1595. Minimum Cost to Connect Two Groups of Points

Hard | Bit Manipulation | Array | Dynamic Programming | Bitmask | Matrix | Leetcode Link

Problem Description

In this problem, we have two distinct groups of points, with the first group containing `size1` points and the second group `size2` points, and it's given that `size1 >= size2`. There is a matrix provided where `cost[i][j]` represents the cost to connect point `i` from the first group to point `j` in the second group.

The objective is to connect every point in both groups such that every point in the first group is connected to one or more points in the second group, and vice versa, with the aim to minimize the total cost of all the connections.

To solve this, we are seeking the minimum total cost to connect both groups following the rule that connections can be between any point in the first group to any point in the second group.

Intuition

The intuition behind the solution involves dynamic programming with bitmasking to efficiently compute the minimum cost while keeping track of the connections that have been made.

Given that the first group has at least as many points as the second, we iterate over each point in the first group and for each, we consider different combinations of connections with the points in the second group. For each point in the first group, we utilize a bitmask to represent the connection state of the second group's points (i.e., whether they are connected to the first group or not).

The 'f' array in the code is initialized with infinity values and represents the minimum cost to reach a given state of connections, where the state is encoded as a bitmask. The array 'g' is a copy of 'f' which is used to keep track of the costs of the next state we're computing.

The function iterates over each point in the first group. For each of these points (indexed as `i`), we explore connecting it to every point `k` in the second group, updating the minimum cost to achieve different connection states as represented by the bitmask `j`.

The essence of the dynamic programming lies in the fact that for each possible connection state 'j', where we only consider second group's points that are already connected (encoded by `j`), we calculate the minimum cost required to connect the current point 'i' to any of the second group's points that the current state 'j' indicates as connected.

The cost is updated based on the previously known minimum costs for other states ('f[j]'), and we consider connecting to a new point in the second group. This keeps track of the minimum cost to maintain a connection from the second group to the first (so that no point is left unconnected).

After iterating over all points in both groups, 'f[-1]' will hold the minimum cost to connect all points in the first group to any point in the second group, having explored all possible connection states.

Solution Approach

To implement the solution, the algorithm leverages dynamic programming and bitwise operations to efficiently track and compute the minimum cost of connecting the groups.

The data structure used to keep track of the state of the second group's connection status is an array `f` of size `2^n` (where `n` is the number of points in the second group). The reason for using `2^n` is to represent all possible combinations of connections between two groups using bit masks. Each bit in the mask can denote whether a respective point in the second group is connected (`1` meaning connected and `0` meaning not connected).

The steps followed in the algorithm are:

1. Initialize `f` with `inf` (infinity) values, except `f[0]` which is set to `0`. Here, `f` is used to store the minimum cost for each state of connections. `f[0] = 0` because initially, no connections are made, and thus the cost is zero.
2. Make a copy of `f` as `g` that will be used to calculate the costs for the next iteration without changing the existing values used as part of the calculation.
3. Iterate through each point `i` in the first group (from `1` to `m`), and for each, iterate over all possible states `j` (which is represented by the bitmasks).
4. Inside the nested loop, the algorithm iterates across each point `k` in the second group and each state of connections to calculate the minimum cost.
5. When considering a connection from point `i` in the first group to point `k` in the second group, the code checks the bits of `j` to determine if point `k` is already connected; if not, `continue` to the next possible point.
6. The minimum cost to add this connection, given the current state `j`, is computed by `c + min(f[j], f[j ^ (1 << k)])`, where `c` is the cost of connecting point `i` to point `k`, `f[j]` is the current minimum cost for the state `j`, and `f[j ^ (1 << k)]` is the cost for the state we'd have if point `k` were not connected (achieved by XOR'ing `j` with a bitmask with bit `k` set).
7. Update the temporary `g[j]` with the calculated cost for this connection, but choose the minimum between the newly calculated cost `x` and the existing `g[j]` to ensure we're keeping the lowest cost as we explore all combinations for this state.
8. After considering all points in the first group and their connections to the second group, copy the temporary cost array `g` to `f` to be used for the next iteration.
9. Once all points are processed, the last element of `f` (i.e. `f[-1]`) will contain the minimum total cost to connect every point in the first group with the points in the second group considering that `f[-1]` represents the state where all points in the second group are connected.

The algorithm makes sure to find an optimum way to make these connections by considering all possible states of the second group's connectivity for each point in the first group.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach. For simplicity, let's say we have the first group of points with `size1 = 3` and the second group with `size2 = 2`. The `cost` matrix is as follows:

```
cost = [
    [1, 2],
    [2, 3],
    [3, 1]
]
```

Where `cost[i][j]` is the cost of connecting point `i` from the first group to point `j` in the second group.

Following the solution approach:

1. We initialize `f` with `inf` values such that `f = [inf, inf, inf, inf]`, but set `f[0] = 0`.
2. We create a copy of `f` and call it `g`. At the start, `g` is the same as `f`: `g = [inf, inf, inf, inf]`, but with `g[0] = 0`.
3. We start by iterating over points in the first group. For the first point `i = 0`, we consider each possible second group connection state represented by `j`. We look at all possible bitmasks from `0 to 3` (`0b00 to 0b11` in binary, because `size2 = 2`).
4. While at point `i = 0`, for each state, we examine potential connections to each point `k` in the second group. Point `k` ranges from `0 to 1`.
5. If state `j` indicates point `k` is not already connected, we calculate the cost to connect and update `g[j]`.
6. For example, group 1 point 0 connecting to group 2 point 0 is considered. The bitmask `j` is `0` (no connections yet), so we calculate a new cost `c = 1` (from `cost[0][0]`). The new state after connection is `j ^ (1 << 0) = 1` (binary `0b01`), and we update `g[1] = min(inf, 0 + 1) = 1`.
7. We repeat this for all connections from group 1 point 0. If we connect to the second point in group 2, the new state would be `j ^ (1 << 1) = 2` (binary `0b10`), and we update `g[2] = min(inf, 0 + 2) = 2`.
8. After processing all connections from the first point in group 1, we copy the costs from `g` to `f`, making `f = [0, 1, 2, inf]`.
9. We now repeat the steps for the next point in the first group (point `i = 1`). We consider the same possible states and update `g` accordingly.
10. After processing all points in the first group, each state `j` of the second group's points indicates what the minimum cost is to reach that state.
11. The final minimum cost to connect all points with this example would be found in `f[-1]`.

For this simple example, the final state of `f` after processing all points could be something like `[0, 1, 2, 4]`, indicating that the minimum cost to connect every point in the first group with every point in the second group is `4`. This represents the state where both points in the second group are connected to at least one point in the first group.

Python Solution

```python
class Solution:
    def connectTwoGroups(self, cost: List[List[int]]) -> int:
        # Get the size of group A and group B
        size_group_a, size_group_b = len(cost), len(cost[0])

        # Initialize the dp array f with infinite value for set states
        dp = [float('inf')] * (1 << size_group_b)
        # Initially, the cost to connect no elements from group B is 0
        dp[0] = 0

        # Initialize a temporary array 'next_dp' the same as 'dp' to store next state values
        next_dp = dp[:]

        # Loop for each element in group A
        for i in range(1, size_group_a + 1):
            # Update states for the new element of group A
            for state in range(1 << size_group_b):
                next_dp[state] = float('inf')

                # Try to connect the i-th element of group A with each element of group B
                for j in range(size_group_b):
                    # If the j-th element of group B is not in the current state, skip
                    if (state >> j & 1) == 0:
                        continue

                    # Calculate the cost to connect the i-th element of group A with the j-th element of group B
                    c = cost[i - 1][j]

                    # Determine the new state by removing the j-th element from the current state
                    new_state = state ^ (1 << j)

                    # Compute the total cost for the new state
                    # and compare with previous costs to find the minimum
                    total_cost = min(next_dp[new_state], dp[state], dp[new_state]) + c

                    # Update the cost for the current state
                    next_dp[state] = min(next_dp[state], total_cost)

            # Move to the next state
            dp = next_dp[:]

        # Return the minimum total cost to connect all elements from both groups
        return dp[-1]
```

Java Solution

```java
class Solution {
    public int connectTwoGroups(List<List<Integer>> cost) {
        int numRows = cost.size();                  // Number of rows in the cost matrix
        int numCols = cost.get(0).size();          // Number of columns in the cost matrix
        final int INFINITY = 1 << 30;              // Define an "infinity" value to be used for comparison

        int[] dp = new int[1 << numCols];          // Dynamic programming (dp) array for the previous round
        Arrays.fill(dp, INFINITY);                 // Initialize all values to infinity
        dp[0] = 0;                                 // Starting with no connections, the cost is zero
        int[] dpNext = dp.clone();                 // Clone the dp array for processing the next row

        // Outer loop processes each row of the cost matrix
        for (int i = 1; i <= numRows; ++i) {
            // Initialize dpNext for each row calculation
            Arrays.fill(dpNext, INFINITY);

            // Go through all the possible states (combinations of connections) in dpNext
            for (int state = 0; state < (1 << numCols); ++state) {

                // Check each column in the current row
                for (int col = 0; col < numCols; ++col) {
                    // If the current state includes a connection to the col-th column
                    if ((state >> col & 1) == 1) {
                        int connectionCost = cost.get(i - 1).get(col);                        // Get the cost for connecting the cu

                        // Update dpNext[state] with the minimum of three possible scenarios:
                        // 1. Connect the current row's col-th column with previous state excluding col-th column and add the cost
                        dpNext[state] = Math.min(dpNext[state], dpNext[state ^ (1 << col)] + connectionCost);
                        // 2. Connect the current row's col-th column with the previous row's state and add the cost
                        dpNext[state] = Math.min(dpNext[state], dp[state] + connectionCost);
                        // 3. Connect the current row's col-th column with the previous row's state excluding col-th column and add
                        dpNext[state] = Math.min(dpNext[state], dp[state ^ (1 << col)] + connectionCost);
                    }
                }
            }

            // Copy the dpNext into dp for the next round/row
            System.arraycopy(dpNext, 0, dp, 0, 1 << numCols);
        }

        // Final answer is at dp[(1 << numCols) - 1], which represents all columns connected
        return dp[(1 << numCols) - 1];
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    int connectTwoGroups(vector<vector<int>>& costs) {
        int numLeft = costs.size(); // Number of nodes on the left
        int numRight = costs[0].size(); // Number of nodes on the right
        const int INF = 1 << 30; // Define an "infinity" value for comparison purposes
        vector<int> currDp(1 << numRight, INF); // Current DP state, initially set to "infinity"
        currDp[0] = 0; // The cost to connect an empty subset is 0
        vector<int> nextDp = currDp; // The next DP state to be calculated

        // Loop through each node on the left
        for (int i = 1; i <= numLeft; ++i) {
            // Calculate next DP state for every subset of right nodes
            for (int mask = 0; mask < (1 << numRight); ++mask) {
                nextDp[mask] = INF; // Reset the next DP state to "infinity"
                // Iterate through each bit/node in the mask
                for (int k = 0; k < numRight; ++k) {
                    // Check if the k-th node on the right is included in the subset
                    if (mask & (1 << k)) {
                        // The cost to connect the i-1-th node on the left and k-th node on the right
                        int connectionCost = costs[i - 1][k];

                        // The new cost is the minimum of three potential previous states plus the new connectionCost
                        int newCost = min({nextDp[mask ^ (1 << k)], currDp[mask], currDp[mask ^ (1 << k)]}) + connectionCost;

                        // Update the next DP state with the new cost if it's smaller than the current cost
                        nextDp[mask] = min(nextDp[mask], newCost);
                    }
                }
            }
            // Swap the current DP state with the next one for the subsequent iteration
            currDp.swap(nextDp);
        }
        // Return the minimum cost to connect all nodes on the left to some subset of nodes on the right
        return currDp[(1 << numRight) - 1];
    }
};
```

Typescript Solution

```typescript
function connectTwoGroups(cost: number[][]): number {
    const groupSizeA = cost.length; // Size of the first group
    const groupSizeB = cost[0].length; // Size of the second group
    const infinity = 1 << 30; // A large number to represent infinity
    const dpCurrent = new Array(1 << groupSizeB).fill(infinity); // DP array to keep track of minimum costs for current iteration
    dpCurrent[0] = 0; // Base case: cost for connecting no elements from Group B
    const dpNext = new Array(1 << groupSizeB).fill(0); // DP array for the next iteration

    // Loop through each element in Group A
    for (let i = 1; i <= groupSizeA; ++i) {
        // Loop through all subsets of Group B
        for (let subsetMask = 0; subsetMask < (1 << groupSizeB); ++subsetMask) {
            dpNext[subsetMask] = infinity; // Reset the next DP array entry to infinity

            // For each element in Group B, try to make connections
            for (let j = 0; j < groupSizeB; ++j) {
                // Check if element j is in the current subset mask
                if (((subsetMask >> j) & 1) === 1) {
                    const currentCost = cost[i - 1][j];

                    // Update dpNext for removing element j from the subset
                    dpNext[subsetMask] = Math.min(dpNext[subsetMask], dpNext[subsetMask ^ (1 << j)] + currentCost);

                    // Update dpNext based on the current dp (previous iteration) keeping element j
                    dpNext[subsetMask] = Math.min(dpNext[subsetMask], dpCurrent[subsetMask] + currentCost);

                    // Update dpNext based on the current dp (previous iteration) removing element j
                    dpNext[subsetMask] = Math.min(dpNext[subsetMask], dpCurrent[subsetMask ^ (1 << j)] + currentCost);
                }
            }
        }

        // Copy the dpNext values to dpCurrent to be used in the next iteration
        dpCurrent.splice(0, dpCurrent.length, ...dpNext);
    }

    // Return the minimum cost for connecting all elements in both groups
    return dpCurrent[(1 << groupSizeB) - 1];
}
```

Time and Space Complexity

The given code is an implementation of a dynamic programming algorithm designed to solve a problem that seems to involve connecting two groups with certain costs. It uses bit masking to represent subsets of connections between the groups.

Time Complexity:

To calculate the time complexity, let's break down the nested loops in the code:

- There is an outer loop that runs `m` times, where `m` is the length of the first dimension of `cost`.
- Inside this loop, there is another loop that runs for all subsets of the second group, which is `2^n` times, where `n` is the length of the second dimension of `cost`.
- Finally, within the second loop, there is an innermost loop that iterates over all `n` elements of the second group.

The innermost loop checks and updates the best cost to connect elements of the first group to a subset of elements of the second group.

Therefore, the time complexity is `O(m * n * 2^n)`, as for each of the `m` elements in the first group and for each of the `2^n` subsets of the second group, we perform `n` checks and updates.

Space Complexity:

As for the space complexity, there are two main data structures to consider: the `f` and `g` arrays. Each of these arrays has a length of `2^n`, as they store subsets of the second group.

Hence, the space complexity is `O(2^n)`, as this is the largest amount of space used by the algorithm at any point in time.