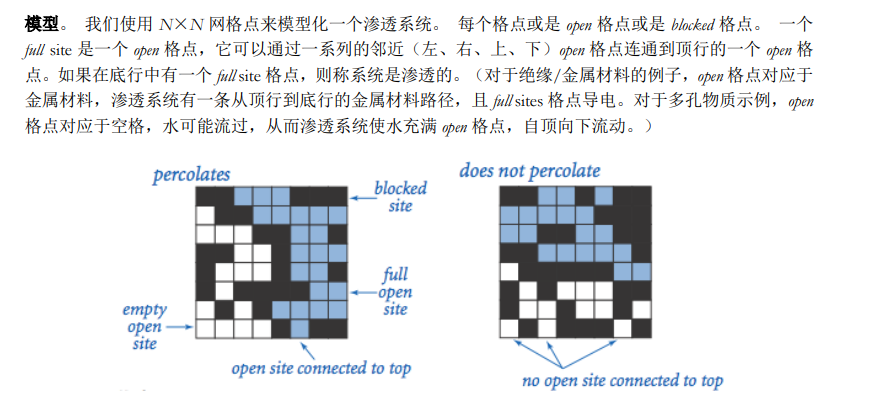
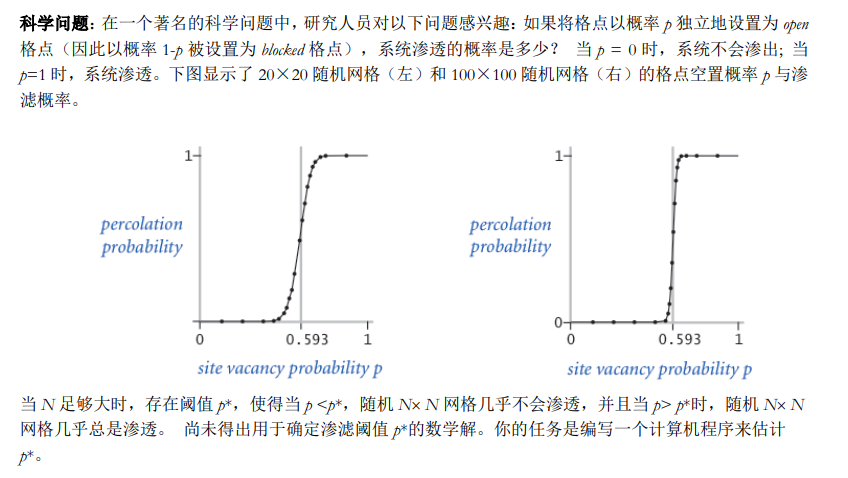
**实验一 渗透问题**

1. 实验描述：

使用合并-查找（union-find）数据结构，编写程序通过蒙特卡罗模拟（Monte Carlo simulation）来估计渗透阈值。







1. 代码实现：

考虑到我对C++的熟练程度更高，在本次实验中我使用的编程语言为C++。

下面是一些在代码中可能用到的变量和头文件准备：

#include <iostream>

#include <utility>

#include <random>

#include <array>

#include <algorithm>

#include <numeric>

#include <cmath>

#include <time.h>

#include <memory>

using namespace std;

const size\_t blocks\_size = 200; *// 用200\*200的网格模拟渗透*

array<array<bool, blocks\_size>, blocks\_size> blocks{};

unsigned seed = 0x89e351ef534eu; *// 固定的随机数种子，以便于复现结果*

default\_random\_engine random\_engine{ seed };

uniform\_int\_distribution<int> random\_generator{ 0, blocks\_size - 1 }; *//采取均匀分布的随机方式*

在本次实验中，我们需要比较不同的并查集方法之间的性能差异，因此我们选择将并查集封装为函数，便于代码的复用，下面是quick\_find的代码：

class union\_find\_set

{

    unique\_ptr<int[]> parents; *//使用智能指针表示网格*

public:

    union\_find\_set(size\_t size)

        : parents(new int[size])

    {

        for (size\_t i = 0; i < size; i++)

            parents[i] = i;

    }

    int find(int i)

    {

        if (i != parents[i])

            parents[i] = find(parents[i]);

        return parents[i];

    }

    void connect(int i, int j)

    {

        parents[find(i)] = find(j);

    }

    bool is\_connected(int i, int j)

    {

        return find(i) == find(j);

    }

};

在完成了并查集代码之后，我们需要一个函数来开始我们的模拟。由于我们所需要模拟的是一个二维网络，而我们所设计的并查集算法只能应用于一维数组，因此我们的网络必须用一维数组来表示，这在准备工作代码部分中已经有所体现。基于此，我们使用特定的方式将其表示为一个二维网格。该模拟函数的返回值为最终我们需要得到的渗透阈值，下面是其代码：

double simulate()

{

    auto calculate\_index = [](int row, int col)

    { return row \* blocks\_size + col; };

    blocks.fill(array<bool, blocks\_size>{});

    union\_find\_set ufs{ blocks\_size \* blocks\_size + 2 };

    int first = blocks\_size \* blocks\_size;

    int last = first + 1;

    for (size\_t i = 0; i < blocks\_size; i++)

    {

        ufs.connect(first, calculate\_index(0, i));

        ufs.connect(last, calculate\_index(blocks\_size - 1, i));

    }

    size\_t space\_count = 0;

    while (!ufs.is\_connected(first, last))

    {

        bool flag = true;

        do

        {

            int row = random\_generator(random\_engine);

            int col = random\_generator(random\_engine);

            if (!blocks[row][col])

            {

                flag = false;

                space\_count++;

                blocks[row][col] = true;

                if (row > 0 && blocks[row - 1][col])

                    ufs.connect(calculate\_index(row, col), calculate\_index(row - 1, col));

                if (col > 0 && blocks[row][col - 1])

                    ufs.connect(calculate\_index(row, col), calculate\_index(row, col - 1));

                if (row < blocks\_size - 1 && blocks[row + 1][col])

                    ufs.connect(calculate\_index(row, col), calculate\_index(row + 1, col));

                if (col < blocks\_size - 1 && blocks[row][col + 1])

                    ufs.connect(calculate\_index(row, col), calculate\_index(row, col + 1));

            }

        } while (flag);

    }

    return static\_cast<double>(space\_count) / (200 \* 200);

}

在该算法的基础上，如果我们想要更换并查集方法，只需要将union\_find\_set类中的函数改写即可。

下面是使用quick\_union算法的union\_find\_set类：

class union\_find\_set

{

    unique\_ptr<int[]> parents; *//使用智能指针定义数组表示网格*

public:

    union\_find\_set(size\_t size)

        : parents(new int[size])

    {

        for (size\_t i = 0; i < size; i++)

            parents[i] = i;

    }

    int find(int i)

    {

        while (i != parents[i]) {

            i = parents[i];

        }

        return i;

    }

    void connect(int i, int j)

    {

        parents[find(i)] = parents[find(j)];

    }

    bool is\_connected(int i, int j)

    {

        return find(i) == find(j);

    }

};

下面是使用weighte\_quick\_union算法的union\_find\_set类：

class union\_find\_set

{

    unique\_ptr<int[]> parents; *//使用智能指针定义数组表示网格*

    unique\_ptr<int[]> treesize;

public:

    union\_find\_set(size\_t size)

        : parents(new int[size]),treesize(new int[size])

    {

        for (size\_t i = 0; i < size; i++)

            parents[i] = i;

        for (size\_t i = 0; i < size; i++) {

            treesize[i] = 1;

        }

    }

    int find(int i)

    {

        while (i != parents[i]) {

            i = parents[i];

        }

        return i;

    }

    void connect(int i, int j)

    {

        int x = find(i);

        int y = find(j);

        if (treesize[x] < treesize[y]) {

            parents[x] = parents[y];

            treesize[y] += treesize[x];

        }

        else {

            parents[y] = parents[x];

            treesize[x] += treesize[y];

        }

    }

    bool is\_connected(int i, int j)

    {

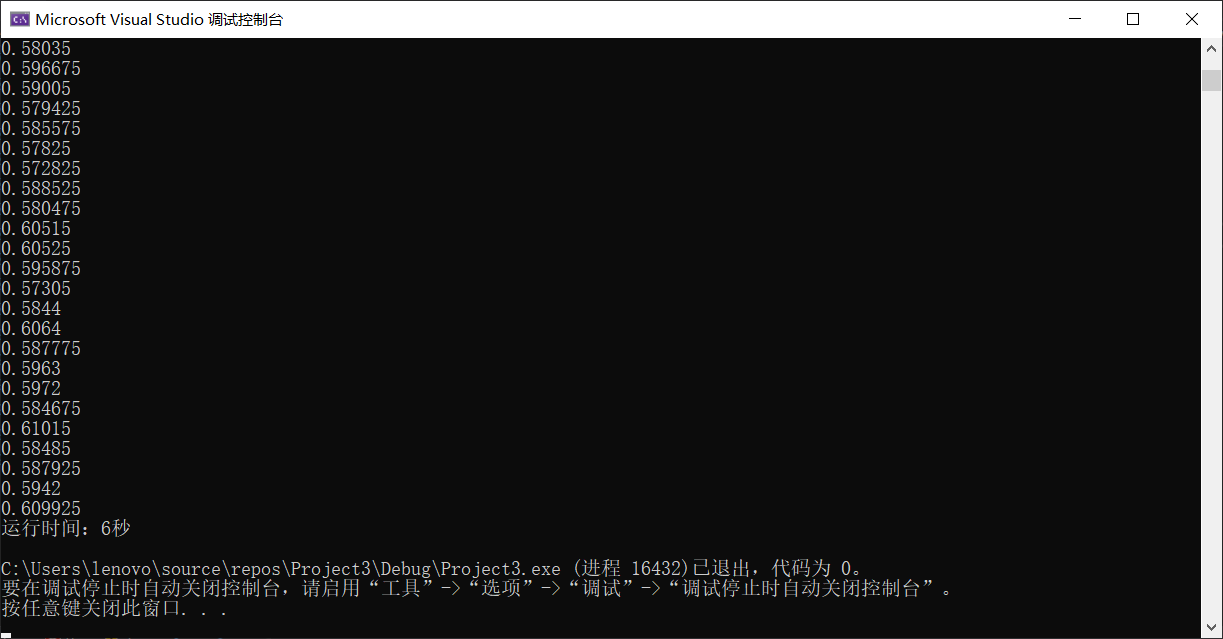
        return find(i) == find(j);

    }

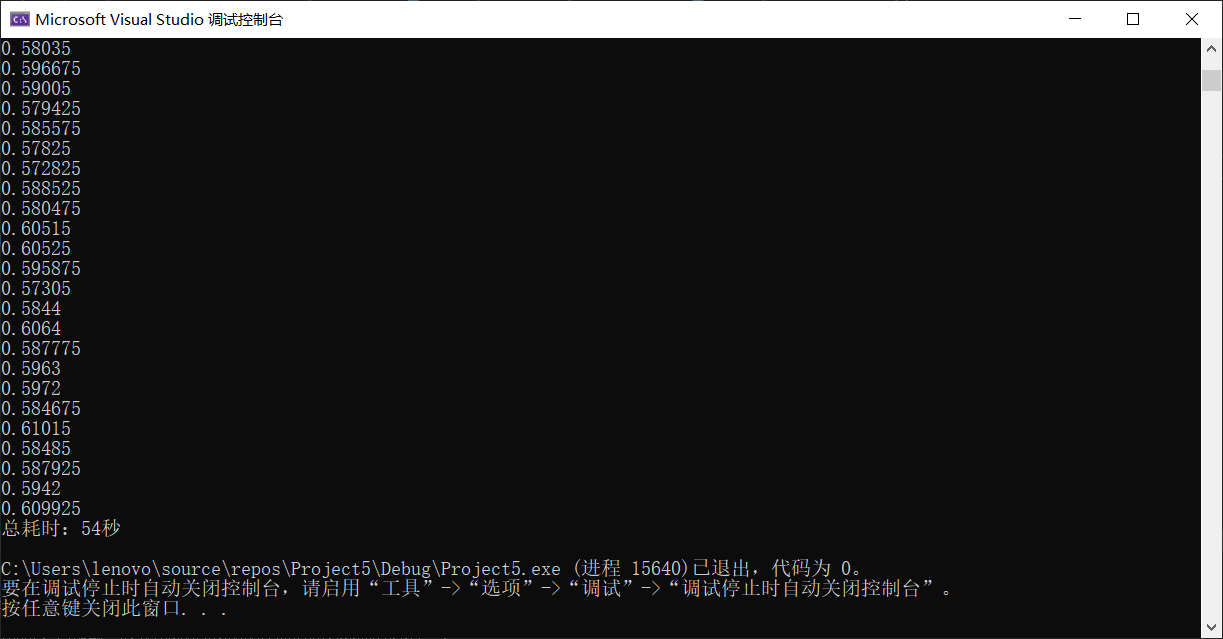
};

三、实验结果：

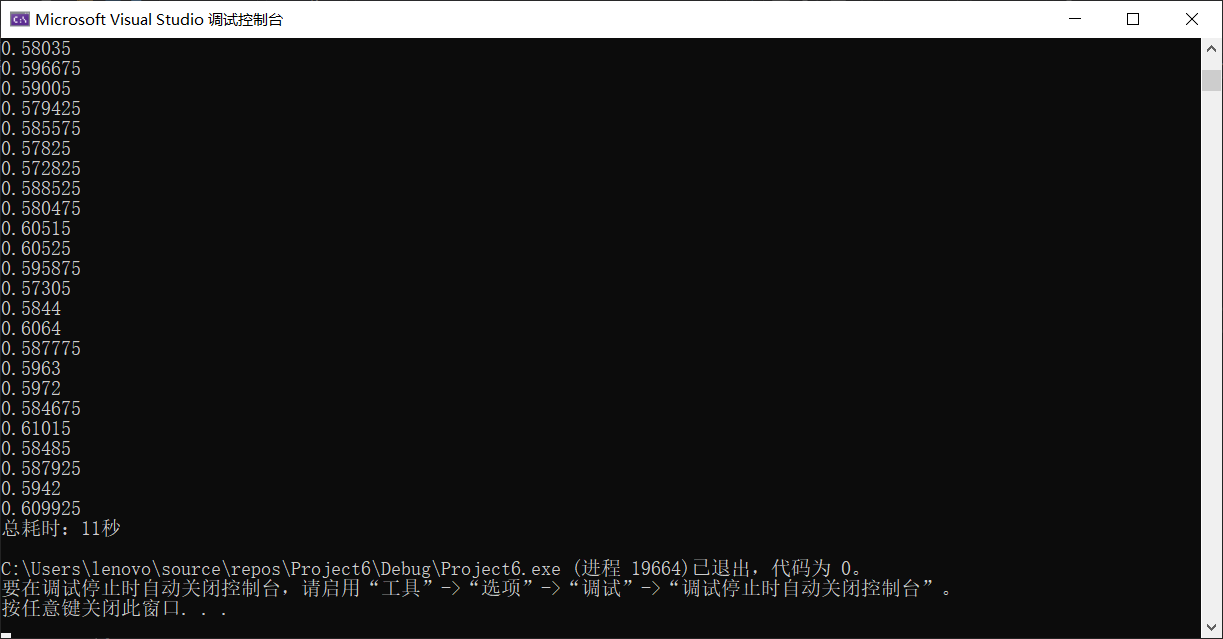
首先是quick\_find的结果分析：由于我们模拟了200次，因此输出了200个结果，最终的执行时间为6秒。



将并查集算法更换为quick\_union之后，输出的结果如下：



再将并查集算法更换为weighted\_quick\_union之后，输出结果为：



四、实验数据分析

我们统计了N=200时各种方法的运行时间对比，我们将其翻倍，分别统计N=400时的运行时间，和N=600时的运行时间对比。通过运行代码我们统计出了如下结果：

|  |  |  |  |
| --- | --- | --- | --- |
|  | N=200 | N=400 | N=600 |
| quick\_find | 6秒 | 23秒 | 53秒 |
| quick\_union | 54秒 | 408秒 | 1341秒 |
| weighted\_quick\_union | 11秒 | 80秒 | 269秒 |

使用近似表示法，我们不难得出：

以上单位均为秒。

最后，我们计算蒙特卡洛模拟中渗透阈值的平均值、方差、标准差以及置信区间，其代码如下：

#include <iostream>

#include <fstream>

#include <vector>

#include <cmath>

using namespace std;

const int N = 10000;

int main()

{

*//读取数据文件*

    ifstream in("simulate.txt", ios::in);

    if (!in.is\_open())

    {

        cout << "open error!" << endl;

        exit(0);

    }

*//将数据文件数据存入数组*

    int i = 0;

    vector<double> a(N);

    while (!in.eof() && i < N)

    {

        in >> a[i];

        i++;

    }

    double sum = 0;

    for (i = 0; i < N; i++) {

        sum += a[i];

    }

    double average = sum / N;

    double σ2 = 0;

    for (i = 0; i < N; i++) {

        σ2 += (a[i] - average) \* (a[i] - average);

    }

    σ2 = σ2 / (N - 1);

    double σ = sqrt(σ2);

    cout << "平均值：" << average << endl;

    cout << "方差：" << σ2 << endl;

    cout << "标准差: " << σ << endl;

    cout << "置信区间: " << "[" << average - 1.96 \* σ / sqrt(N) << ", "

        << average + 1.96 \* σ / sqrt(N) << "]" << endl;

    return 0;

}

运行结果如下：

