外部排序-k路归并

09022313 李晶晶

问题描述/需求分析

为优化外部排序，对归并路数进行优化。采用多路归并的方式。

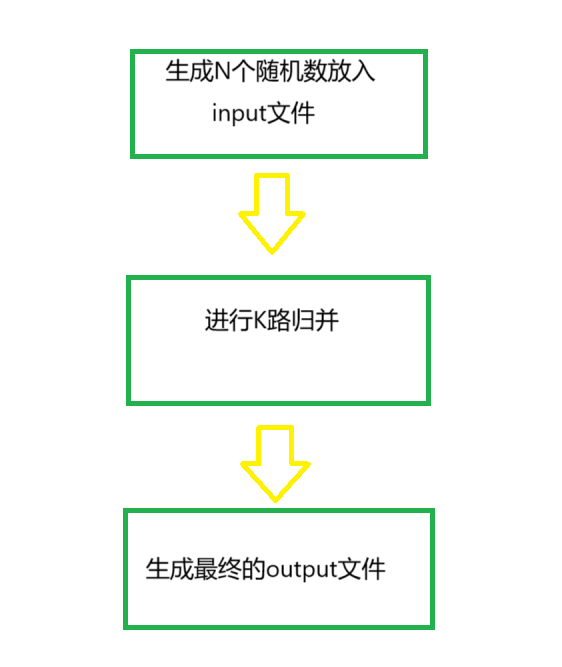
系统结构/算法设计

生成N个随机数放入input文件，设定初始顺串的长度BUFFER\_SIZE，生成初始顺串并存储至临时文件。

通过最小堆对所有临时文件进行排序，生成最终的output文件

功能模块设计

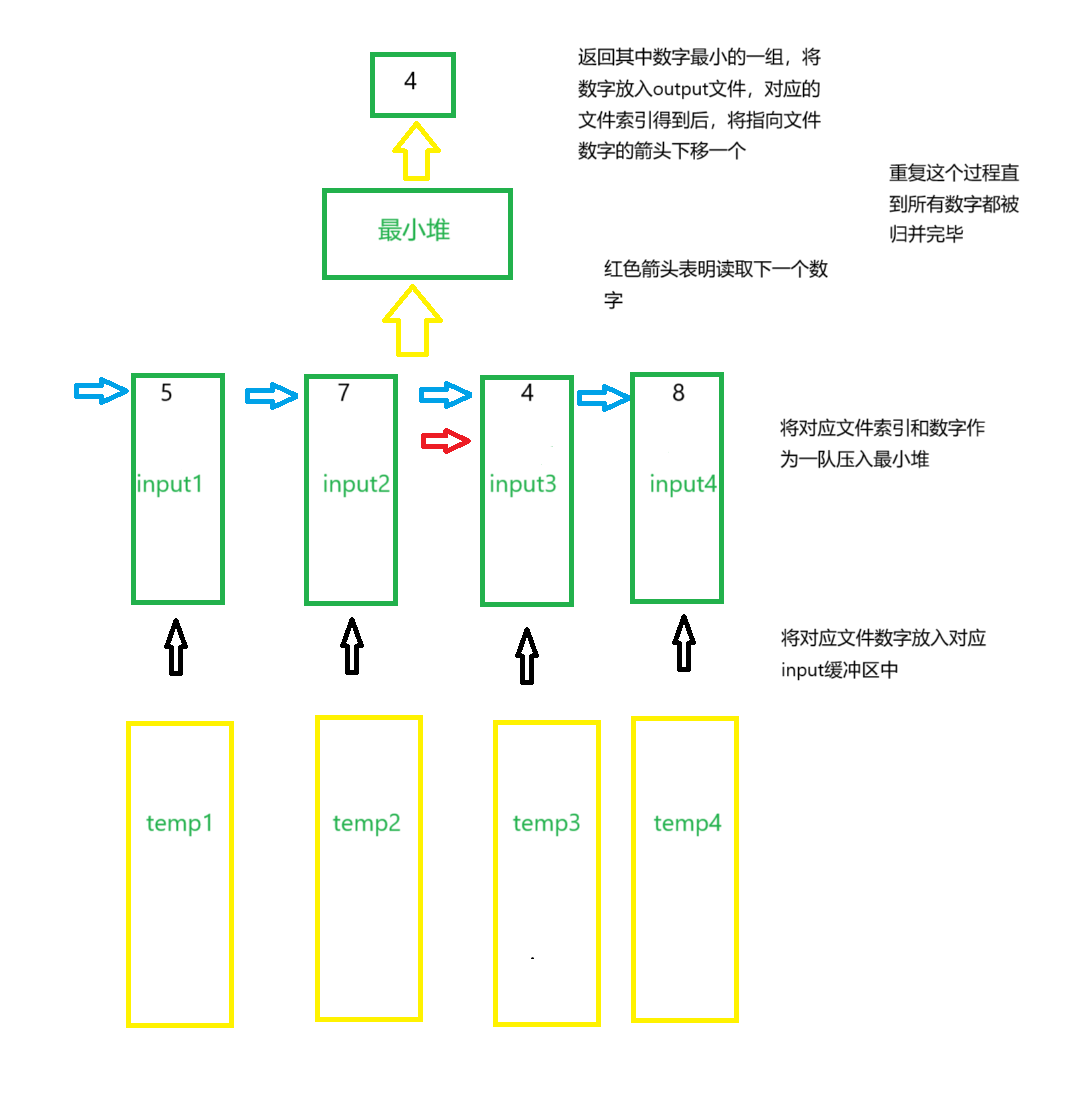
流程



K路归并

图例显示生成的初始顺串temp1-4，放入临时文件temp1-4中，排序时先从temp1-4文件中取出BUFFER\_SIZE大小的数据放入input1-4的buffer中，然后再进行k路归并。

进行k路归并时，将对应文件索引和当前的数字作为一对压入最小堆，再弹出其中最小的数字，将该数字放入outputbuffer中，对应的文件索引可以读入下一个数字，并重复上述过程，如果有buffer数字读空，则从对应文件中补充，如果文件已经读完，则结束。



结果测试与分析

运行结果与预期相符

图形用户界面

中度可信度描述已自动生成

应用程序

低可信度描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

图形用户界面, 文本, 应用程序

描述已自动生成

可以看到，output文件中的数字已经排序完毕。

IO性能分析

表格

描述已自动生成

图中，横坐标为N，纵坐标为IO次数，蓝色线代表优化前，橙色线代表进行顺串生成优化，绿色线代表进行k路归并优化。

从图表中可以非常明显的看到k路归并对于IO次数优化有着很好的作用。

实验总结

事实上，k路归并可以采用最小堆进行处理，过程比较清楚明了。同时在本实验中，采用的k直接为对应的初始顺串数，此时归并的性能会比较好，但在处理外部数据排序时，需要选取好初始顺串大小，避免出现因初始顺串大小过小，初始顺串过多而导致的麻烦。

附录

#include <iostream>

#include <fstream>

#include <vector>

#include <queue>

#include <algorithm>

#include <string>

#include <sstream>

#include <ctime> // 添加时间头文件

using namespace std;

const int BUFFER\_SIZE = 10;

// Function to generate an input file with N random numbers

void generateInputFile(const string& filename, int N)

{

ofstream output(filename);

if (!output.is\_open())

{

cerr << "Error creating input file." << endl;

return;

}

// Seed the random number generator

srand(static\_cast<unsigned int>(time(0)));

for (int i = 0; i < N; ++i)

{

output << rand() % 100 << " ";

}

output.close();

}

void sortAndWriteChunk(ifstream& input, int& chunkIndex, int& ioReadCount, int& ioWriteCount)

{

vector<int> numbers;

int number;

// 从文件中读取数字到缓冲区

while (input >> number)

{

numbers.push\_back(number);

// 达到缓冲区大小后，进行排序并写入临时文件

if (numbers.size() == BUFFER\_SIZE)

{

ioReadCount++;

sort(numbers.begin(), numbers.end());

ofstream out("temp" + to\_string(chunkIndex++) + ".txt");

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

{

out << numbers[i] << " ";

}

ioWriteCount++; // 每次写入文件增加写计数

out.close();

numbers.clear(); // 清空缓冲区

}

}

// 如果有剩余数字，进行最后的排序和写入

if (!numbers.empty())

{

sort(numbers.begin(), numbers.end());

ofstream out("temp" + to\_string(chunkIndex++) + ".txt");

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

{

out << numbers[i] << " ";

}

ioWriteCount++; // 每次写入文件增加写计数

out.close();

}

}

void mergeChunks(int numChunks, const string& outputFile, int& ioReadCount, int& ioWriteCount)

{

auto compare = [](const pair<int, int>& a, const pair<int, int>& b)

{

return a.first > b.first; // 最小堆

};

priority\_queue<pair<int, int>, vector<pair<int, int>>, decltype(compare)> minHeap(compare);

vector<ifstream> chunkFiles(numChunks);

vector<int> currentNumbers(numChunks);

// 打开所有临时文件

for (int i = 0; i < numChunks; ++i)

{

chunkFiles[i].open("temp" + to\_string(i) + ".txt");

if (chunkFiles[i] >> currentNumbers[i])

{

minHeap.push({ currentNumbers[i], i }); // {数字, 文件索引}

}

}

ofstream output(outputFile);

// 归并过程

while (!minHeap.empty())

{

auto topElement = minHeap.top();

int value = topElement.first;

int chunkIndex = topElement.second;

minHeap.pop();

output << value << " "; // 写入最终的有序文件，数字之间用空格分隔

// 从相应的临时文件中读取下一个数字

if (chunkFiles[chunkIndex] >> currentNumbers[chunkIndex])

{

minHeap.push({ currentNumbers[chunkIndex], chunkIndex });

}

}

// 关闭文件

for (int i = 0; i < numChunks; ++i)

{

chunkFiles[i].close();

}

output.close();

ioWriteCount++; // 写入完成时增加写计数

}

int main()

{

const string inputFileName = "input.txt";

const int N = 10000;

// 初始化 IO 计数器

int ioReadCount = 0;

int ioWriteCount = 0;

generateInputFile(inputFileName, N);

ifstream input(inputFileName);

if (!input.is\_open())

{

cerr << "无法打开文件!" << endl;

return 1;

}

// 第一步：分割输入文件并排序

int chunkIndex = 0;

sortAndWriteChunk(input, chunkIndex, ioReadCount, ioWriteCount);

input.close();

// 第二步：归并所有已排序的部分

mergeChunks(chunkIndex, "output.txt", ioReadCount, ioWriteCount);

cout << "排序完成，结果已写入 output.txt" << endl;

cout << "文件读取次数: " << ioReadCount << endl;

cout << "文件写入次数: " << ioWriteCount << endl;

cout << ioReadCount + ioWriteCount << endl;

return 0;

}