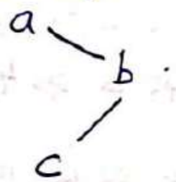
GRAPH DRILL SESSION

1. Satisfiability of Equality Equations

→ If a==b, it means a equals to b then b also equals to a. So we can say that a is connected to b with an undirected ~~ege~~ edge.



→ ~~So~~ b==c

implies a==c → TRUE

If given $a \neq c$ → then we will return ans = False.

Approach :- //O(V+E)

1. Make a graph with the help of adjacency list.
2. In the adjacency list, if a==b then a will have b and vice-versa.
3. Perform DFS to find connected components.
4. Iterate through the inequality equation to check for contradictions.

→ If two variables in an inequality are found in the same component, it's a logical contradiction.

If d!=f, but the graph place them both in component 2, the conditions cannot be satisfied and result is false.

All Submissions

```
public:
    bool equationsPossible(vector<string>& equations) {

        map<char, int> component;
        map<char, vector<char>> edges;
        set<char> st;
        map<char, bool> visited;

        for(int i=0; i<equations.size();i++){
            char x, y;

            x = equations[i][0];
            y = equations[i][3];

            st.insert(x);
            st.insert(y);

            if(equations[i][1] == '='){
                edges[x].push_back(y);
                edges[y].push_back(x);
            }
```

```
int comps = 0;

for(auto it=st.begin(); it!=st.end(); it++){

    char ch = *it;

    if(visited.count(ch) == 0){

        comps++;

        queue<char> q;

        q.push(ch);

        while(!q.empty()){

            char f = q.front();

            q.pop();
```

```
while(!q.empty()){

    char f = q.front();

    q.pop();

    visited[f] = 1;

    component[f] = comps;

    for(int i=0; i<edges[f].size(); i++){

        char v = edges[f][i];

        if(visited.count(v) == 0){
            q.push(v);
        }
    }

}
```

```
for(int i=0; i<equations.size(); i++){

    char x = equations[i][0];
    char y = equations[i][3];

    if(equations[i][1] == '!'){
        if(component[x] == component[y])
            return false;
    }

    if(equations[i][1] == '='){
        if(component[x] != component[y])
            return false;
    }
}

return true;

  }
};
```

2. Fox and Two Dots

This problem asks to find if there is a cycle of four or more dots of the same color on a grid. The grid itself can be treated as a graph.

Approach: Cycle Detection in an Undirected graph

1. DFS Traversal: Iterate through each cell of the grid. If a cell hasn't been visited, start a DFS from it.

2. During the DFS from a cell (i, j), only explore adjacent neighbouring cells that have the same color.

3. A cycle is detected when the traversal from the current node encounters a neighbour that has already been visited in the current DFS path and is not its immediate parent. This previously visited node is an ancestor.

4. To identify, use a map or visited array specific to the current traversal path.

5. Cycle must have a length of atleast 4. This can be verified by tracking the depth of each node in the DFS traversal. If a back edge is found to an ancestor, the length of cycle = difference in depths + 1. The distance must be ≥ 4.

6. If a cycle meeting all conditions (same color, length ≥ 4) is found, return true. If the entire grid is traversed without finding such a cycle, return false.

3. Maze

Goal : Convert exactly k empty cells into walls such that the remaining empty cells form a connected component.

Key Idea :-

- Treat the maze as a graph where :
  - → Vertices = empty cells
  - → Edges = adjacency b/w cells
- If we block a cell, it loses its neighbours.
- Need to ensure the remaining graph stays connected.

Observation :-

1. Leaf Nodes in DFS Traversal
   - Leaf Node = empty cell that cannot move further in DFS (only 1 connection left).
   - If we want to block cells, starts from leaf node first.
   - Remove them one by one until k ~~blocks~~ walls are placed.

2. Order of Removal
   - Always prioritize removal of leaf node with lowest degree (fewer connection)
   - After removing a leaf, its parent may become a new leaf → continue

Approach :-

1. Count total open cells S.
2. Required open cells after blocking = S-K
3. Start DFS / BFS from any empty cell.
4. Traverse until visiting S-K cells.
5. Remaining unvisited empty cells → mark as walls (X).

Multiple connected components

If maze has multiple disconnected open regions :
→ Choose one connected component.
→ Maintain S-K open cells inside it.

Exa : # . # .
. # . #
. . . .

Open cells = 6
If K = 3 → Keep 3 cells connected block rest.