519. Random Flip Matrix

July 26, 2018 | 2.6K views

You are given the number of rows `n_rows` and number of columns `n_cols` of a 2D binary matrix where all values are initially 0. Write a function `flip` which chooses a 0 value uniformly at random, changes it to 1, and then returns the position `[row.id, col.id]` of that value. Also, write a function `reset` which sets all values back to 0. **Try to minimize the number of calls to system's Math.random()** and optimize the time and space complexity.

Note:

1. `1 <= n_rows, n_cols <= 10000`
2. `0 <= row.id < n_rows` and `0 <= col.id < n_cols`
3. `flip` will not be called when the matrix has no 0 values left.
4. the total number of calls to `flip` and `reset` will not exceed 1000.

Example 1:

```
Input:
["Solution","flip","flip","flip","flip"]
[[2,3],[],[],[],[]]
Output: [null,[0,1],[1,2],[1,0],[1,1]]
```

Example 2:

```
Input:
["Solution","flip","flip","reset","flip"]
[[1,2],[],[],[],[]]
Output: [null,[0,0],[0,1],null,[0,0]]
```

Explanation of Input Syntax:

The input is two lists: the subroutines called and their arguments. `Solution`'s constructor has two arguments, `n_rows` and `n_cols`. `flip` and `reset` have no arguments. Arguments are always wrapped with a list, even if there aren't any.

Solution

Preface:

Because the matrix is given to us in an abstract way, we must design our own representation of the matrix. In addition to being accurate, we would want our representation to have the following properties:

1. It utilizes less space than $O(\texttt{n\_rows} \cdot \texttt{n\_cols})$, since $\texttt{n\_rows}$ and $\texttt{n\_cols}$ can be very large.
2. It allows us to generate a random 0 position efficiently, using significantly less than $O(\texttt{n\_rows} \cdot \texttt{n\_cols})$ time and only one call to the random number generator.

Described below are two different representations which have these properties.

Approach 1: Virtual Array

Intuition

Let assume that we are given an array-like data structure V of size $\texttt{n\_rows} \cdot \texttt{n\_cols}$, where all cells $V[i]$ have been initialized to ith entry in the matrix, and each assignment and access takes $O(1)$ time.

This data structure can encode the state of the matrix. How can it encode the state of the matrix in a way that enables efficient generation of random 0 positions?

Also, since this data structure isn't actually given to us, how can we construct this data structure in less than $O(\texttt{n\_rows} \cdot \texttt{n\_cols})$ time?

Algorithm

Data structure V, described above, obviously cannot be constructed in less than $O(\texttt{n\_rows} \cdot \texttt{n\_cols})$ time if the initializations are explicit. We must approach it in another way.

We create a new data type which is a slight modification of HashMap, where access to uninitialized key k will initialize $V[k]$ to the kth entry in the matrix and then return it, rather than throwing an error. In this way, it is implied that all $V[k]$ are initialized to kth entry in the matrix.

Let the number of 0 entries remaining in the matrix be denoted as $\texttt{rem}$. As we perform `flip` and reset operations, we update V to maintain the invariant that $V[0] \dots V[\texttt{rem} - 1]$ map to all 0 entries and $V[\texttt{rem}] \dots V[\texttt{n\_rows} \cdot \texttt{n\_cols} - 1]$ map to all 1 entries.

`flip` will change the 0 entry stored at $V[k]$ to 1, where k is a random integer in the range $[0, \texttt{rem})$. Then, it will decrement rem and swap $V[k]$ with $V[\texttt{rem}]$.

reset will clear V of all assigned values and set $\texttt{rem}$ to $\texttt{n\_rows} \cdot \texttt{n\_cols}$.

C++ Java Copy

```
class Solution {
public:

    unordered_map<int,int> V;
    int nr, nc, rem;

    //c++11 random integer generation
    mt19937 rng(random_device{}());
    //uniform random integer in [0, bound)
    int randint(int bound) {
        uniform_int_distribution<int> uni(0, bound);
        return uni(rng);
    }

    Solution(int n_rows, int n_cols) {
        nr = n_rows, nc = n_cols, rem = nr * nc;
    }

    vector<int> flip() {
        int r = randint(--rem);
        int x = V.count(r) ? V[r] : V[r] = r;
        V[r] = V.count(rem) ? V[rem] : V[rem] = rem;
        return {x / nc, x % nc};
    }

    void reset() {
        V.clear();
```

Complexity Analysis

- Time Complexity: $O(1)$ preprocessing. $O(1)$ flip. $O(\min(\texttt{F}, \texttt{n\_rows} \cdot \texttt{n\_cols}))$ reset, where $\texttt{F}$ is the total number of flips.
- Space Complexity: $O(\min(\texttt{F}, \texttt{n\_rows} \cdot \texttt{n\_cols}))$.

Approach 2: Square-Root Decomposition

Intuition

Say that we have $\texttt{rem}$ 0 entries left in the matrix and have randomly chosen the kth 0 entry to be flipped, where $0 \leq k < \texttt{rem}$. Traversing through all $\texttt{n\_rows} \cdot \texttt{n\_cols}$ cells to find its position is too costly. If we split the matrix into roughly $\sqrt{\texttt{n\_rows} \cdot \texttt{n\_cols}}$ contiguous groups of roughly size $\sqrt{\texttt{n\_rows} \cdot \texttt{n\_cols}}$ each, we can find the containing group in $O\left(\sqrt{\texttt{n\_rows} \cdot \texttt{n\_cols}}\right)$ time and then search through that group in $O\left(\sqrt{\texttt{n\_rows} \cdot \texttt{n\_cols}}\right)$ time to find the kth 0 entry.

Algorithm

Create roughly $\sqrt{\texttt{n\_rows} \cdot \texttt{n\_cols}}$ buckets, and have the ith entry in the matrix belong to bucket number $\frac{i}{\sqrt{\texttt{n\_rows} \cdot \texttt{n\_cols}}}$.

Each bucket has a size attribute which represents the number of entries in the matrix that map to it. Also, each bucket tracks which of its entries are 1-valued vs 0-valued by storing its 1-valued entries in a HashSet.

To find the kth remaining 0 entry in the matrix, loop through the list of buckets and use the size and the count of 1s in each bucket to calculate which bucket contains the kth remaining 0 entry. Then, loop through all entries which belong to this bucket, checking which are 0-valued and which are not, to find out what the kth remaining 0 entry is.

C++ Java Copy

```
class Solution {
public:

    int nr, nc, rem, b_size;
    vector<unordered_set<int>> buckets;

    //c++11 random integer generation
    mt19937 rng(random_device{}());
    //uniform random integer in [0, bound)
    int randint(int bound) {
        uniform_int_distribution<int> uni(0, bound - 1);
        return uni(rng);
    }

    Solution(int n_rows, int n_cols) {
        nr = n_rows, nc = n_cols, rem = nr * nc;
        b_size = sqrt(nr * nc);
        for (int i = 0; i < nr * nc; i += b_size)
            buckets.push_back({});
    }

    vector<int> flip() {
        int c = 0;
        int c0 = 0;
        int k = randint(rem);
        for (auto& b1 : buckets) {
            if (c0 + b_size - b1.size() > k) {
```

Complexity Analysis

- Time Complexity: $O\left(\sqrt{\texttt{n\_rows} \cdot \texttt{n\_cols}}\right)$ preprocessing. $O\left(\sqrt{\texttt{n\_rows} \cdot \texttt{n\_cols}}\right)$ flip. $O(\sqrt{\texttt{n\_rows} \cdot \texttt{n\_cols}} + \min(\texttt{F}, (\texttt{n\_rows} \cdot \texttt{n\_cols})))$ reset, where $\texttt{F}$ is the total number of flips.
- Space Complexity: $O(\sqrt{\texttt{n\_rows} \cdot \texttt{n\_cols}} + \min(\texttt{F}, (\texttt{n\_rows} \cdot \texttt{n\_cols})))$.

Rate this article:

Previous Next

Comments: 2 Sort By

Type comment here... (Markdown is supported)

Preview Post

hello_world_cn ★285 September 30, 2018 10:31 AM

Optimize approach 1:

```
vector<int> flip() {
        ...
        int x = V.count(r) ? V[r] : r;
```

Read More

0 Share Reply

edaengineer ★328 July 30, 2018 1:17 AM

The above approach-2 is always worse than approach-1, correct?

0 Share Reply

SHOW 1 REPLY