1820. Maximum Number of Accepted Invitations Medium Array Backtracking Matrix

Leetcode Link

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

Problem Description

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 x 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.

To tackle this problem, we can use a graph-based approach. We can create a bipartite graph from the grid, 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

match.

Intuition

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.

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 matches 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 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 everyone including boy i can have a

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 <u>Hungarian algorithm</u> or the <u>Ford-Fulkerson algorithm</u> in its DFS version.

Data Structures Used: • grid: An m x 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

• vis: A set that keeps track of the visited girls in the current DFS iteration to prevent cycles and revisitations.

Whether girl j is not already visited during the current DFS iteration (j not in vis).

• If a match is found (the find function returns True), the count ans is incremented by 1.

○ Create the match list with all elements initialized to -1, representing that no invitations have been made yet.

3. DFS Function - find:

1. Initialization:

2. Main Loop:

Algorithms and Patterns:

girl.

Here's a breakdown of the implementation:

 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.

• The find function attempts to find an invitation for boy i. It iterates through all the girls and checks two conditions for each

∘ 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

Finally, after the main loop has finished, the total count ans represents the maximum possible number of accepted invitations,

or can replace an existing one (allowing the displaced boy to find another girl), it contributes to a larger number of overall invitations.

The algorithm complexity is 0(m * n * n), where the factor n * n comes from the find function, which, in the worst case, could be

Adjustments continue until no more matches can be made, ensuring the maximum number of invitations is reached.

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: Whether boy i can invite girl j (indicated by grid[i][j] == 1).

Execution of the Algorithm:

Example Walkthrough

girls.

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

called for each girl (n) in each iteration for every boy (m).

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

our match array becomes [0, -1, -1].

match array to [0, 1, -1].

The ans count is incremented to 3.

def can_invite(person_from_a):

return False

visited = set()

return total_invitations

For each boy, the DFS find function is called.

girl j, and we update the match[j] to i and return True.

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],[0, 1, 0], [1, 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

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

○ 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

• 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

class Solution:

10

12

13

14

15

16

17

27

28

29

30

31

32

33

34

35

36

37

38

39

• Boy 2:

 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.

In this small example, the algorithm goes through each boy and ensures all possible matches are made by rearranging previous

Check if the current person from "group B" has not been visited and there's a relation

If the person from "group B" is not matched or can be matched to another person

if matches[colleague_index] == -1 or can_invite(matches[colleague_index]):

Match person_from_a with colleague_index from "group B"

matches if necessary, which is in line with the DFS technique described. All boys are able to invite a girl, and this example therefore

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.

We increment ans to 1 as a successful match has been made.

already invited by Boys 1 and 2, so we have to check if we can rearrange.

successfully illustrates the solution approach in practice. Python Solution

def maximumInvitations(self, grid: List[List[int]]) -> int:

visited.add(colleague_index)

return True

for person_from_a in range(num_people_a):

if can_invite(person_from_a):

total_invitations += 1

Return the total number of invitations

private int[][] grid; // The grid representing invitations

int rows = grid.length; // Number of rows in the grid

columns = grid[0].length; // Number of columns in the grid

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

function<bool(int)> tryMatch = [&](int row) -> bool {

if (grid[row][col] && !visited[col]) {

for (int col = 0; col < colCount; ++col) {</pre>

match[col] = row;

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

// Iterate over rows to find all possible matches.

let visited: boolean[] = new Array(GRID_SIZE_LIMIT).fill(false);

let match: number[] = new Array(GRID_SIZE_LIMIT).fill(-1);

// Iterate over rows to find all possible matches

// Return the total number of successful invitations

// Recursive function to perform DFS and find if a match can be made for a row

// If there's a potential match and column is not yet visited

if (match[col] === -1 || tryMatch(match[col])) {

// Make the match and return true

// If the column is not matched or the previously matched row can be matched elsewhere

// Reset visitation status for each row

// Reset match array for each new grid.

for (let row = 0; row < rowCount; ++row) {</pre>

// Increment if a match is found

totalInvitations++;

function tryMatch(row: number): boolean {

for (let col = 0; col < colCount; ++col) {</pre>

// Mark column as visited

match[col] = row;

visited[col] = true;

return true;

// No match found for this row

62 // console.log(maximumInvitations(grid));

return false;

[1, 0, 0, 1],

[1, 0, 0, 0],

[0, 0, 1, 0],

// [0, 0, 1, 1]

Space Complexity

55 // Example usage:

56 // const grid = [

if (grid[row][col] && !visited[col]) {

visited[col] = true;

return true;

for (int row = 0; row < rowCount; ++row) {</pre>

// Lambda function to perform DFS and find if a match can be made for a row.

// If there's a potential match and column is not yet visited.

if (match[col] == -1 || tryMatch(match[col])) {

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.

// Make the match and return true.

matched = new int[columns]; // Initialize matches array

int invitations = 0; // Initialize invitation count

visited = new boolean[columns]; // Initialize visited array for tracking

// Iterate over all rows (left side people) to find maximum matchings

Arrays.fill(matched, −1); // Fill the matches array with −1 indicating no match

Arrays.fill(visited, false); // Reset the visited array for each iteration

invitations++; // If a match is found, increment the invitation count

Helper function to find a match for a person from "group A"

Return False if no match is found for person_from_a

if has_relation and colleague_index not in visited:

matches[colleague_index] = person_from_a

Loop through each person in "group A" to find a matching person in "group B"

If a match is found, increment the total number of invitations

Set of visited indices in "group B" for the current person from "group A"

private boolean[] visited; // To track if a column (person in right set) has been visited

for colleague_index, has_relation in enumerate(grid[person_from_a]):

19 # Number of people in "group A" and "group B" (rows and columns of the grid) 20 num_people_a, num_people_b = len(grid), len(grid[0]) 21 22 # Initialize matches for each person in "group B" to -1 (indicating no matches) $matches = [-1] * num_people_b$ 24 25 # Total number of matches (invitations) we can make 26 total_invitations = 0

private int[] matched; // To store matched left-side people to right-side people private int columns; // Number of columns in the grid 8 // Method to compute the maximum number of invitations 9 10 public int maximumInvitations(int[][] grid) {

11

14

15

16

17

18

19

20

21

22

23

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

43

44

46

45 };

};

Typescript Solution

2 let grid: number[][] = [];

3 let rowCount: number = 0;

4 let colCount: number = 0;

match.fill(-1);

1 const GRID_SIZE_LIMIT: number = 210;

let totalInvitations: number = 0;

visited.fill(false);

if (tryMatch(row)) {

return totalInvitations;

Java Solution

1 import java.util.Arrays;

public class Solution {

this.grid = grid;

for (int i = 0; i < rows; ++i) {

if (tryFindMatch(i)) {

```
24
 25
 26
             return invitations; // Return the maximum number of invitations
 27
 28
 29
         // Helper method to find a match for a person in the left set
 30
         private boolean tryFindMatch(int personIdx) {
             for (int j = 0; j < columns; ++j) {
 31
 32
                 // If there's an invitation from personIdx to right set person 'j' and 'j' is not visited
 33
                 if (grid[personIdx][j] == 1 && !visited[j]) {
 34
                     visited[j] = true; // Mark 'j' as visited
 35
                     // If 'j' is not matched, or we can find a match for 'j's current match
 36
                     if (matched[j] == -1 || tryFindMatch(matched[j])) {
 37
                         matched[j] = personIdx; // Match personIdx (left set) with 'j' (right set)
 38
                         return true; // A match was successful
 39
 40
 41
 42
             return false; // No match was found for personIdx
 43
 44
 45
C++ Solution
  1 #include <vector>
  2 #include <cstring>
    #include <functional>
    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();
 10
 11
 12
             // Initialize visitation status and match arrays.
 13
             bool visited[210];
             int match[210];
 14
             memset(match, -1, sizeof(match));
 15
```

// If the column is not matched or the previously matched row can be matched elsewhere.

// Function to calculate the maximum number of invitations that can be sent function maximumInvitations(inputGrid: number[][]): number { grid = inputGrid; 11 12 rowCount = grid.length; 13 colCount = grid[0].length; 14 totalInvitations = 0;

8

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

31

33

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

54

63

53 }

61 //];

32 }

```
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 m times where m is the number of rows in grid. This loop calls the find function.

  • Within the find function, there is a loop that iterates n times in the worst case, where n 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

iterations if every column is linked to a new row. Hence, the recursive calls can happen n times in the worst case.

 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.

Combining these factors, the time complexity of the entire algorithm is $0(m * n^2)$ in the worst case.

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

This also results in space complexity of O(n).

stack. As these do not depend on each other and only the maximum will determine the overall space complexity, the space complexity of

• The recursive find function can go up to n levels deep due to recursive calls, which could potentially use space 0(n) on the call

the algorithm is O(n).