1066. Campus Bikes II

Medium Bit Manipulation Array Dynamic Programming Backtracking Bitmask   Leetcode Link

Problem Description

The LeetCode problem described above presents a scenario where there is a campus represented as a 2D grid, which contains a certain number of workers (`n`) and bikes (`m`) located at different coordinates on this grid. With a condition where `n <= m`, implying there are at least as many bikes as there are workers, possibly more.

The challenge is to assign each worker exactly one bike in such a way that the total sum of the Manhattan distances between each worker and their assigned bike is as small as possible. Manhattan distance is a way to measure distance on a grid, calculated by taking the absolute difference of the x-coordinates and the y-coordinates of two points and adding them together.

The task is to return this minimum possible sum of the Manhattan distances after optimally assigning the bikes to the workers.

Intuition

The intuition behind the solution is to use dynamic programming to explore all possible assignments of bikes to workers. The state space is represented by a 2D array `f`, where the dimensions are the number of workers (`n`) plus one, and the number of possible states which is `2^m` (since `m` is the number of bikes, and a bike can either be assigned or not, representing a binary state).

We initialize a 2D array `f` with infinity, which represents the minimum cost of assigning bikes to workers up to a certain point. The first dimension `i` is iterating through workers, and the second dimension `j` is a bitmask representing which bikes have been taken.

In this algorithm, we start from worker 1 and try to assign bikes to workers up to that point. For each worker `i`, we look up all possible combinations of bike assignments for previous workers (`j`). Then, we iterate over all bikes and try to assign an available bike `k` to the current worker, updating the minimum cost.

We do this by taking the assignment cost for the previous worker without bike `k` (`f[i - 1][j ^ (1 << k)]`, where `^` is the XOR operator and `<<` is the left shift operator, used here to manipulate bits) and adding the Manhattan distance between worker `i` and bike `k`. This gives us the cost of assigning bike `k` to worker `i`. We take the minimum of these values for all bikes and all combinations of previous assignments.

Finally, we are interested in the minimum cost when all workers have been assigned a bike which is the minimum of the last row `f[n]` of our array.

This approach ensures that we find the minimum sum of distances without having to consider all permutations, which would be too expensive to compute for larger numbers of workers and bikes.

Solution Approach

The solution employs dynamic programming (DP) to efficiently solve the problem by breaking it down into smaller subproblems. The key data structure in use is a 2D array `f`, which is used to store the minimum cost of assigning bikes to all workers up to a certain point.

Here's how the solution approach works:

1. **Initialize DP array**: We create a 2D array `f` with dimensions `(n+1) x (2^m)`, where `n` is the number of workers and `m` is the number of bikes. Each element of `f` is initialized to infinity, which represents the cost of assigning bikes to workers. The first row is initialized to 0 since at this point no bikes have been assigned and the cost is 0.

2. **Iterating workers**: We loop through all workers (starting from 1 since we skip the base case of 0 workers) using `i`, and within this loop, we scan all possible bike assignments using a bitmask representation `j`. This bitmask is an integer where the k-th bit is 1 if bike k has been assigned to a worker, and 0 otherwise.

3. **Assigning bikes**: For each worker `i` and each possible previous assignment `j`, we iterate through all the bikes using `k` to check which bikes can be assigned to the current worker. Here, `(x1, y1)` represents the current worker's position, and `(x2, y2)` are the coordinates for bike `k`.

4. **Updating DP array**: We calculate the Manhattan distance between worker `i` and bike `k` and consider the minimum cost so far for worker `i-1` without bike `k` (`f[i-1][j ^ (1 << k)]`). If bike `k` is available in the current assignment `j` (indicated by `j >> k & 1`), we update the DP array for the current worker `i` as the minimum of the current stored value and the new value which is the sum of the previous workers' assignments (`f[i-1][j ^ (1 << k)]`) and the Manhattan distance for worker `i` to bike `k`.

5. **Finding the result**: After populating the DP array, the minimum possible sum that we require will be the minimum value of the last row `f[n]`. This is because the last row of the DP array represents the minimum cost of assigning bikes to all `n` workers, considering all combinations.

The algorithm effectively uses bit manipulation to handle combinations of bike assignments and dynamic programming to avoid redundant calculations of the minimum distance sums. This turns an otherwise exponential-time permutation problem into a solvable problem with a time complexity that is exponential in `m` (which is acceptable for the constraints of the problem, as `m` is limited and there are efficient ways to manage bits in most programming languages).

Example Walkthrough

Let's go through a small example to illustrate the solution approach. Consider a grid where we have 2 workers (`n=2`) and 2 bikes (`m=2`). The grid looks like this with workers `W` and bikes `B` at the coordinates shown:

```
(0,0) (0,1)
 W     B

(1,0) (1,1)
 B     W
```

Now, we will walk through the algorithm steps for our small grid:

1. **Initialize DP array**: We have 2 workers and 2 bikes. Our `f` array will have a size of `(2+1) x (2^2)`, which is `3 x 4`. We initialize all values to infinity except `f[0]` which will be all zeroes indicating no bikes have been assigned yet.

2. **Iterating workers**: We will iterate through our workers (1 and 2). Let's denote the bitmask states as follows: `00` represents no bikes are taken, `01` represents the first bike is taken, `10` represents the second bike is taken, and `11` represents both bikes are taken.

3. **Assigning bikes**:

- For worker 1, and the initial state `00`, we try assigning both bikes:
  - Assigning bike 1, we calculate the Manhattan distance from (0,0) to (0,1), which is `1`. So, `f[1][01] = min(infinity, 0 + 1) = 1`.
  - Assigning bike 2, we calculate the distance from (0,0) to (1,0), which is also `1`. So, `f[1][10] = min(infinity, 0 + 1) = 1`.
- Moving to worker 2, for the state `01`, we can only assign bike 2:
  - The distance from (1,1) to (1,0) is `1`. Since bike 1 is taken (`01` state), we look at `f[1][01]` which is `1` and add the new distance. So, `f[2][11] = min(infinity, f[1][01] + 1) = min(infinity, 1 + 1) = 2`.
- For the state `10`, only bike 1 is available:
  - The distance from (1,1) to (0,1) is also `1`. We check `f[1][10]`, add the new distance, and update if it's the minimum. So, `f[2][11] = min(f[2][11], f[1][10] + 1) = min(2, 1 + 1) = 2`.

4. **Updating DP array**: We've updated our DP array with the minimum distances considering all possible assignments. `f[1][01]` and `f[1][10]` are both `1`, and `f[2][11]` is `2`.

5. **Finding the result**: The result is in the last row of the `f` array. The minimum value is `f[2][11]`, which is `2`, indicating that the minimum sum of the Manhattan distances after optimally assigning the bikes to the workers is 2.

With this approach, we systematically evaluate each state and assign bikes to workers such that we minimize the total Manhattan distance. The example walk-through demonstrates the principles of the approach on a small scale, showing how dynamic programming helps avoid redundant calculations and delivers the minimum possible sum for the problem.

Python Solution

```python
from typing import List

class Solution:
    def assignBikes(self, workers: List[List[int]], bikes: List[List[int]]) -> int:
        num_workers, num_bikes = len(workers), len(bikes)

        # Initialize a 2D memoization table with infinity where
        # `dp[i][state]` represents the minimum distance to assign `i` workers
        # with the `state` of bikes availability.
        dp = [[float('inf')] * (1 << num_bikes) for _ in range(num_workers + 1)]
        dp[0][0] = 0  # Base case: no workers assigned, no bikes taken.

        # Iterate over all workers.
        for worker_index, (worker_x, worker_y) in enumerate(workers, start=1):
            # Iterate over all possible bike states.
            for state in range(1 << num_bikes):
                # Iterate over all bikes.
                for bike_index, (bike_x, bike_y) in enumerate(bikes):
                    # Check if the bike at `bike_index` is available in current `state`.
                    if state >> bike_index & 1:
                        # Calculate the Manhattan distance between
                        # the worker and the bike.
                        distance = abs(worker_x - bike_x) + abs(worker_y - bike_y)

                        # Calculate the previous state by turning off
                        # the bike_index-th bit in `state`.
                        previous_state = state ^ (1 << bike_index)

                        # Update the memoization table by considering this new distance.
                        dp[worker_index][state] = min(
                            dp[worker_index][state],
                            dp[worker_index - 1][previous_state] + distance,
                        )
        # Return the minimum distance across all states when all workers are assigned.
        return min(dp[num_workers])

# Example usage:
solution = Solution()
workers = [[1, 2], [3, 4]]
bikes = [[1, 1], [2, 3]]
result = solution.assignBikes(workers, bikes)
print(result)  # Output will be the minimum sum of distances for assigning bikes.
```

Java Solution

```java
class Solution {
    public int assignBikes(int[][] workers, int[][] bikes) {
        int numWorkers = workers.length;
        int numBikes = bikes.length;

        // dp array, dp[i][j] will hold the minimum distance sum for i workers and state j for bikes
        // state j is a bitmask representing which bikes have been assigned
        int[][] dp = new int[numWorkers + 1][1 << numBikes];

        // Initialize dp array with large values, as we will be taking the minimum
        for (int[] row : dp) {
            Arrays.fill(row, Integer.MAX_VALUE / 2); // Use Integer.MAX_VALUE / 2 to avoid overflow
        }

        // No workers means no distance
        dp[0][0] = 0;

        // Calculate distance sum for each worker (i) and each possible bikes state (j)
        for (int i = 1; i <= numWorkers; ++i) {
            for (int j = 0; j < (1 << numBikes); ++j) {
                for (int k = 0; k < numBikes; ++k) {
                    // Check if bike k is available in state j
                    if ((j & (1 << k)) != 0) {
                        // Calculate the Manhattan distance between worker i-1 and bike k
                        int distance = Math.abs(workers[i - 1][0] - bikes[k][0])
                                     + Math.abs(workers[i - 1][1] - bikes[k][1]);

                        // Update dp: try to assign bike k to worker i-1 and see if we get a better solution
                        dp[i][j] = Math.min(dp[i][j], dp[i - 1][j ^ (1 << k)] + distance);
                    }
                }
            }
        }

        // Find and return the minimum distance from the last row of the dp array
        return Arrays.stream(dp[numWorkers]).min().getAsInt();
    }
}
```

C++ Solution

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

class Solution {
public:
    int assignBikes(std::vector<std::vector<int>>& workers, std::vector<std::vector<int>>& bikes) {
        int numWorkers = workers.size(); // Number of workers
        int numBikes = bikes.size();    // Number of bikes
        int dp[numWorkers + 1][1 << numBikes]; // dp array to store min distance

        // Initialize the dp array with a large value
        memset(dp, 0x3f, sizeof(dp));

        // Base case: no workers assigned, so the distance is 0
        dp[0][0] = 0;

        // Start from the first worker
        for (int i = 1; i <= numWorkers; ++i) {
            // Go through all possible combinations of the bikes
            for (int mask = 0; mask < (1 << numBikes); ++mask) {
                // Check each bike
                for (int k = 0; k < numBikes; ++k) {
                    // If the k-th bike is available in this combination
                    if (mask & (1 << k)) {
                        // Calculate the Manhattan distance from the i-1-th worker to the k-th bike
                        int distance = std::abs(workers[i - 1][0] - bikes[k][0]) +
                                       std::abs(workers[i - 1][1] - bikes[k][1]);
                        // Update the dp value considering the current bike-worker pair
                        // Flip the k-th bit to mark the bike as used for the current worker
                        dp[i][mask] = std::min(dp[i][mask], dp[i - 1][mask ^ (1 << k)] + distance);
                    }
                }
            }
        }

        // Find and return the smallest value from the last row of dp array
        // which represents the assignment of all workers with a valid bike combination
        return *std::min_element(dp[numWorkers], dp[numWorkers] + (1 << numBikes));
    }
};
```

Typescript Solution

```typescript
function assignBikes(workers: number[][], bikes: number[][]): number {
    // n represents the number of workers
    const numWorkers = workers.length;
    // m represents the number of bikes
    const numBikes = bikes.length;
    // Large number representing 'infinity'
    const INF = 1 << 30;
    // f is a 2D array to hold the minimum distance calculation results
    // for each combination of workers and used bikes
    const minDistances: number[][] = new Array(numWorkers + 1)
                                     .fill(0)
                                     .map(() => new Array(1 << numBikes).fill(INF));
    // Initialize the minimum distance for 0 workers to be 0
    minDistances[0][0] = 0;

    // Iterate through each worker
    for (let workerIndex = 1; workerIndex <= numWorkers; ++workerIndex) {
        // Iterate through all combinations of bike assignments
        for (let mask = 0; mask < (1 << numBikes); ++mask) {
            // Iterate through each bike
            for (let bikeIndex = 0; bikeIndex < numBikes; ++bikeIndex) {
                // Check if the current bike is already used in the combination specified by mask
                if (((mask >> bikeIndex) & 1) === 1) {
                    // Calculate the Manhattan distance between the current worker and bike
                    const distance = Math.abs(workers[workerIndex - 1][0] - bikes[bikeIndex][0]) +
                                     Math.abs(workers[workerIndex - 1][1] - bikes[bikeIndex][1]);
                    // Calculate the new state removing the current bike from the combination of bikes
                    const prevMask = mask ^ (1 << bikeIndex);

                    // Update minimum distance for current worker with the current combination of bikes
                    minDistances[workerIndex][mask] = Math.min(minDistances[workerIndex][mask],
                                                               minDistances[workerIndex - 1][prevMask] + distance);
                }
            }
        }
    }
    // Return the minimum distance for all workers using all bikes
    return Math.min(...minDistances[numWorkers]);
}
```

Time and Space Complexity

The provided code snippet appears to solve an assignment problem using bit masking and dynamic programming to match `workers` with `bikes` in a way that minimizes the sum of the Manhattan distances between assigned pairs.

Time Complexity

The time complexity of the algorithm can be determined by analyzing the nested loops within the code.

- The outer most loop runs `n` times, where `n` is the number of workers. This loop iterates through each worker starting at 1: `for i in range(1, n + 1):`.

- The second loop runs through all combinations of bike assignments using a bitmask, with `1 << m` possible combinations (where `m` is the number of bikes). This is a loop over subsets: `for j in range(1 << m):`.

- The innermost loop iterates over each bike `m` times: `for k in range(m):`.

Considering the loops are nested, we need to multiply their time complexities. So, the time complexity of this algorithm is `O(n * 2^m * m)`, where `n` is the number of workers and `m` is the number of bikes.

Space Complexity

For space complexity, we look at the amount of memory allocated for the algorithm, which primarily depends on the size of the `f` list.

The list `f` is initialized as a two-dimensional list with size `(n+1) x (1 << m)`. This means there's an element for each worker and each combination of bike assignments. So, the space complexity is `O(n * 2^m)` since this is the largest data structure that holds the state of the problem.