Problem Set 5

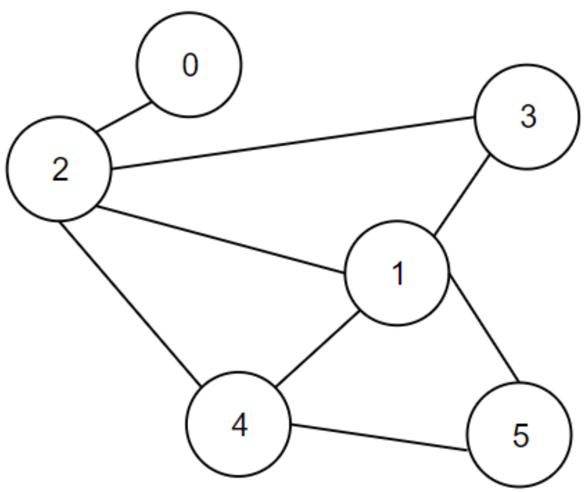
Name: Thomas Ren

Collaborators: None

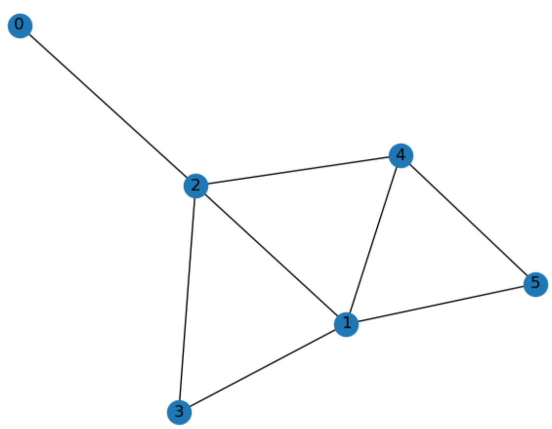
Problem 5-1.

(a) Here, I use the $app.diagrams$ and $networkx$ to draw this Graph, represented as follows:

app.diagrams



networkx



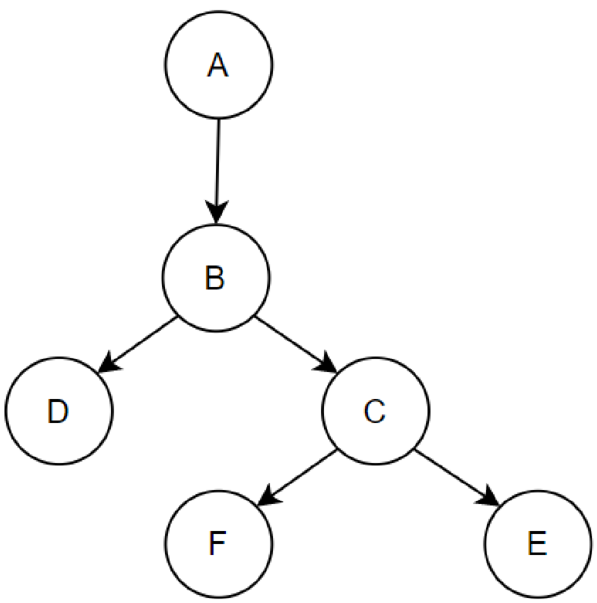
(b) The Adj list satisfying the requirements is as follows:

```
Adj = {'A':['B'],
       'B':['C','D'],
       'C':['E','F'],
       'D':['E','F'],
       'E':None,
       'F':['D','E']}
```

(c) 1. **BFS:**

```
['A','B','C','D','E','F']
```

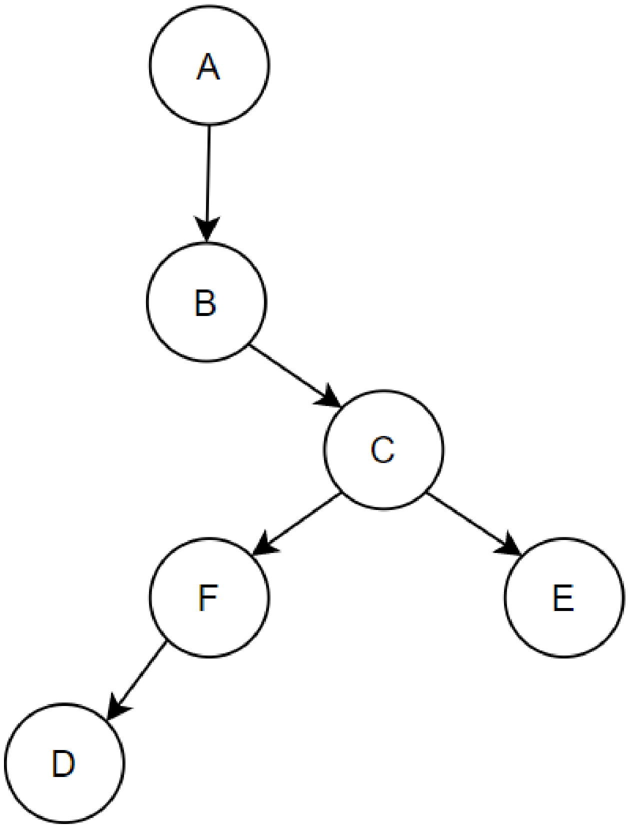
visiting tree



2. **DFS:**

```
['A','B','C','E','F','D']
```

visiting tree



(d) The reason why this graph is cyclic is the **reciprocal edges** between **Node D** and **Node F**.
A **DAG** can be easily achieved by removing either edge (D, F) or (F, D).

1. (D, F) is removed.
   Topological Order:

   ```
   ['A','B','C','F','D','E']
   ```

   Since E is the finishing node that has no outgoing edge. Then except for outgoing edge pointing to E, D has no outgoing edge, so D comes second in the **finishing order**. After doing this recursively, we can achieve the **finishing order** which is the reverse of the **topological order**.

2. (F, D) is removed.
   Topological Order:

   ```
   ['A','B','C','D','F','E']
   or
   ['A','B','D','C','F','E']
   ```

The order of D and C can be either way since there is no path between those two nodes. The rest of the order can be obtained the same way as the aforementioned method.

Problem 5-2. According to the subject condition, we know that each power plant is in a connected component, we designate each connected component as $V_i, i = 0, 1, \cdots, n-1$ with edge sets $E_i, i = 0, 1, \cdots, n-1$. The whole graph can be represented as $G(V, E)$, where $V = V_0 \cup V_1 \cup \cdots \cup V_{n-1}, E = E_0 \cup E_1 \cup \cdots \cup E_{n-1}$.

What we care about is whether a building will be powered or not, i.e. the reachability of each building. So we will use **DFS** running on every power plant.

We have already known that DFS's running time is $O(|E|)$, so for the i_{th} power plant, the running time is $O(|E_i|)$. Thus making the running time of the full-DFS to be $O(|E|) = O(\sum_{i=0}^{n-1} |E_i|)$

Then, our goal is to find out an upperbound for the number of edges. We know that edge set can be represented $V \times V$, so for n^2 buildings and n power plants, $|V| = n^2 + n$, and $|E| = O(|V|^2) = O(n^4)$.

After we run full-DFS, we need to find out the max value, which takes $O(n)$-time to find.

Thereby, the whole running time of full-DFS algorithm is $O(n^4)$.

Problem 5-3. This problem can be viewed as a coloring problem, in this situation, we need to find out whether the party members are **2-colorable**.
To construct a graph first, we map every member that is invited to this party as vertices and connect short-circuiting pairs with undirected edges.

$$G(V, E), \quad |E| = n$$

Remember that in $6.042j$, the coloring problem is **NP hard**, so it takes at least linear time for a general situation. To traverse all edges possible, we run a **full-DFS** on this constructed graph G.
Besides running the **full-DFS** algorithm, we assign each node with a color index, which is either 0 or 1. The procedure is as follows:

1. If ever we need to assign a different color to some node which has been already set a color, e.g. set $node_i$ with $color_1$ when $node_i.color = color_0$, we terminate the algorithm and return **False** that we **cannot** arrange two parties without encountering short-circuit. Otherwise, we have traversed all edges possible without termination, we return **True** that we **can** satisfy the requirements.

2. During traversing, i.e. the **full-DFS** algorithm, we assign the first node in a connected component to be, WLOG, $color_0$ and assign the following nodes on the path to be $color_1$ and $color_0$ in turn.

Problem 5-4. The essence of this problem is to construct a graph given a grid structure.
The subject asks for a **shortest path** from $euphris$ to $tigrates$, so **BFS** will be perfect to search such **shortest path**.
One problem remaining is that the $euphris$ contains more than one node and we can not afford running BFS on every node of $euphris$, which make this task not **single source**.
$Problem \quad Solved$: by adding a **virtual node** connecting to all $euphris$ nodes so that if we BFS on this virtual node to any other node, the shortest path is the same as starting from any $euphris$ node. **Besides**, we do the same thing for $tigrates$ nodes.
The subject requires not bypass crops, so before we run BFS, we need to preprocess the graph to make sure our BFS won't count banned path.
$Problem \quad Solved$: by removing edges connecting the same farmer. To eliminate every such edges, we need to traverse the whole graph. Since traversing the whole graph takes $O(n^2)$-time, it doesn't hurt our algorithm's running limit.
To sum up: we list the procedure of our algorithm from scratch:

1. Construct a grid-like graph directly mapping to the grid structure, i.e. nodes map to square grid, edges connecting adjacent nodes in the grid. Takes $O(n^2)$-time.
2. Add two virtual nodes, one connecting all $euphris$ nodes, the other connecting all $tigrates$ nodes, designate by S and t. Takes worst case $O(n^2)$-time to connect edges to the virtual nodes.
3. Preprocessing the graph, eliminating all edges that belongs to the same farmer. Needs to traverse the whole graph, takes $O(n^2)$-time.
4. Run BFS from s to t, in $O(|V| + E|)$-time. Since the $|V| + |E|$ is worst case $O(n^2)$-time, the whole algorithm from scratch runs in $O(n^2)$-time, which is linear to the graph size.

Problem 5-5. The subject is quite long, we will break down the essential parts of the problem to know what algorithm to use to solve each part of the problem.

1. Only one location e is the **entrance/exit**, so Liza has to start and terminate at the same node while carrying a pizza during the procedure.

2. Some of the doors are **one-way**, some are **two-way**. We use directed edge to represent one-way door and reciprocal edge to represent two-way door.
3. For doors that needs card access, we initially set the weight to **float('inf')** or doesn't connect that edge at all. Besides, we add some operation to the known locations l_t that if that location is been traversed, we add edges to the graph.
4. For the requirement of the subject, we need to find the minimized number of doors in our path, i.e. finding the shortest path. So **BFS** with both source and terminate at the same node.

The key part here is to construct our graph:

- For each location, construct a node with an object whose attributes includes:
  - p whether there contains a pizza. All the locations initially contain a pizza, record the number of pizzas there. After Liza traverses and takes one from a certain location, decrement this attribute.
  - $k_{SeeSail}, k_{TOPS}, k_{S3C}, k_{DPW}$: contains whether this location contains a certain type of key. If so, set the corresponding type of k to 1 and 0 otherwise. If one of them is 1, then after Liza visit this location, change the graph and set the corresponding type of k to 0.
- For each door $d = (l_1, l_2)$, if it's one-way, connect l_1, l_2 with a directed edge (l_1, l_2). Otherwise, if it's two-way, connect l_1, l_2 with reciprocal edges (l_1, l_2) and (l_2, l_1)

Constructing the graph takes linear time.
Running BFS on the graph also takes linear time.
At last, we spend linear time to decide the shortest path.

Problem 5-6.

(a) If we consider that any configuration is possible without moving the obstacles, i.e. we assume that the s sliders can be in any place in the configuration and that configuration is reachable from the initial configuration B.
Then for s sliders and b obstacles, the number of possible configurations is $\binom{possible\ places}{s}$.
Here, possible places is the number of places that is not occupied by obstacles, i.e. $n^2 - b$.

$$\begin{aligned}\binom{n^2-b}{s} &= \frac{(n^2-b)!}{s!(n^2-b-s)!} \\ &= \frac{(n^2-b)(n^2-b-1)\cdots(n^2-b-s+1)}{s!} \\ &= \frac{1}{s!}\prod_{i=0}^{s-1}(n^2-b-i)\end{aligned} \tag{1}$$

$\square$

Q.E.D.

(b) For a given configuration, it can only be transformed into at most 4 configurations, i.e. move {'up','down','left','right'}.
However, it can be transformed from a bunch of configurations. For every slider, wlog moved from move('left'), the slider can be previously in any place in that row, which has exactly n possibilities including stay at the same place.
Thereby, for all s sliders, there is at least $\Omega(n^s)$ possibilities.

(c) We construct a graph where vertices are possible configurations, and directed edges (u, v) if v can be reached with one move from u.
Then run a modified BFS from the original configuration. Since every node has at most 4-outdegree and our constructed graph is connected, $O(|E|) = O(|V|)$ and $O(|V| + |E|) = O(|V|)$.
Our algorithm terminate after traversing the level in which contains a configuration that solves the puzzle, which is $O(r^k)$. If there the configuration is not solvable, we need to traverse all possible configuration, which is $O(C(n, b, s))$. Besides, for each new configuration, it takes $O(n^2)$-time to generate the new configuration after one move.
Thereby, our algorithm takes $O(n^2 min\{r^k, C(n, b, s)\})$-time.

(d) Submit your implementation to `alg.mit.edu`.

```
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)
    '''
    M = []
    ##################
    # YOUR CODE HERE #
    ##################

    move_paths = {} # map a configuration to its moves from original B
    Transform = ('up', 'down', 'left', 'right')
    level = [] # records all different configurations ordered by number of moves from
    level.append([B])
    transform = []
    move_paths[B] = tuple(transform)
    no_victory = not(victory(B,t))
    # Adj = build_adjacency(B)
    # level.append(Adj)
    while no_victory:
        level.append([])
        for u in level[-2]:
            for adj in build_adjacency(u):
                level[-1].append(adj)
            for i, b in enumerate(build_adjacency(u)):
                transform = list(move_paths[u])
                transform.append(Transform[i])
                if b not in move_paths:
                    move_paths[b] = tuple(transform)
                if victory(b, t):
                    M = move_paths[b]
                    no_victory = False

        # for b in level[-1]:
        #     if victory(b,t):
        #         M = move_paths[b]
        #         break


    return M

def victory(B,t):
    '''
    Input:  B | Starting board configuration
            t | Tuple t = (x, y) representing the target square
    Output: Bool value | True if a slider hit the target, False otherwise
    Done in O(1)-time
    '''
    if B[t[1]][t[0]] == 'o':
        return True
    else:
        return False

def build_adjacency(B):
```

```python
    '''
    Input:  B | Starting board configuration
    Output: B_ | the four possible configuration after one tilt from B
    Done in O(1)-time
    '''
    B_ = []
    Transform = ('up','down','left','right')
    for transform in Transform:
        B_.append(move(B,transform))

    return B_

####################################
# USE BUT DO NOT MODIFY CODE BELOW #
####################################
def move(B, d):
    '''
    Input:  B  | Board configuration
            d  | Direction: either 'up', down', 'left', or 'right'
    Output: B_ | New configuration made by tilting B in direction d
    '''
    n = len(B)
    B_ = list(list(row) for row in B)
    if d == 'up':
        for x in range(n):
            y_ = 0
            for y in range(n):
                if (B_[y][x] == 'o') and (B_[y_][x] == '.'):
                    B_[y][x], B_[y_][x] = B_[y_][x], B_[y][x]
                    y_ += 1
                if (B_[y][x] != '.') or (B_[y_][x] != '.'):
                    y_ = y
    if d == 'down':
        for x in range(n):
            y_ = n - 1
            for y in range(n - 1, -1, -1):
                if (B_[y][x] == 'o') and (B_[y_][x] == '.'):
                    B_[y][x], B_[y_][x] = B_[y_][x], B_[y][x]
                    y_ -= 1
                if (B_[y][x] != '.') or (B_[y_][x] != '.'):
                    y_ = y
    if d == 'left':
        for y in range(n):
            x_ = 0
            for x in range(n):
                if (B_[y][x] == 'o') and (B_[y][x_] == '.'):
                    B_[y][x], B_[y][x_] = B_[y][x_], B_[y][x]
                    x_ += 1
                if (B_[y][x] != '.') or (B_[y][x_] != '.'):
                    x_ = x
    if d == 'right':
```

```
        for y in range(n):
            x_ = n - 1
            for x in range(n - 1, -1, -1):
                if (B_[y][x] == 'o') and (B_[y][x_] == '.'):
                    B_[y][x], B_[y][x_] = B_[y][x_], B_[y][x]
                    x_ -= 1
                if (B_[y][x] != '.') or (B_[y][x_] != '.'):
                    x_ = x
    B_ = tuple(tuple(row) for row in B_)
    return B_

def board_str(B):
    '''
    Input:  B | Board configuration
    Output: s | ASCII string representing configuration B
    '''
    n = len(B)
    rows = ['+' + (' - '*n) + '+']
    for row in B:
        rows.append(' |' + '  '.join(row) + '|')
    rows.append(rows[0])
    S = '\n'.join(rows)
    return S
```