3 网络传输任务管理

Requirement

1. 解释代码逻辑和自己使用的方法思路。
2. 分析问题中算法的复杂度。
3. 和暴力解决相比，线段树方案的性能提升分析和实验分析（选做）

数据传输任务受理判断系统：判断新任务是否应该被受理

用线段树（一种特殊的树）来完成

有C台计算机，编号为 1, 2, 3·····，C。

这些计算机被C-1条光纤连接成一条直线型状的链路，光纤编号为 1, 2, 3·····，C-1。

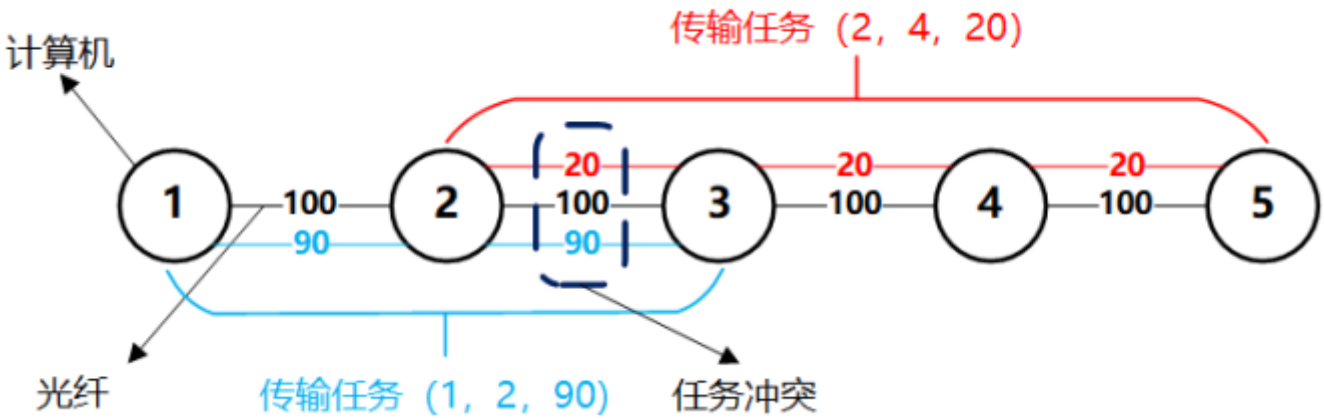
第 i 条光纤连接第 i 个和第 i+1 个计算机。

也就是说，每两个相邻计算机之间用一条光纤连接。

光纤的带宽不同。

例如第 i 条光纤有100M/s 带宽，则第i个计算机与第i+1个计算机之间的通信速率最大为 100M/s。

本题要求用程序实现一个数据传输任务受理判断系统，判断每一个申请的任务是否应该被运行在上述的链路中。



数据传输任务：

一个数据传输任务可以用（sender，receiver，workload）来描述。

sender 表示发送数据的电脑的编号。

receiver表示接受数据电脑的电脑的编号。

worklead表示这个传输这个任务所需的带宽。

其中，1 =< sender <= C, 1 =< receiver <= C, sender ≠ receiver。

例如数据传输任务（2, 4，20）则表示这项任务是从第编号为2的计算机传到编号为4的计算机，

需要占用编号为2和编号为3的光纤各20M/s的带宽。

判断新任务是否应该被受理

多个数据传输任务按顺序依次被判断是否应该被受理。

这里的 “受理” 是指是否应该把这个任务运行在链路上。

很显然，同一条光纤上负载的所有任务的workload之和不能超过这条光纤的带宽。

举个例子，假如有三台计算机，编号为 1, 2, 3。

有两条光纤，编号为 1， 2。

两条光纤的带宽都为100M/s。

现在已经在这个链路中受理运行了（1, 2，80）和（2, 3，50）两项任务。

那么新的任务（1, 3, 50）就不能被这条链路执行。

虽然编号为2的光纤仍剩下 50M/s 的带宽可用，能满足新的任务的需求，

但是编号为1的光纤只剩下 20M/s 的可用带宽，不能运行新的任务。

所以是否能运行新的任务（m, n, w）需要检查编号为 m，m+1， ·····，n-1 的光纤中可用带宽的最小值，

如果这个值小于w，则不能受理这个新任务。

如果这个值大于等于w，则受理新任务，

并且编号为 m，m+1， ·····，n-1 的光纤的剩下的可用带宽都要减去w。

假设每个计算机的存储空间无限，并且一个数据传输任务一经受理则会永远运行，即永远占用着带宽。

要求用线段树（一种特殊的树）来完成这个数据传输任务受理判断系统。

你需要自己定义所需要的线段树结构和相关操作，来实现数据传输任务受理判断系统。

然后，根据输入的参数来进行判断，输出每个任务的判断结果。

关于线段树的资料很多，可以自行学习：

- 算法学习笔记(14): 线段树知乎 (zhihu.com) https://zhuanlan.zhihu.com/p/106118909

线段树 从入门到进阶（超清晰，简单易懂）_进阶线段树-CSDN博客

https://blog.csdn.net/weixin_45697774/article/details/104274713

使用线段树的motivation

setting：

数据传输任务里需要频繁进行的操作是在[sender, receiver-1]内查询最小带宽，以确定区间是否满足新任务的带宽需求

若区间内最小带宽满足新任务需求，就更新区间带宽

problem:对大的网络，有操作频繁的问题，且每次查询或更新的区间范围不同。简单线性扫描和更新在时间复杂度上不可接受，需要设计高效数据结构。

Why SegmentTree?

1.dynamic array: Efficient interval query and update on dynamically changing arrays is supported

2.efficient query:O(logn)

3.efficient update:O(logn)

Advantages

1.Improve the efficiency of quering and updating

2.Be able to tackle large scale data

3.Scalability

算法复杂度：

（1）构建线段树时间复杂度：

构建线段树需要遍历每个节点一次，时间复杂度为O(n)，n是光纤数量

（2）查询区间最小带宽的时间复杂度：

单次查询O(log n)，因为线段树高度为log n，每次查询最多访问树的高度层数

（3）更新区间带宽的时间复杂度：

单次更新O(log n)，因为线段树高度为log n，更新最多涉及树的高度层数的节点

总体复杂度：

m个任务，每个任务进行一次查询、一次更新，n是光纤数量

总的时间复杂度O(m log n)

构建线段树的时间复杂度为 O(n)，因为只需要在初始化时执行一次，所以整体的时间复杂度应包括构建线段树的时间复杂度和处理任务的时间复杂度。

总的时间复杂度为： O(n+mlogn)

当 $m \log n \gg n$ 时，O(n)可以相对忽略，总时间复杂度接近 O(mlogn)。但在一般情况，特别是在分析理论复杂度时，我们应保留所有重要项，因此： O(n+mlogn)

【对比】

暴力算法：适合小规模数据以及对性能要求不高的场景

（1）查询O(n)

（2）更新O(n)

（3）总体O(m*n)

线段树法代码

主要的代码逻辑

1.定义数据结构

①SegmentTreeNode线段树节点

最小带宽min_bandwidth

懒惰标记lazy

②Task数据传输任务

发送方sender

接收方receiver

工作负载workload

2.构建线段树build_segment_tree

初始化每个节点的最小带宽和懒惰标记

if叶子节点->设置带宽

if非叶子节点->递归构建子节点，设置带宽为2个子节点带宽最小值

3.懒惰传播propagate

当前节点有懒惰标记->将其传播到子节点并更新当前节点的带宽

4.查询最小带宽query_min_bandwidth

查询给定区间内的最小带宽，结合懒惰传播保证结果正确性

递归查询左右子节点，返回查询区间的最小带宽

5.更新线段树update_segment_tree

给定区间的带宽，用懒惰标记减少更新的复杂度

递归更新左右子节点，重新计算当前节点的最小带宽

6.主函数main

读取输入数据：计算机数量、光纤带宽

构建线段树

读取任务列表，依次处理每个任务，判断其能否受理

对每个任务，查询传输路径上的最小带宽

满足要求->受理任务并且更新

否则拒绝任务

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define INF 0x3f3f3f3f

typedef struct
{
    int min_bandwidth;
    int lazy;
} SegmentTreeNode;

typedef struct
{
    int sender;
    int receiver;
    int workload;
} Task;

void build_segment_tree(SegmentTreeNode *tree, int *bandwidths, int node, int start, int end)
{
    if (start == end)
    {
        tree[node].min_bandwidth = bandwidths[start];
        tree[node].lazy = 0;
    }
    else
    {
        int mid = (start + end) >> 1;
        int left_child = 2 * node + 1;
        int right_child = 2 * node + 2;
        build_segment_tree(tree, bandwidths, left_child, start, mid);
        build_segment_tree(tree, bandwidths, right_child, mid + 1, end);
        tree[node].min_bandwidth = (tree[left_child].min_bandwidth < tree[right_child].min_bandwidth) ? tree[left_child].min_bandwidth : tree[right_child].min_bandwidth;
        tree[node].lazy = 0;
    }
}

void propagate(SegmentTreeNode *tree, int node, int start, int end)
{
```

```
    if (tree[node].lazy != 0)
    {
        tree[node].min_bandwidth -= tree[node].lazy;
        if (start != end)
        {
            tree[2 * node + 1].lazy += tree[node].lazy;
            tree[2 * node + 2].lazy += tree[node].lazy;
        }
        tree[node].lazy = 0;
    }
}

int query_min_bandwidth(SegmentTreeNode *tree, int node, int start, int end,
int l, int r)
{
    propagate(tree, node, start, end);
    if (r < start || l > end)
    {
        return INF;
    }
    if (l <= start && r >= end)
    {
        return tree[node].min_bandwidth;
    }
    int mid = (start + end) >> 1;
    int left_child = 2 * node + 1;
    int right_child = 2 * node + 2;
    int left_query = query_min_bandwidth(tree, left_child, start, mid, l, r);
    int right_query = query_min_bandwidth(tree, right_child, mid + 1, end, l,
r);
    return (left_query < right_query) ? left_query : right_query;
}

void update_segment_tree(SegmentTreeNode *tree, int node, int start, int end,
int l, int r, int value)
{
    propagate(tree, node, start, end);
    if (start > end || start > r || end < l)
    {
        return;
    }
    if (start >= l && end <= r)
    {
        tree[node].lazy += value;
        propagate(tree, node, start, end);
        return;
    }
```

```
    int mid = (start + end) >> 1;
    int left_child = 2 * node + 1;
    int right_child = 2 * node + 2;
    update_segment_tree(tree, left_child, start, mid, l, r, value);
    update_segment_tree(tree, right_child, mid + 1, end, l, r, value);
    tree[node].min_bandwidth = (tree[left_child].min_bandwidth < tree[right_child].min_bandwidth) ? tree[left_child].min_bandwidth : tree[right_child].min_bandwidth;
}

int main()
{
    int num_computers;
    int *bandwidths;
    Task *tasks;
    int num_tasks = 0;
    int i;

    scanf("%d", &num_computers);

    bandwidths = (int *)malloc((num_computers - 1) * sizeof(int));
    for (i = 0; i < num_computers - 1; i++)
    {
        scanf("%d", &bandwidths[i]);
    }

    tasks = (Task *)malloc(100 * sizeof(Task));
    getchar();
    char line[100];
    while (fgets(line, sizeof(line), stdin) && line[0] != '\n')
    {
        if (sscanf(line, "%d %d %d", &tasks[num_tasks].sender, &tasks[num_tasks].receiver, &tasks[num_tasks].workload) == 3)
        {
            num_tasks++;
        }
    }

    int tree_size = 2 * (1 << (int)(ceil(log2(num_computers - 1)))) - 1;
    SegmentTreeNode *segment_tree = (SegmentTreeNode *)malloc(tree_size * sizeof(SegmentTreeNode));
    build_segment_tree(segment_tree, bandwidths, 0, 0, num_computers - 2);

    for (i = 0; i < num_tasks; i++)
    {
        int sender = tasks[i].sender - 1;
        int receiver = tasks[i].receiver - 1;
```

```
        int workload = tasks[i].workload;

        if (sender > receiver)
        {
            int temp = sender;
            sender = receiver;
            receiver = temp;
        }

        int min_bandwidth = query_min_bandwidth(segment_tree, 0, 0, num_computers - 2, sender, receiver - 1);

        if (min_bandwidth >= workload)
        {
            printf("YES\n");
            update_segment_tree(segment_tree, 0, 0, num_computers - 2, sender, receiver - 1, workload);
        }
        else
        {
            printf("NO\n");
        }
    }

    free(bandwidths);
    free(tasks);
    free(segment_tree);
    return 0;
}

```

暴力算法代码

```
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int sender;
    int receiver;
    int workload;
} Task;

int main() {
    int num_computers;
```

```c
    int *bandwidths;
    Task *tasks;
    int num_tasks = 0;
    int i, j;

    scanf("%d", &num_computers);

    bandwidths = (int *)malloc((num_computers - 1) * sizeof(int));
    for (i = 0; i < num_computers - 1; i++) {
        scanf("%d", &bandwidths[i]);
    }

    tasks = (Task *)malloc(100 * sizeof(Task));
    getchar();
    char line[100];
    while (fgets(line, sizeof(line), stdin) && line[0] != '\n') {
        if (sscanf(line, "%d %d %d", &tasks[num_tasks].sender, &tasks[num_tasks].receiver, &tasks[num_tasks].workload) == 3) {
            num_tasks++;
        }
    }

    for (i = 0; i < num_tasks; i++) {
        int sender = tasks[i].sender - 1;
        int receiver = tasks[i].receiver - 1;
        int workload = tasks[i].workload;

        if (sender > receiver) {
            int temp = sender;
            sender = receiver;
            receiver = temp;
        }

        // Find the minimum bandwidth on the path
        int min_bandwidth = bandwidths[sender];
        for (j = sender + 1; j < receiver; j++) {
            if (bandwidths[j] < min_bandwidth) {
                min_bandwidth = bandwidths[j];
            }
        }

        if (min_bandwidth >= workload) {
            printf("YES\n");
            for (j = sender; j < receiver; j++) {
                bandwidths[j] -= workload;
            }
        } else {
```

```
58              printf("NO\n");
59          }
60      }
61
62      free(bandwidths);
63      free(tasks);
64      return 0;
65  }
66
```

不同规模的测试

正确性测试

3个

输入

3

100 100

1 2 80

2 3 50

1 3 50

3 2 50

3 2 10

第一行的3代表链路中有3个计算机。

第二行的2个数字表示编号从小到大的光纤的带宽，用空格分开。

接下来每一行都表示一个任务。每一行的三个数字分别代表这个任务的 sender receiver workload，用空格分开。

任务的判断顺序为自上到下依次进行。

预期输出

YES

YES

NO

YES

NO

上面的输出表示判断结果。任务（1,2,80）（2,3,50）（3,2,50）被受理，而任务（1,3,50）（3,2,10）被拒绝。

很显然，（2,3,50）和（3,2,50）在本题中没有区别。

10个

测试数据

10

100 150 200 250 300 350 400 450 500

1 3 80

2 5 120

4 6 100

5 8 50

1 10 200

3 7 300

7 10 250

2 4 70

1 9 150

6 10 200

预期输出

YES

NO

YES

YES

NO

NO

YES

YES

NO

NO

15个

15

120 150 180 200 220 240 260 280 300 320 340 360 380 400

1 5 100

2 6 130

3 7 150

4 8 180

5 9 200

6 10 220

7 11 240

8 12 260

9 13 280

10 14 300

11 15 320

1 10 150

5 15 200

2 11 180

6 14 220

预期输出

YES

NO

NO

NO

YES

NO

NO

NO

YES

NO

NO

NO

NO

NO

NO

经过测试，发现两种算法均能输出正确结果

用脚本生产不同规模的数据以比较性能提升

```python
import random

def generate_test_data(num_computers, num_tasks, filename):
    try:
        with open(filename, 'w') as f:
            # 写入计算机数量
            f.write(f"{num_computers}\n")

            # 生成带宽数据
            bandwidths = [random.randint(1, 100) for _ in range(num_computers - 1)]
            f.write(" ".join(map(str, bandwidths)) + "\n")

            # 生成任务数据
            for _ in range(num_tasks):
                sender = random.randint(1, num_computers)
                receiver = random.randint(1, num_computers)
                while receiver == sender:  # 确保发送者和接收者不相同
                    receiver = random.randint(1, num_computers)
                workload = random.randint(1, 100)
                f.write(f"{sender} {receiver} {workload}\n")
        print(f"Test data generated successfully: {filename}")
    except Exception as e:
        print(f"An error occurred while generating test data: {e}")

# 逐步增加数据规模，确保生成成功
try:
    generate_test_data(64, 32, 'test_data_64.txt')  # 64规模，32个任务
    generate_test_data(128, 64, 'test_data_128.txt')  # 128规模，64个任务
    #generate_test_data(10, 5, 'test_data_small.txt')    # 小规模
    #generate_test_data(100, 50, 'test_data_medium.txt') # 中等规模
    #generate_test_data(10000, 5000, 'test_data_large.txt') # 大规模
    #generate_test_data(100000, 50000, 'test_data_very_large.txt')  # 更大规模
except Exception as e:
    print(f"Error during test data generation: {e}")

```

在cmd运行

brute_force.exe < test_data_64.txt

segment_tree.exe < test_data_64.txt

```
C:\Users\azq\OneDrive\桌面\ds_24spring\21300720003\PJ3>brute_force.exe < test_data_64.txt
NO
NO
NO
NO
NO
YES
NO
NO
NO
NO
NO
NO
NO
NO
NO
YES
NO
NO
NO
NO
NO
NO
NO
NO
NO
NO
NO
YES
NO
NO
NO
NO
NO
0.004645
```

```
C:\Users\azq\OneDrive\桌面\ds_24spring\21300720003\PJ3>segment_tree.exe < test_data_64.txt
NO
NO
NO
NO
NO
YES
NO
NO
NO
NO
NO
NO
NO
NO
NO
YES
NO
NO
NO
NO
NO
NO
NO
NO
NO
NO
YES
NO
NO
NO
NO
NO
0.003238
```

brute_force.exe < test_data_128.txt

segment_tree.exe < test_data_128.txt

C:\Users\azq\OneDrive\桌面\ds_24spring\21300720003\PJ3>brute_force.exe < test_data_128.txt

...省略32行输出

0.010188

C:\Users\azq\OneDrive\桌面\ds_24spring\21300720003\PJ3>segment_tree.exe < test_data_128.txt

...省略32行输出

0.008363

可以发现segment_tree相比brute_force有性能提升