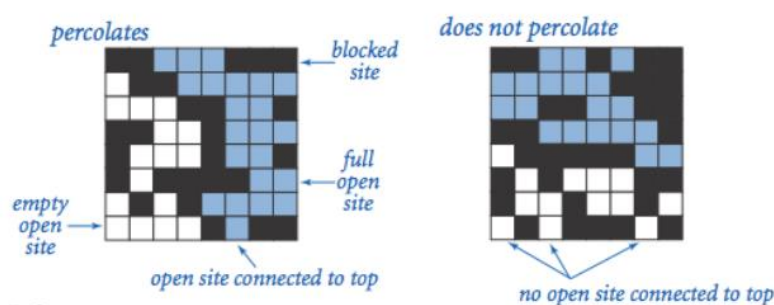


实验一 渗透问题

一、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。

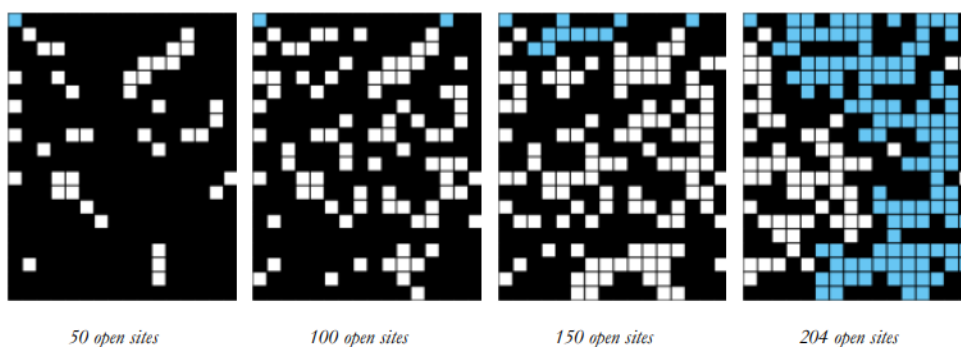
模型。 我们使用 $N \times N$ 网格点来模型化一个渗透系统。每个格点或是 *open* 格点或是 *blocked* 格点。一个 *full site* 是一个 *open* 格点,它可以通过一系列的邻近(左、右、上、下) *open* 格点连通到顶行的一个 *open* 格点。如果在底行中有一个 *full site* 格点,则称系统是渗透的。(对于绝缘/金属材料的例子, *open* 格点对应于金属材料,渗透系统有一条从顶行到底行的金属材料路径,且 *full sites* 格点导电。对于多孔物质示例, *open* 格点对应于空格,水可能流过,从而渗透系统使水充满 *open* 格点,自顶向下流动。)



蒙特卡罗模拟 (Monte Carlo simulation) . 要估计渗透阈值, 考虑以下计算实验:

- 初始化所有格点为 *blocked*。
- 重复以下操作直到系统渗出:
 - 在所有 *blocked* 的格点之间随机均匀选择一个格点 (row i , column j)。
 - 设置这个格点(row i , column j)为 *open* 格点。
- *open* 格点的比例提供了系统渗透时渗透阈值的一个估计。

例如, 如果在 20×20 的网格中, 根据以下快照的 *open* 格点数, 那么对渗透阈值的估计是 $204/400 = 0.51$, 因为当第 204 个格点被 *open* 时系统渗透。



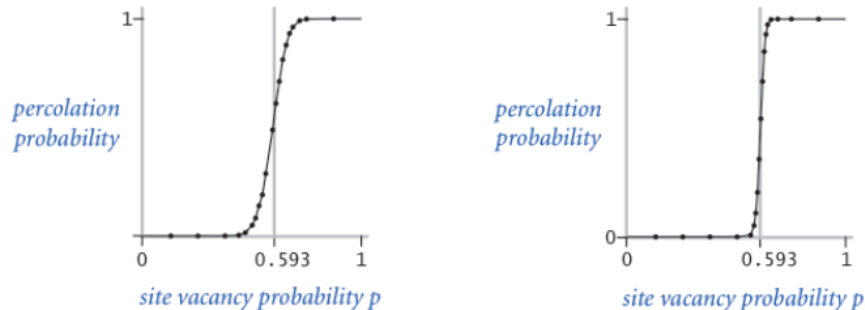
通过重复该计算实验 T 次并对结果求平均值, 我们获得了更准确的渗透阈值估计。令 x_i 是第 i 次计算实验中 *open* 格点所占比例。样本均值 μ 提供渗透阈值的一个估计值; 样本标准差 σ 测量阈值的灵敏性。

$$\mu = \frac{x_1 + x_2 + \dots + x_T}{T}, \quad \sigma^2 = \frac{(x_1 - \mu)^2 + (x_2 - \mu)^2 + \dots + (x_T - \mu)^2}{T-1}$$

假设 T 足够大 (例如至少 30), 以下为渗透阈值提供 95% 置信区间:

$$\left[\mu - \frac{1.96\sigma}{\sqrt{T}}, \mu + \frac{1.96\sigma}{\sqrt{T}} \right]$$

科学问题: 在一个著名的科学问题中, 研究人员对以下问题感兴趣: 如果将格点以概率 p 独立地设置为 *open* 格点 (因此以概率 $1-p$ 被设置为 *blocked* 格点), 系统渗透的概率是多少? 当 $p = 0$ 时, 系统不会渗出; 当 $p=1$ 时, 系统渗透。下图显示了 20×20 随机网格 (左) 和 100×100 随机网格 (右) 的格点空置概率 p 与渗透概率。



当 N 足够大时, 存在阈值 p^* , 使得当 $p < p^*$, 随机 $N \times N$ 网格几乎不会渗透, 并且当 $p > p^*$ 时, 随机 $N \times N$ 网格几乎总是渗透。尚未得出用于确定渗透阈值 p^* 的数学解。你的任务是编写一个计算机程序来估计 p^* 。

二、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];
    }

}

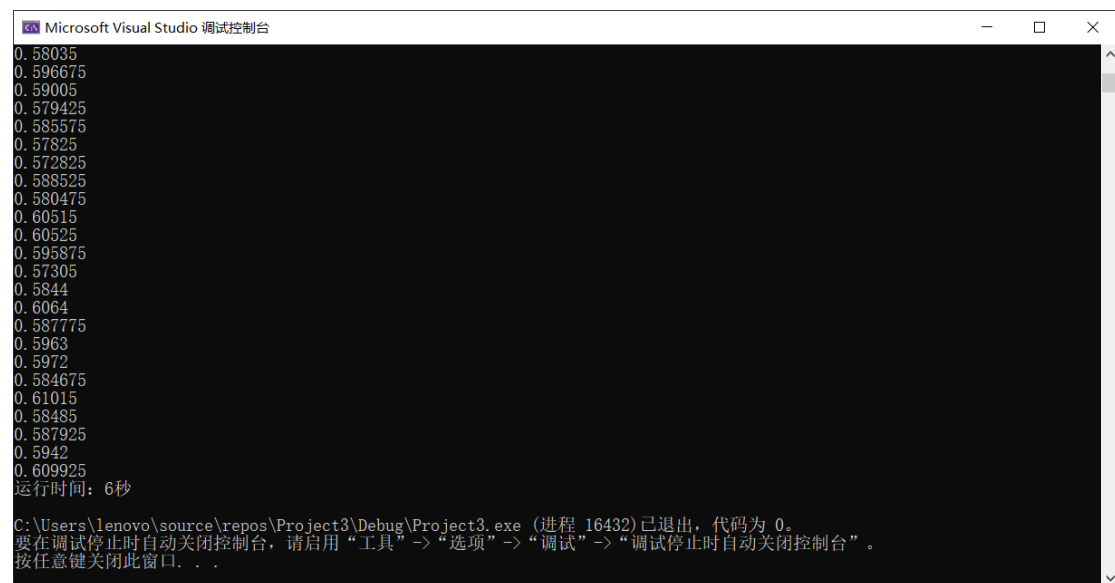
bool is_connected(int i, int j)
{
    return find(i) == find(j);
}

};

```

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。



```

Microsoft Visual Studio 调试控制台
0.58035
0.596675
0.59005
0.579425
0.585575
0.57825
0.572825
0.588525
0.580475
0.60515
0.60525
0.595875
0.57305
0.5844
0.6064
0.587775
0.5963
0.5972
0.584675
0.61015
0.58485
0.587925
0.5942
0.609925
运行时间: 6秒
C:\Users\lenovo\source\repos\Project3\Debug\Project3.exe (进程 16432) 已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。

```

将并查集算法更换为 quick_union 之后，输出的结果如下：

```
Microsoft Visual Studio 调试控制台
0.58035
0.596675
0.59005
0.579425
0.585575
0.57825
0.572825
0.588525
0.580475
0.60515
0.60525
0.595875
0.57305
0.5844
0.6064
0.587775
0.5963
0.5972
0.584675
0.61015
0.58485
0.587925
0.5942
0.609925
总耗时: 54秒
C:\Users\lenovo\source\repos\Project5\Debug\Project5.exe (进程 15640) 已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

再将并查集算法更换为 weighted_quick_union 之后, 输出结果为:

```
Microsoft Visual Studio 调试控制台
0.58035
0.596675
0.59005
0.579425
0.585575
0.57825
0.572825
0.588525
0.580475
0.60515
0.60525
0.595875
0.57305
0.5844
0.6064
0.587775
0.5963
0.5972
0.584675
0.61015
0.58485
0.587925
0.5942
0.609925
总耗时: 11秒
C:\Users\lenovo\source\repos\Project6\Debug\Project6.exe (进程 19664) 已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

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:

$$T_{qf} = 0.75 \times T \times N^2 \times 10^{-6}$$

$$T_{qu} = 1.27 \times T \times N^2 \times \ln N \times 10^{-6}$$

$$T_{wqu} = 0.25 \times T \times N^2 \times \ln N \times 10^{-6}$$

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;

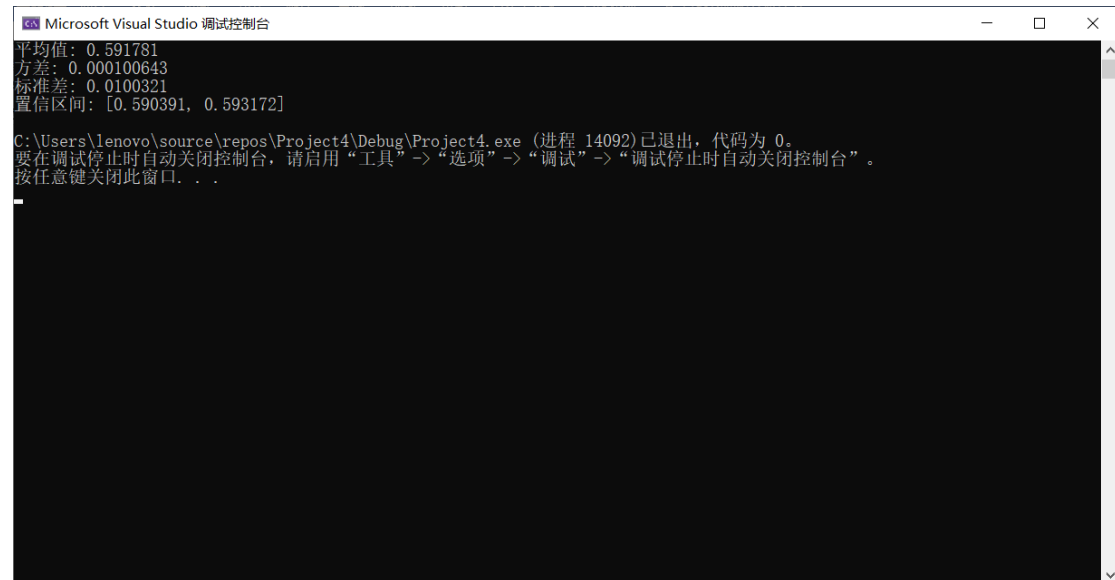
const int N = 10000;

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:



The screenshot shows the Microsoft Visual Studio Debug Console window. The title bar reads "Microsoft Visual Studio 调试控制台". The output text is as follows:

```
平均值: 0.591781  
方差: 0.000100643  
标准差: 0.0100321  
置信区间: [0.590391, 0.593172]  
  
C:\Users\lenovo\source\repos\Project4\Debug\Project4.exe (进程 14092) 已退出, 代码为 0。  
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。  
按任意键关闭此窗口. . .
```