7/03/18

DAA by Sudheer Sir

Dynamic Programming (Algorithmic paradice)

→ Use for such programs whose subproblems are not independent

→ Also called Dynamic optimization

→ Divide → Solve → Save the solution and whenever in future you require to solve that problem again then take the past solution and copy.

4 Steps performed for applying dynamic programming

1. find the substructure of a problem
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal sol^n in a bottom-up fashion
4. Compute an optimal solution.

Eg → Assembly line Scheduling

$a_{1,1}$ → shop 1 in assembly line 1

$a_{2,1}$ → shop 1 in assembly line 2

① → Switch b/w assembly lines, ~~reducing~~ such that our task takes minimal time.

② → Be in a sigal assembly line for its whole task.

What should be the rule for switching b/w the assembly line?

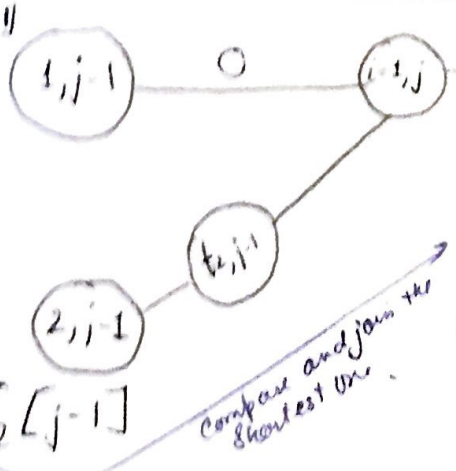
Total possible paths are 2^n → n shops, 2 → assembly lines

→ Entry times e_1 and e_2 and exit times x_1 and x_2.

→ $S_{1,j}$ and $S_{2,j}$ perform same fns but can take different amt of time $a_{1,j}$ and $a_{2,j}$

→ Stay on same line, at no cost of

Eg. $f_1[j]$



$$f_2[j] = \min\left(f_2[j-1] + 0 + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}\right) \quad \text{--- (1)}$$

($a_{2,j}$: time taken to perform that task in j^{th} shop)

→ First we will have a loop and loop will run from 2 to n. I.e. if we have n shops

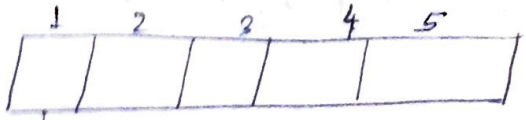
→ We will require base values $f_1[1]$ and $f_2[1]$

$f_1[1] = e_1 + a_{1,1}$
$f_2[1] = e_2 + a_{2,1}$
} initially we calculate this and then move iteratively.

→ loop recursive will start from $j=2$ to n.

In this loop we require to calculate $f_1[j]$ and $f_2[j]$

→ A =



1^{st} shop in which assembly line

→ for ($j=2$; $j <= n$; $j++$)

→ { If $f_1[j]$ is smaller than $f_2[j]$
mention 1; → $A[j] = 1$
else
mention 2; → $A[j] = 2$

}

→ Implement this assembly line theorem to find the optimal path to complete the task.

Put the values in eq^n (1) and then take min of $f_1[2]$ or $f_2[2]$

eg → $f_1[2] = \min(f_1[1] + 0 + a_{1,2},\ f_2[1] + t_{2,1} + a_{1,2})$

$f_2[2] = \min(f_2[1] + 0 + a_{2,2},\ f_1[1] + t_{1,1} + a_{2,2})$

9/03/18

① fastest-way (t , a , n, e , x)

starting time → e, ending time → x

t : transition time.

a : time required to complete a job in particular step

$f_1[1] \leftarrow e_1 + a_{1,1}$ O(1)

$f_2[1] \leftarrow e_2 + a_{2,1}$ O(1)

for $j \leftarrow 2$ to n

{ if $f_1[j-1] \leq f_2[j-1] + t_{2,j-1}$

$f_1[j] \leftarrow f_1[j-1] + a_{1,j}$

$l_1[j] \leftarrow 1$

else

$f_1[j] \leftarrow f_2[j-1] + t_{2,j-1} + a_{1,j}$

$l_1[j] \leftarrow 2$

} 3O(1)

{ if $f_2[j-1] \leq f_1[j-1] + t_{1,j-1}$

$f_2[j] \leftarrow f_2[j-1] + a_{2,j}$

$l_2[j] \leftarrow 2$

else

$f_2[j] \leftarrow f_1[j-1] + t_{1,j-1} + a_{2,j}$

$l_2[j] \leftarrow 1$

} 3O(1)

end

3O(1) {

if $f_1[n] + x_1 \leq f_2[n] + x_2$ → O(1)

$f^* \leftarrow f_1[n] + x_1$ → O(1)

$l^* \leftarrow 1$ → O(1)

else

$f^* \leftarrow f_2[n] + x_2$

$l^* \leftarrow 2$

$l_1[2]$ ← Carries information of in which assembly line, previous step will it belong, for the optimal path.

② → Pro

Eg

Solu

Comple

Asym

Actu

②

→ Print (L, n) (Prints the assembly line and, in decreasing order of stops)

O(1) $i \leftarrow L^*$ ← print 'line' i 'station' n O(1)

for $j \leftarrow n$ to 2

$i \leftarrow L_i[j]$ O(1)

Print 'line' i 'station' j-1 O(1)

Eg → We have this inf.

2	3	4	5	6
1	1	1	2	1
1	1	2	2	1

→ This info we get from our Previous algo.

$L^* \leftarrow 1$

3O(1)

So i) then $i \leftarrow 1$

line 1 station 6
line 1 station 5
line 2 station 4

Complexity:-

Asymptotically Complexity → O(n) → when we have large n, but here we write not have the condition of Loarge n.

3O(1)

Actual complexity →

For algo 1 Total complexity → $T_n = 6O(n) + 5O(1)$ (n of loop)

For algo 2 Total complexity → $T_n' = 2O(n) + 2O(1)$

Total → $8O(n) + 7O(1)$

space complexity of $f_i[n]$ → 2O(n) (f variable)

" " $L_i[n]$ → 2O(n)

↳ L variable

space complexity of other variable → 3O(1)

Total → $4O(n) + \underbrace{3O(1)}_{j, f^*, L^*} + \underbrace{2O(1)}_{i, j}$

→ $4O(n) + 5O(1)$

] what ever is not given in the Passed parameter, count those variables in space complexity.

If we remove the L part (Just fill the first end (1-) is kept and rest is removed.

$4O(n) + 2O(1)$

Other way -

OPath (f_1, t, n, x)

for $i \leftarrow 2$ to n

$3\Theta(1)$

- if $f_1[i] \leq f_2[i]$
  - if $f_1[i-1] \leq f_2[i-1] + t_{2,i-1}$
    - Print 'line' 1 Station $(i-1)$
  - else
    - Print line 2 station $(i-1)$
- else if $(f_1(i-1) + t_{1,i-1} \leq f_2(i-1))$
  - Print line 1 station $(i-1)$
- else
  - Print line 2 station $(i-1)$

$T_n^* = 40(n) + 20(1)$

$T_{n_2} = 30(n) + 20(1)$

$\text{Total} = 70(n) + 40(1)$

If $f_1(n) + x_1 < f_2(n) + x_2$

$L(n) = 1$ print line 1 station n

else

$L(n) = 2$

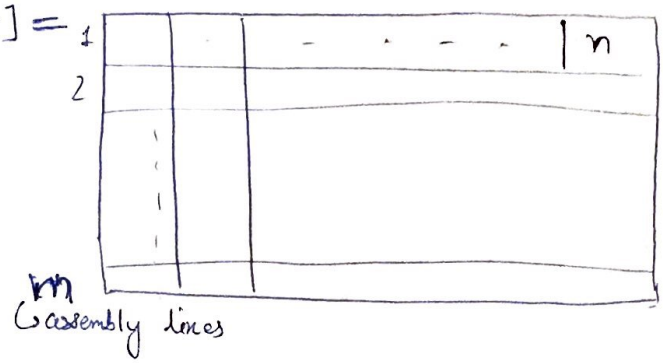
print line 2 station n

12/03/18

for m assembly lines

$f_i[j]$ → The optimal cost to reach j^{th} shop in i^{th} assembly line.

$f_i[j] =$




```
for j ← 2 to n
    for i ← 1 to m
        f_i[j] = Min_oc(f, t, j, i, m) + a_ij
```

```
Min_oc (f, t, j, i, m)
    for k ← 1 to m                          ⟶ if (k != i)
        M[k] ← f_k[j-1] + t_{k,j-1}
    M[i] ← f_i[j-1]
    oc ← min (M)
```

as, if the assembly line is same then transition cost is not considered.

↳ is a one-D array, and we have to just find a min. out of this.

$O(nm^2)$

Correctness Proof :-

Look up that our algorithm gives result, as we want or not.

① Proof by induction

when every suboptimal solution is a part of our optimal solution.

② Proof by Contradiction.

We choose some alternate Sol^n other than optimal Sol^n, and then Prove that the choosen Sol^n is not optimal.

If our algo, doesn't accepts the alternate Sol^n, then it means that our algo. known which Sol^n is a optimal Sol^n.

Eg →

	2	3	4	5
	1	2	1	1
	1	2	1	2

$f_1[j]/l_1[j]$	1	2	3	4	5
	9	18	20	20	32
$f_2[j]/l_2[j]$	12	16	22	25	30

	1	2	3	4	5
	1	2	1	1	1

→ Assembly line in which the chips are

$j=4$ $j=5$

If we purposely change the above array

	1	2	1	2	1

← go from back.

By algo. we see that

→ your algorithm is not accepting our above alternate solution.

It contradicts the soln which we have chosen.

Correctness Prove

Step-1 → Check the suboptimal soln is a part of soln or not

Step-2 → Use contradiction or induction

Step-3 → If use contradiction, then choose an alternative soln.

Step-4 → Wheather our algo. accept the alternative soln or not.

Matrix-Chain Multiplication Problem :→

first check the dimensions.

A_1, A_2, A_3

$P_0 * P_1$, $P_1 * P_2$, $P_2 * P_3$

P_0 ↙ dimensions

$((A_1, A_2)A_3) = P_0P_1P_2 + P_0P_2P_3$

$= P_0P_2(P_1+P_3)$

$(A_1 (A_2 A_3)) = P_0P_1P_3 + P_1P_2P_3$

~~= 15(30)~~

A_i & A_{i+1}

$A'_{P_0 * P_2} = A_1A_2 = P_0P_1P_2$ (As we have three loops in matrix multi.)

Cost of $(A_1A_2)A_3 = P_0 \times P_2 \times P_3$

Let there be j matrices

$A_i \; A_{i+1} \; \cdots \; A_j$

$A_{i \cdots j} = A_{i \cdots k} * A_{k+1 \cdots j}$

Order of every matrix → $A_i = P_{(i-1)} * P_{(i)}$

	5
0	32
5	30

$A_{i---j} = A_{i---k} * A_{k+1---j} + C$

$P_{(i-1)} * P_{(k)} \qquad P_{(k)} * P_{(j)}$

$$m[i,j] = \min_{i \le k < j} \left(m[i,k] + m[k+1, j] + P_{i-1} P_k P_j \right)$$

Optimal cost of matrix-chain Product

for optimal $i \le k < j$ **

A subproblem can also be a single matrix.

Eg :→ What is the optimal cost of product where we have $m(1,1) = 0$ as when $i = j = 1$ then this is only 1 matrix, hence its cost of product = 0.

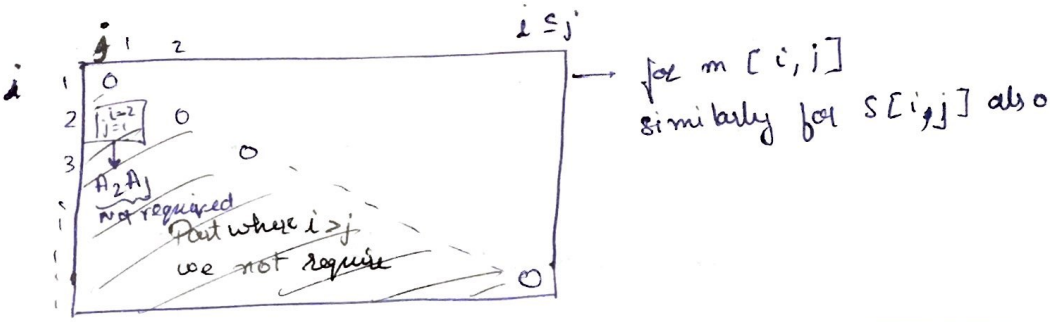
if $i < j$ ⟶ then $m[i,j] = \min_{i \le k < j} (m[i,k] + m[k+1, j] + P_{i-1} P_k P_j)$

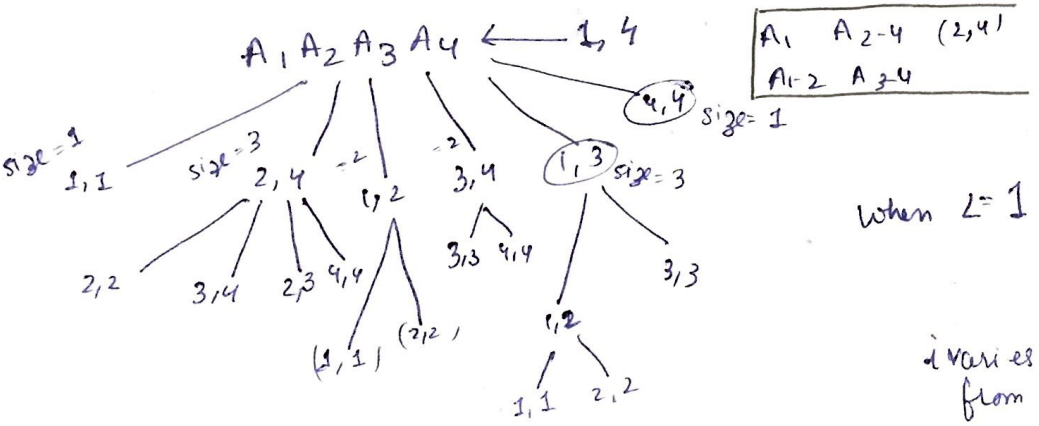
if $i = j$ ⟶ then 0

→ In Algorithm

$S[i,j]$ → Carries the information from where we break our problem into subproblem.

↳ Carries that value of k, for which corresponding $m[i,j]$ is minimum.





A_1	A_{2-4} (2,4)
A_{1-2}	A_{3-4}

when $i = 1$ → (1,1) (2,2) (3,3) (4,4)

i varies from 1 to 4

$n = 4$ (Size of original Problem)

$L = 2 \Rightarrow (3,4), (2,3), (1,2)$

$L = 3 \Rightarrow (2,4), (1,3)$

As the problem size ↑, the values of i of these subproblems decreases.

Loop over i :→

$i = 1$ to $n - L + 1$

Loop over j :→

when $L:1 \rightarrow j = i+0$
$L:2 \rightarrow j = i+1$
$L:3 \rightarrow j = i+2$

} Thus $j = i + L - 1$

From induction, we can say ~~L:1 to ...~~

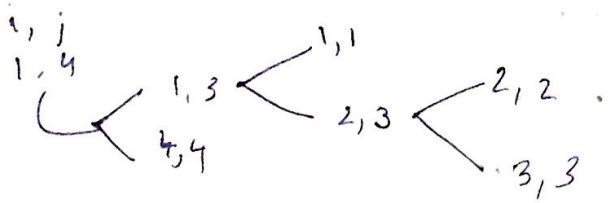
for $k \leftarrow i$ to $j-1$ → from where we have to break and make a subproblem.

Construct the optimal solution.

→ S ~~array~~ array, carries the optimal break, where we placed it.

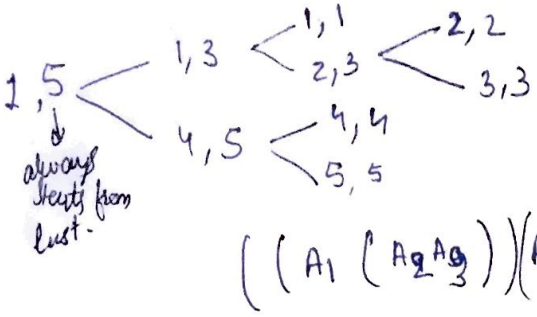
```
Print - OPT - Parens (S, i, j)
  if i = j
     then print "A";
     else print "A";
        else print "("
                Print - OPT Parens (S, i, S[i,j])
                Print - OPT Parens (S, S[i,j]+1, j)
                Print ")".
```

→ first time we enter in else part.



$((A_1 (A_2 A_3)) A_4)$

Q: Optimal Paranthesis for $A_1 A_2 A_3 A_4 A_5$



$((A_1 (A_2 A_3))(A_4 A_5))$

i

j	1	2	3	4	5	6
6	3	3	3	5	5	
5	3	3	3	4		
4	3	3	3	-		
3	1	2	-			
2	1	-				
1						

Complexity

$l \leftarrow 2$ to n → $O(n)$

→ for $k \leftarrow i$ to $j-1$ $O(n)$

$i+l-1$

$l-1$

~~$l=2$~~ ~~$i=1$ to $n-1$~~ → $2(n-1)$ + constant

→ $L : 2$ → $L(n-l+1(2+(L-1)(3)))$

→ $L((n-L+1)(2+3(L-1)))$

$\approx L^3$

and order of L is n.

→ logn complexity of Construct optimal BST algo.

$T_n \Rightarrow 2T(n/2) + C$

Space complexity

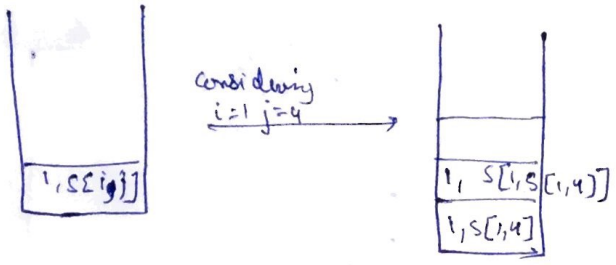
Size 1 → l, K, n, ...

Single $\left(\frac{1}{2}n^2\right)$ → m, s

↓ Eg only the upper array is required.

→ Space complexity for recursive algorithm :→

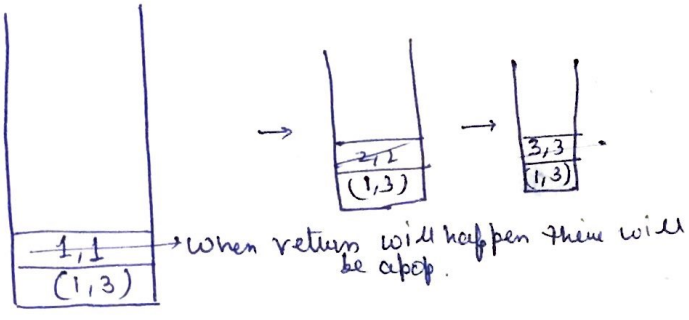
When we recursively call, in the memory, a stack is maintained, as we require some space, where when we return, we know where should he go.



Eg:→ We call the construct optimal solution algorithm for 1, 6

Calling - push

return - pop



$A_1 | A_2 | A_3 | A_4 | A_5 | A_6$

if divide from here, less complexity

k can be anywhere

Maximum size of stack when k is at $\frac{1}{2}$.

↳ If we divide the problem from $\frac{1}{2}$ part it will lead to $\log(n)$

Correctness Proof

Proof by contradiction:→

If any alternate solution is not accepted by our algorithm, then we say that our algorithm is correct.

$A_1 A_2 A_3$

$(A_1 (A_2 A_3)) \rightarrow$ [illegible] $= (P_1 P_2 P_3) + (P_0 P_1 P_3)$
4500

$((A_1 A_2) A_3) \rightarrow 1750 = (P_0 P_1 P_2) + (P_0 P_2 P_3)$

$P_0 \times P_1 \quad P_1 \times P_2 \quad P_2 \times P_3$
$10 \times 15 \quad 15 \times 5 \quad 5 \times 20$

Now apply the algorithm:→

$n = 3$

$L \rightarrow 2$ to 3

for $L \rightarrow 2$

$i \rightarrow 1$ to 2

for $i=1$
$j=2$
$S[1,2]$ •

for $i=2$
$j=3$
$S[2,3]$

$L=3$
$i \leftarrow 1$ to 1
$j=3$
$S[1,3]$ ✓

$k \leftarrow 1$ to 2

for $k=1$ $q \leftarrow m[1,1] + m[2,3] + P_0 P_1 P_3$

$m[1,1]$ → Cost → 0
$m[2,3]$ → Cost → $P_1 P_2 P_3$ → 1500
$P_0 P_1 P_3$ → 3000

for $k=2$ $q \leftarrow m[1,2] + m[3,3] + P_0 P_2 P_3$

$m[1,2]$ → 750
$m[3,3]$ → 0
$P_0 P_2 P_3$ → 1000

So our algorithm say 2 should be stored at $S[1,3] \leftarrow k=2$

Thus our alternate solution say $k=1$ should be stored at $S[1,3]$, so thus our algorithm accepts just the optimal solution.

Recursive version of the above algorithm

Recursive_MatrixChain(P, i, j) (i = 1, j = n)
↳ (carries the dimension of those matrices)

```
if i=j
    return 0
m[i,j] ← ∞
for k ← i to j-1
    q ← Recursive_Matrix chain (P, i, k)
        + Recursive_Matrix chain (P, k+1, j)
        + P_{i-1} P_k P_j

if q < m[i,j]
    m[i,j] ← q

return m[i,j]
```

Complexity → $T(n) \geq \sum_{k=1}^{n-1} T(k) + T(n-k)$

$T(n)$ → length of the matrix chain

$$T(n) \geq 2 \sum_{k=1}^{n-1} T(k)$$

(asymptotically we can say)

$T(1) \geq 1 = 2^0$

$T(2) \geq 2\,O(1)$

$T(3) \geq 2^2 O(1)$

$T(4) \geq 2^3 O(1)$

$T(n) \geq 2^{n-1}$

→ The first algorithm was better.

→ What is the issue with this algorithm, that we got a exponential complexity.
Coz the defination of dynamic programming is, divide a problem into subproblems and then store their solution for use in future, but we are not doing so.

```
Memorized MatrixChain(P)
    n ← length(P) − 1
    for i ← 1 to n
        for j ← i to n
            m[i,j] ← ∞
    return lookup_chain
        (P, 1, n)

Lookup Chain
    if m[i,j] < ∞
    then return m[i,j]
    if i=j
    then m[i,j] ← 0 ← else.
        for k ← i to j−1
            q ← lookup chain(P, i, k)
                + lookup chain(P, k+1, j)
                + P_{i-1} P_k P_j
            if q < m[i,j]
                m[i,j] ← q
            return m[i,j]
```

10/03/18

Longest Common Sequence (LCS)

S_1: A C T G C A T G C C A (T) → x_i (symbol in S_1 at i^{th} position)

S_2: A C C G G T T C A G C (T) → y_j (symbol in S_2 at j^{th} element)

→ LCS which is common in both the sequences.

→ Used in DNA testing

Find the length of LCS.

Lets say S_1 is of some size i

Lets say S_2 is of some size j

first check whether the last symbol is same or not. if not then it means that one of the 2 last symbol will not be in optimal solution for $x_i = y_j$

① $LCS(S_1(i), S_2(j)) \Rightarrow LCS(S_1(i-1), S_2(j-1)) + 1$

↳ this will find one common symbol T at the end

② When the last symbol is not same :→

Our solution is the one which has larger value.

We will divide in subproblems and the subproblem having higher value will be of our use.

S_1 : A C T G C A T G C C A (T)

S_2 : A C C G G T T C A G C (A)

$$\max\left(LCS(S_1(i-1), S_2(j)), LCS(S_1(i), S_2(j-1))\right)$$

$$m[i,j] = \begin{cases} m[i-1, j-1] + 1 & x_i = y_j \\ \max\left[m[i-1,j], m[i,j-1]\right] & x_i \neq y_j \end{cases} \quad n \geq i > 0,\ m \geq j > 0$$

(i → seq 1 of length i; j → seq 2 of length j)

Base Case :→ for $i \leftarrow 0$ to n : $m[i,0] \leftarrow 0$ $\quad 0 \leq i \leq n$

for $j \leftarrow 1$ to m : $m[0,j] \leftarrow 0$ $\quad 1 \leq j \leq m$

↳ as $m[0,0]$ is already calculated in above

Code :→

Iterative algo → Bottom to top

```
for i ← 1 to n
    for j ← 1 to m
        if x_i == y_j
            m[i,j] = m[i-1, j-1] + 1
                                  → b[i,j] ← "↖"
        else
```

~~$m[i,j] = \max[m[i-1,j], m[i,j-1]]$~~

else if $m[i-1,j] \geq m[i,j-1]$

then $m[i,j] \leftarrow m[i-1,j]$ $\quad b[i,j] \leftarrow$ "↑"

else

$m[i,j] \leftarrow m[i,j-1]$

$b[i,j] \leftarrow$ "←"

eg: → $X = \{A, B, C, B, D, A, B\}$

$Y = \{B, D, C, A, B, A\}$

		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	1 ↖	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	4

→ i must be decremented

Result = B C B A

$m \rightarrow O(n)$

$n \rightarrow O(n)$

complexity = $O(n^2)$

→ Write the printing algorithm.

```
Print-LCS (b, X, i, j)
  if i = 0 or j = 0
     then return
  if b[i, j] = "↖"
     then Print-LCS (b, X, i-1, j-1)  → Print x_i
  else if b[i, j] = "↑"
     then Print-LCS (b, X, i-1, j)
  else Print-LCS (b, X, i, j-1)
```

Procedure takes time $O(m+n)$

23/03/18

0/1 knapsack

We have n items each item have weight and value.
There is a K.S of capacity W.
Maximize the total value of the K.S by putting items in it.
Put the items in such a ways in the knapsack.
The value of total item that we fill in this knapsack is max.

→ When we keep one item in the knapsack

→ remaining capacity = initial − capacity of that particular item
no. of available items = no. of available items − 1

→ This problem can be solved if we have a solution of the subproblem

KSOV(i, W) ← available capacity of the KS
↑ no. of available items

→ KSOV(i−1, W−W(i)) + V(i)
↑ weight of ith item
V(i) ← value that we have kept in the K.S

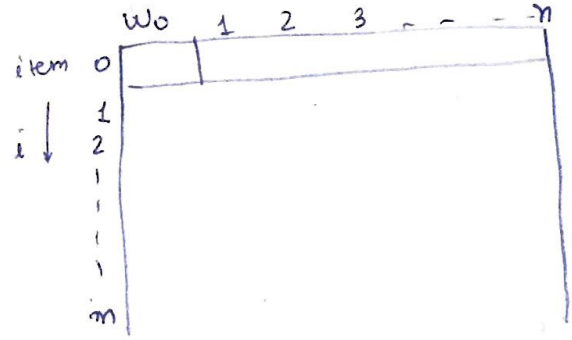
KSOV(i−1, W)
when we not find the item to be suitable to keep in the K.S

Conditions →
1) If the available capacity < weight of that item

KSOV(i−1, W−W(i)) + V(i), KSOV(i−1, W)
V, V'
V''
choose the one which is maximum

Algorithm

m[i, j]
↓ items
↳ capacity



KSOV(i, W)

for i=0 to m
  for j=0 to n
    if w(i) > j
      m[i, j] = m[i−1, j]
    else if m[i−1, j−w(i)] + V(i) ≥ m[i−1, j]
      m[i, j] ← m[i−1, j−w(i)] + V(i)
    else
      m[i, j] ← m[i−1, j]

Eg.

	1	2	3	4
Wt	3	2	4	1
Value	100	20	60	40

$W = 5$ = KS capacity.

Base case → either $i=0$ or $j=0$

$m[0,j] \leftarrow 0$ for $0 \le j \le n$

$m[i,0] \leftarrow 0$ for $1 \le i \le m$

item \ W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	100	100	100
2	0	0	20	100	100	120
3	0	0	20	100	100	120
4	0	40	40	100	140	140

Optimal value which we can achieve is 140. (with items 4 and capacity 5)

① $m[1,3]$
↳ in else part.
$m[0,0] + 100 \ge m[0,3]$
$m[1,3] \leftarrow 100$

② $m[1,4]$
in else part.

③ $m[2,1]$
if part
$m[2,1] = m[1,1]$

④ $m[2,2]$
in else part
$m[2,2] \leftarrow 20$

Algorithm to print the items

```
i = m & w = W
while (i & w > 0)
    if m[i,w] != m[i-1, W]
        print i
        w = w - w(i)
        i = i - 1
```

Fraction KS

Find out value per item

Eg →	1	2	3	4
Wt	3	2	4	1
Value	100	20	60	40
V/wt	100/3	20/2	60/4	40/1
	33.33	10	15	40

→ our decision should be based on value per weight

We use greedy algorithm here.

check out which item has more. V/wt , keep in the K.S.

$W = W - w(i)$

Value = Value + V(i)

28/03/18

Graphs

→ G(V, E)

↓ Set of Vertices it have

↳ set of edges it have.

①

→ A directed graph is one in which each edge is a directed pair of vertices

$\langle v_0, v_1 \rangle \; ! = \langle v_1, v_0 \rangle$

tail → head.

→ In undirected graph any direction is possible.

②

Complete graphs are those graph in which ~~we can form~~ a edge from any edge to other remaining edges exists.

Complete graph → # edges in UDCG

n → no. of vertices.

$(n-1)+(n-2)+(n-3) \; - - - +3+2+1$

$\#edges = \frac{n(n-1)}{2}$

Complete graph → # edges in DCG

$\Rightarrow n(n-1)$

③ Adjacent.

④ Self edge

If we find a loop → it is called a self edge.

→ Any subset of a graph is a subgraph.

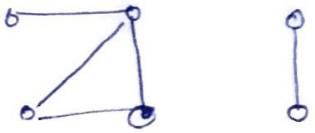
→ path

→ length of path → no. of edges we travel to reach that vertex

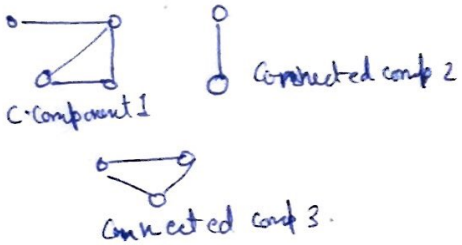
→ Simple path
No vertex in between should be repeated.

Connected graph
All the Vertices can be ~~traced~~ traversed through the set of edges.

Eg :→ Unconnected graph



Eg → Connected Component



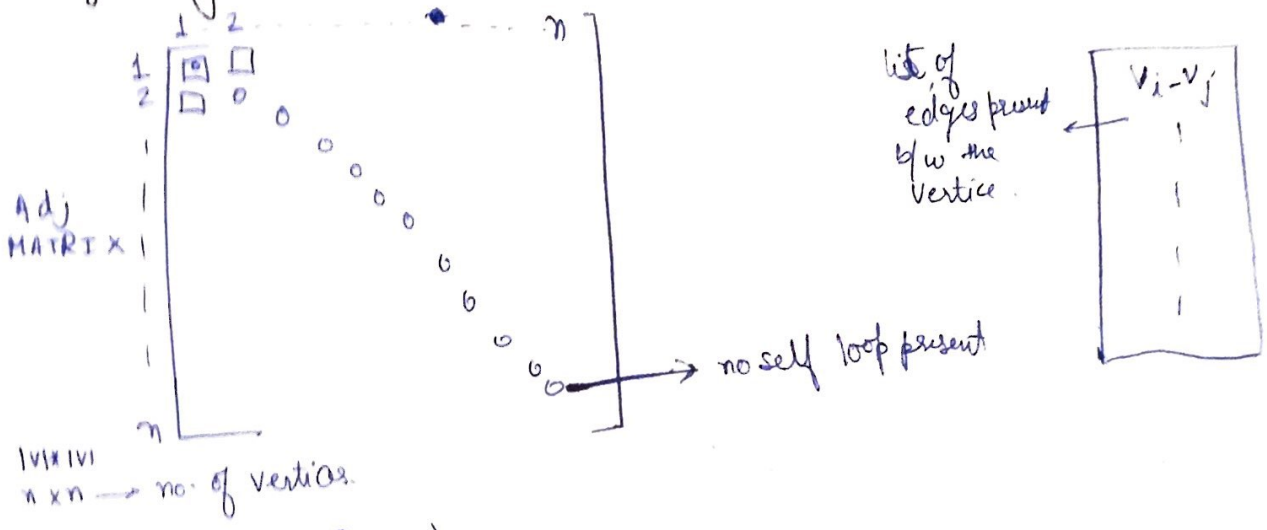
→ In undirected
Degree

→ In directed
① in-degree
② out-degree.

Graph Representations

1) Adjacency Matrix
2) Adjacency Lists
3) Adjacency Multilists

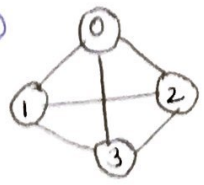
- Adjacency Matrix



Complexity → $O(n^2)$

→ Adjacency list

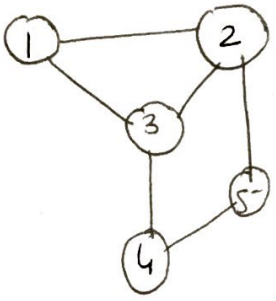
eg①



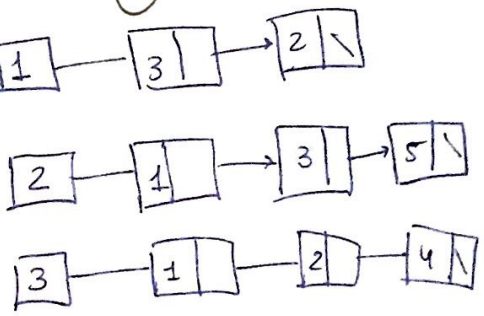
|E| = ⑥
|V| = 5
nodes = ⑫

$O(2|E|) + O(|V|)$
addⁿ cost

& ②



Corresponding to Vertice 1



```
#define Max_Vertices 50
typedef struct node *node_pointer;
typefed struct nodes
        int vertex,
        Struct node * link;
    };
node_pointer graph [MAX_VERTICES];
int n=0;   /* vertices currently in use*/
```

→ Which DS should we use
Adjacency Matrix
Adjacency list ?

① If |V| >> |E| then use Ajacency list

② Complexity of implementation is lower for Adjacency matrix.

③ complexity of iteration is lower for " list.

ADT for graph →

structure graph is

objects: a non empty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all graph ∈ Graph, V, V_1 and V_2 ∈ vertices

Graph Create ()

Graph Insert_Vertex (graph, V)

" Insert_edge (graph, V_1, V_2)

Boolean IsEmpty (graph) → no vertices, no edge.

Adj - MtR - G

Insert Vertice (V)

Adj - MTR - G' = [[ADJ_MATRIX G]]

[0 0 0 0 0 0] ← eg no edges are added

→ update a Vertice → is slightly difficult.

2nd April 2018

Breadth First Search

$d[v]$: Shortest distance from source s to vertex v.

$\pi[v]$: vertex which is predecessor to v if shortest path is followed.

→ for each vertex : we have one variable : color

White : not discovered vertex

Gray : vertex is discovered.

Black : finished.

↓

if all vertices adjacent to it have been discovered.

→ Also require queue.

BFS(G, s)

for each vertex u is $V[G] - \{s\}$

do color[u] ← white
   d[u] ← ∞
} initialization

π[u] ← nil

color[s] ← gray (discovered source)

d[s] ← 0

π[s] ← nil

Q ← ϕ

enqueue(Q, s) ← Movement a vertex is discovered, put it into queue.

while Q ≠ ϕ

do u ← dequeue(Q)

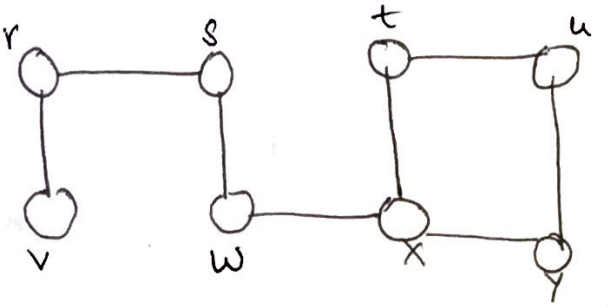
for each v in Adj[u]

do if color[v] = white

then color[v] ← gray

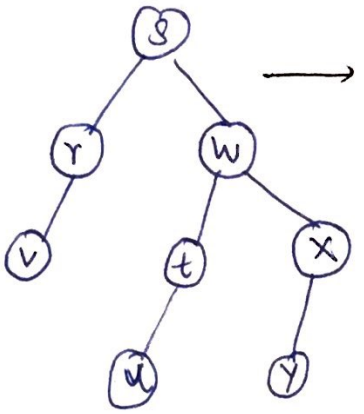
- for each ---
  do ---
    d[v] ← d[u] + 1
    π[v] ← u
    enqueue (Q, v)
  --- color [u] ← black

Ex



Q : s w r x v t y u
0 1 1 2 2 3 3 4

d[value] BFS :



level 0

→ 1 energy is required to pass an edge (let)

level 1

level 2

level 3 — node farthest from source is at last level

→ If give energy = 3δ ∈ ⇒ can reach to each node

(Max energy required = 3δ∈)

Appⁿ of BFS

→ diameter of graph (D)

$\delta(u, v)$ ← shortest path distance from u to v.

$D = \max(\delta(u, v))$

Here, $D = 5$

How to find D?

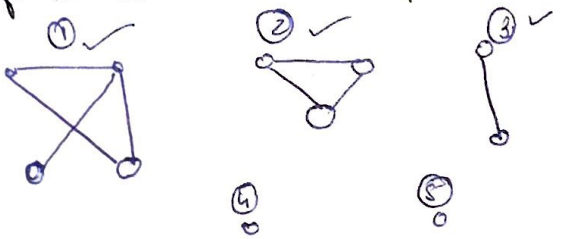
for, s → we can get most distant (at last level)

Do same for each node as source and find max among all.

In last of the code write :

```
return max[d]
for all v ∈ V
    D[v] ← BFS(v, G)
Diameter = max[D]
```

To find connected components



No. of connected components here = 5

Variable color :→ check whether vertex is discovered or not.

Choose any vertex whose color is still white → run on all vertices

can have counter how many time BFS is running.

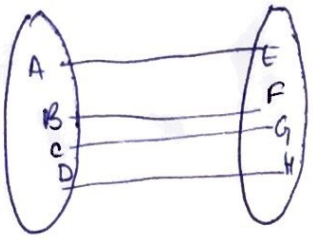
Value of counter = no. of connected components.

Keep on doing till all nodes are black.

logic :→

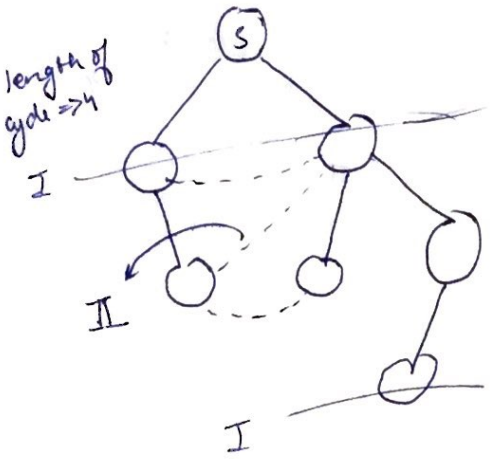
```
for v in V[G]
    if (color[v] == white)
        BFS(v, G)
        count++;
```

⇒ To check whether a graph is bipartite or not



end of edges are divided into two categories ⇩ bipartite

(if edge b/w A & B ⇒ no bipartite)



- if ∃ a edge b/w < vertices at same level ⇒ cycle length = 3 (odd)
- if vertices at diff. level may have cycle len = even

cycle len. odd ⇒ not bipartite

cycle len. even ⇒ bipartite

4th April 18

BFS(G, S) (Only works for connected graphs)

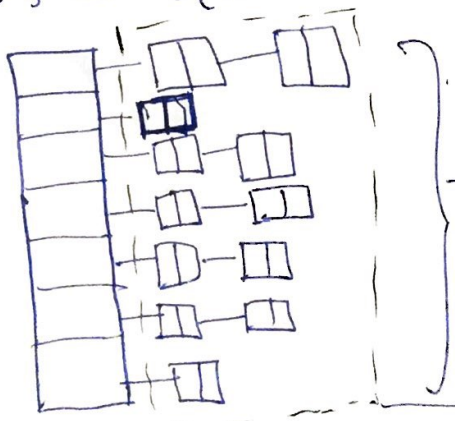
```
1.  for each vertex u in V[G] - {S}
2.      do color[u] ← white
3.         d[u] ← ∞                                  } O(|V|)

4.
 |                                                  ] → O(1)
 |
9.
10. while Q = ϕ
11.       do u ← dequeue(Q)
12.             for each v in Adj[u]
13.                 do if color[v] = white
14.                       then color[v] ← grey
15.                            d[v] ← d[u] + 1
16.                            π[v] ← u
17.                            enqueue(Q, v)

18.             color[u] ← black.
```

1 - 3 → O(|V|)

4 - 9 → O(1)



→ By this adjacency matrix we get the estimation of no. of edges and vertices.

For loop in 12 line we are performing exactly this much operation.

10 - 18 → O(2|E| + 2|V|) → for 11 and 18

→ O(|E| + |V|)

Total complexity → $O(|E|+|V|)$

In tree → min no. of edges in a connected graph → $|V|-1$

So

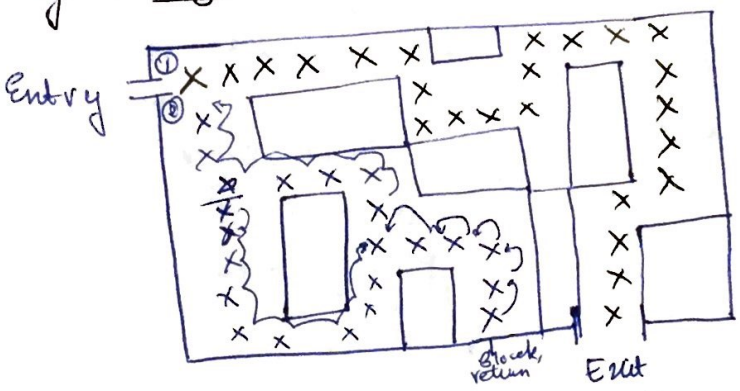
$\Rightarrow |E| \geq |V|-1$

So asymtotically we can say $|V| = O(|E|)$

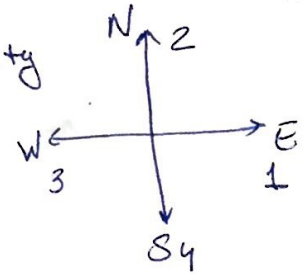
So complexity → $O(|E|+|V|)$
→ $O(|E|+|E|)$
→ $O(|E|)$

Depth first Search (DFS)

Eg → Maze

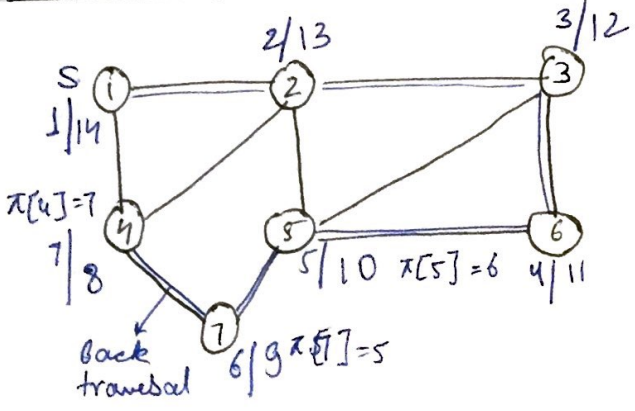


We will set a priority



→ we not repeat the path which we have followed earlier.

Eg → undirected graph



Decide one Source.

① once a vertex is discovered change its colour

② We also need to note the time at which the status of the vertex changed from undiscovered to discovered.

d → discovered time
f → finished time (when all the vertices present in the adjacency list of the vertex is covered)

Diffeunce b/w DFS and BFS

In BFS we are not able to traverse the unconnected part of a unconnected graph. not so in DFS.

→ $\pi[u]$ contains the predecessor.

DFS(G)

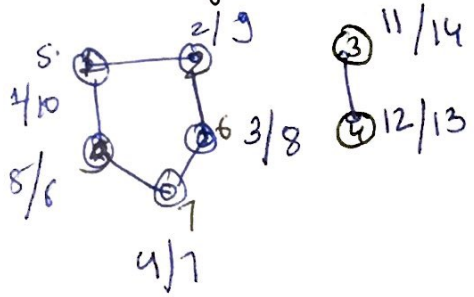
for each vertex $u \in V[G]$
do color[u] ← white
$\pi[u]$ ← nil

time ← 0
for each vertex $u \in V[G]$
do if colour[u] = white
then DFS-visit(u)

DFS-visit(u)

1. Colour[u] ← GRAY ▽ colour of vertex u has been discovered
2. time ← time + 1
3. d[u] = time + 1
4. for each $v \in Adj[u]$ → 2 (|E|)
5. do if Color[v] = white
6. then $\pi[v]$ ← u
7. DFS-visit(v)
8. color[u] ← Black ▽ Blackens u it is finished
9. f[u] ← time ← time + 1

Eg:→ check for a unconnected graph :→

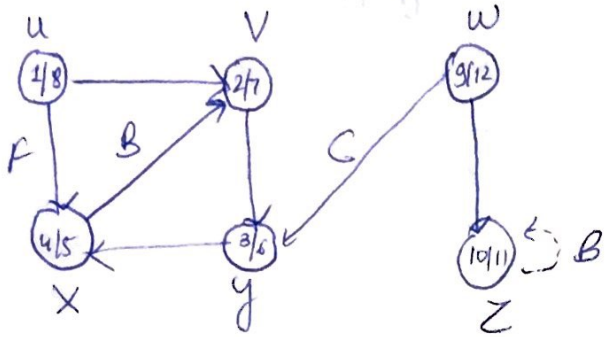


Yes, DFS works

Complexity : → O(|E|+|V|)

6th April 2018

Example (DFS)



With undirected graph we get → only 1 tree

With the directed graph we get → more than 1 tree

Undirected graph, if you run DFS it will return a DFS tree :→

So Edges which are part of DFS Tree are called tree edges

while the other edges are considered as back edges.

directed graph, if you run DFS it will return a DFS forest.

edges which are part of trees, are considered as Tree edges

other edges — forward edge
Back edge
Cross Edge.

Parenthesis Theorem

d(u)/f(u) d(v)/f(v)

u ⟶ v → (directed)

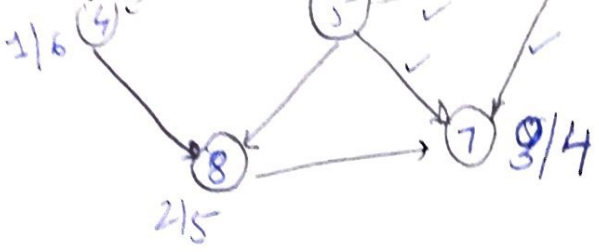
Ancestor / Parent vertex.

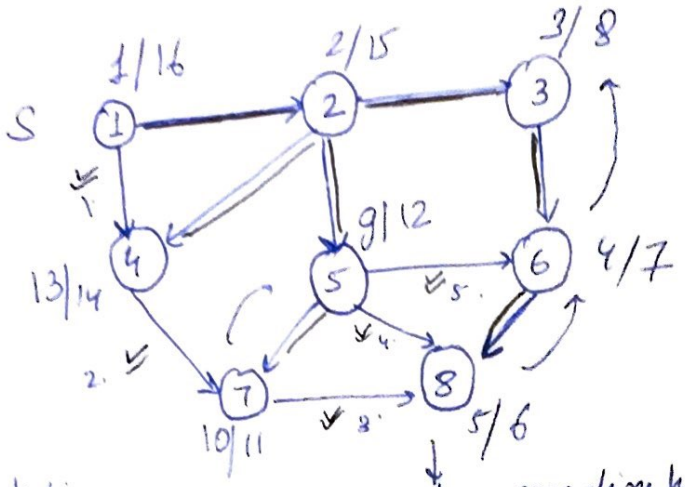
Descendant Vertex

1) If $d(u) < d[v] < f(v) < f(u)$ and v is a descendant of u. (Forward edge)

2) If $d[v] < d[u] < f[u] < f(v)$ and u is a descendant of v. (Back edge)

3) $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither u nor v is a descendant of the other. (Cross edge).





black line are edges and rest both of other edges.

After reaching here, is this an vertex in its adjacency list → No
so back track and mark on the finishing time.

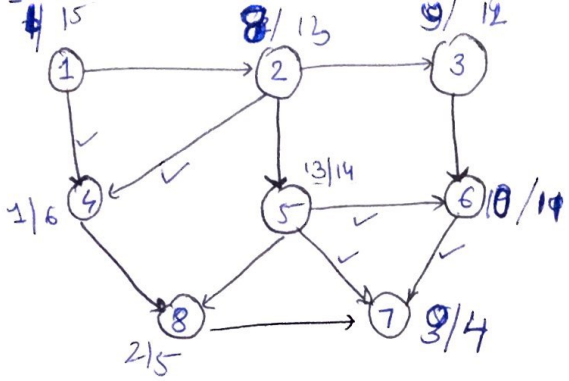
→ Forward edge.
→ Cross edge.
→ Cross edge.
→ "
→ Cross edge.

plication of DFS

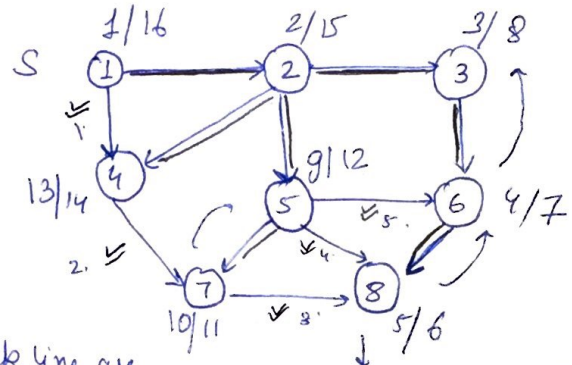
If this is a back edge in a graph while running a DFS, then it means this is a loop.

→ Topological Sorting.

Eg →



Sol":



The black line are tree edges and rest are back of other edges.

1. → Forward edge.
2. → Cross edge.
3. → Cross edge.
4. → "
5. → Cross edge.

After reaching here, is this an vertex in its adjacency list → No
So back track and mark on the finishing time.

Application of DFS

(1) → If this is a back edge in a graph while running a DFS, then it means this is a loop.

(2) → Topological Sorting.

Example

Ancestor

Just put T

(3) → How

9/04/18

Q. To find

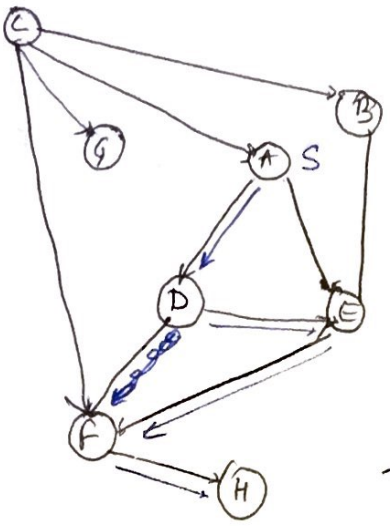
→ If a part of graph

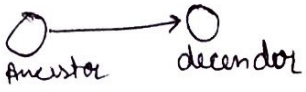
→ If entire strongly

Eg →

Topological Sort (DFS)

Example.





T

H	F	E	D	A	B	G	C

↓ first element completed

Just print it in reverse order

C G B A D E F H

Topological Sort

Just put the statement T ← u b/w 9 & 8.

(3) → How to check how many strongly connected component a graph has.

↓ part of graph which a source that can reach any vertex. Start any vertex we

9/04/18

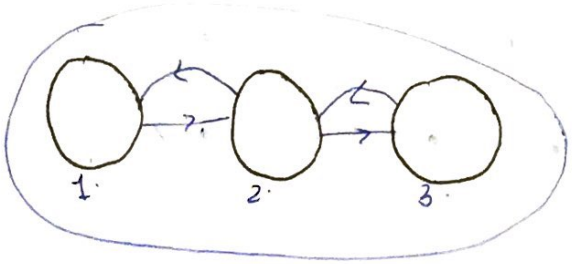
Q. To find No of Strongly connected components of a graph.

→ If a part of a graph is strongly connected → Then we say that, that component of graph is a strongly connected components

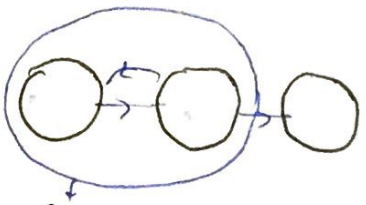
→ If entire graph is a strongly connected then we say that their is a 1 strongly connected component.

Eg →

(1)



(2)



SCC

⇒ Kosaraju Algorithm (Proposed by S Rao Kosaraju in 1978)

Step 1 → Apply DFS
As a vertex change its status to finished, push the vertex into a stack. → $O(n)$ (n: no. of vertices.) $O(|E|+|V|)$

When this above process gets completed then →

Step 2 → Get a transpose of the graph. → $O(|E|)$

↓
(is a graph which have edge dirⁿ opposite to the original)

No. of operation :→ no. of edges.

Step 3 → Run DFS (Running it on transposed graph)

Source → Consider the vertex at the top of stack as the source.

As the vertex are finished, pop from the stack.

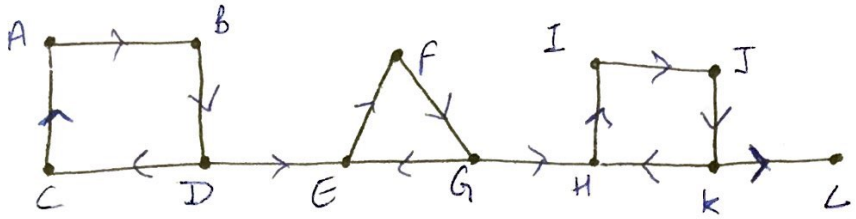
DFS(G)

5 → u = pop(stack)
6 → if u ≠ NULL
7 → the DFS-Visit(u)

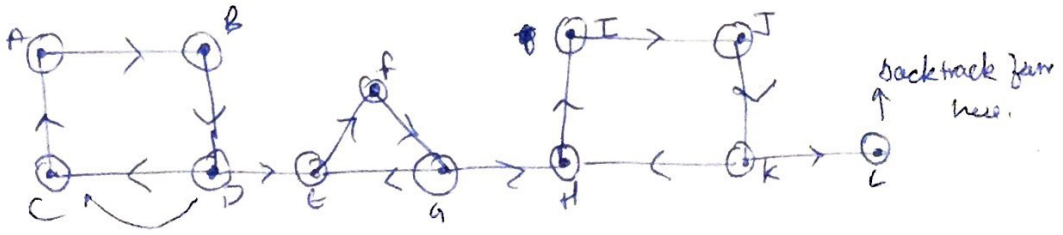
DFS Visit(u)
8. colour[u] ← Black it is finished
9. pop (② time DFS) / push (① time DFS)

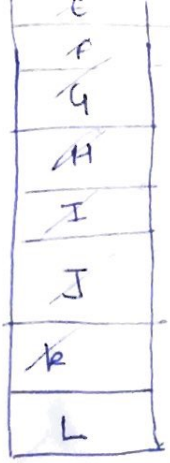
→ As we finished a vertex we have to pop it from the stack.
→ The vertex caused in DFS-Visit that are strongly connected.

Eg:→

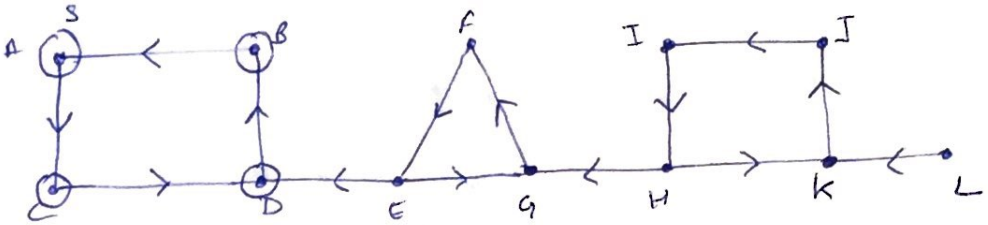


Soln





→ Transpose



Source → pop from the stack => A

1 → B D C A (pop all these vertexes from the stack)
↓ ↳ 1st component that is strongly connected.
Status changes from discovered to finished.

2 → F G E

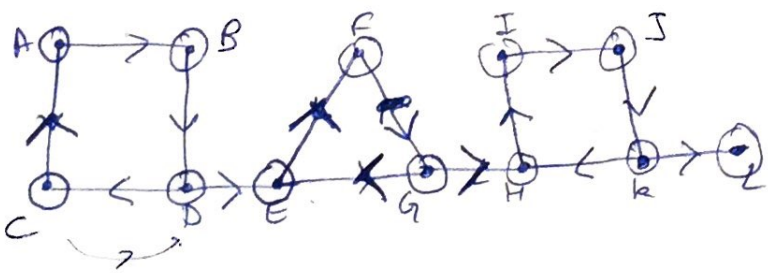
3 → I J K H

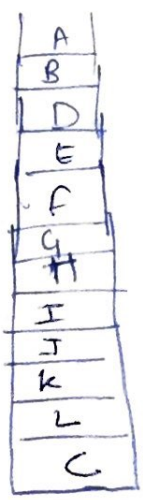
4 → L

5 → NULL → so end.

In this graph we find out 4 strongly connected component

M2





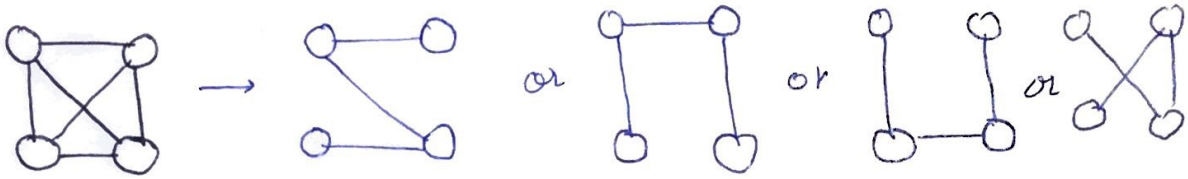
As 'C' in the end of the stack so there will be a problem in deleting, so for deleting here we can use L.L.

pop ——replaced with——> Delete.

11/04/18

Minimum spanning Tree

It is A subgraph which we construct from a graph which consist all the vertices of the graph, and also follow the defination of a tree.
(there should not be any loop)

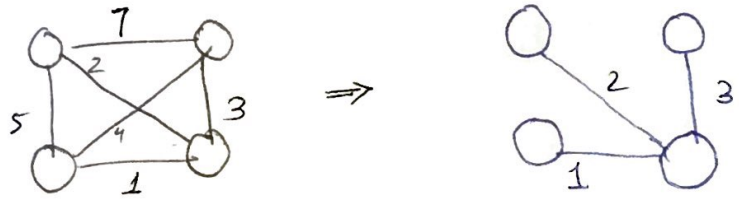
Eg → 

→ Cost of a spanning tree = Sum of edges weights of a spanning tree.

→ A spanning tree which have minimum weight is a minimum spanning tree.

→ Algorithms for obtaining the Minimum Spanning Tree
Time complexity (mlogn)

1. Kruskal's Algorithm
2. Prim's Algorith.
3. Barvka's Algo :→ A Hybrid algo and is better in application as the data structure is simpler.

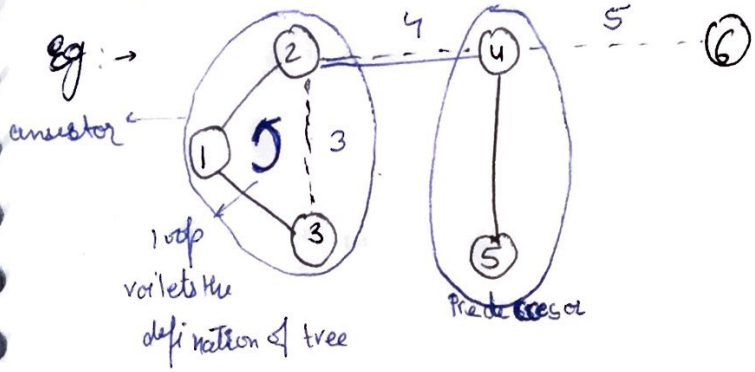
Eg: → 

n → no. of vertices = no. of trees

Kruskal's Algorithm :→

1. First draw all the ~~edges~~ vertices without edge. which will result into n number of trees.
2. Sort the edges from min. edge weight to max.
3. Whenever a graph is being updated, then will be merging of two trees.

Eg :→



(u, v)
↓ ↓
ancestor predecessor.

4. If the vertex (u,v) b/w which we are forming the edge, and both u, v belongs to same tree than it will definately be a loop.

How to find it :→

We should maintain a table in which we maintain the info. for all vertices, that which tree part they form and the tree's root.

a. T_Root[V] → in which tree vertex v belong and what is its root.

b. find-T-Root[v], find-T-Root[u]

```
if (find_T_Root[V] != find-T-Root[u])
    include (u, v);
```

After joining the vertex (u,v)

(update) T_root[v] = T_root[u]

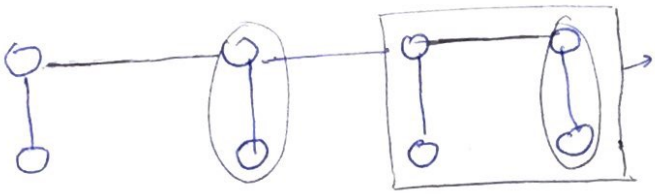
→ At first initialize all the vertices with T.root[v] = v ∀ v ∈ V

Complexity :→

1. A ← ϕ O(1)
2. for loop O(|V|)
3. Sorting edge of E → O(|E|) → ~~non~~ linear Sort (for small graph)
4. for each edge (u,v) ∈ E → m edges
   find-Set → called 2 times for each edge.
   ⇒ 2m times

→ UNION (u, v)

Worst case :→



→ updated exactly $\frac{n}{2}$ times

$\frac{n}{2} + \frac{n}{4} + \frac{n}{2} = \frac{n}{2}\log_2 n$

3 times

$3 = \log_2 8$ •

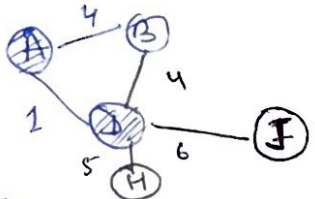
13/04/2018

Prism's Algorithm

We need to choose Vertices and find for which part the incident edge has lowest weight.

→ Once a Verten is available, you can find its adjacency list.

→ Out of the 3 Vertexes visible at a Time, A is discovered and B, D are not.



When D gets exhausted Verten, B, H & F are visible.

→ The edge weight tells which Verten has to be choose out of available vertexes (B & D, we choose D)

→ For applying the above algo:→

Let key [V ∈ V] → Thus we have some info. related to every Verten.

→ This variable is update always when a adjacency list of a Verten is discovered.

if we find $v \leftarrow Adj[u]$

(u, w) → mean there is a edge b/w u & v.

if $w(u,w) < key[v]$ ← infinite

1

weight of edge

$key[v] \leftarrow w(u, v)$

$\pi[v] \leftarrow u$ → ancestor of v

Eg :→ Adj[A] → B, D

D ← Adj[A]

if w(A, D) < key[D]

key[D] ← w(A, D)

π[D] ← A

B ← Adj[A]

if w(A, B) < key[B]

key[B] ← w(A, B)

π[B] ← A

→ min key value selected. For this min heap, data structure is used.

→ We have to repeat these steps till the heap contains element.

1st step. initialize key[v∈V] = ∞

π[D] ← NULL (for all)

2nd step :→ construct a min heap for key elements.

3rd step :→ Extract min element from the heap

4th step → look into the adjacency list of the element which you have extracted from the heap and make the above comparisons.

5th step → go to step

6th step → when heap is empty terminate

↳ for (i ← no. of vertices)

Algorithm :→

If the source or root of the tree is given, that we should start from this vertex then make its

key[v] = 0 so that in

min heap it will come at top (root)

Complexity

1. for each $u \in V[G]$ } $O(|V|)$
   do key[u] $\leftarrow \infty$
   $\pi[u] \leftarrow NIL$

key[r] $\leftarrow 0$
$Q \leftarrow V[G]$

while $Q \neq \phi$

do $u \leftarrow$ Extract-Min(Q) (Heapify here only)

eg next time jha pe update ho rha hai sirf wahi change hega so $|E| \log |V|$

$|E| \log |V|$

for each $v \in Adj[u]$ → ($O(2|E|)$)

do if $v \in Q$ and $w(u,v) < key[v]$

then $\pi[v] \leftarrow u$

$key[v] \leftarrow w(u,v)$

total of element the adjacency list have → $2|E|$

⇒ Borukva's Algorithm

→ Initially we get a forest, and no. of trees in that forest = no. of vertex.

→ What Are the possible edges which have vertex A (choose vertex) and as the incident vertex.

→ The vertex is arriving on the basis of edge weight.

-> If the weights of certain edges of a vertex (which have edges being incident on it) then choose the vertex which is a part of tree, which is already processed.

After traversing the again

→ The No. of edges incidenting on the vertex of the tree, choose the one which has min. weight.