1820. Maximum Number of Accepted Invitations

Medium Array Backtracking Matrix

Leetcode Link

Problem Description

In this problem, we are presented with a class consisting of m boys and n girls who are planning to attend an upcoming party. Each boy can invite only one girl, and each girl can accept only one invitation from a boy.

We are given a grid represented as an $m \times n$ integer matrix. The cell `grid[i][j]` can either be 0 or 1. If `grid[i][j] == 1`, it means the i-th boy has the option to invite the j-th girl to the party. Our task is to find the maximum number of invitations that can be accepted under these conditions.

In other words, our objective is to pair as many boys with girls as possible, with the constraint that each boy can invite only one girl and each girl can accept only one boy's invitation.

Intuition

To tackle this problem, we can use a graph-based approach. We can create a bipartite graph from the cell, with boys on one side and girls on the other. An edge is drawn between a boy and a girl if the boy can invite that girl (`grid[i][j] == 1`). The solution to the problem is then to find the maximum matching in this bipartite graph, which represents the largest number of pairs (invitations) that can be formed without overlap.

The intuition behind finding the maximum matching is to iterate through each boy and try to find a girl that he can invite. If the girl is already invited by another boy, we then check if that other boy can invite a different girl. This process is similar to trying to find an augmenting path in the graph—a path that can increase the number of matched pairs.

The solution code defines a function `find(i)` which performs a depth-first search (DFS) for a given boy i. This function attempts to find an available girl or to rearrange the current matches (if a girl is already invited) such that an invitation from boy i can have a match.

Inside the main function `maximumInvitations`, we maintain an array `match` where `match[j]` is the index of the boy who has invited girl j. We initialize all entries as -1, meaning initially, no girls have been invited. We also keep a count `ans` to keep track of the number of invitations successfully made.

The main function then iterates over each boy, trying to find a match for them using the `find` function while keeping track of visited girls in the `vis` set to avoid revisiting during the DFS of the current iteration. For each iteration, if a new match is found for the current boy, we increment the `ans` by 1.

In summary, the code employs a greedy depth-first search algorithm to iteratively build the maximum number of boy-girl pairings for the party invitations, following principles used to solve the maximum matching problem in bipartite graphs.

Solution Approach

The solution uses a depth-first search (DFS) algorithm to implement a technique often used for finding maximum matchings in bipartite graphs known as the Hungarian algorithm or the Kuhn-Fulkerson algorithm in its DFS version.

Here's a breakdown of the implementation:

Data Structures Used:

- `grid` An $m \times n$ matrix that represents available invitations, where m is the number of boys and n is the number of girls.
- `match`: A list where `match[j]` stores the index of the boy who has currently invited the j-th girl. It has n elements, one for each girl.
- `vis`: A set that keeps track of the visited girls in the current DFS iteration to prevent cycles and revisitations.

Algorithms and Patterns:

1. **Initialization:**
   - Create the `match` list with all elements initialized to -1, representing that no invitations have been made yet.
2. **Main Loop:**
   - Iterate through each boy, trying to find an invitation for them. For each iteration, create a new empty set `vis` to keep track of visited girls to ensure the DFS does not revisit nodes within the same path.
3. **DFS Function - `find`:**
   - The `find` function attempts to find an invitation for boy i. It iterates through all the girls and checks two conditions for each girl:
     - Whether boy i can invite girl j (indicated by `grid[i][j] == 1`).
     - Whether girl j is not already visited during the current DFS iteration (`not in vis`).
   - If both conditions are met, it adds girl j to the set `vis` and checks if girl j is not matched (`match[j] == -1`) or if the boy currently matched with girl j can find an alternate girl (`find(match[j])`). In both of these cases, the current boy i can invite girl j, and we update the `match[j]` to i and return `True`.

Execution of the Algorithm:

- For each boy, the DFS `find` function is called.
- If a match is found (the `find` function returns `True`), the count `ans` is incremented by 1.
- Finally, after the main loop has finished, the total count `ans` represents the maximum possible number of accepted invitations, which is returned as the result.

Essentially, the algorithm explores potential matches for each boy. When a potential match is found that either adds a new invitation or can replace an existing one (allowing the displaced boy to find another girl), it contributes to a larger number of overall invitations. Adjustments continue until no more matches can be made, ensuring the maximum number of invitations is reached.

The algorithm complexity is $O(n * m * \alpha)$, where the factor $n * m$ comes from the `find` function, which, in the worst case, could be called for each girl (n) in each iteration for every boy (m).

Example Walkthrough

Let's consider a small example where we have a 3×3 grid representing 3 boys and 3 girls as follows:

```
1 grid = [[1, 0, 1],
2         [1, 1, 0],
3         [0, 1, 1]]
```

Here, the 1st boy can invite either the 1st or the 3rd girl, the 2nd boy can invite the 2nd girl, and the 3rd boy can invite any of the girls.

Now, let's apply our solution approach to this example:

1. We initialize our `match` array to -1, which gives us `[-1, -1, -1]`, indicating no girl has received an invite yet.
2. We set our `ans` (answer) count to 0 as no pairs have been formed yet.

We will now iterate through each boy and try to find a match using DFS:

- **Boy 1:**
  - We attempt to invite Girl 1 since `grid[0][0] == 1`. Since Girl 1 has not been invited yet (`match[0] == -1`), we invite her, and our `match` array becomes `[0, -1, -1]`.
  - We increment `ans` to 1 as a successful match has been made.
- **Boy 2:**
  - Now we try to invite Girl 2 as `grid[1][1] == 1`. Again, she has not been invited yet (`match[1] == -1`), so we update the `match` array to `[0, 1, -1]`.
  - The `ans` count is increased to 2.
- **Boy 3:**
  - Boy 3 could invite Girl 1, 2, or 3 (since `grid[2][0] == 1`, `grid[2][1] == 1`, and `grid[2][2] == 1`). However, Girls 1 and 2 are already invited to Boys 1 and 2, so we have to check if we can rearrange.
  - We first try for Girl 1. Since Boy 1 can also invite Girl 3, we assign Girl 3 to Boy 1, rearranging the match to `[0, 1, 0]`, and then assign Girl 1 to Boy 3, updating the `match` array to `[2, 1, 0]`.
  - Notice that we had to backtrack and rearrange the existing pair to maximize the number of invitations.

The `ans` count is incremented to 3.

After iterating through all the boys, we find that the maximum number of invitations is 3, which is also the total number of boys in our example. Each girl has been matched with a boy, which is our optimal solution.

In this small example, the algorithm goes through each boy and ensures all possible matches are made by rearranging previous matches if necessary, which is in line with the DFS technique described. All boys are able to invite a girl, and this example therefore successfully illustrates the solution approach in practice.

Python Solution

```python
class Solution:
    def maximumInvitations(self, grid: List[List[int]]) -> int:
        # Helper function to find a match for a person from "group A"
        def can_invite(person_from_a):
            # Iterate through all potential matches in "group B"
            for person_from_b in range(len(grid[person_from_a])):
                # Check if the current person from "group B" has not been visited and there's a relation
                if has_relation and not visited[person_from_b]:
                    visited.add(person_from_b)
                    # If the person from "group B" is not matched or can be matched to another person
                    if matches[person_from_b] == -1 or can_invite(matches[person_from_b]):
                        # Match the person from "group A" with the person from "group B"
                        matches[person_from_b] = person_from_a
                        return True
            # Return False if no match is found for person_from_a
            return False

        # Number of people in "group A" and "group B" (rows and columns of the grid)
        num_people_a, num_people_b = len(grid), len(grid[0])

        # Initialize matches for each person in "group B" to -1 (indicating no matches)
        matches = [-1] * num_people_b

        # Total number of matches (invitations) we can make
        total_invitations = 0

        # Loop through each person in "group A" to find a matching person in "group B"
        for person_from_a in range(num_people_a):
            # Set of visited indices in "group B" for the current person from "group A"
            visited = set()

            # If a match is found, increment the total number of invitations
            if can_invite(person_from_a):
                total_invitations += 1

        # Return the total number of invitations
        return total_invitations
```

Java Solution

```java
import java.util.Arrays;

public class Solution {
    private int[][] grid; // The grid representing invitations
    private boolean[] visited; // To track if a girl (column in right set) has been visited
    private int[] matches; // To store matched left-side people to right-side people
    private int columns; // Number of columns in the grid

    // Method to compute the maximum number of invitations
    public int maximumInvitations(int[][] grid) {
        int rows = grid.length; // Number of rows in the grid
        columns = grid[0].length; // Number of columns in the grid
        this.grid = grid;
        visited = new boolean[columns]; // Initialize visited array for tracking
        matched = new int[columns]; // Initialize matches array
        Arrays.fill(matched, -1); // Fill the matches array with -1 indicating no match
        int invitations = 0; // Initialize invitation count

        // Iterate over all left side people to find maximum matchings
        for (int i = 0; i < rows; ++i) {
            Arrays.fill(visited, false); // Reset the visited array for each iteration
            if (findPath(i)) {
                invitations++; // If a match is found, increment the invitation count
            }
        }
        return invitations; // Return the maximum number of invitations
    }

    // Helper method to find a match for a person in the left set
    private boolean findPath(int personIdx) {
        for (int j = 0; j < columns; ++j) {
            // Check if there is a relation between the person(idx to right set person 'j' and 'j' is not visited
            if (grid[personIdx][j] == 1 && !visited[j]) {
                visited[j] = true; // Mark 'j' as visited
                // If 'j' is not matched, or we can find a match for 'j's current match
                if (matched[j] == -1 || findPath(matched[j])) {
                    matched[j] = personIdx; // Match person(idx (left set) with 'j' (right set)
                    return true; // A match was successful
                }
            }
        }
        return false; // No match was found for personIdx
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <cstring>
#include <iostream>

class Solution {
public:
    // Method to calculate the maximum number of invitations that can be sent.
    int maximumInvitations(vector<vector<int>>& grid) {
        int rowCount = grid.size();
        int colCount = grid[0].size();

        // Initialize visitation status and match arrays.
        bool visited[colCount];
        int matches[colCount];
        memset(matches, -1, sizeof(matches));

        int totalInvitations = 0; // Counter for total invitations.

        // Lambda function to try and find if a match can be made for a row.
        function<bool(int)> tryMatch = [&](int row) -> bool {
            for (int col = 0; col < colCount; ++col) {
                // If there is a potential match and column is not yet visited.
                if (grid[row][col] && !visited[col]) {
                    visited[col] = true;

                    // If the column is not matched or the previously matched row can be matched elsewhere,
                    // then match this row with the column.
                    if (matches[col] == -1 || tryMatch(matches[col])) {
                        // Make the match and return true.
                        matches[col] = row;
                        return true;
                    }
                }
            }
            return false; // No match found for this row.
        };

        // Iterate over rows to find all possible matches.
        for (int row = 0; row < rowCount; ++row) {
            memset(visited, 0, sizeof(visited)); // Reset visitation status.
            totalInvitations += tryMatch(row); // Increment if a match is found.
        }

        return totalInvitations; // Return the total number of successful invitations.
    }
};
```

Typescript Solution

```typescript
const GRID_SIZE_LIMIT: number = 250;
let grid: number[][] = [];
let rowCount: number = 0;
let colCount: number = 0;
let visited: boolean[] = new Array(GRID_SIZE_LIMIT).fill(false);
let match: number[] = new Array(GRID_SIZE_LIMIT).fill(-1);
let totalInvitations: number;

// Function to calculate the maximum number of invitations that can be sent
function maximumInvitations(inputGrid: number[][]): number {
    grid = inputGrid;
    rowCount = grid.length;
    colCount = grid[0].length;
    totalInvitations = 0;

    // Reset match array for each new grid.
    match.fill(-1);

    // Iterate over rows to find all possible matches
    for (let row = 0; row < rowCount; ++row) {
        // Reset visitation status for each row
        visited.fill(false);

        // Increment if a match is found
        if (tryMatch(row)) {
            totalInvitations++;
        }
    }

    // Return the total number of successful invitations
    return totalInvitations;
}

// Recursive function to perform DFS and find if a match can be made for a row
function tryMatch(row: number): boolean {
    for (let col = 0; col < colCount; ++col) {
        // If there's a potential match and column is not yet visited
        if (grid[row][col] && !visited[col]) {
            // Mark column as visited
            visited[col] = true;

            // If the column is not matched or the previously matched row can be matched elsewhere
            if (match[col] == -1 || tryMatch(match[col])) {
                // Make the match and return true
                match[col] = row;
                return true;
            }
        }
    }

    // No match found for this row
    return false;
}

// Example usage:
// const grid = [
//   [1, 0, 1],
//   [1, 1, 0],
//   [0, 1, 1],
// ];
// console.log(maximumInvitations(grid));
```

Time and Space Complexity

Time Complexity

The given code performs a modification of the Hungarian algorithm for the maximum bipartite matching problem. The time complexity is derived from the nested loops and the `find` function, which is a depth-first search (DFS):

- There is an outer loop that iterates n times where n is the number of rows in `grid`. This loop calls the `find` function.
- Within the `find` function, there is a loop that iterates m times in the worst case, where m is the number of columns, to try to find a matching for each element in the row.
- The depth-first search (DFS) within the `find` function can traverse up to n elements in the worst-case scenario.
- Since `match[j]` is called recursively within `find` (specifically, `find(match[j])`), in the worst case, this can lead to another n operations if every column is linked to a new row. Hence, the recursive calls can happen n times in the worst case.

Combining these factors, the time complexity of the entire algorithm is $O(n * m^2)$ in the worst case.

Space Complexity

The space complexity can be considered based on the data structures used:

- The `match` list uses space proportional to n, which is $O(n)$.
- The `vis` set can store up to n unique values in the worst case, as it keeps track of the visited columns for each row during DFS. This also results in space complexity of $O(n)$.
- The recursive `find` function can go up to n levels deep due to recursive calls, which could potentially use space $O(n)$ on the call stack.

As these do not depend on each other and only the maximum will determine the overall space complexity, the space complexity of the algorithm is $O(n)$.