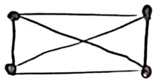
23-4-2024
TUESDAY

CSE4203
Md. Ridwan Kabir sir

Types of graphs: Let n be the no. of vertices.

1. Complete Graphs (K_n):

For every possible vertices, there'll be exactly one dot. $K_1 \Rightarrow \cdot$ $\quad K_4 \Rightarrow$ 

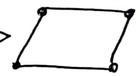
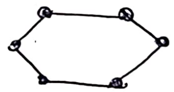
Complete Graphs follow both the theorems and thus are proved to be simple graphs.

From handshaking theorem, we get, $2m = \sum \deg(v)$

$$\sum \deg(v_o) + \sum \deg(v_e) \Rightarrow 2.6 = 4.3 \Rightarrow 12 \otimes 1 = 12$$

$(4.3) + (0.0)$

2. Cycles ($n \geq 3$):

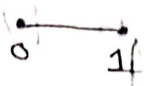
$[C_n] \Rightarrow$  $C_4 \Rightarrow$  $\quad C_6 \Rightarrow$ 

3. Wheels (W_n): $\Rightarrow C_n + 1$

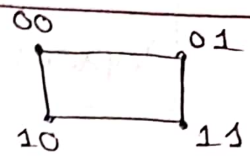
$\therefore W_3 = C_3 + 1 \Rightarrow$ 

4. n-cubes (Q_n): here, vertices represent bit strings

$Q_1 \Rightarrow 2^1 \longrightarrow$ 0, 1

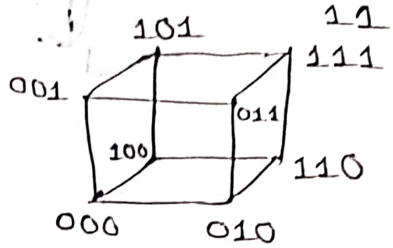


$Q_2 \Rightarrow 2^2 \longrightarrow$ 00, 01, 10, 11



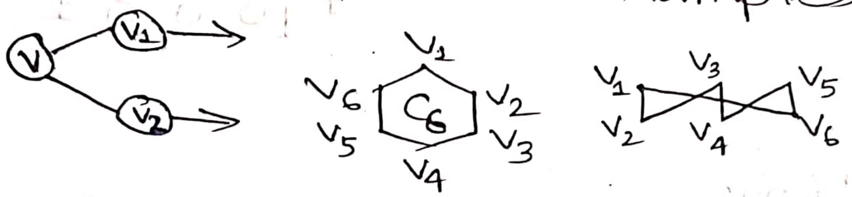
$Q_3 \Rightarrow 2^3 \longrightarrow$

000	100
001	101
010	110
011	111



Each adjacent vertices must be different at only one bit position

5. Bipartite Graphs: The examples of disjoint sets follow;

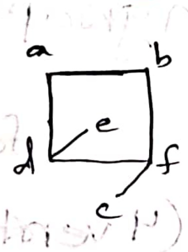


6. Complete Bipartite Graph: These are bipartite graphs but every node is connected to all other nodes.

※ To colour a graph properly, we must calculate the chromatic no. of graph (i.e., the least no. of colours needed to paint all the nodes so that no adjacent nodes share the same colour).

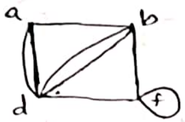
※ Graph Representations:

1. Adjacency List:



V	Adj. V
a	b, d
b	a, f
c	f
d	a, e, f
e	d
f	c, d, b

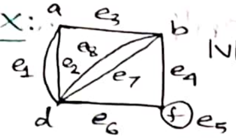
2. <u>Adjacency Matrix:</u>



$|V| \times |V| \Rightarrow$

$$\begin{array}{c|cccc} & a & b & d & f \\ \hline a & 0 & 1 & 2 & 0 \\ b & 1 & 0 & 2 & 1 \\ d & 2 & 2 & 0 & 1 \\ f & 0 & 1 & 1 & 1 \end{array}$$

3. <u>Incidence Matrix:</u>

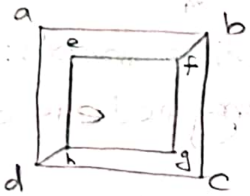


$|V| \times |E| \Rightarrow$

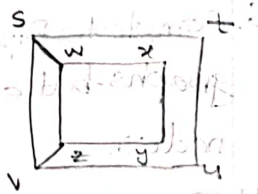
$$\begin{array}{c|cccccccc} & e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \\ \hline a & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ b & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ d & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ f & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{array}$$

☆ <u>Graph Isomorphism:</u>

2 graphs ~~may~~ will be isomers if —



G_1



G_2

i. Same no. of vertices
ii. Same no. of edges
iii. Same no. of vertices w/ same degree
iv. Derive isomorphism function based on the adjacency matrix

example — Let's use the 2 drawn graphs (G_1 and G_2) as an example:

i. 8, 8
ii. 10, 10
iii. degree is lowest at 2 (4 vertices) and highest at 3 (4 vertices for each graph)

iv.

$$G_1 = \begin{array}{c|cccccccc} & a & b & c & d & e & f & g & h \\ \hline a & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ b & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ c & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ e & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ f & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ g & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ h & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{array}$$

$$G_2 = \begin{array}{c|cccccccc} & s & t & u & v & w & x & y & z \\ \hline s & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ t & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ u & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ v & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ w & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ x & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ y & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ z & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{array}$$

Now, our task is to determine whether the matrices' rows and columns can be rearranged (based on the vertices w/ same degree) to make them look alike.

We get — $f(a) = u$ [i.e., a's degree in G_1 is same to u's degree in G_2]

Similarly, we figure — $f(b) = v$, $f(c) \neq t$.

Therefore, it's impossible for the graphs to be isomers.

It can also be noticed in the graphs as well —
pairs of adjacent vertices w/ deg(3) in $G_1 \Rightarrow$ (d, h) and (f, b)
pairs of adjacent vertices w/ deg(3) in $G_2 \Rightarrow$ (s, w), (s, v), (w, z) and (z, v)

$\therefore G_1 \not\cong G_2$

Chapter-10.4

Connectivity

* A Path starts at a vertex 'x' [let] and traverses through n [let] number of edges (Path Length) to reach the vertex 'y' [let].
* n = no. of nodes in the path − 1
* Sequence and repetition doesn't matter for paths.
* But it won't be a Simple Path (no repeating nodes), otherwise it'll be a Complex Path.
* A Circuit is basically a closed path. Similar to a self-loop, a circuit starts and ends at the same vertex.
* Euler and Hamilton Path:

i. Passes through every vertices of a graph exactly once ⇒ Hamiltonian Path (edges may be left out)

ii. Traverses every edges of a graph exactly once ⇒ Euler Path (vertices may be repeated)

* Euler and Hamilton Circuits:

i. Passes through every other vertices exactly once and the terminal one twice ⇒ Hamiltonian Circuit

ii. Passed through each edge exactly once for a closed path (to accomplish so, all vertices must have even degrees) ⇒ Euler Circuit

* For directed graphs, connectivity is divided into Strongly Connected (being able to travel from one node to another using any combination of paths) and Weakly Connected (not being able to do so).
* Theorem-1: A connected multigraph has an EP and not an EC iff there is exactly 2 vertices of odd degree.

Theorem-2: A connected multigraph w/ at least 2 vertices have an EC iff each of the vertices has even degree.

* Dirac's theorem: Let G be a simple graph w/ n no. of vertices [$n \geq 3$] and $\deg(\forall_i \in n) \geq n/2$. Then G has an HC.
* Ore's theorem: G ⟶ simple graph

n vertices, $n \geq 3$

$\deg(u) + \deg(v) \geq n$ for every pair of non-adjacent vertices (u, v)

Then, G has an HC.

* Iff Diree's and Ore's theorem converge affirmatively, the graph will have an HC. If only one of them ~~verdi~~ offers an affirmative verdict, there's a possibility of the existence of an HC for that graph.

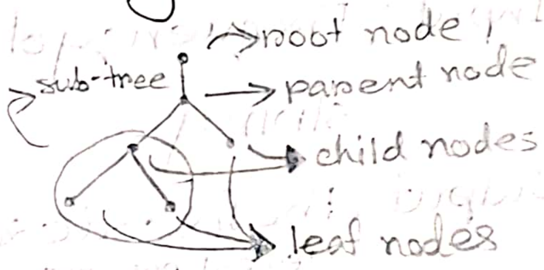
1-5-2024
SATURDAY

CSE 4203 (DM)
Md. Ridwan Kabir sir

Chapter-11
Applications of trees

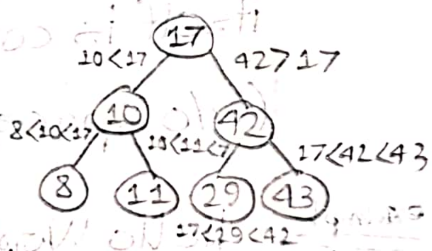
* Trees can be basically defined as graphs without simple circuits. A tree must have a Root Node from where n no. of branches may fan out.

$n=2 \longrightarrow$ binary tree
$n=m \longrightarrow$ m-ary tree



* **Binary search trees:**

1. 2 nodes/vertices → left child (less than ~~the~~ its parent node)
   → right child (~~more~~ greater or equal to than its parent node)
2. The root node will be provided / 17 (root) 10, 42, 11, 8, 29, 43
3. If child < parent → left child
   Else if child ≥ parent → right child



* **Decision trees:** It's a rooted tree where every node can have multiple children.

* **Prefix Codes:**

1. Encodes alphabets w/ bitstrings.
2. frequently used letter → shorter bitstring
   less frequent letter → longer bitstring
3. start or end of any characters in the message can be

determined using ~~Half Man~~ Huffman Coding.

example – eat ⇒ ~~e → 0, a → 1, t → 01~~ Conflicting

✱ Huffman Coding:

Input: Frequency of occurence of characters in a string

Output: Prefix codes to encode the string using the ~~least amount of~~ fewest possible bits

Goal: To produce a rooted binary tree

✱ Huffman Coding can vary based on whether space itself is considered as a character (w/ spaces) or not (w/o spaces)

example – Hello World ⇒ w/ spaces

$n = 11 \Rightarrow H \to 1,\ e \to 1,\ l \to 3,\ o \to 2,\ \_\_ \to 1,\ w \to 1,\ r \to 1,\ d \to 1$

→ probability

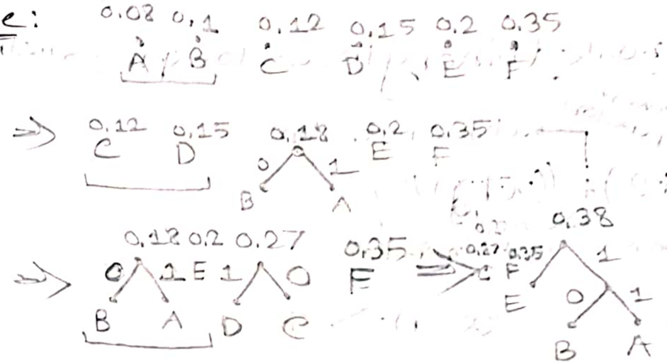
$P(H) = \frac{1}{11} = P(e) = \frac{1}{11} = P(\_) = P(w) = P(r) = P(d)$

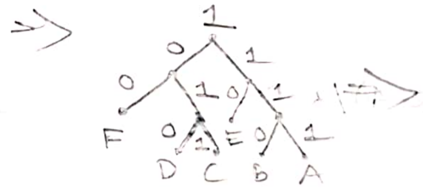
$P(l) = \frac{3}{11} \quad P(o) = \frac{2}{11}$

$\sum P = 1$

✱ Huffman code ~~sorts~~ sorts the characters in ascending order and then add the probability of 1st two characters (assigning 0 to the character w/ higher probability and 1 to the another).

example:





⇒ A = 111, B = 110, C = 011, D = 010, E = 10, F = 00

To code a character, we must start from the root node always (but the root node isn't mentioned in the final code).

✱ $\boxed{\text{Avg. no. of bits for Huffman coding} = \sum_{i \in M} |\text{bitstring}|_{\text{length of}} * \text{probability}}$

↳ for each character

✱ Algorithm to decipher Huffman coding proceeds by examining each bit and accumulating the bits from the string until a match is found for any of the ~~code's~~ character's codification.

* No character's Huffman code overlaps at their starting part. So, no character's string will be comprised of other characters' strings.

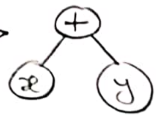
* Ordered rooted binary trees to express mathematical eqⁿs:

power operator; a binary operator

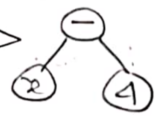
example: $((x+y)\uparrow 2) + ((x-4)/3)$

left operand: $((x+y)\uparrow 2)$; right operand: $((x-4)/3)$

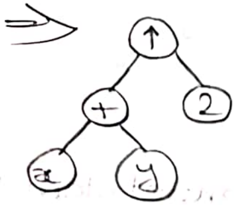
$(x+y) \Rightarrow$



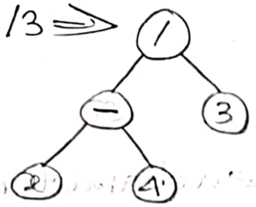
$(x-4) \Rightarrow$



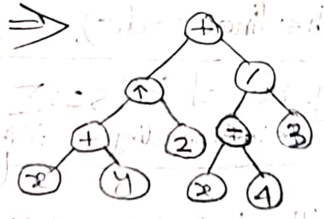
$(x+y)\uparrow 2 \Rightarrow$



$(x-4)/3 \Rightarrow$



$\therefore ((x+y)\uparrow 2)+((x-4)/3) \Rightarrow$

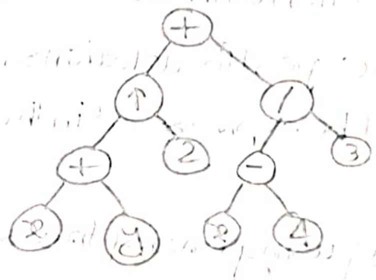


Ordered rooted tree from the given expression

* A tree can be traversed in 3 ways:

1. Preorder $\Rightarrow$ P/R_t, $L \rightarrow R$ $\Rightarrow$ Prefix notation
2. In-order $\Rightarrow$ $L, P/R_t, L \rightarrow R$ $\Rightarrow$ In-fix notation
3. Post-order $\Rightarrow$ $L \rightarrow R, P/R_t$ $\Rightarrow$ Post-fix notation

example



Prefix: $+ \uparrow /$

$\Rightarrow + \uparrow + 2 / - 3$

$\Rightarrow + \uparrow + x\ y\ 2 / - x\ 4\ 3$

Postfix: $\uparrow / +$

$\Rightarrow + 2 \uparrow - 3 / +$

$\Rightarrow x\ y + 2 \uparrow x\ 4 - 3 / +$

Infix: $\uparrow + /$

$\Rightarrow + \uparrow 2 + - / 3$

$\Rightarrow x + y \uparrow 2 + x - 4 / 3$ (same as the original expression except the bracket)

* Prefix and postfix are computationally more efficient whereas infix is better for human understanding.

* Evaluating prefix notations: $R \rightarrow L$

example: $+ - * 2\ 3\ 5 / \uparrow 2\ 3\ 4$

$2 \uparrow 3 = 8$

$\Rightarrow + - * 2\ 3\ 5 / 8\ 4$

$\Rightarrow + - * 2\ 3\ 5\ 2$

$\Rightarrow + - 6\ 5\ 2$

$\Rightarrow + 1\ 2 \Rightarrow 3$

Similarly, the opposite direction with the same method is used to evaluate postfix notations. In these calculations, fractions must be expressed in their decimal values.

* Representing each encoding using binary trees:

0 ⟶ left child, 1 ⟶ right child, characters ⟶ leaves.