作业04 BloomFliter

> 在本次作业中需要自行实现一个简单的 Bloom Filter，并按照实验步骤探究 Bloom Filter 各项参数与误报率（False Positive）的关系。

一、设计思路

（a）哈希函数的设置

我测试了一下关于stl里面的Hash函数，发现 `Hash<int> hash_int 的这个函数，效果非常不好。他直接返回的就是 `hash_int(i) == i`，这显然没有任何意义，如果我插入1-100的数据，然后检测101-200的数据，这显然没一个命中的，所以我对于哈希函数进行了重置！由于我另一门课的云操作系统刚做完Lab1里面的网络数据包加密正好用到了字符串加密，所以我就借鉴了一下这个思想。这个函数如下：

```
// 三个参数，第一个是要hash的X，第二个是第i个Hash函数
// 根据PDF要求，hash_i = hash(x+i)
int myHashI(int x,  int times, int limit){
    int num = x + times;
    string num_str = to_string(num);
    return hash_str(num_str) % limit;
}
```

- 函数会界接受三个参数，第一个是要hash的X，第二个是第i个Hash函数。
- 根据作业要求，后续的哈希函数$H_i(x) = H_1(x+i)$可由第一个哈希函数简单变化生成
- 为了让这个哈希函数能够控制范围为$[0, limit-1]$，我通过去模的方式进行处理。

（b）布隆过滤器

- 整个过滤器的构建其实不是很难，只需要对外暴露相应的元素操作的函数接口即可！
- 在实例初始化的时候，需要指定哈希数组的大小、哈希函数的个数。实例化的时候会根据数组大小动态创建数组，回收的时候delete。
- 然后在插入一个元素的时候，会依次计算hash，把数组对应的元素修改一下。
- 查找元素的时候，会到对应的数组里面的去看（计算出来哈希的位置是否都是1），如果都是1说明可能是已经存在的，反之就是不存在的，把这个结果返回

```
class BloomFilter{
private:
    int m;  // 哈希数组的大小
    int k;  // hash函数的个数
    int *data;
public:
    BloomFilter(int set_m, int set_k){
        m = set_m;
        k = set_k;
        data = new int[set_m];

```

```
        for(int i = 0; i < m; i++){
            data[i] = 0;
        }
    }
    ~BloomFilter(){
        if(data != NULL)
            delete[] data;
    }
    // 插入一个元素
    void insertNum(int num){
        for(int i = 0; i < k; i++){
            int hashed_val = myHashI(num, i, m);
            data[hashed_val] = 1;
            //cout << "set hashed val to 1 " << hashed_val << endl;
        }
    }
    // 返回是否存在某个元素
    bool findNum(int num){
        //cout << "try find! ############ "  << endl;
        for(int i = 0; i < k; i++){
            int hashed_val = myHashI(num, i, m);
            //cout << "try find! hashed val is : " << hashed_val << endl;
            if(data[hashed_val] == 0){
                return false;
            }
        }
        return true;
    }
};
```

（c）更优雅的输出表格

- 为了让表格能够更优雅的输出（类似MySQL的输出，表格能够根据内容长度适度伸展），我百度了一下，找到了一个开源库！ **p-ranav/tabulate: Table Maker for Modern C++ (github.com)**
- 然后为了把库导入：
  - 首先需要引入头文件，并且把 `/include/tabulate` 文件夹移动到项目目录里面：

    ```
    #include "tabulate/table.hpp"
    ```

  - 然后，在编译的时候，需要加上包含的 `include` 文件夹

    ```
    g++ -std=c++17 ./main.cpp -o main -I ./
    ```

  - 补充：根据项目要求，必须使用c++17，才能让这个库正常使用。
- 为了验证这个库效果如何。编写一个测试代码：

```cpp
#include <tabulate/table.hpp>
using namespace tabulate;

int main() {

  Table universal_constants;
  universal_constants.add_row({"Quantity", "Value"});
  std::cout << universal_constants << std::endl;
}
```

- 如何动态的插入元素呢，我仔细检查了他的源代码，add_row必须要传入一个Row_t类型！

```cpp
tabulate::Table resultTable;
using Row_t = std::vector<variant<std::string, const char *, string_view,
tabulate::Table>>;
Row_t t;
t.push_back("1");
t.push_back("1");
t.push_back("1");
resultTable.add_row(t);
```

（d）makefile的编写

- 这次我尝试了一下自己编写makefile的文件，尽管可能写的不是很好。

```
main:main
  g++ -std=c++17 ./main.cpp -o main -I ./
clean:
  rm ./main
run:
  ./main
```

二、测试思路

（a）实现单次测试的抽象

- 要测试这个布隆过滤器，我们把每次一组（m、n、k）抽象为一次测试
- 我测试的是会分两组生产数据，并**确保两组数据完全不一样！不会重复**
  - 一组用来插入，生成的数量根据n来决定
  - 一组用来测试，测试的数据要保证绝对不能曾经被插入
  - 此外不仅要保证组之间的数据具有唯一性，组内的数据也必须要有唯一性。
- 把数据生成好了之后，我们只需要检测**测试组**里面的数据，用我自己编写的查找函数，并统计有多少个找到了。
- 为了让测试数据的量也具有可控制性，noneExitNum给上层提供设置的权利
- 最终，返回错误的概率$P = \frac{\text{测试组被误判为存在的数量}}{\text{测试组数据的数量}}$

```cpp
double bloomTest(int m, int n, int k, int noneExitNum){
    BloomFilter bloomFilter(m, k);
    map<int,bool> preparedData;
```

```
    vector<int> noneExistNumData;
    for(int i = 0; i < n; i++){
        int randVal = rand();
        while(preparedData.count(randVal) == 1){
            randVal = rand();
        }
        preparedData[randVal] = 1;
        bloomFilter.insertNum(randVal);
    }

    for(int i = 0; i < noneExitNum; i++){
        int randVal = rand();
        while(preparedData.count(randVal) == 1){
            randVal = rand();
        }
        noneExistNumData.push_back(randVal);
    }

    int wrongNum = 0;
    for(int i = 0; i < noneExitNum; i++){
        if(bloomFilter.findNum(noneExistNumData[i]) == true){
            wrongNum++;
        }
    }

    // return wrongNum;
    return wrongNum * 1.0 / noneExitNum;
}
```

（b）实现整次测试的抽象

- 首先，因为我在测试的时候，主要依赖于随机数的生成！随机数依赖于时间，所以，需要初始化种子。
- 然后，我们要设置好对应的k, m, n。由于作业PDF里面$\frac{m}{n}$可以取值为$[2, 8]$，所以我决定固定$m = 403200$，为啥呢，这玩野可以正好被$[2, 8]$所有的整数整除！
- 剩下的就只是调用函数的事情了：`bloomTest(m, n, k, 10000)`
- 代码如下！

```
void runTest(){
    srand(time(NULL));
    int m = 403200;
    using Row_t = std::vector<variant<std::string, const char *, string_view,
tabulate::Table>>;
    tabulate::Table resultTable;
    Row_t row_header;
    row_header.push_back("data");
    for(int k = 1; k <= 8; k++){
        string str = "k = " + to_string(k);
        row_header.push_back(str);
    }
    resultTable.add_row(row_header);

```

```
14        for(int i = 2; i <=8; i++){
15            Row_t row_data;
16            string str = "m/n = " + to_string(i);
17            row_data.push_back(str);
18            for(int k = 1; k <= 8; k++){
19                int n = m / i;
20                // cout << bloomTest(m, n, k, 10000) << "\t";
21                row_data.push_back(to_string(bloomTest(m, n, k, 10000)));
22            }
23            resultTable.add_row(row_data);
24        }
25        cout << resultTable << endl;
26    }
27
```

四、实验结果：

实验结果如下表格所示（可以说和助教给出的数据基本高度一致）：

```
ziqianzhang@ziqiandeMac-Studio Code % make run
./main
+---------+----------+----------+----------+----------+----------+----------+----------+----------+
| data    | k = 1    | k = 2    | k = 3    | k = 4    | k = 5    | k = 6    | k = 7    | k = 8    |
+---------+----------+----------+----------+----------+----------+----------+----------+----------+
| m/n = 2 | 0.388600 | 0.405100 | 0.473100 | 0.556700 | 0.649300 | 0.744000 | 0.804600 | 0.863500 |
+---------+----------+----------+----------+----------+----------+----------+----------+----------+
| m/n = 3 | 0.286300 | 0.240800 | 0.251200 | 0.294700 | 0.359200 | 0.414600 | 0.487800 | 0.557700 |
+---------+----------+----------+----------+----------+----------+----------+----------+----------+
| m/n = 4 | 0.218500 | 0.151600 | 0.147300 | 0.167500 | 0.186300 | 0.215700 | 0.260800 | 0.305800 |
+---------+----------+----------+----------+----------+----------+----------+----------+----------+
| m/n = 5 | 0.184100 | 0.110200 | 0.097600 | 0.093300 | 0.098800 | 0.119100 | 0.141500 | 0.156500 |
+---------+----------+----------+----------+----------+----------+----------+----------+----------+
| m/n = 6 | 0.152100 | 0.076600 | 0.058100 | 0.051600 | 0.055600 | 0.063600 | 0.079200 | 0.083700 |
+---------+----------+----------+----------+----------+----------+----------+----------+----------+
| m/n = 7 | 0.128900 | 0.067200 | 0.042400 | 0.035900 | 0.034600 | 0.035700 | 0.043500 | 0.048800 |
+---------+----------+----------+----------+----------+----------+----------+----------+----------+
| m/n = 8 | 0.120400 | 0.049500 | 0.029500 | 0.021500 | 0.023000 | 0.021900 | 0.023700 | 0.023300 |
+---------+----------+----------+----------+----------+----------+----------+----------+----------+
ziqianzhang@ziqiandeMac-Studio Code %
```

这个是给出的标准值：

m/n	*k*	*k*=1	*k*=2	*k*=3	*k*=4	*k*=5	*k*=6	*k*=7	*k*=8
2	1.39	0.393	0.400						
3	2.08	0.283	0.237	0.253					
4	2.77	0.221	0.155	0.147	0.160				
5	3.46	0.181	0.109	0.092	0.092	0.101			
6	4.16	0.154	0.0804	0.0609	0.0561	0.0578	0.0638		
7	4.85	0.133	0.0618	0.0423	0.0359	0.0347	0.0364		
8	5.55	0.118	0.0489	0.0306	0.024	0.0217	0.0216	0.0229	

观察规律总结如下：

- 当$\frac{m}{n}$的值固定的时候，随着k的增加，误报率先逐渐递减，然后再逐渐递增。
- 根据理论推到的结果，极值点在$k = ln2(\frac{m}{n})$的时候取到最小值，具体在表中的数据反应就是：m/n=2的时候，大概在k=2的时候最小，m/n=4的时候，大概在k=3的时候最小0.14左右。

- 从理论分析：为什么k太大了或者k太小了都不好呢?
  - 如果k太小了每次插入只会标记数组里面的1个位置，不同的元素比较容器出现哈希碰撞，哈希碰撞的概率大大提高，所以K不能太小。
  - 如果k太大了也不好，每次插入标记了太多的元素（试想一下整个表都全部标记了！）那哈希冲突也会发生。
  - 所以基于此来说，K不能太大，也不能太小。
- 当K的值固定的时候，随着m/n的增加，发现误报率逐渐增大，这是为什么呢？因为插入的元素越来越多，到了后面显然就更加容易冲突了，这也是符合我们的理论认知的。

附录：完整代码

```cpp
#include<iostream>
#include <functional>
#include <vector>
#include <string>
#include <map>
#include <time.h>
#include "tabulate/table.hpp"

using namespace std;
std::hash<string> hash_str;

// 三个参数，第一个是要hash的X，第二个是第i个Hash函数
// 根据PDF要求，hash_i = hash(x+i)
int myHashI(int x,  int times, int limit){
    int num = x + times;
    string num_str = to_string(num);
    return hash_str(num_str) % limit;
}


class BloomFilter{
private:
    int m;  // 哈希数组的大小
    int k;  // hash函数的个数
    int *data;
public:
    BloomFilter(int set_m, int set_k){
        m = set_m;
        k = set_k;
        data = new int[set_m];

        for(int i = 0; i < m; i++){
            data[i] = 0;
        }
    }
    ~BloomFilter(){
        if(data != NULL)
            delete[] data;
    }
```

```
40        // 插入一个元素
41        void insertNum(int num){
42            for(int i = 0; i < k; i++){
43                int hashed_val = myHashI(num, i, m);
44                data[hashed_val] = 1;
45                //cout << "set hashed val to 1 " << hashed_val << endl;
46            }
47        }
48        // 返回是否存在某个元素
49        bool findNum(int num){
50            //cout << "try find! ############ "  << endl;
51            for(int i = 0; i < k; i++){
52                int hashed_val = myHashI(num, i, m);
53                //cout << "try find! hashed val is : " << hashed_val << endl;
54                if(data[hashed_val] == 0){
55                    return false;
56                }
57            }
58            return true;
59        }
60    };
61
62    // 不妨固定 m = 403200 (1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 = 40320, 然后我再乘以了100)
63    // m/n      2      3         4       5       6       7       8
64    // n      202600  134400   100800   80640   67200   57600   50400
65
66    double bloomTest(int m, int n, int k, int noneExitNum){
67        BloomFilter bloomFilter(m, k);
68        map<int,bool> preparedData;
69        vector<int> noneExistNumData;
70        for(int i = 0; i < n; i++){
71            int randVal = rand();
72            while(preparedData.count(randVal) == 1){
73                randVal = rand();
74            }
75            preparedData[randVal] = 1;
76            bloomFilter.insertNum(randVal);
77        }
78
79        for(int i = 0; i < noneExitNum; i++){
80            int randVal = rand();
81            while(preparedData.count(randVal) == 1){
82                randVal = rand();
83            }
84            noneExistNumData.push_back(randVal);
85        }
86
87        int wrongNum = 0;
88        for(int i = 0; i < noneExitNum; i++){
89            if(bloomFilter.findNum(noneExistNumData[i]) == true){
90                wrongNum++;
91            }
```

```
92          }
93
94          // return wrongNum;
95          return wrongNum * 1.0 / noneExitNum;
96      }
97
98
99
100     void runTest(){
101         srand(time(NULL));
102         int m = 403200;
103         using Row_t = std::vector<variant<std::string, const char *, string_view,
    tabulate::Table>>;
104         tabulate::Table resultTable;
105         Row_t row_header;
106         row_header.push_back("data");
107         for(int k = 1; k <= 8; k++){
108             string str = "k = " + to_string(k);
109             row_header.push_back(str);
110         }
111         resultTable.add_row(row_header);
112
113         for(int i = 2; i <=8; i++){
114             Row_t row_data;
115             string str = "m/n = " + to_string(i);
116             row_data.push_back(str);
117             for(int k = 1; k <= 8; k++){
118                 int n = m / i;
119                 // cout << bloomTest(m, n, k, 10000) << "\t";
120                 row_data.push_back(to_string(bloomTest(m, n, k, 10000)));
121             }
122             resultTable.add_row(row_data);
123         }
124         cout << resultTable << endl;
125     }
126
127
128     int main(){
129         runTest();
130         return 0;
131     }
```