DSA ESE

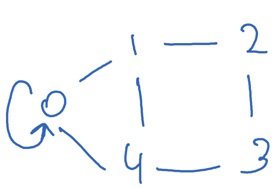
04 November 2023 10:14 AM

Types of graph:

1) Null: No vertex
2) Trivial: single vertex
3) Complete: direct path b/w any two nodes
4) Regular: All nodes have same degree
5) Bipartite: divided into two graphs with vertices set v_1 & v_2 st $v_1 \cap v_2 = \emptyset$ and there are no direct paths b/w vertices within v_1 & v_2
6) Weighted & Unweighted
7) Directed & Undirected
8) Cyclic: Atleast one cycle in it
9) Connected & Disconnected

Representation.

1) Adjancy List.



0 → 0 1 4
1 → 0 2 4
2 → 1 3
3 → 2 4
4 → 0 1 3

2) Matrix representation.

	0	1	2	3	4
0	1	1	0	0	1
1	1	0	1	0	1
2	0	1	0	1	0
3	0	0	1	0	1
4	1	1	0	1	0

* replace by weights if given

TRAVERSALS

BFS (Breadth First Search)

→ traverses at the same height first, then goes to next depth.

→ uses Queue & visited array

→ used for getting shortest distance from the starting node to any node.

→ $O(V+E)$ V = vertices   E = edges

colours :   white → not visited π → parent

gray → visited   d → dist from source

black → no more use.

```
BFS ( G, S )
for each vertex u in G - {s}
        u.colour = white
        u.dist = ∞
        u.π = nill

S.colour = gray
s.d = 0
s.π = nill
enqueue (Q, S)
while Q != empty
      u = dequeue (Q)
      for each v in G.adj [u]
            if v.colour = white
```

for each v in G.adj[u]
    if v.colour = white
        v.colour = gray
        v.d = u.d+1
        v.π = u
        enqueue(Q,v)
    u.colour = black.

DFS (Depth First Search)

- uses stack & visited array
- first searches the entire depth of a node, then moves on to another node at same depth.
- explore neighbours of neighbours before sibling
- may not give shortest path.
- $O(V+E)$

```
DFS(G)                                   // Main program
{   for each u in V                      // Initialize
    {   color[u]=W; pred[u]=NIL; }
    time=0;
    for each u in V
        if(color[u]==W) DFSVisit(u);     // Start new tree
}

DFSVisit(u)                              // Process vertex u
{   color[u]=G;                          // Vertex discovered
    d[u]=++time;                         // Time of discovery
    for each v in adj[u]
        if(color[v]==W)                  // Visit undiscovered
        {   pred[v]=u;  DFSVisit(v);}    //    neighbours
    color[u]=B;                          // Vertex finished
    f[u]=++time;                         // Time of finish
}
```

BFS

Web crawlers
topological sort
Shortest dist
GPS navigation

DFS

Detecting cycles in graph
finding a path
checking if graph is bipartite
back tracking
topological sort

if branches are many. BFS stuck.

if vertexes are may. DFS stuck

BFS gives a guarenteed sol^n. DFS doesn't.

Spanning Tree.

A tree which contains all the vertices of the graph but has no cycles in it.

edges = # vertices − 1   E = V − 1
spanning trees are acyclic
Connected and contains all vertices

} props

Minimum Spanning tree = path with least weights. Maynot be unique.

KRUSKALS ALGO (greedy approach)

→ Sort all edges in increasing order
→ Pick the lowest cost edge and add it to the MST such that it doesnt create a cycle

Time complexity: Sorting E log E

doesn't create a cycle

Time complexity: Sorting $E \log E$

union find algo for a edge $= \log V$ to detect cycles

for E edges UFA $= E \log V$

$E \log V > E \log E$

$\therefore O(E \log V)$

Make-set (u) ⇒ makes a set with element u.
find-set (u) ⇒ finds a set with u as an element
union (x, y) ⇒ union of sets x & y

Disjoint set DS

```
KRUSKAL(G):
A = ∅
For each vertex v ∈ G.V:
    MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
    if FIND-SET(u) ≠ FIND-SET(v):
    A = A ∪ {(u, v)}
    UNION(u, v)
return A
```

PRIMS ALGO (greedy approach)

start with any random edge.

See to which all vertices the edge is connected to.

Add the min possible weight which doesn't create a cycle.

Now again repeat. Now select the min wt

Main idea: Always add the edge which will add a new vertice to the MST and has the least weight. At all times the tree is connected but not cyclic.

→ $O(E \log V)$

SEARCHING ALGO

Linear Search.

best : $O(1)$

worst : $O(n)$

Binary Search

→ requires sorted algo

best : $O(1)$

worst : $O(\log n)$

Ternary Search

Fibonacci Search

Exponential Search.

SORTING ALGO

Selection Sort.

find min element in array

put it on the first index

decrease size of array by one and repeat

```
for i=1 to i=n-1
    for j=i+1 to j=n
        if ( a[i] > a[j] )
            Swap( a[i], a[j] )
```

time = $O(n^2)$ best / worst / avg

space = $O(1)$

swaps = $O(n)$

Stable = No

in place = yes.

Bubble Sort:

Time: worst $O(n^2)$

best $O(n)$

Space: $O(1)$

inplace: yes

stable: yes

```
for i=1 to i=n
    for j=0 to j=n-i-1
        if ( a[j] > a[j+1] )
            swap ( a[j], a[j+1] )
```

INSERTION SORT

Time: worst $O(n^2)$

best $O(n)$

Space: $O(1)$

inplace: yes

stable: yes

```
for i=2 to n
    Key = A[i]
    j = i-1
    while (j>0 && A[i] > key)
        a[j+1] = a[j]
        j--
    a[j+1] = Key
```

if our input is close to the best case, then we can use this

if our input data is small.

Basically our array is divided into two parts sorted & unsorted.

in each iteration we take an element from unsorted part and put it in its correct position of sorted array.

MERGE SORT

time. best $O(n\log n)$

→ based on DNC

MERGE SORT

- → based on DNC
- → Divide array then combine in sorted manner
- → merge sort better than heap sort

time. best $O(n \log n)$
avg $O(n \log n)$
worst $O(n \log n)$

space: $O(n)$

inplace: No

stable: yes.

```
Merge ( A, start, mid, end)
    len1 = mid-start -1
    len2 = end- mid
    leftArr [len1]      rightArr [len2]

    for i=0 to i= len1-1
        LeftArr [i]= A [ Start +i]

    for j=0 to j= len2-1
        rightArr [j] = A[ mid + 1+j ]

    i=0  j=0  k =Start
    while i < len1 && j < len2
        if leftArr [i] < = rightArr [j]
            A[k] = leftArr [ i]
            i++

        else
            A[k] = right Arr [j]
            j++

    k++
```

```
while ( i < len1 )
    A[k] = leftArr[i]
    i++
    k++
while ( j < len2 )
    A[k] = rightArr[j]
    j++
    K++
```

```
Merge Sort ( A, start, end )
    if start < end
        mid = start + (end - start)/2

        Merge Sort ( A, start, mid )
        Merge Sort ( A, mid+1, end )
        Merge      ( A, start, mid, end )
```

Quick Sort

Time: $O(n \log n)$ best
$O(n^2)$ worst

Space: $O(1)$ for non stable
$O(n)$ for stable

in-place = depending on version

Stable = No

pivot → first element
last element
median element

* An element (pivot) is put to its correct place in a single pass.

* An element (pivot) is put to its correct place in a single pass.

```
Partion (A,p,r)
  i = p-1    j = p    pivot = A[r]
  while (j<r)
     if ( A[j] < pivot)
          i++
          swap (A[i], A[j])
     j++
  i++
  swap ( A[i], pivot)
  return i
```

```
Quick Sort ( A, p, r )
   if (p < r)
       return
   i = Partition (A,p,r)
   Quick Sort (A,p,i-1)
   Quick sort (A, i+1, r)
```

p = Start
r = end

General Stuff

→ comparison based algo have lower bound of $O(n \log n)$

because no. of permutations of input = $n!$

$2^h > n!$ $n! = c\,n^n$ Sterling approx

$\therefore h \geq n \log n$

h = height of BT . h = no. of comparisons to be made.

Time Complexity of Quick Sort.

K = pivots correct position.

$\underbrace{\qquad}_{k-1}$ $\underbrace{\quad}_{k}$ $\underbrace{\qquad}_{n-k}$

Sub1 Sub2

$T(n) = T(n-k) + N$ → for splitting the array (traversing in partition func^n)

How to check whether a graph is bipartite using DFS

How to check whether a graph is bipartite using DFS

→ Assign a colour to a node
0 = not visited
1 = colour 1
2 = colour 2

→ if node colour = 0 ; give correct colour
→ if node colour ≠ 0 ; its parent & its own colour need to be diff
else its not bipartite.

SORTING Rev

	Time Best	Avg	Worst	Space	in place	stable
selection	n^2	n^2	n^2	1	Yes	No
bubble	n	n^2	n^2	1	Yes	Yes
insertion	n	n^2	n^2	1	yes	yes
Quick	nlogn	nlogn	n^2	1	Yes	No
Merge	nlogn	nlogn	n^2	n	No	Yes
Heap	nlogn	nlogn	nlogn	1	Yes	No

Time complexity of Merge Sort.

$$T(N) = 2\,T\left(\frac{N}{2}\right) + O(N)$$

↳ for traversing

$O(n \log n)$

$$O(\underbrace{n}_{\text{for merging}} \underbrace{\log n}_{\text{height of tree.}})$$

unstable = S Q H → Selection, Quick, Heap

not inplace = Merge.