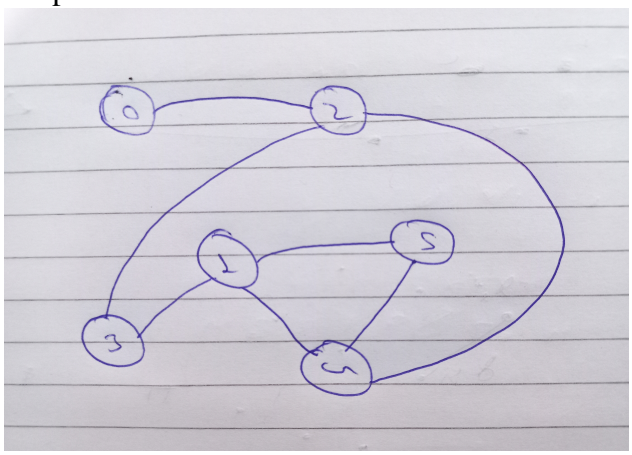
Problem Set 5

Name: Vivek Verma

Collaborators: None

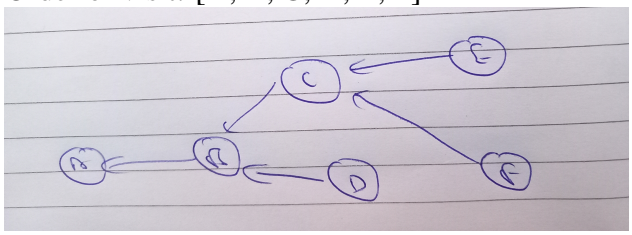
Problem 5-1.

(a) Graph

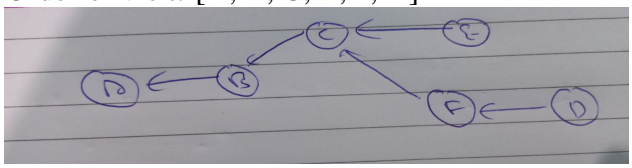


(b) { A: [B],
B: [C, D],
C: [E, F],
D: [E, F],
E: [],
F: [D, E] }

(c) BFS
Order of visit: [A, B, C, D, E, F]



DFS
Order of visit: [A, B, C, E, F, D]



(d) Topological order after removing (D, F): [A, B, C, F, D, E]
Topological order after removing (F, D): [A, B, C, D, F, E]

Problem 5-2.

Construct a Graph G where power plants and buildings are nodes of the graph and Wires are edges.

Since there are n power plants and there can be atmost one wire connected to each power plant. Also there are n^2 buildings and each building has to be powered only once. Therefore the number of wires are upper bounded by $O(n^4)$.

$$\because |E| \leq \left(\frac{|V|}{2}\right)$$

The power plant can be connected to one building and that building in turn may be recursively connected to zero or more buildings.

Maximum depth of recursion represents most number of buildings powered by the power plant.

Using Full DFS/BFS, connected components of the graph can be computed in $O(n^4)$ time.

Then size of each connected component can be computed in $O(n^2)$ time and The power plant present in maximum size connected component should be the one where emergency generator should be installed.

Problem 5-3.

Construct a graph G, where nodes are the friends and vertices represents that two nodes of the edge short-circuit each other.

A **bipartite graph** is a graph whose vertices can be divided into two sets such that in each set, there is not an edge between any two pair of vertices of that set.

If the graph G is a bipartite graph, then the nodes can be divided into two sets, such that in each set, there is not an edge between any pair of vertices.

To check whether G is bipartite or not, the two coloring algorithm can be used which is as follows:

Modifying BFS,

Initialize two colors c_1 and c_2.
Assign color c_1 to the source vertex.
Then assign c_2 to the neighbors of source.
Assign c_1 to the neighbors of neighbors of source vertex and so on.

If for any vertex, there is a neighbor which is colored same as the current vertex, then the graph is not bipartite since there is an odd cycle.

The algorithm runs in $O(n)$ time since there are n edges in the graph.

Problem 5-4. *— After looking at solution —*

Construct a graph G where vertices are squares of the grid, i.e each vertex is represented by (r, c) $r, c \in \{0, ..., n\}$.

Connect vertices $v_1 = (r, c-1)$ and $v_2 = (r, c)$ when v_2 is owned by some farmer and v_1 and v_2 are owned by different farmers.
And connect vertices $v_3 = (r-1, c)$ and $v_4 = (r, c)$ when v_4 is owned by some farmer and v_3 and v_4 are owned by different farmers.

The graph now has the property that a path between any two vertices is a route on the map that does not tramble crops.

For each $euphris$ vertex (r, c), mark vertices $\{(r, c), (r+1, c), (r, c+1), (r+1, c+1)\}$ as E. Mark in the same way for $tigrates$ vertices as T.

Combine vertices marked as E in a supernode S. Now run BFS from S to vertices in T in $O(n^2)$ time.

Return the shortest path from S to T by traversing the parents.

$\because$ there are $O(n^2)$ edges and running BFS takes $O(n^2)$ time. Traversing back to find shortest path takes $O(n^2)$ time. $\therefore$ Total running time is $O(n^2)$.

Problem 5-5.

Problem 5-6.

(a) For an empty n x n board, there are n^2 positions. If b fixed obstacles are placed on the board, then there are $n^2 - b$ available positions for the sliders.
Choosing s positions from $n^2 - b$ available positions is the number of possible configurations of the board. i.e

$\binom{n^2-b}{s} = \frac{(n^2-b)}{(n^2-b-s)!s!} = \frac{1}{s!}\prod_{i=0}^{s-1}(n^2 - b - i)$

(b) After a single move the slider will either be at top row, bottom row, left column or right column (assuming no obstacle).
Let's consider the last row/column as $n - 1$ row/column. If the slider after the move shifts to the $n - 1$ row/column, then there could be $0, ..., n - 2$ positions from which it could have before. Similarly for leftward/rightward move, there can be $n-1$ positions from which from which the slider could move to left or right column. $\therefore$ the lower bound on number of predecessors of B are n^s i.e $\Omega(n^s)$

The board configuration B can take 4 possible moves, left, right, down, up. If the reverse the move, the board will be in exact previous state. $\therefore$ there are $O(1)$ successors of a board configuration.

(c) *— After looking at solution —*
A graph G will be constructed as we compute next states from the source state.

Nodes of the graph are configurations and edges of the graph are moves.

We will run BFS on G from the source configuration. and construct the graph as next configurations are computed.

If the target configuration is after k moves, then by definition of BFS, it will take k moves to reach target configuration from the source configuration.

At any configuration, there are $r = 4$ possible moves. If the target configuration is after k moves, then $O(r^k)$ configurations are explored during the search.
If B is not solvable, then every configuration may need to be searched, and there are total $C(n, b, s)$ configurations.

Also each move takes $O(n^2)$ time.

$\therefore$ Total time complexity is $O(n^2 min\{r^k, C(n, b, s)\})$

(d)

```
def solve_tilt(B, t):
    '''
    Input:  B | Starting board configuration
            t | Tuple t = (x, y) representing the target square
    Output: M | List of moves that solves B (or None if B not solvable)
    '''
    # Base case i=0
    M = []
    P = {B: None}
    L = []
    L.append([B])

    # Inductive case
    while L[-1]:
        # To compute current level
        l = []

        # iterate over vertices u of previous level
        for u in L[-1]:
            # get the neighbors of u as v
            for direction in ['up', 'down', 'left', 'right']:
                v = move(u, direction)

                # if v does not appear in any level before
                if v not in P:
                    # set parent of v equals to u
                    P[v] = (u, direction)
                    # append v to the current level
                    l.append(v)

                    # if v is the target configuration
                    if v[t[1]][t[0]] == 'o':
                        # iteratively get the parents of v until
                        # source vertex is reached
                        while P[v]:
                            v, direction = P[v]
                            M.append(direction)
                        # reverse the path
                        M.reverse()
                        return M
        # append the current level to levels
        L.append(l)
    return None
```