1601. Maximum Number of Achievable Transfer Requests

Hard | Bit Manipulation | Array | Backtracking | Enumeration | Leetcode Link

Problem Description

We are given `n` buildings, and an array of employee transfer requests between these buildings. Each request is represented by a pair `[from, to]`, meaning an employee wants to transfer *from* one building *to* another. A request array is deemed *achievable* if the transfers can happen without altering the total number of employees in each building; that is to say, the incoming employees must balance the outgoing ones for every building. The objective is to return the maximum number of such achievable transfer requests.

To put it simply, we have to find the largest subset of the given requests that can be satisfied simultaneously, ensuring that every building ends up with the same number of employees it started with.

Intuition

The solution approach is based on the idea of checking every possible combination of requests, from no requests being satisfied to all of them being considered. To do this efficiently, a bitmask representation is utilized.

A bitmask approach involves creating a mask for each possible subset of requests. Each bit in the mask corresponds to a decision whether to include or exclude a particular request. The total number of bitmasks to check will be `2^len(requests)`, because that's how many subsets are possible.

The intuition behind using a bitmask is that it allows us to efficiently iterate over all subsets of requests, including the empty set and the set containing all requests. By incrementally checking each bitmask, we determine whether that particular combination of requests leads to a balanced transfer where each building's employee count remains unchanged.

Checking a mask involves updating a list of net change in employees for each building (`cnt`). If a request is included in the mask (indicated by the corresponding bit being `1`), we decrement the employees count from the `from` building and increment it in the `to` building. At the end of this process, we check if all buildings' counts are zero. If they are, it means that particular combination is achievable.

The final step keeps track of the maximum number of requests that form a valid set (`ans`). This is done by comparing the number of requests in the current mask (`cnt`) with the highest number found so far. If the current mask is both a larger set and a valid transfer set, it updates the maximum found.

This brute force method ensures that all possible request combinations are checked, and the largest valid subset is found.

Solution Approach

The implementation of the solution employs a brute force approach with a bit manipulation technique to iterate through all possible subsets of the transfer requests.

Here's the step-by-step breakdown of the code:

1. Define a helper function `check(mask: int) -> bool`:
   - This function takes a bitmask as an argument, representing a subset of requests.
   - It creates an array `cnt` of size `n`, initialized with zeros. The array represents the net change in the number of employees in each building.
   - The function iterates over all requests and for each request, checks if the corresponding bit in the mask is set to `1`. If it is, it means the request is included in the current subset and the function updates the `cnt` array by decrementing the count for the `from` building and incrementing it for the `to` building.
   - After iterating through all requests, it checks if all buildings have a net change of zero, and returns `True` if they do, indicating that the subset is achievable.
2. Initialize a variable `ans` with `0` to keep track of the maximum number of achievable requests found.
3. Iterate through all possible masks/subsets using a for-loop:
   - The range of the loop is `1 << len(requests)`, which gives us all possible combinations of including or excluding each request.
   - The variable `mask` represents the current subset of requests being considered.
4. Inside the loop, calculate the number of requests included in the current subset using `mask.bit_count()`, which returns the number of set bits (or `1`s) in the bitmask.
5. If the current mask has more requests than `ans` (the previous maximum), and the `check(mask)` function confirms that the current subset is achievable, update `ans` with the count of the current subset.
6. After considering all possible subsets, return `ans` as the final result. This value represents the maximum number of requests that can be accommodated without changing the number of employees in each building.

Overall, the key data structures used in this approach are:

- An integer `mask` to represent a subset of requests and facilitate bit manipulation.
- An array `cnt` for tracking the net change in employees for each building within a subset.

This algorithm makes use of combinatorial logic (to generate all possible subsets of requests) and bitwise operations (to manage and evaluate these subsets efficiently). The time complexity of this approach is O(2^m * n) where `m` is the number of requests and `n` is the number of buildings, since for each of the `2^m` subsets, we perform `n` operations to validate it.

Example Walkthrough

Let's take a small example to illustrate the solution approach.

Suppose we have `n = 3` buildings and a list of `4` employee transfer requests, as follows:

```
requests = [[0, 1], [1, 2], [0, 2], [2, 0]]
```

Here's a step-by-step walkthrough using a bitmask approach for the given requests:

1. **Initialize Maximum Achievable Requests (`ans`):**

   We start by setting `ans` to `0`, as we have not processed any requests yet.

2. **Iterate Through All Possible Subsets:**

   With `4` requests, there are `2^4 = 16` possible combinations of these requests, from no requests (bitmask `0000`) to all requests (bitmask `1111`).

3. **Evaluate Each Subset (Bitmask):**

   For each bitmask, we perform the following steps:

   - **Bitmask `0000`:** No requests are included, thus `ans` remains `0`.
   - **Bitmask `0001`:** The subset includes just the last request `[2, 0]`. This is achievable as we can transfer one employee from building `2` to `0`.
   - **Bitmask `0010`:** The subset includes the request `[0, 2]`. This is also achievable in isolation, but `ans` remains at `1` because we already found a subset of `1` request.
   - **...and so on for each combination...**
   - **Bitmask `1011`:** This subset includes requests `[0, 1]`, `[1, 2]`, and `[0, 2]`. Upon checking, the count for each building after these transfers would be `0`, so the subset is achievable. Since this subset has `3` transfers, `ans` is updated to `3`.
4. **Check Each Subset For Balance (Using `check` Function):**

   When we evaluate each set, the `check` function will update an array `cnt` that tracks the net change in number of employees at each building, considering which requests are included in the current subset. If all entries in `cnt` are `0`, it means that the current subset of requests is balanced and thus achievable.

5. **Find Maximum Number of Achievable Requests:**

   As we evaluate all possible subsets, we use the `ans` variable to keep track of the greatest number of requests in a balanced subset encountered so far. In our example, upon checking all subsets, the maximum achievable number is `3`, which would be the final answer.

For this example, the final `ans` represents that there is a subset of `3` transfer requests which can be satisfied simultaneously, maintaining the balance of employees in all buildings.

Python Solution

```python
from typing import List

class Solution:
    def maximumRequests(self, n: int, requests: List[List[int]]) -> int:
        # Internal function to check if the current combination of requests
        # satisfies the balance of incoming and outgoing requests for each building.
        def is_valid(combination: int) -> bool:
            balance = [0] * n  # Initialize a list to keep track of balance for each building
            for idx, (from_building, to_building) in enumerate(requests):
                if combination >> idx & 1:  # If the current request is included in the combination
                    balance[from_building] -= 1  # Decrement the balance for the 'from' building
                    balance[to_building] += 1  # Increment the balance for the 'to' building
            return all(value == 0 for value in balance)  # Return True if all balances are zero

        maximum_val = 0  # Initialize the maximum number of requests that can be satisfied

        # Iterate over all possible combinations of requests represented by bitmask
        for combination in range(1 << len(requests)):  # 1 << len(requests) is 2 raised to the power of the number of requests
            count = bin(combination).count('1')  # Count how many requests are included in this combination
            if maximum_val < count and is_valid(combination):  # If this combination has more requests than the max found so far, anc
                maximum_val = count  # Update the maximum value

        return maximum_val  # Return the maximum number of requests that can be satisfied

# Example usage:
# sol = Solution()
# result = sol.maximumRequests(n, requests)
# where 'n' is the number of buildings and 'requests' is the list of request pairs [from, to].
```

Java Solution

```java
class Solution {
    private int numRequests; // Total number of requests
    private int numBuildings; // Total number of buildings
    private int[][] requestsArray; // Array containing requests

    public int maximumRequests(int numBuildings, int[][] requestsArray) {
        this.numRequests = requestsArray.length;
        this.numBuildings = numBuildings;
        this.requestsArray = requestsArray;
        int maxRequests = 0; // Maximum number of requests that can be fulfilled without imbalance

        // Iterate over all possible combinations of requests
        for (int mask = 0; mask < (1 << numRequests); ++mask) {
            int requestCount = Integer.bitCount(mask); // Count of requests in the current combination
            // If the current combination has more requests and is balanced, update maxRequests
            if (maxRequests < requestCount && isBalanced(mask)) {
                maxRequests = requestCount;
            }
        }
        return maxRequests; // Return the maximum number of requests that can be fulfilled
    }

    // Helper method to check if a combination of requests is balanced
    private boolean isBalanced(int mask) {
        int[] balance = new int[numBuildings]; // Array to keep track of the balance of each building

        // Apply requests in the current combination to the balance array
        for (int i = 0; i < numRequests; ++i) {
            if ((mask >> i & 1) == 1) { // If the i-th request is in the combination
                int from = requestsArray[i][0], to = requestsArray[i][1];
                --balance[from]; // Decrement the count of the 'from' building
                ++balance[to]; // Increment the count of the 'to' building
            }
        }

        // Check if all buildings are balanced, i.e., have a zero balance
        for (int v : balance) {
            if (v != 0) {
                return false; // If any building is unbalanced, return false
            }
        }
        return true; // If all buildings are balanced, return true
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <cstring> // for memset

class Solution {
public:
    // Function to find the maximum number of requests that can be fulfilled without leaving any building imbalanced.
    int maximumRequests(int n, std::vector<std::vector<int>>& requests) {
        int requestCount = requests.size();
        int maxFulfilledRequests = 0; // Variable to store the maximum number of requests fulfilled.

        // Lambda function to check if the selected requests sequence balances the building.
        auto checkBalance = [&](int mask) -> bool {
            int balance[n]; // Array to hold the net balance of each building.
            std::memset(balance, 0, sizeof(balance)); // Initialize all balances to zero.

            for (int i = 0; i < requestCount; ++i) { // Traverse each request.
                if (mask >> i & 1) { // Check if the i-th request is chosen in the current combination (mask).
                    int from = requests[i][0], to = requests[i][1]; // Get the 'from' and 'to' buildings for the request.
                    --balance[from]; // Decrement balance for the 'from' building.
                    ++balance[to]; // Increment balance for the 'to' building.
                }
            }

            // Check if all buildings are balanced, i.e., have a net balance of zero.
            for (int value : balance) {
                if (value) { // If any building is not balanced, return false.
                    return false;
                }
            }
            return true; // All buildings are balanced, return true.
        };

        // Iterate over all combinations of requests.
        for (int mask = 0; mask < (1 << requestCount); ++mask) {
            int currentCount = __builtin_popcount(mask); // Count the number of bits set in mask, which equals the number of reques
            // If the current combination has more requests than maxFulfilledRequests and is balanced.
            if (maxFulfilledRequests < currentCount && checkBalance(mask)) {
                maxFulfilledRequests = currentCount; // Update the maximum number of requests fulfilled.
            }
        }
        // Return the final answer, which is the maximum number of requests that can be fulfilled.
        return maxFulfilledRequests;
    }
};
```

Typescript Solution

```typescript
// Function to calculate the maximum number of requests that can be fulfilled without any building ending up in a deficit.
function maximumRequests(n: number, requests: number[][]): number {
    const numberOfRequests = requests.length;
    let maximumFulfilledRequests = 0;

    // Function to check if the given combination of requests keeps all buildings balanced.
    const checkBalancedRequests = (mask: number): boolean => {
        const balanceCounter = new Array(n).fill(0);
        for (let i = 0; i < numberOfRequests; ++i) {
            if ((mask >> i) & 1) {
                const [fromBuilding, toBuilding] = requests[i];
                // Decrement for the 'from' building, and increment for the 'to' building.
                --balanceCounter[fromBuilding];
                ++balanceCounter[toBuilding];
            }
        }
        // Check if all the buildings are balanced (end up with 0 transfers).
        return balanceCounter.every(value => value === 0);
    };

    // Iterate over all possible combinations of requests.
    for (let mask = 0; mask < (1 << numberOfRequests); ++mask) {
        const numberOfSetBits = bitCount(mask);  // Count the number of bits set in mask.
        if (maximumFulfilledRequests < numberOfSetBits && checkBalancedRequests(mask)) {
            maximumFulfilledRequests = numberOfSetBits;  // Update max if a better combination is found.
        }
    }
    return maximumFulfilledRequests;  // Return the maximum number of requests that can be fulfilled.
}

// Function to count the number of bits set to 1 in the binary representation of a number.
function bitCount(i: number): number {
    i = i - ((i >>> 1) & 0x55555555);
    i = (i & 0x33333333) + ((i >>> 2) & 0x33333333);
    i = (i + (i >>> 4)) & 0x0f0f0f0f;
    i = i + (i >>> 8);
    i = i + (i >>> 16);
    return i & 0x3f;  // Return the number of bits set.
}
```

Time and Space Complexity

Time Complexity

The time complexity of the code is primarily determined by two nested operations:

1. The outer loop, which iterates over all possible subsets of requests. There are `len(requests)` requests, and for each request, there are two possibilities (either the request is fulfilled or it is not). Therefore, there are `2^(len(requests))` possible subsets, resulting in a time complexity of `O(2^m)` for this loop, where `m` is the length of `requests`.

2. The inner function `check(mask)`, which is called for each subset to verify whether choosing a particular subset of requests satisfies the balance criterion (every building ends up with the same number of people). This function iterates through all requests and then all buildings to ensure balance, contributing a complexity of `O(m + n)`, where `n` is the number of buildings.

The combined effect of these operations leads to a total time complexity of `O(2^m * (m + n))`.

Space Complexity

The space complexity of the provided code can be analyzed by looking at the additional memory used by the algorithm. The key factors influencing space complexity here are:

1. The counter array `cnt` for building balances, which uses `O(n)` space, where `n` is the number of buildings.

2. The bit mask, which does not add significant space complexity, as it is just an integer value storing the current subset being checked.

Thus, the overall space complexity of the algorithm is `O(n)`, as it's primarily dependent on the number of buildings to store the balance counter for each building.