Unit 2 [1 Marks]

1) Disjoint Set:- A disjoint set is a collection of sets where no element is present in more than one set

Ex:- If S_i & S_j, $i \neq j$ are two sets, then there is no element in both S_i & S_j

2) Two operations performed on disjoint sets
   1) Union - Combines two disjoint sets into one set
   2) Find - Determines the set to which a given element belongs.

3) Define find in disjoint set
The given element i [find i] determines the set containing i, by finding the root of tree representing that set.

4) Application of union find
To find the connected components in a graph (Eg:- in Kruskal's minimum spanning tree algorithm)

5) Backtracking:- Backtracking is a logical way of trying out various sequences of decisions until a solution to a problem is found, discarding the paths that fails to satisfy constraints

6) Which data structure is used in union find algorithm
Tree representation (implemented using arrays with parent pointers)

successfully placed on the board (i.e, k=n) print or record the solution.

8) Application of backtracking :- N-Queens problem.

9) Which graph problem can be used to solve backtracking?

→ Graph coloring problem.

10) output of find(5) in a set where 5→3→1?

The output is 1.

---

5 Marks

1) Explain union & find with example.

Union:- Combines 2 disjoint sets into one set. In tree representation, make one tree a subtree of the other.

Find:- Determines the set containing a given element by following parent links until reaching the root.

Ex:- $S_1 = \{1, 7, 8, 9\}$

$S_2 = \{2, 5, 10\}$

$S_3 = \{3, 4, 6\}$

Union$(S_1, S_2) = \{1, 7, 8, 9, 2, 5, 10\}$

Find(4) → returns S_3.

2) Pseudocode for 'find' operation, using path compression

```
function FIND(x):
    If parent[x] != x:  // If x is not the root
    parent[x] = FIND(parent[x])  // path compression step
    return parent[x]
```

i) Base condition :-

If parent[x] == x, it means x is the root of its set so we return it.

2) Recursive step :- If x is not the root we recursively find the root of parent[x]

3) path compression :- while returning, we directly set parent[x] to the root, flattening the tree for faster future queries

---

3) 4-Queen problem :-

The problem is to place 4 queens on a 4x4 chessboard so that;

1) No 2 queens are in the same row
2) No 2 queens are in the same column
3) " " diagonal

Here N = 4

Backtracking: We place queens one row at a time

→ Try placing a queen in each column of a current row

→ If placement is safe (no attacks), move to the next row.

→ If not safe, try next column.

→ If no position is safe in a row, backtrack to the previous row & move that queen.

Example: Q = queen — = empty cell

Step 1: Place in row 1 - Place queen in column 2 (safe)

```
- Q - -
- - - -
- - - -
- - - -
```

Step 2: Place in Row 2

```
- Q - -
Q - - -
- - - -
- - - -
```

Step 3: Row 3

```
- Q - -
Q - - -
- - Q -
- - - -
```

Step 4: Row 4

```
- Q - -
Q - - -
- - Q -
- - - Q
```

This is the valid solution//

---

4) Steps of solving N-Queens problem using back tracking

1) Start from the 1st row

2) Place a queen in the first safe column

3) Move to the next row & repeat

4) If a row has no safe column, backtrack to the previous row & move the queen to next safe column
5) Continue until all n queens are placed or all configurations are tried.

Can write example also

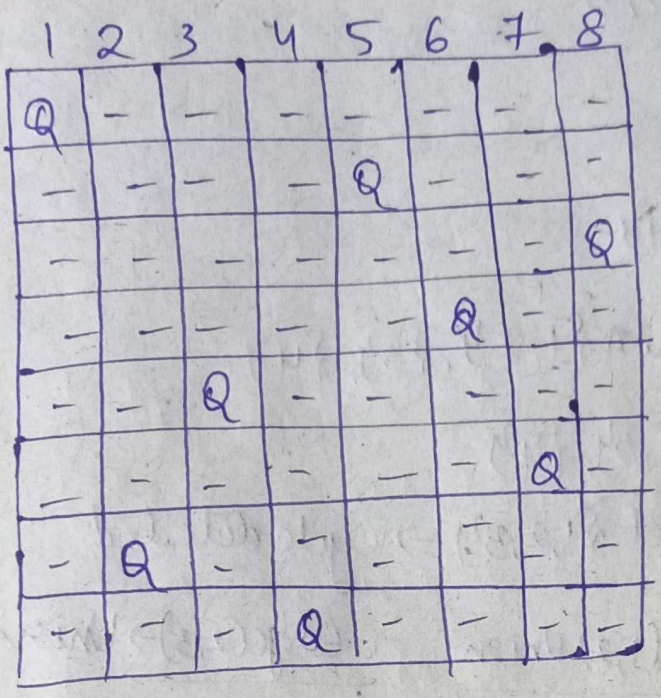
---

5) Backtracking algorithm for subset sum:-

```
Algorithm sumofsubsets (i, weight, total)
    // i → index of the current elements
    // weight → current sum of selected manner
    // total → sum of remaining elements

if weight == m then
    print the solution vector x[1..n]
    return

if (weight + total >= m) and (weight + w[i] <= m) then
    x[i] ← 1

sumofsubsets(i+1, weight+w[i], total-w[i])

if (weight + total - w[i] >= m) and (weight + w[i] <= m)
    then x[i] ← 0

sumofsubsets(i+1, weight, total-w[i])
```

6) 8-Queen problem with example.

Same as 4-queens problem (theory)



Expand them into steps & write [sodhi] :

---

7) In a undirected graph, a cycle occurs when adding an edge connects 2 vertices that are already connected by some path.

Steps :-

1) Initialize :- Create a disjoint set for all vertices (each vertex is its own parent)

2) Process each edge (u, v) :- Find the root (parent) of u & v using FIND()

→ If both have same root, adding this edge cycle will be created → Cycle detected

→ Otherwise, UNION the sets of u & v [merge them]

3) Repeat :- A cycle is found → Stop, All edges processed

Example:

Vertices: {1, 2, 3, 4}

Edges: (1,2), (2,3), (1,3) (3,4)

Step-by-step:

Initially: {1}, {2}, {3}, {4}

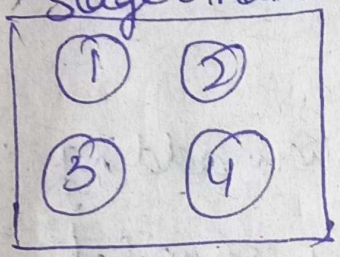
→ Edge (1,2) Diff sets → union {1,2}, {3}, {4}

→ Edge (2,3) Diff sets → {1,2,3}, {4}

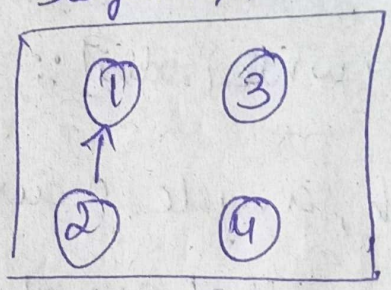
→ Edge (1,3) Both in same set {1,2,3} → Cycle detected

diagram:

Stage 0: Make set



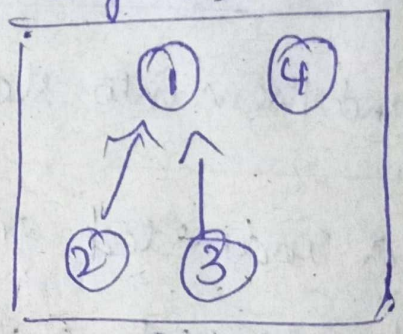
Stage 1 (1,2) union



1 ≠ 2

No cycle

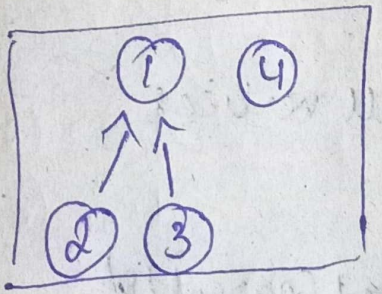
Edge (2,3) → Union



2 = 1 3 = 3

Union

No cycle



Stage 3 Edge (1,3)

Cycle

(1) = 1 3 = 1

Cycle detected //

8) Time complexity of union-find with path compression

FIND(x) :- Find the root of set containing element x

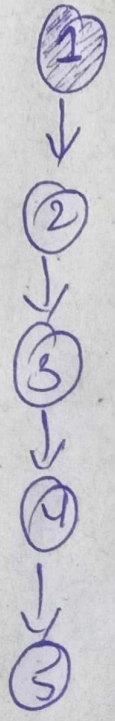
UNION(x,y) → Merge the sets containing x & y

1) path compression :- when we find the root of a node, we make each visited node point directly to the root
→ Flattens the tree.

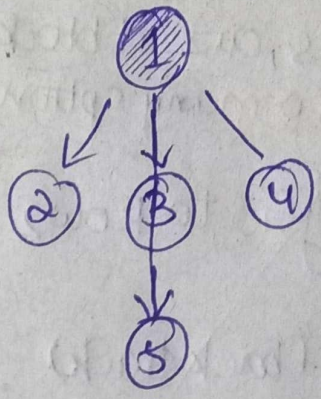
2) Union by Rank/Size :- Attach the smaller tree under the root of the larger tree. (x) not needed
→ prevents tall trees.

→ Path compression makes trees very flat over time
The cost of find is much smaller than O(n).

Before path compression



After



Not confident abt this ans :-

9) Recursive nature of the backtracking.

Write abt backtracking :-

Recursive nature :-

Backtracking problems are naturally recursive because:

1) At each decision point, we choose one option & the recursively attempt to solve the smaller sub problem
2) If recursive call finds a valid soln, we accept it
3) If we do not backtrack, & undo the choice, try next option & call the same function recursively again
4) The tree represents all possible states of the problem

Example :- N=4 Queen problem.

---

10) Solution tree for N=4 Queen problem

Row1 :-

Try (1,1) - fails, cuz it blocks too many options

(1,2) - safe } Go to Row2

(1,3) - safe

(1,4) - fails (backtrack)

Row2 :- (1,2) → (2,1) fails

(2,3) diagonal attack fails

(2,4) safe → go to row 3

(1,3) → (2,1) safe → go to row 3

(2,2) (2,4) attack from diagonal fail

10) N=4 Queen problem.

→ No 2 queens are in the same row, column & diagonal.

Row1 :- Column 2 Safe position as board is empty

Row2 :- " 4 No conflict with queen at (1,2)

Row3 :- " 1 Not in the same column as (1,2) (2,4)
Not on the diagonal of the previous queen.

Row4 :- " 3 No column & diagonal conflict

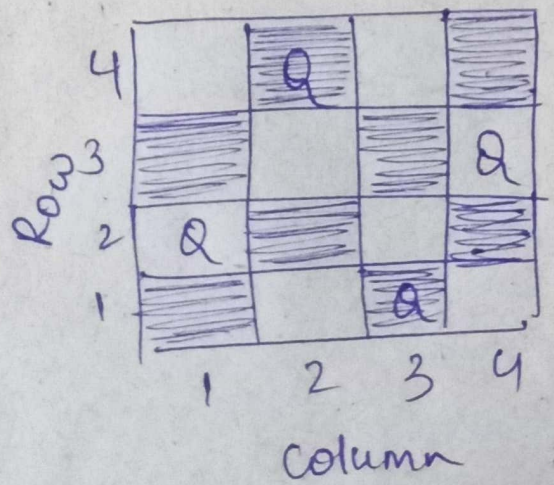
→ Each queen is on a diff row.

→ Each column positions are 2,4,1,3 → all are unique

→ The diff b/w rows & columns for any 2 queens are never the same, so no diagonal attack.

Queens are at

(1,2) (2,4) (3,1) (4,3)



One valid sol'n

Another valid sol'n is (3,1,4,2)

(1,3) (2,1) (3,4) (4,2)