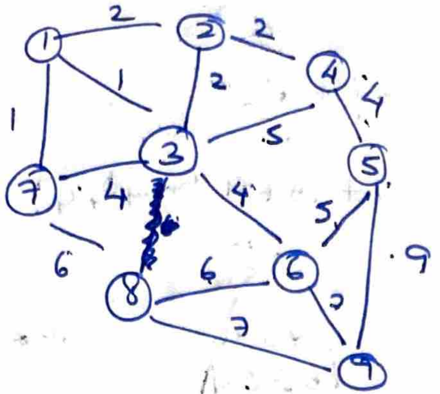
4 bf 4 cf 5 af 5 df

cd 6 8

Construction of M.S.T



eg..

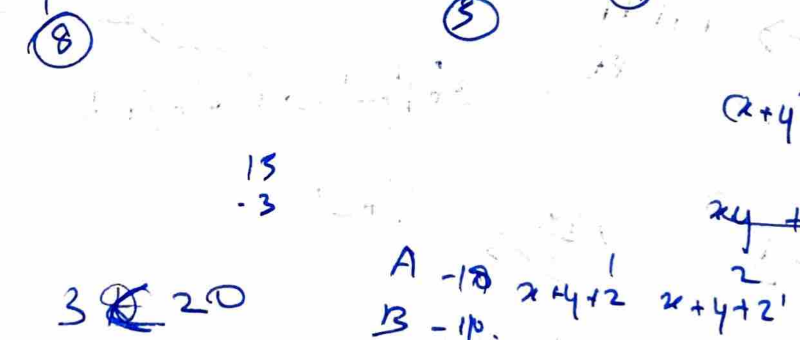


1 1/3 1/7 2/12 2/23 2/24

4/37 4/36 4/45 5/34 5/36 6/78 6/68

7/69 7/89 9/59

16 | 54



18
13
3

(x+y) (x+z')

15
3

xy + x.z'

3 20

A - 10 x+y+z x+y+z' x+y'+z'

B - 10.

C - 12. x'+y+z 4 x+y'+z

D - 13. x'+y'+z 5 x'+y+z'

E - 12.

F - 14

G -

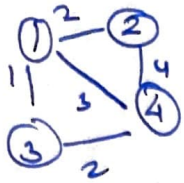
This was repeated until all vertices were present in tree.

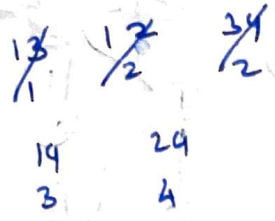
⇒ Disjoint set use.

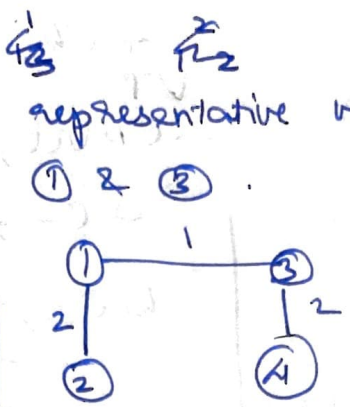
Makeset — Find — Union — can be performed, individual deleted

↳ individual

all vertices created individually

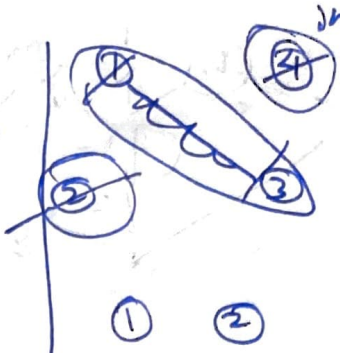






representative vertex

1 & 3.



representation diff. then diff. subset. then union performed

representative

1, 2, 3 → 1

4 → 4.

after merging of 4

1, 2, 3, 4 → 1.

no. of times representation changes $< \log_2 n$

Complexity

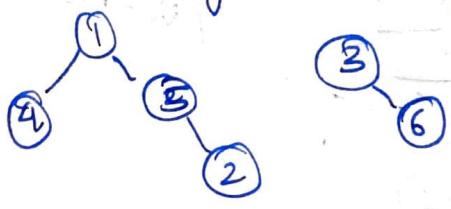
- Makeset (x) : $\Theta(1)$
- Find (x) : $\Theta(\log n)$ → it is not n as we will never have skewed repn.

Union (x, y) : $\Theta(1)$

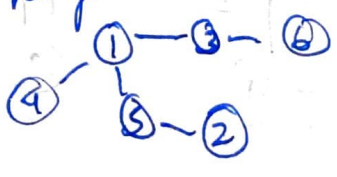
For (n-1) union & m find : $\Theta(n + m \log n)$

Union by Rank:

→ measuring rank of tree based on ht. (how many levels sub tree has & perform union by rank

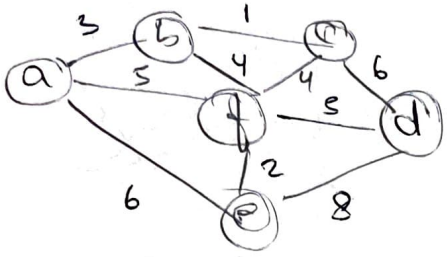


⇒



Prim's Algorithm. (Less Kruskals from 'one' else)

* any vertex can be selected as starting vertex
* every vertex represented by a (,)
* find out if vertex has any other adjacent vertex in MST



~~a (b, 3)~~

VT

a (NULL, NULL)

a (-, -)

b (a, 3)

c (b, 1)

f (b, 4)

e (f, 2)

Remaining (for a in MST)

b (a, 3)	e (a, 6)
c (-, -)	f (a, 5)
d (-, -)	

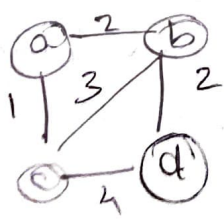
c (b, 1)	e (a, 6)
d (-, -)	f (b, 4)

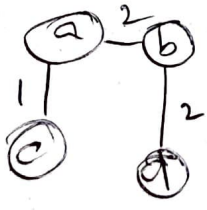
(adjacent to more than one vertex choose edge with min. wt)

d (e, 6)	e (a, 6)
f (b, 4)	

d (f, 5) e (f, 2)

d (f, 5)





b (a, 2)

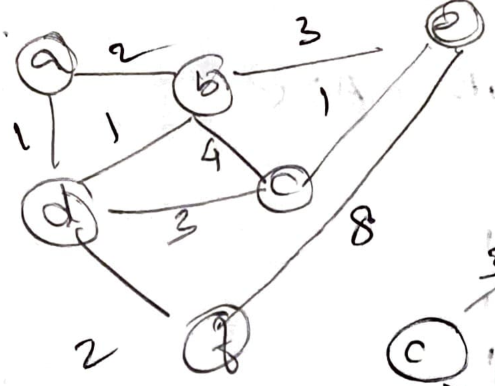
c (a, 1)

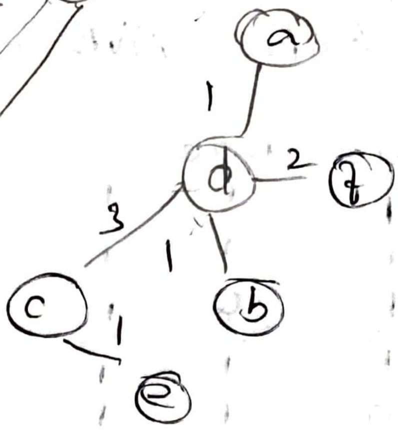
d (-, 2)

b (a, 2)

d (c, 4)

d (b, 2)





~~b (a, 2)~~
c (- - -)
d (a, 1)
e (- - -) f (- -)

~~b (d, 1)~~ e (-, -)
c (d, 3) f (d, 2)

c (bd, 3) e (b, 3)
~~f (d, 2)~~

~~c (d, 3)~~ e (b, 3)
e (c, 1)

$\forall x \; F(x) \rightarrow P(x)$

$\forall x \; \sim F(x) \vee P(x)$ $\exists x \; F(x) \wedge \sim P(x)$

$\forall x \; (\exists$ ~~y α~~$) \ldots \exists y \, (\alpha) \vee \exists z \, (\sim \beta))$

$\forall x \; (\; \forall y \, (\alpha) \rightarrow \exists z \, (\sim \beta)$ Every Graph V connected

$P \subset Q \Rightarrow (Q \rightarrow P)$ $\sim Q \vee P$.

$(P \vee Q)$ $\sim Q \rightarrow P$.

$P \subset \bar{Q}$

~~∃x ∃y (+sa(x) ∧ equivalent~~

$(\forall x \; f(sa(x)))$ (x loves y at t.) $\forall x \; \exists y \; \exists z$ x^{f}

$\forall z \; \forall y$ (~~x~~ loves y at t) $(P \wedge Q) \vee (P \wedge \sim Q)$ $P \vee Q$

Q

$P \wedge (Q \vee \sim Q)$

V_T - vertex E_T - edges (v,u) such that v in MST, u

Prim's algorithm

// Prim's alg. for MST

// i/p: a weighted connected graph $G = (V, E)$

// o/p: E_T, set of edges comprising MST of G

$V_T \leftarrow [v_0]$ // the set of tree vertices can be initialized with any vertex

$E_T \leftarrow \emptyset$

for $i \leftarrow 1$ to $|V| - 1$ do

find a min. weight edge $e^* = (v^*, u^*)$ among all edges such that v is in V_T & u is in $V - V_T$.

$V_T \leftarrow V_T \cup [u^*]$

$E_T \leftarrow E_T \cup [e^*]$

return E_T.

T_0 - one vertex

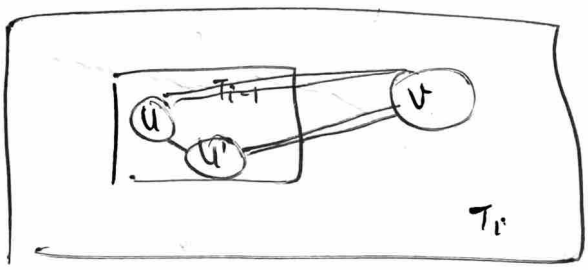
T_1 - 2 vertices

$T_i = T_{i-1}$

Proof by contradiction.

Assume T_{i-1} is a MST

T_i is not MST.



v u!

min. wt edge

Complexity

graph is represented as adjacency list & priority queue is implemented as a min heap

1. No of deletes $|V|-1$

   No of verifications : $|E|$

changing the priority in a min heap takes place atmost $|V|$ times.

- Complexity : $\log|V|$

Complexity $(|V|-1+|E|)\log V = |E|\log V$

as $|V|-1 < |E|$ in a connected graph

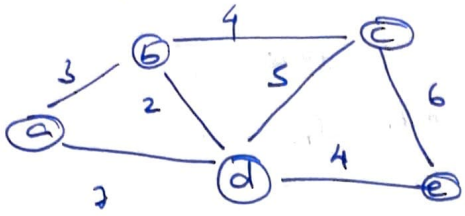
(can't not get shortest path)

Single source shortest path

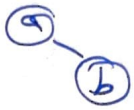
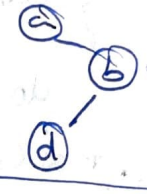
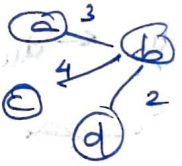
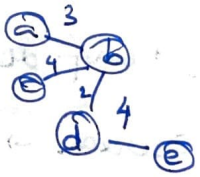
Dijkstra — source to all other vertex shortest path.

- Applicable to undirected & directed graphs with non negative weights only.

example



// from source through vertex, shortest path //

a(-,-)	b(a,3) c(-,∞) d(a,7) e(-,∞)	
	c(b,7) through b from a. d(b,5) e(-,∞)	
	c(b,7) thro b. e(d,9) thro b,d.	
	e(d,9) thro b,d c(d,11) thro d. e(d,16) thro b,d,c e(d,13) thro b,c e(d,11) thro b,c,d	

Algorithm Dijkstra(G, s)

// i/p: A weighted connected graph G(V, E) w. nonnegative wts and its vertices

// o/p: The length d_v of a shortest path from s to v & its penultimate vertex p_v for every vertex v in V.

Initialize(Q) // priority queue to empty

for every vertex v in V

$d_v \leftarrow \infty$; $P_v \leftarrow$ NULL

Insert(Q, v, d_v) // initialise vertex priority in the priority queue.

$d_s \leftarrow 0$; Decrease(Q, s, d_s) // update priority of s with d_s.

$V_T \leftarrow \emptyset$.

for $i \leftarrow 0$ to $|V|-1$ do

$u^* \leftarrow$ DeleteMin(Q) // delete min. priority elemen.

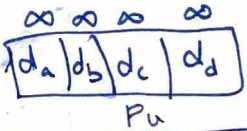
$V_T \leftarrow V_T \cup \{u^*\}$

for every vertex u in $V - V_T$ that is adjacent to u^* do

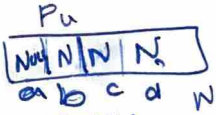
if $d_{u^*} + w(u^*, u) < d_u$

$d_u \leftarrow d_{u^*} + w(u^*, u)$; $P_u \leftarrow u^*$

Decrease(Q, u, d_u)



P_u

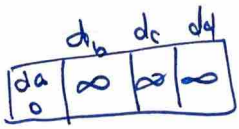


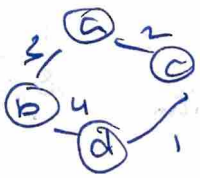
NULL

d_s → ~~any~~ dist. from source to source.

Decrease → fun. updating the priority queue.

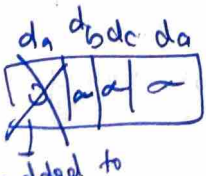
$d_s \leftarrow 0$





V_T ← shortest path graph

Delete Min(Q) → delete elements with min. priority & makes it u^*.

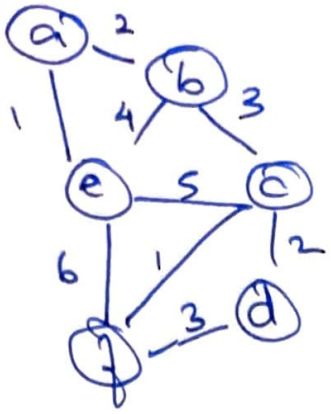


if $d_{u^*} + w(u, u^*) < d_u$

then $d_u = d_{u^*} + w(u, u^*)$

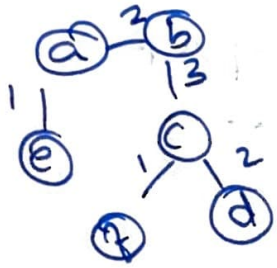
Prep

penultimate → pu ← u*.



a (-, -)

b(a, 2)	a, e
c(-, ∞)	
d(-, ∞)	
e(a, 1)	
f(-, ∞)	
b(a, 2)	a, b, e
c(e, 6) through e	
d(-, ∞)	
f(e, 7) through e	
c(b, 5) through b	a, b, e, c
d(-, ∞)	
f(e, 7) thro e	
d(c, 7) thro b, c	a, b, e, c, f
f(c, 6) thro b, c	
d(c, 7)	



Complexity:

Adjacency matrix & priority queue as an unordered array: $O(|V|^2)$ $V \rightarrow$ vertices.

Adjacency lists a priority queue implemented as a min heap: $O(|E| \log |V|)$

Dynamic programming:

- technique for problems with overlapping subproblems.
- These subproblems arise from recurrence relating a given problem's soln to solns of its smaller subproblems.
- principle of optimality: an optimal soln to any instance of optimization problem is composed of optimal solns to its sub instances

Fibonacci numbers

$F(n) = F(n-1) + F(n-2)$ for $n > 1$

$F(0) = 0$. $F(1) = 1$

$F(2) = F(1) + F(0) = 1$. $F(3) = F(2) + F(1) = 1+1 = 2$.

all are subproblems and overlapping.

→ this is an eg. of dynamic programming.

Knapsack Problem:

- Given n items of known weights $w_1, \ldots, w_n$ (+ve int) and values $v_1, \ldots v_n$ (dec.) & a knapsack of capacity W (+ve int) find the most valuable subset of items that fit into knapsack.
- Assume that all weights & knapsack capacity are +ve int. The item values don't have to be int.

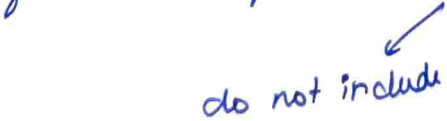
0/1 → item is taken completely / not taken at all.

Soln:

1, 2, 3 . . . . i $\quad 1 \le i \le W$

1. excluding i^{th}
2. may / may not including i^{th}

- Consider an instance defined by first i items $1 \le i \le n$, with wts $w_1 \ldots w_i$ values $v_1 \ldots, v_i$ & knapsack capacity j, $1 \le j \le W$.
- Let $F(i,j)$ be value of opt. soln. i.e the value of most valuable subset of first i items that fit into knapsack of capacity j.
- We can divide all subsets of first i items that fit the knapsack (j) into 2 categories



F(2,3) means most valuable among 1st and 2nd when we have knapsack of capacity of 3.

1. Not including ith.

F(i, j) => we have i-1 items. = F(i-1, j)

F(i, j) = F(i-1, j)

2. may or may not include i:

if i is not valuable

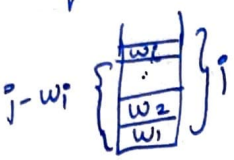
F(i, j) = F(i-1, j) (same as 1st case)

if i is also valuable,

$F(i,j) = F(i-1, j-w_i) + v_i$

↓

value of ith item.



$$F(i,j) = \begin{cases} \max(F(i-1,j),\ v_i + F(i-1, j-w_i)) & j-w_i \geq 0 \\ F(i-1,j) & j-w_i < 0. \end{cases}$$

capacity of knapsack = 0. then value = 0.

F(i, 0) = 0.

no items

F(0, j) = 0.

item	wt.	value
1	w_1 2	$12 v_1
2	w_2 1	$10 v_2
3	w_3 3	$20 v_3
4	w_4 2	$15 v_4

W = 5.

F(1, 4) = max (F(1, 1) + F(2, 0)

F(1, 5):

max (F(1,3) + F(i-1, j-wi)

F(1,

Bottom up approach

→ individual subproblems to goal.

item represented as rows, Capacity columns

i ↓ / j →	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

$F(i,0) = 0$

$F(0,j) = 0$

$12 + F(i-1, j-w_i)$

$12 + (20$

$j - w_i \geq 0$

$10 + F(1, 0)$

Max { $F(i-1, j)$, $v_i + F(i-1, j-w_i)$ }

$F(i-1, j)$ if $j - w_i < 0$

$F(2,1)$

{ $F(1,1)$, $10 + F(1,0)$

$F(2,2) =$ { $F(1,2)$, $10 + F(1,1)$

$F(2,2) =$ { $F(1,2)$, $10 + F(1,1)$

(12)

Complexity

w

n

$(n+1)(w+1)$

$O(nW)$

Top-Down approach - memory fun. (same formula)

virtual initialisation.

initialise to -1.

i / j	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	–	12	22	–	22
3	0	–	–	22	–	32
4	0	–	–	–	–	(37)

$F(4,5) = \max[F(3,5), F(3,3) + v_5]$

$F(3,5) = \max[F(2,5), F(2,2) + 20]$

$F(3,3) = \max[F(2,3), F(2,0) + 20]$

$F[2,3) = \max[F(1,3), F(1,2) + 10]$

$F(1,3) = \max[F(0,3), F(0,1) + 12]$

$F(1,3) = 12$

$F(1,2) = \max[F(0,2), F(0,0) + 12]$

$= 12.$

$F(2,3) = 22$

$F(3,3) = 22$

$F(2,5) = \max[F(1,5), F(1,4) + 10]$

$F(1,5) = \max[F(0,5), F(0,3) + 12] = 12.$

$F(1,4) = 12.$

$F(2,5) = 22$

$F(i-1, j) \quad v_i + F(i-1, j-w_i)$

In top down approach, the values (kept as blank) was computed in bottom up approach. which is a waste of time. => top down approach is preferred.

Complexity $O(\frac{nW}{2})$ => $O(nW)$

Algorithm for top down approach

MF knapsack (i, j)

// implements memory fun. method for knapsack problem.

//i/p : a non neg. int. i → no. of first items being considered & non neg. int j → capacity of knapsack

//o/p : value of optimal feasible subset of first i items.

Note: Use global var. Weights[1...n] values [1...n] & F[0...n, 0...W] where entries are initialized with -1 except row 0 & column 0 initialized with 0.

if F[i,j] < 0

if j < Weights[i]

value ← MFKnapsack(i-1, j)

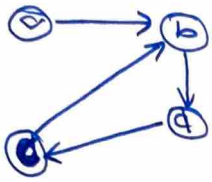
else

value ← Max (MFKnapsack[i-1, j], Values[i] + MFKnapsack(i-1, j - Weights[i]))

F[i,j] ← value

return F[i,j].

Transitive closure



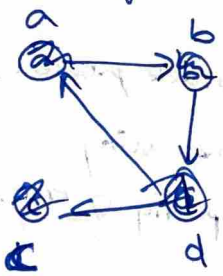
Length of path
a to c = 2
a to d = 3

Matrix

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	1		
d				

Warshall's Algorithm

Transitive closure of a directed graph with n vertices can be defined as a n x n boolean matrix T={tij} in which the element in ith row & jth column is if there exists a non trivial path (i.e directed path of +ve length) from ith vertex to jth vertex, otherwise 0.



A

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

R^0

R_0 → path matrix o no. of intermediate vertices

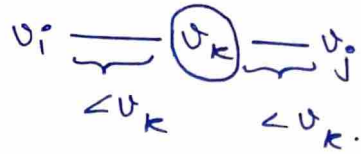
R_1 → path matrix with intermediate vertices no. not higher than 1.

R_2 → ≤ 2 R_3 → ≤ 3.

R^k → path matrix with intermediate vertices ≤ k.

there will be a path btw v_i & v_j

R^k :

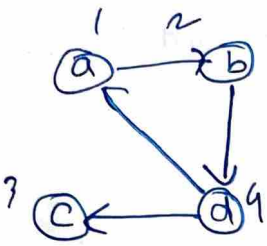


if not v_k, the intermediate vertices $< k$ has 1,

$R_{ij}^{k-1} = 1$ then there is a path.

Warshall's alg. construct the transitive closure of $n \times n$.

a list of intermediate vertices each numbered $< k$.



① ↓ a

$$R^1 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

o intermediate ~~as~~ 'a' as intermediate vertex.

R^0

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

0, a, b.

$$R^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

$R^1 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix}$

0, a, b, c

$$R^3 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

0, a, b, c, d

$$R4 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

- Warshall (A[1...n, 1...n])

// to compute transitive closure
// i/p adjacency matrix of A 'n' vertices
// o/p transitive closure of digraph

$R^{(0)} \leftarrow A$

for k ← 1 to n do
  for i ← 1 to n do
    for j ← 1 to n do

$R^{(k)}[i,j] \leftarrow R^{k-1}[i,j]$ or ($R^{k-1}[i,k]$ and $R^{k-1}[k,j]$)

return $R^{(n)}$

Complexity: $\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} (OR) = n^3$.

Basic oper: or / and (∵ equal no. of both present)

Floyd's Algorithm: (All pairs shortest paths)

→ computes the distance matrix of weighted graph with n vertices through a series n×n matrices: $D^{(0)}, \ldots D^{(k-1)}, D^{(k)} \ldots D^{(n)}$

⇒ Dist. btw a&a, b&b, c&c, d&d = 0.

⇒ given weight matrix R^0.

D^k → distance matrix w. intermediate vertices numbered not less than = k.

v_i — v_k — v_j

$\underbrace{\quad}_{\leq k}$ $\underbrace{\quad}_{\leq k}$

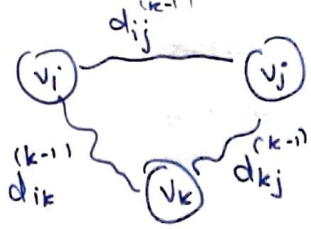
d_1 d_2

D^{k-1} D^{k-1}

ik kj

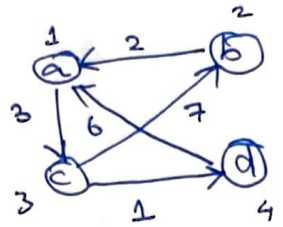
distance btw v_i & v_j (via v_k) = $d_1 + d_2$

$\min(D_{ij}, D_{ik} + D_{kj})$ v_i & v_k v_k & v_j



$$\min\left(d_{ij}^{k-1},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

eg:



$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 0 & 3 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 7 & 0 & 1 \\ 6 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

not able to differentiate dist. of (a to a case) & no edge

so, no path then represent it with ∞.

$$D = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & \infty & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

$$D' = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ \infty & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

$$D^2 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \infty & 3 & \infty \\ 2 & 0 & 5 & \infty \\ 9 & 7 & 0 & 1 \\ 6 & \infty & 9 & 0 \end{bmatrix} \end{array}$$

$$D^3 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 9 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

$$D^4 = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array}\begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

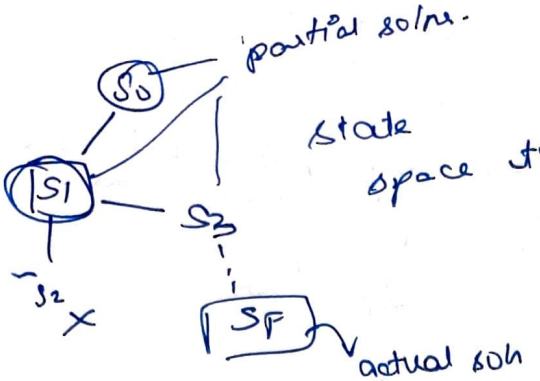
$O(V^3)$

Backtracking

Idea: construct solns one component at a time & evaluate such partially constructed candidates as follows:

1. If it can be developed further without violating problem's constraints, it is done by taking the first remaining legitimate option for the next component
2. If no legitimate option for next component, no alternatives for any remaining component need to be considered.

3. In this case, the alg. backtracks to replace the last component of the partially constructed soln. with its next optn.



State space tree. (root is the initial condn.)

State space tree: tree of choices

→ roots → an initial state.

next level → partial soln

Promising: scope for expanding & moving towards a final soln.

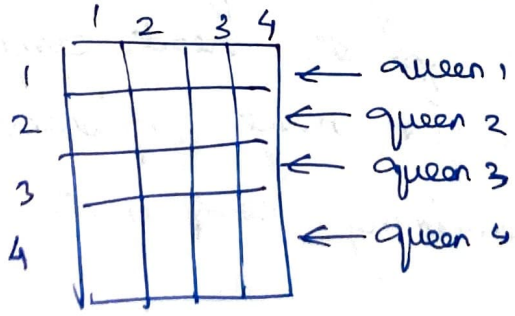
A node in state space tree is promising if it corresponds to a partially constructed soln that may still lead to a complete soln.

Leaves represent either non promising dead ends / complete solns found by alg.

N Queens problem

no two queens can be placed in same row / same column / same diagonal.

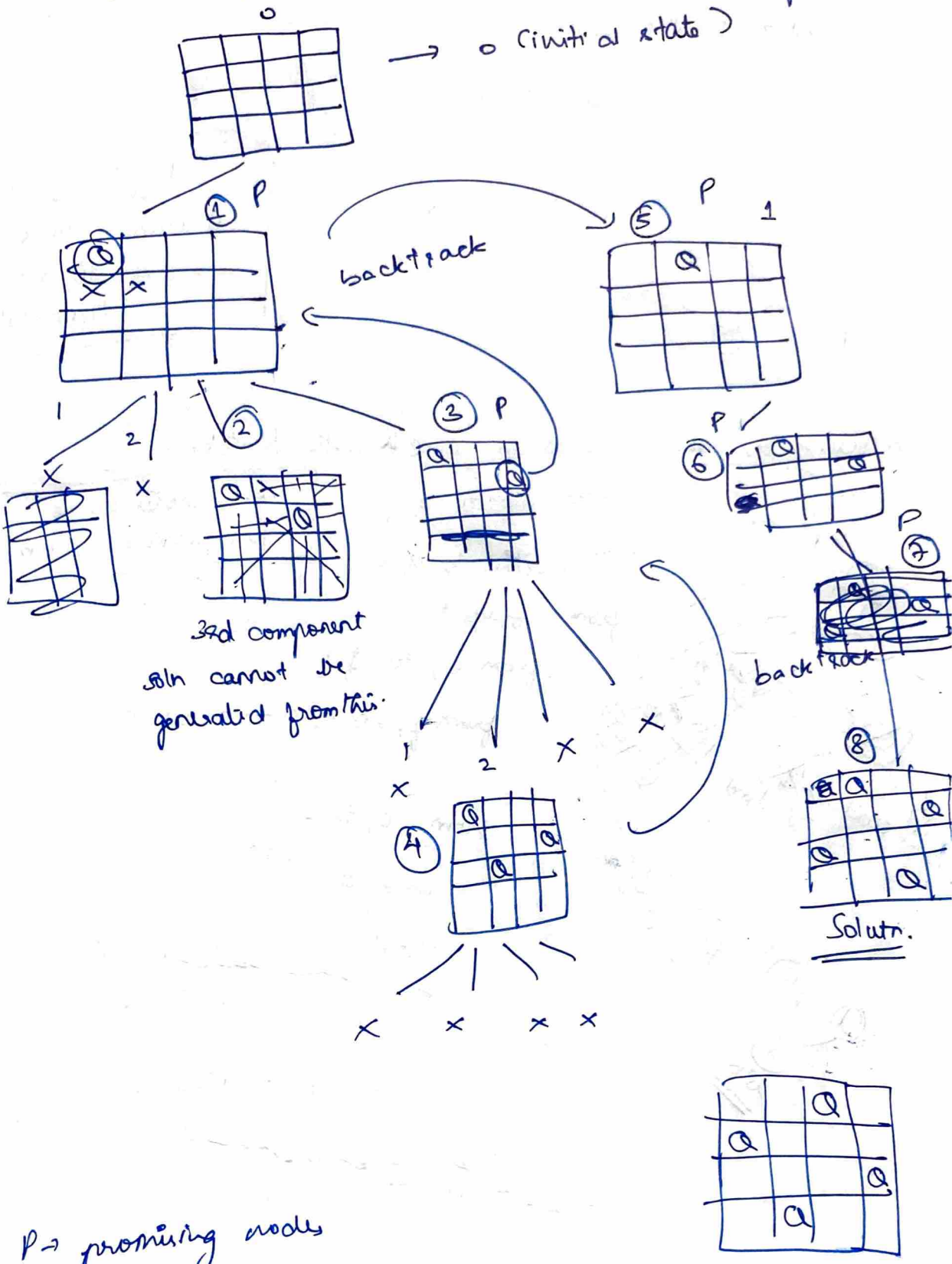
⇒ n×n chessboard, n queens



soln: N=1 soln exists N=2, 3: no soln.

4 queens problem

4 queens - 2 soln
8 queens - 92 soln

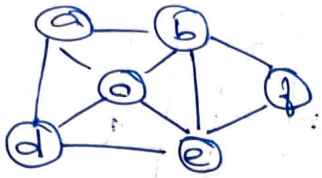


P → promising nodes

Hamiltonian Circuit Problem

* Start from a vertex, visit all other vertices exactly once & return to the same vertex
* Starting vertex visited twice.

diff btw. H & salesman → min. distance traversal



In vertex A, take a decision whether path to take.

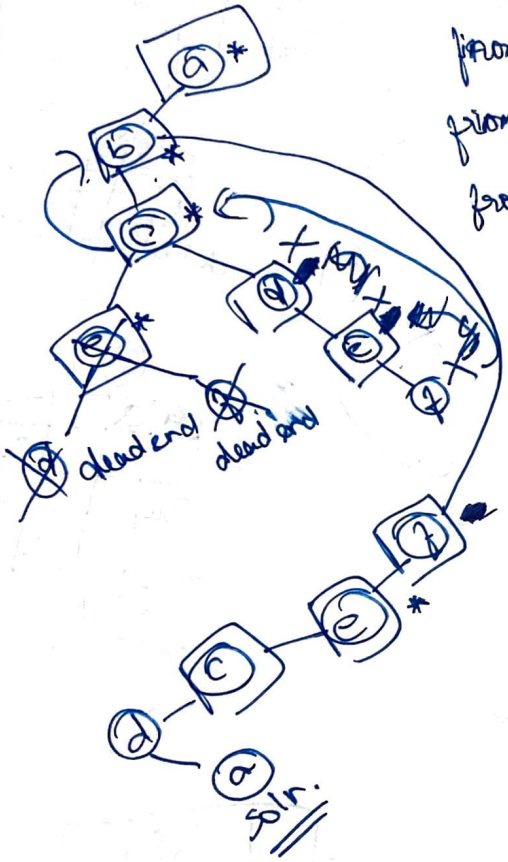
from b, either to c/f.
from c, either to d/e
from d, only e.
from e, to f.
from f, no optn → dead end

from c, to e
from e to d/f.

dead ends → non promising
others are promising

D → promising.

* → denotes where you wanna return



Branch & Bound

similarity: state space tree.

diff: objective is maximized in B & B (optimizatn) but not in backtracking
↳ not for (optimizatn)

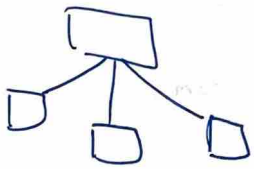
Optimizatn problem:
↳ seeks to minimize/maximize some objective fn., subject to some constraints.

- Feasible soln.: a point that satisfies problem's constraints

Optimal: a feasible soln with best value of objective fn. (Hamiltonian) (Shortest Hamiltonian circuit)

Backtracking → non optimizatn

Branch & Bound → optimizatn.
↓
Best first order
1. all possible soln. in 1st level
2. choose best possible from this then proceed further.

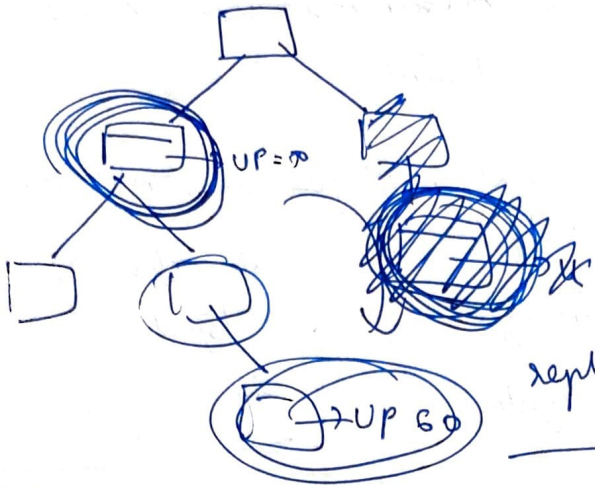


- A way to provide for every state space tree a bound on objective fn on any soln that can be obtained by adding further components to partially constructed soln rep. by node
- The value of best soln. so far.
- Best first order

Terminatn condn:

- value of node's bound is not better than value of best soln seen so far. (considered) (30) (20)
- node represents no feasible soln because the constraint of problem are already violated

3. subset of feasible solns. rep. by the node consists of a single point (& hence no further choices can be made) - in this case, we compare the value of objective fun. for the feasible soln. with that of the best soln. seen so far & update the latter with the former if the new soln. is better.



replace this with that.

Knapsack problem.

- Order items in desc. order by their value to wt ratios.
- The first item gives best payoff per wt unit & last one gives the worst payoff per weight unit, with ties resolved arbitrarily.

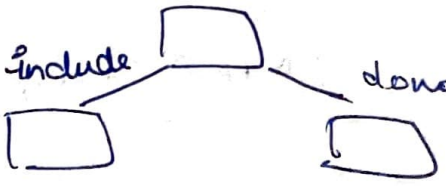
$$v_1/w_1 \geq v_2/w_2 \geq \ldots v_n/w_n$$

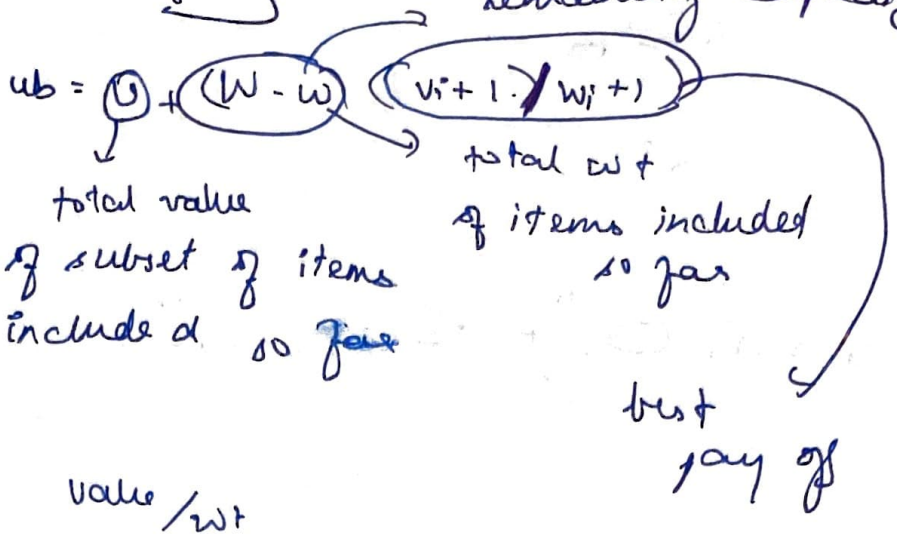
- $ub = v + (w - w_i)(v_{i+1}/w_{i+1})$

v: total value of items.

w: total weight of items.

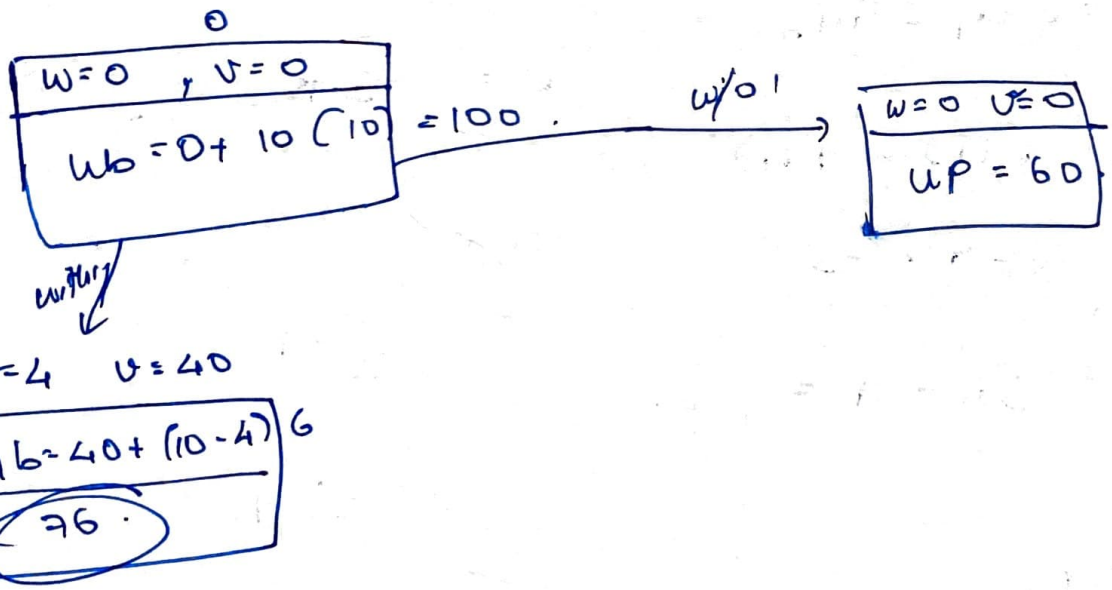
State space tree for B&B for knapsack problem





item	wt	value	value/wt
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

knapsack's capacity is (10)

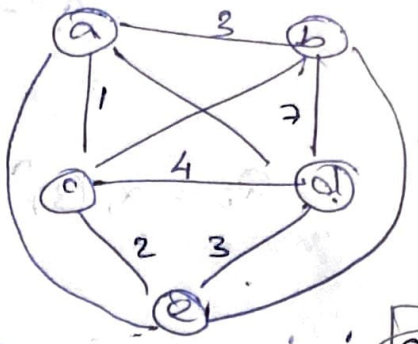


lb = (3+8) + (3+6) + (6+4) + (3+4) + (8+3)/2 = 34.

(8+5) + (3+6) + (6+2 + (5+3)

(3+2)/2 = 19

8



lb
abc
abcd
24

abce
19

lb
abd
abdce
abdeca

(3+8) + (7+3)
(4+7) (2+4)
(8+2) = 24

(7+3)
(3+7)
(2+3)
(1+2)
16

1. An alg. solves a problem in polynomial time if its worst case time eg. belong to O(p(n)) where p(n) is a polynomial of problem's i/p size n. eg: n, log n.

2. Problems that can be solved in polynomial time are called tractable & problems that cannot be solved in polynomial time)

intractable.

t(n) ≤ p(n)
O(p(n))

P class:

Informally, we can think about problems that can be solved in polynomial time as the set that C.S theoreticians call P

All decision problems are in P

class P → Class of decision problems that can be solved in polynomial time by deterministic alg.

This class of problems → polynomial

Decidable & undecidable Problem.

NP problems.

* Hamiltonian circuit problem: Decision counterpart
* Traveling salesman problem: Decision counterpart
* Knapsack problem: Decision counterpart.
* Partition problem: Given n +ve int, determine whether it is possible to partition them to two distinct subsets with same sum

Euler circuit can be solved in polynomial time $O(n^2)$

start vertex, visit every edge only once & return back to same vertex.

$O(n^2)$ ← matrix of $n \times n$

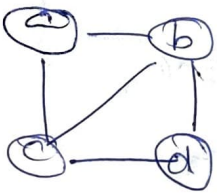
Class NP

A nondeterministic alg is 2-stage procedure that takes as its i/p an instance I of a decision problem Π does:

1. Nondeterministic ("guessing") stage: An arbitrary string S is generated that can be thought of as a candidate soln. to the given instance.

2. Deterministic ("verification") stage: A deterministic alg. takes both I and S as its i/p & o/p yes iff S represent a soln. to instance I.

Hamiltonian circuit.



1st stage: arbitrary string

1. abcda

2. No.

Randomized alg: Deterministic

- A non deterministic alg. solves a decision problem iff every yes instance of the problem if returns yes on execution.

— able to "guess" soln at least once & be able to verify validity

→ A Non Det. alg is said to be non det. polynomial if time eff. of verification stage is polynomial

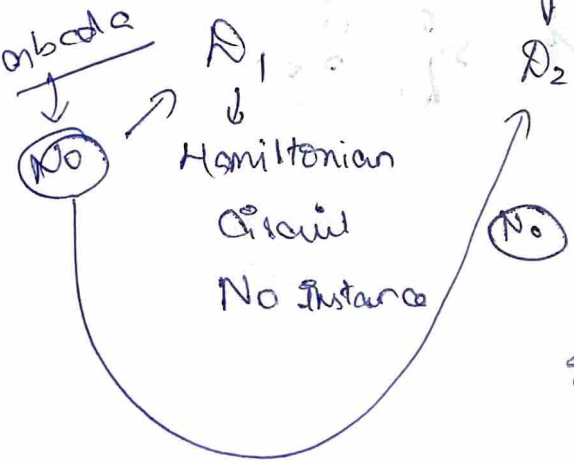
Class NP

→ Class of decision problems that can be solved by Nondeterministic polynomial alg. This class of problems → nondeterministic polynomial

$P \subseteq NP$ - Decision problems.

D_1 (Decision problem) is said to be polynomially reducible to a decision problem D_2, if there exists a fun. t that transforms instances of D_1, to instances of D_2 such that:

- t maps all yes instances of D_1 to yes instances of D_2 & all no instances of D_1 to no instances of D_2
- t is computable by a polynomial time alg.



If for D_1 we get no, D_2 also should give D_2.
If D_1 gives yes, D_2 should yes.

NP Complete

- A decision problem D is NP complete if
  - it belongs to class NP
  - every problem in NP is polynomially reducible to D.

CNF satisfiability Problem
Conjunctive Normal Form

$(a \vee b \vee c) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c)$

$a = T \quad b = F \quad c = T$

$T \wedge T \wedge (F)$ → Change to true

(F) → T.

Check whether truth values to vars. such that the final expression is true.