華中科技大學

课 程 实 验 报 告

课程名称： 并行编程原理与实践

专业班级：	**IOT1601**
学号：	**U201614898**
姓名：	**潘翔**
指导教师：	**陆枫**
报告日期：	**2019.7**

计算机科学与技术学院

目 录

1 实验一

1.1 实验目的与要求

在串行环境下编写计算斐波那契数列的C语言小程序，并按要求输出对应的斐波那契数列。

1.2 实验内容

斐波那契数列(Fibonacci sequence)，又称黄金分割数列、因数学家列昂纳多·斐波那契(Leonardoda Fibonacci)以兔子繁殖为例子而引入，故又称为“兔子数列”。

$$f(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ f(n-1)+f(n-2) & n>1\end{cases}$$

公式 1.1 Fibonacci 递推

1.3 实验结果

代码 1.1 串行斐波那契数列

```
#include<stdio.h>
#define int long
int a[100];
int f(int i)
{
    if(a[i]!=-1)
    {
        return a[i];
    }
    else
    {
        if(i==0)
        {
            a[i]=0;
            return a[i];
        }
        if(i==1)
        {
            a[i]=1;
            return a[i];
        }
        if(i>1)
```

```
        {
            a[i]=f(i-1)+f(i-2);
            return a[i];
        }
    }

}

int main()
{
    //init
    for(int i=0;i<100;i++)
    {
        a[i]=-1;
    }

    int n;
    scanf("%ld",&n);
    for(int i=1;i<n+1;i++)
    {
        printf("%ld",f(i));
        if(i!=(n))
        {
            printf(" ");
        }
    }
    printf("\n");
    return 0;
}
```

Time:0.678s

1.4 实验总结

实验十分简单，主要是为了与并行程序进行对比

2 实验二

2.1 Thread

2.1.1 实验目的与要求

编写使用多线程计算斐波那契数列的C语言小程序，并按要求输出对应的斐波那契数列。

2.1.2 实验环境

1) 本地环境：
2) 在线环境：educoder

2.1.3 算法描述

采用公式计算解决数据依赖，利用多线程(两线程)并行计算奇数和偶数。

$$a_n = \frac{1}{\sqrt{5}}[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n]$$

利用多线程进行计算写入数组，最后等待计算完成，串行

2.1.4 实验方案

1) 算法过程

图 2.1 thread 算法流程图

2) 算法实现

代码 2.1 thread 并行算法

```
#include <stdio.h>
#include <pthread.h>
#include <math.h>
#include <stdlib.h>
#define ll long long
```

```
ll a[100];
int n=0;
int thread_num=16;


void* par_fun(void * tn)
{
        // int tn=*((int *)tn);
    int par_tn=*((pthread_t *)tn);
    // printf("par_tn:%d\n",par_tn);
        for(int i=par_tn;i<=n+1;i+=thread_num)
        {

        a[i]=(long)(1.0/sqrt(5.0)*(pow((1.0+sqrt(5.0))/2.0,i)-pow((1.0-s
qrt(5.0))/2.0,i)));
        }
}



int main()
{
        scanf("%d",&n);
    // thread_num=(int)(n/2+0.5);
    pthread_t threads[thread_num];
    int tn;
    for(tn=0;tn<thread_num;tn++)
    {
        pthread_create(&threads[tn],NULL,par_fun,(void *)&tn);
        usleep(20);
    }
    for(tn=0;tn<thread_num;tn++)
    {
        pthread_join(threads[tn],NULL);
    }
        for(int i=1;i<n;i++)
        {
                printf("%lld ",a[i]);
        }
    printf("%lld\n",a[n]);
    return 0;
}
```

2.1.5 实验结果分析

educoder显示时间：1.092 秒

因为线程的创建需要时间，虽然是共享相应的变量和函数空间而不用进行拷贝，可是分配资源和函数调用需要，而测试集整体较小，所以效率上可能不如串行。

2.2 OpenMP

2.2.1 实验目的

编写使用`OpenMP`计算斐波那契数列的`C`语言小程序，并按要求输出对应的斐波那契数列。

2.2.2 实验环境

本地环境:
编译器：gcc version 9.1.0 (MacPorts gcc9 9.1.0_2)
并行库：OpenMP
在线环境: educoder

2.2.3 算法描述

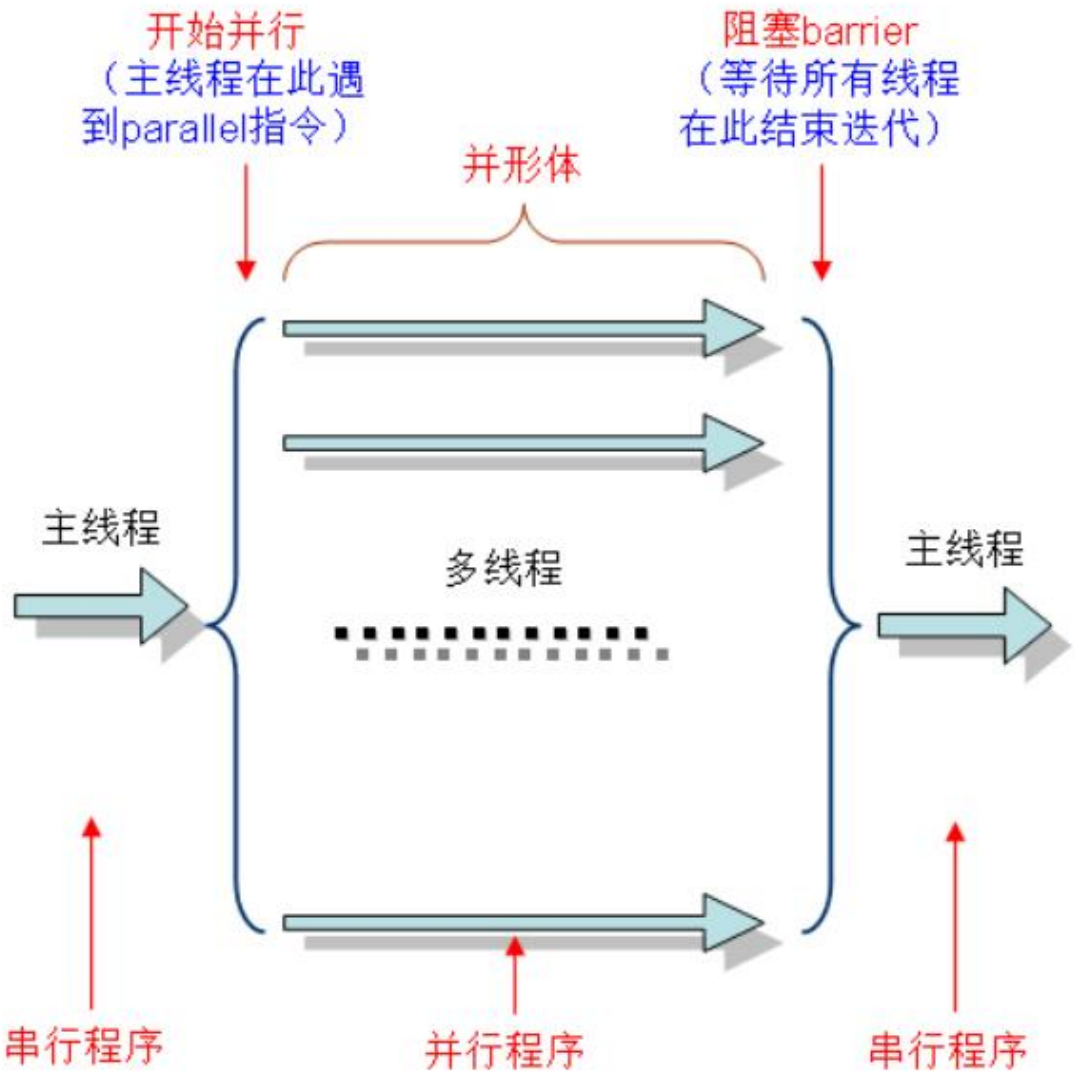

图 2.2 OpenMp 并行流程

OpenMP采用fork-join的执行模式。开始的时候只存在一个主线程，当需要进行并行计算的时候，派生出若干个分支线程来执行并行任务。当并行代码执行完成之后，分支线程会合，并把控制流程交给单独的主线程。

直接利用#pragma omp parallel for对条件循环部分进行并行从而OpenMP自动进行线程级别并行。

2.2.4 实验方案

代码 2.2 OpenMP 解决 fibonacci 数列

```
//OpenMp
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>

#define ll long long
ll a[100];

ll F(int n)
{
    return
(ll)(1.0/sqrt(5.0)*(pow((1.0+sqrt(5.0))/2.0,n)-pow((1.0-sqrt(5.0))/2.0,n
)));
}

int main()
{
        int n=0;
        omp_set_num_threads(2);
        scanf("%d",&n);
        getchar();
        a[0]=1;
        a[1]=1;
        #pragma omp parallel for
        for(int i=2;i<n;i++)
        {
                a[i]=F(i);
        }
        for(int i=0;i<n-1;i++)
    {
```

```
        printf("%lld ",a[i]);
    }
        printf("%lld\n",a[n-1]);
        return 0;
}
```

2.2.5 实验结果分析

educoder时间:1.01s，相较于自己手动创建线程有了一定程度的提升，可能是框架层面的优化，也可能是循环的控制导致没有进行更多的冗余计算。

2.3 MPI

2.3.1 实验目的

编写使用`MPI`计算斐波那契数列的`C`语言小程序，并按要求输出对应的斐波那契数列。

2.3.2 实验环境

本地环境：
gcc version 9.1.0 (MacPorts gcc9 9.1.0_2)
MPI: clang-mpi(Thread model: posix)
在线环境：educoder

2.3.3 算法描述

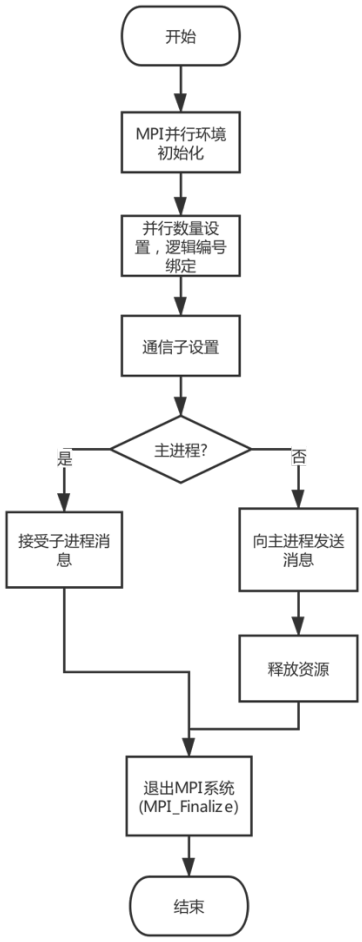

图 2.3 MPI 并行程序设计

2.3.4 实验方案

由于在实验平台中，并未指定参数输入，导致需要等待std输入流造成线程卡顿，故直接采用最大值进行计算从而进行输出，实际过程中，创建线程仍然是最大开销。

代码 2.3 MPI 解决 fibonacci

```
//MPI
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <string.h>
#define ll long long

ll F(int n)
{
```

```
    return
(ll)(1.0/sqrt(5.0)*(pow((1.0+sqrt(5.0))/2.0,n)-pow((1.0-sqrt(5.0))/2.0,n
)));
}
int main(int argv,char* argc[])
{
    int n = 0;

    int max_len=55;

    int process_id=0;
    int process_num=0;

    MPI_Status status;
    MPI_Init(&argv,&argc);
    MPI_Comm_rank(MPI_COMM_WORLD,&process_id);
    MPI_Comm_size(MPI_COMM_WORLD, &process_num);

    ll* message = (ll*)malloc(sizeof(ll)*max_len);
    ll* recv = (ll*)malloc(sizeof(ll)*max_len);
    memset(message, 0, max_len);
    memset(recv, 0, max_len);

    int loop = ceil(max_len/(process_num-1));
    if(process_id != 0) // send message
    {
        for(int i = ( process_id - 1 ) * loop; i < process_id*loop; i++)
        {
            message[i] = F(i);
        }
        MPI_Send(message, max_len, MPI_LONG_LONG_INT, 0, 0,
MPI_COMM_WORLD);
    }
    else//receive message
    {
        scanf("%d", &n);
        for(int i = 1; i < process_num; ++i)
        {
            MPI_Recv(recv, max_len, MPI_LONG_LONG_INT, i, 0,
MPI_COMM_WORLD, &status);
            for(int j = (i-1)*loop; j < i*loop; j++)
            {
                message[j] = recv[j];
            }
```

```
        }
        for(int i = 1; i < n; ++i)
        {
            printf("%lld ", message[i]);
        }
        if (n != 1)
            printf("%lld\n", message[n]);
        else
            printf("1\n");
        free(message);
        free(recv);
    }
    MPI_Finalize();
    return 0;
}
```

2.3.5 实验结果分析

Educoder时间：1.605 s
由于并未严格按照输入数据范围进行计算，直接打大表，造成一定的开销，如果允许命令行参数的话，在本地测试过程中，可以直接初始化需要计算的范围，不会造成程序卡顿，同时由于存在通信开销，故时间略长。

2.4 CUDA

2.4.1 实验目的

编写使用`CUDA`计算斐波那契数列的`C`语言小程序，并按要求输出对应的斐波那契数列。

2.4.2 实验环境

本地环境：
OS：Manjaro 18.04
GCC：9.1
CUDA：9.0
在线环境：educoder

2.4.3 算法描述

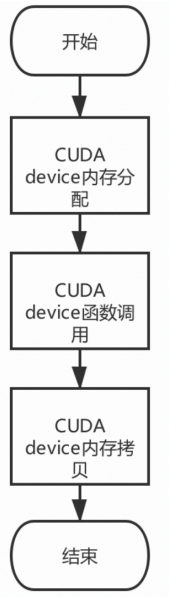

图 2.4 CUDA 流程

2.4.4 实验方案

根据输入的n值进行内存分配，然后分配线程进行计算，其中可以采用公式进行运算，也可是采用递推形式，但是采用递推形式的话不解决数据依赖并没有达到并行优化的效果。

如果采用公式的话，由于device不能使用host function，而采用内置的_powf精度较低，当n比较大的时候会出错。故采用原始函数空间的pow。此时的pow没有重载，需要使用(double,double参数)。

代码 2.4 CUDA 解决 fibonacci 问题

```
//CUDA
#include <stdio.h>
// #include <math.h>
#define ll long long

ll F(int n)
{
    return
(ll)(1.0/sqrt(5.0)*(pow((1.0+sqrt(5.0))/2.0,n)-pow((1.0-sqrt(5.0))/2.0,n
)));
}
```

```
__global__ static void fibo_cuda(ll* result){
    int tid =threadIdx.x + 1;
    ll f3 = 0;
    if(tid==0)
    {
        f3 = 0;
    }
    else if(tid==1) // print from 1
    {
        f3 = 1;
    }
    else
    {
        ll n=tid;

f3=(ll)(1.0/sqrt(5.0)*(pow((1.0+sqrt(5.0))/2.0,(double)n)-pow((1.0-sqrt(
5.0))/2.0,(double)n)));
    }
    result[tid] = f3;
}


int main()
{
    ll* p_result;
    ll result[100];
    int n;
    scanf("%d",&n);
    int N = sizeof(long)*(n+1); //add zero to the arr
    cudaMalloc((void **)&p_result,N);

    fibo_cuda<<<1,n>>>(p_result);

    cudaMemcpy(&result,p_result,N,cudaMemcpyDeviceToHost);
    for(int j = 1;j <n;j++)
    {
        printf("%lld ",result[j]);
    }
    printf("%lld\n",result[n]);
        return 0;
}
```

2.4.5 实验结果分析

Educode时间：1-2s

由于通关最初采用递推形式(后期无时间显示)，最终自行测试时间求和约在1-2s,存在拷贝空间和资源分配等高耗时操作。

2.5 串行算法复杂度分析

2.5.1 描述

分析串行算法的复杂度，并设计测试案例予以证明。

2.5.2 分析

采用递推的方式进行串行计算，其
时间复杂度：O(N)
空间复杂度：O(1)

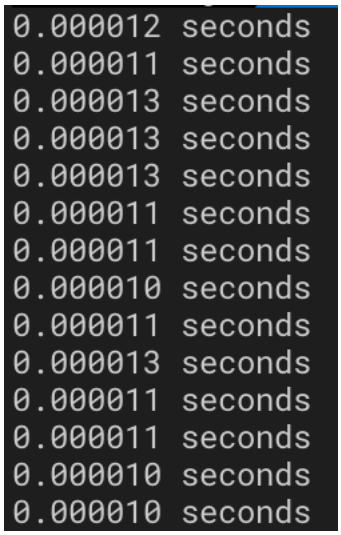

图 2.5 串行时间分析(1-100)部分

经过测试，发现程序主要的时间在IO上，即输出结果。在小数据范围内，显著差异，加上输出，则基本线性。

2.6 分析并行实现的正确性

在“斐波那契数列计算”中实现的多种并行实现中，挑选一种并分析其正确性，并请给出方案确保并行实现的正确性。

Thread形式：
分奇数和偶数采用双线程进行利用斐波拉契数列的公式进行计算，写入数组，最后串行数组，由于不存在数据依赖和输出的相关考虑，则程序结果正确。

2.7 大数场景分析

2.7.1 描述

1) 设计大数计算场景下“**斐波那契数列计算**”的测试用例并分别使用串行和并行的方式实现。
2) 分析并行实现的加速比。
3) 分析并行实现加速比的正确性。

2.7.2 分析

由于long
long的精度限制，当n=100时已经出现明显溢出，实际上算法的主要时间仍然在IO上面。

- 纯计算过程，并行加速比<并行线程数
- 整个过程中考虑初始的串行初始化过程和最后的串行输出过程
- 当 n 越大时，约趋近于并行线程数

实际过程中，由于小数据和大数据情况下具有较大的差异，实际上仍有许多书剑用于IO，所以在测量上不进行输出。对同一个程序进行多次测量的时间上可能由于缓存的原因仍有一定的差异。

对于时间的测量有两种方式：

1) 利用系统调用进行时间的测量
   此时直接采用命令行的方式进行参数给定，避免等待IO的时间误差
2) 利用 time 指令进行时间测量
   Time 指令包括三部分
   a. 实际时间(real time)：从 command 命令行开始执行到运行终止的消逝时间
   b. 用户 CPU 时间(user CPU time)：命令执行完成花费的用户 CPU 时间，即命令在用户态中执行时间总和
   c. 系统 CPU 时间(system CPU time)：命令执行完成花费的系统 CPU 时间，即命令在核心态中执行时间总和

测量过程如下：

```
hover@wings ~/Desktop/Labs/HUST_ParallelProgramming_Labs/fibonacci master time ./lab1.out
90
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 241
57817 39088169 63245986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 4807526976 7778742049 12586269025 20365011074 32951280099 53316291173 8626757
1272 139583862445 225851433717 365435296162 591286729879 956722026041 1548008755920 2504730781961 4052739537881 6557470319842 10610209857723 17167680177565 27777890035288 4494557021285
3 727234602481 41 117669030460994 190392490709135 308061521170129 498454011879264 806515533049393 1304969544928657 2111485077978050 3416454622906707 5527939700884757 8944394323791464 14
472334024676221 23416728348467685 37889062373143906 61305790721611591 99194853094755497 160500643816367088 259695496911122585 420196140727489673 679891637638612258 1100087778366101931
177997941600471418 2880067194370816120
0.000100 seconds
./lab1.out  0.00s user 0.00s system 0% cpu 1.278 total
```

图 2.6 时间测量过程

我们发现，采用函数调用的测量精度高于time最后的时间结果。

表 2.1 时间测量结果

方法	函数调用时间

串行	0.000097 seconds
Thread	0.000694 seconds
OpenMP	0.000478 seconds
MPI	0.001041 seconds
CUDA	0.003254 seconds

2.8 并行优化方案设计

2.8.1 描述

1) 设计大数场景并行实现的优化方案
2) 测试并分析优化方案的加速比与正确性。

进行并行数的搜索，所得的并行数应该具有最高的并行加速比。

对于并行来说，对于小计算量来说，更重要的是框架的选择，因为没有太多的优化技巧，其中值得注意的就是并行量的选择，由于计算量太小，那么多开一个线程的开销可能已经远远大于并行节省的时间，此时可以考虑选择较为轻型的框架进行计算比如thread和OpeMP。

2.9 实验总结

实验过程中使用了不同的并行程序库进行同一个基础算法的比较，主要的任务是解决数据依赖和如何进行并行结果的稳定输出。由于本地MacOS且不想采用虚拟机，在环境方面有一段时间的折腾(Unix与Linux部分API存在差异)，且代码迁移需要一定的修改。

实验的具体过程中，在cuda中，由于device有自己相关的函数定义，不能使用主空间的函数，所以在重载方面卡了一会，且并行编程的一个问题是参数传递，不支持命令行餐参数的情况下测试程序尚不完备，容易造成IO流卡顿。

首先并行需要解决的是数据相关性，在原始的递推形式中，由于依赖前项，从而有数据相关，当采用公式进行计算而消除了数据相关性之后，算法的性能得到了很大的提升

将串行的优化为并行，其效果分析，我们首先从串行的复杂度进行分析，串行算法的复杂度进行分析，在实际的测试过程中，发现占用性能的主要是I/O而不是计算过程，而由于本地计算性能相较于问题来说还行，CPU相应的缓存和编译的自动优化导致时间非常小，不容易测量，造成一定的麻烦。

在并行的测试上面，对于不同的框架进行了对比，从而对于不同框架的轻重程度有了直观的认识，在进行轻型并行的计算时，应该采用更加轻型化的计算框架，同时对于并行数量的选择也需要注意，注意并行造成的加速和相应的

开销的对比，同时需要注意本地硬件的限制，比如本地的CPU是6核12线程，那么通常来说，并行数不应该超过12，在实际的编译过程和没有数据依赖的重型任务时候，加速比的提升和线程数的设置是线性的。

并行过程中可能涉及通信如MPI，实际上可以看到是posix:thread的高层封装，此时性能差于直接thread是可以预见的。

当设计到不同数据之间进行拷贝的时候比如CUDA可能设计显存和内存的拷贝，则其时间相对更长，同时进行设计的时候副本的拷贝也是需要考虑的。

总体上来书，软件方面：不同的框架体现了不同的特点，其瓶颈可能也不同，涉及存储和通信两个基本部分，体会了并行程序涉及需要考虑的各种因素。硬件方面：平台上所有的并行结果最后都是负优化，同时并未给明平台的硬件性能，从而说明在实际并行过程中，需要关注硬件性能和参数，从而能够获得最后的加速效果。

3 Project

3.1 实验目的

1) 了解 Akari 问题，探索其解决方法。
2) 了解回溯法的概念，掌握回溯法的原理，并能够使用一种编程语言通过使用回溯法解决问题。
3) 培养和锻炼分析问题与解决问题的能力。

3.2 实验环境

本地环境:
OS: 64bit Mac OS X 10.14.5 18F203
Kernel: x86_64 Darwin 18.6.0
G++: Apple clang version 11.0.0 (clang-1100.0.31.5)
在线环境：
educoder

3.3 算法描述

3.3.1 Akari 问题

Akari问题有时又被称为Light up或者Beleuchtung，源于日本逻辑解密游戏系列Nikoli。

游戏规则很简单。点灯游戏的棋盘是一张方形格网，其中的格子可能是黑色也可能是白色。游戏目标是在格网中放置灯泡，使之能照亮所有的白色方格。如果一个方格所在的同一行或同一列有一个灯泡，并且方格和灯泡之间没有黑色格子阻挡，那么这个方格将被灯泡照亮。同时，放置的每个灯泡不能被另一个灯泡照亮。

某些黑色格子中标有数字。这些数字表示在该格子四周相邻的格子中共有多少个灯泡。

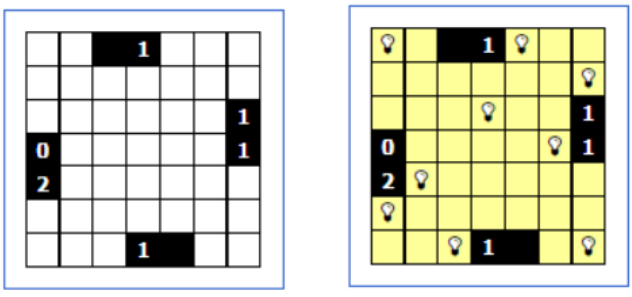

图 3.1 Akari 的初始状态与解

3.3.2 回溯法

回溯法是一种迭代算法。在回溯法中，首先将问题的解决分为若干步，其次通过枚举每一步的选择构造解空间树。在此基础上通过深度优先搜索遍历此

解空间树，若当前节点的局部解不能构造出全局解，则向上回溯，否则向下扩展。重复此步骤直到找到全局解。

回溯法的关键点在于：

问题可分步并且可枚举每一步的选择。可以迅速的判断出当前局部解可否构造出全局解。

若问题可分为N步，每一步有M种选择，易知其时间复杂度为$O(m^n)$，虽然和穷举法有着相同的时间复杂度，但因为是逐步搜索并不断摒弃局部解，因此在实际应用中比穷举法效率高的多。实际上，回溯法是进行了减枝操作.

3.3.3 串行算法描述

首先考虑有数字周围黑色方块放灯的情况，选择其中一种情况，然后再考虑其他剩余白块的放灯情况。考虑白灯也会有很多种情况，当考虑完毕放完灯后发现仍有白块未照亮，则说明此方案不可行，需要回溯，先从最近的一次方案开始回溯，直到将所有白块回溯完毕后发现仍不可行，则需要回溯数字黑块周围的放灯情况，直到有一种情况满足条件。

1) 对问题进行划分

根据黑色方格中数字的大小，按从大到小的顺序进行排序，并将“在一个有数字的黑色方格周围放置‘车’”定义为一步。

2) 选择枚举

枚举每一步的选择。若黑色方格中的数字为4，则其相邻的周围格子的“灯泡”的放置方式为1种，即上下左右均放置一个“灯泡”。

若黑色方格中的数字为3，则其相邻的周围格子的“灯泡”的放置方式为4种，即在左右下、下右上、右上左、上左下的格子中放置“灯泡”。

若黑色方格中的数字为2，则其相邻的周围格子的“灯泡”的放置方式有6种，即在上左、上下、上右、左下、左右、下右的格子中放置“灯泡”。

若黑色方格中的数字为1，则其相邻的周围格子的“灯泡”的放置方式有4种，即在上、下、左、右的格子中放置“灯泡”。

若黑色方格中的数字为0，则其相邻的周围格子的“灯泡”的放置方式有1种，即所有周围格子均不放置车。

3) 构造解空间

根据上述讨论构造解空间，初始状态为解空间树的根节点，从编号最大的黑色格子开始尝试填入“灯泡”，填入后判断是否为一个可行解，若为可行解，则解空间向下进行分枝，否则向上进行回溯。解空间树的结构大致如图2所示。

3.4 串行实验方案

3.4.1 数据结构定义

代码 3.1 数据结构定义

```
typedef struct point
{
        int x;
        int y;
        int val;
```

```
} point;

point black[100];
```

其中(x,y)为相应的坐标，val为当前点的值
利用black进行黑格的存储，从而进行顺序搜索，每一层解空间满足一个黑格子的条件。

3.4.2 操作定义

1) 放灯(on)

在(x,y)处进行放灯，并且将横和列进行点亮

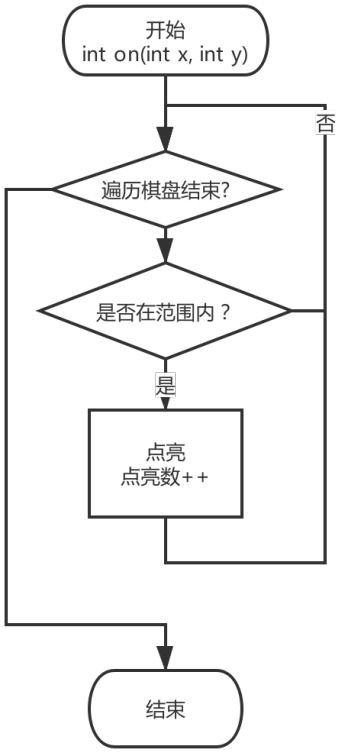

图 3.2 点亮流程图

代码 3.2 放灯操作

```
int on(int x, int y)// put in (x,y)
{
        int ret=1; //light num
        int i = 0;
        int j = 0;
        for(int k=0;k<4;k++)
        {
                i=x+dx[k];
                j=y+dy[k];
                while (in_range(i,j)&& type[i][j] != 2)
                {
```

```
                        cover[i][j]++;
                        if(cover[i][j] == 1)
                        {
                                ret ++;
                        }
                        i+=dx[k];
                        j+=dy[k];
                }
        }
        return ret;
}
```

2)　熄灯

同亮灯一样，进行熄灯操作。

代码 3.3 熄灯操作

```
int off(int x,int y)
{
        int ret=1; //light num
        int i = 0;
        int j = 0;
        for(int k=0;k<4;k++)
        {
                i=x+dx[k];
                j=y+dy[k];
                while (in_range(i,j) &&type[i][j] != 2)
                {
                        cover[i][j]--;
                        i+=dx[k];
                        j+=dy[k];
                }
        }
        return ret;
}
```

3.4.3 结果判定定义

1)　重叠判定

对当前黑色格子的状态进行判定，是否满足，如果满足返回0，如果未满足，返回1，如果出现错误重叠，返回2。

代码 3.4 重叠判定

```
int check_overlap()
{
    int i,j;
    int tmp;
```

```
    int ret = 2;
    for(i = 0;i < b;i++) // check black
    {
        if(black[i].val == -1)
            continue;
        tmp = 0;
        for(j = 0;j < 4;j ++)//check black neibor
        {
            int tx = black[i].x + dx[j];
            int ty = black[i].y + dy[j];
                        if(!in_range(tx,ty))
                        {
                                continue;
                        }
            if(type[tx][ty] == 5)
                tmp ++;
        }
        if(tmp > black[i].val)
            return 0;
        if(tmp != black[i].val)
            ret = 1;
    }
    return ret;
}
```

2)　边界判定

代码 3.5 边界判定

```
bool in_range(int x, int y)
{
        return (x >= 1 && y >= 1 && x <= n && y <= m);
}
```

3.4.4 操作流程

此处最初进行DFS搜索，每次放置一个灯而不是一组灯，故采用DFS搜索，但是后来发现并行过程并不合适，故之后改为成组放置(每次满足一个黑格)。在实现过程中，可以进行黑格的数量进行判断是否满足，也可以采用白格的数量进行满足。

1)　黑色格子 DFS 搜索(单个放置)

遍地黑色格子序列，对于每个黑色格子进行DFS搜索，如果不满足则进行回溯，解空间比较大，且存在重叠，但是实现简单。

代码 3.6 黑格搜索

```
void dfs_for_black(int id,int count,int depth)
{
```

```
        if(ok)
        {
                return;
        }
        if(id == b)
    {
        if(check_overlap() == 2)// this situ should not
        {
            dfs_for_white(1,1,count,depth);
        }
        return;
    }
        int i=0;
        int j=0;
        for(i=0;i<4;i++) // chekc neibor
        {
                if(type[black[id].x + dx[i]][black[id].y + dy[i]] == 5)
            j++;
        }
        if(j > black[id].val)
    {
        return;
        }
        else
    {
        if(j == black[id].val)
            dfs_for_black(id + 1,count,depth);
        else
        {
            for(i = 0;i < 4;i ++)
            {
                int tx = black[id].x + dx[i];
                int ty = black[id].y + dy[i];
                if(tx < 1 || ty < 1 || tx > n || ty > m)
                    continue;
                if(type[tx][ty] == 0 && cover[tx][ty] == 0)
                {
                    type[tx][ty] = 5;
                    if(check_overlap() == 0)
                    {
                        type[tx][ty] = 0;
                        continue;
                    }
                    int tp = on(tx,ty);
```

```
                    dfs_for_black(id,count + tp,depth + 1);
                    off(tx,ty);
                    type[tx][ty] = 0;
                }
            }
        }
    }
}
```

2) 白色格子 DFS 搜索(单个放置)

完成黑格搜索之后，判定黑格是否满足，从而继续进行白格的DFS搜索。

代码 3.7 白格搜索

```
void dfs_for_white(int x,int y,int count,int depth)
{
        if(ok)
        {
                return;
        }
        if(count == m*n-b) //light all
        {
                ok=true;
                for(int i=1;i<=n;i++)
                {
                        for(int j=1;j<=m;j++)
                        {
                                map[i][j]=type[i][j];
                        }
                }
        }
        for(int i = x;i <= n;i ++)
    {
        for(int j = 1;j <= m;j ++)
        {
            if(type[i][j] == 0 && cover[i][j] == 0)
            {
                type[i][j] = 5;
                if(check_overlap() != 2) //overlap
                {
                    type[i][j] = 0;
                    continue;
                }
                int light_num = on(i,j);
                dfs_for_white(i,j,count + light_num,depth + 1);
                off(i,j);
```

```
                type[i][j] = 0;
            }
        }
    }
        return;
}
```

3) 整体操作流程

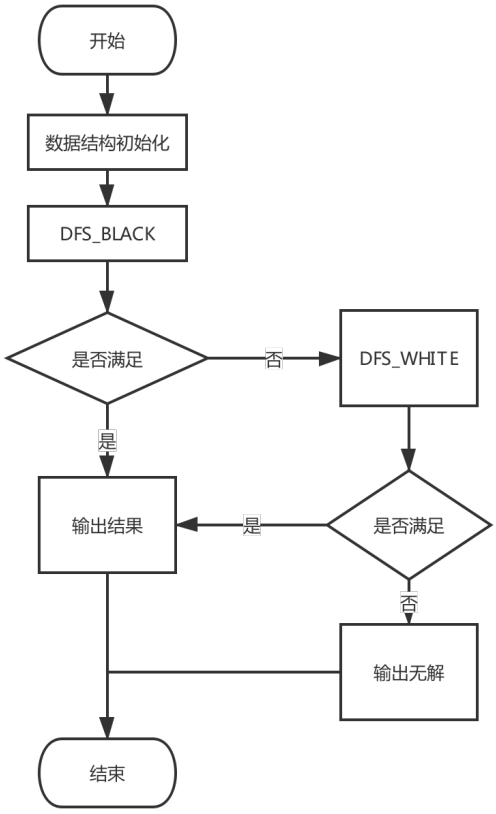

图 3.3 整体操作流程图

代码 3.8 整体操作流程

```
vector<vector<int> > solveAkari(vector<vector<int> > &g)
{
        n=g.size();
        m=g[0].size();
        b=0;
        for(int i=0;i<g.size();i++) // add black to struct
        {
                for(int j=0;j<g[i].size();j++)
                {
                        if(g[i][j]>=-1)
                        {
                                black[b].x=i+1;
                                black[b].y=j+1;
```

```
                                black[b++].val=g[i][j];
                                type[i+1][j+1]=2;
                        }
                }
        }
        sort(black,black+b,cmp);
        ok=false;
        int i;
        for(i=0;i<b;i++)
        {
                if(black[i].val>0)
                {
                        break;
                }
        }
        while(1)
        {
                if(ok||ans>m*n-b)
                {
                        break;
                }
                dfs_for_black(i,0,0);
                if(ok)
                {
                        break;
                }
                ans++;
        }
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++){
                if(map[i][j]==5)
            {
                g[i - 1][j - 1] = 5;
            }
                }
    return g;
}
```

3.5 并行实验方案

由于最初仅仅为了实现的简单而采用单个放置的搜索，后来发现这样不利于并行(解空间重叠)造成效率损失，故先更改串行方式进行成组搜索，然后在解空间层面进行并行。

3.5.1 并行算法描述

串行算法并行化的过程中的核心问题是问题分解和解除数据相关的问题。所谓的问题分解问题即将串行算法采取分治的理念分成可并行计算的子问题，解除数据相关的问题就是采取一定的冗余策略或锁策略使各子问题的私有数据之间不相互影响。

回溯法是易于并行化的典型算法之一，在Akari问题的回溯算法中，可以清楚的认识到回溯的具体路线的选择对产生解的正确性没有影响，因此该回溯法中每个节点的计算过程以及节点的分裂过程均是可以并行计算的。同时由算法的性质决定了每个解空间节点各自数据不会相互影响，没有任何数据相关。

并行的Akari回溯算法的示意图如图2.1所示。

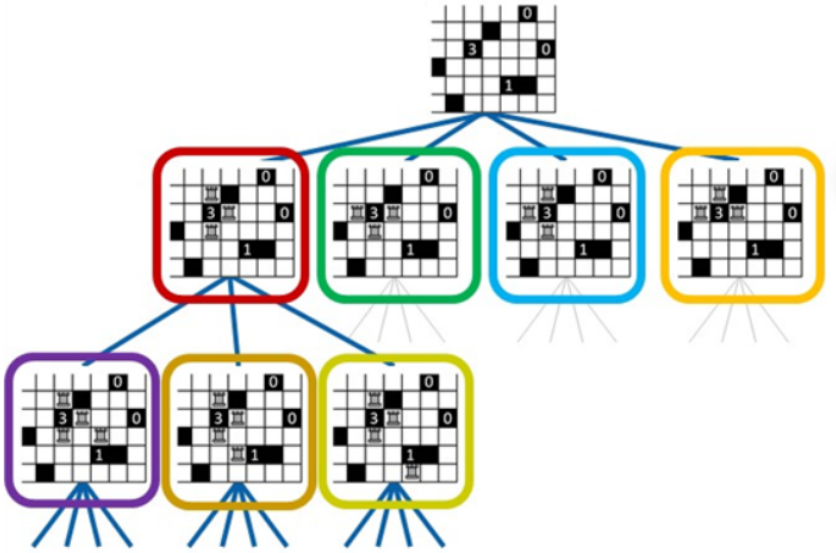

图 3.4 并行的 Akari 回溯算法

图中彩色的矩形表示可以并行执行的单元，可以看出，解空间树中相同层的执行是可以并行化的，不同层之间的执行依然可以并行化。

在Linux 下 C++ 环境中并行编程方法：使用pthread进行多线程编程

3.5.2 并行具体过程

1) 搜索

根据数字黑块上的数字，根据排列组合创建相应数量的线程完成并行递归回溯。具体数量如下：

数字为0时，无灯，只有1种情况。

数字为4时，全是灯，只有1种情况。

数字为1或3时，共有4种情况，创建4个线程。

数字为2时，共有6种情况，创建6个线程。

当所有线程递归回溯完毕后，选择一种可行的解输出。

代码 3.9 并行搜索解成组绑定

```
int step[4][2] = {{-1,0},{1,0},{0,-1},{0,1}};
int solution4[1][4] = {{1,1,1,1}};
int solution3[4][4] = {{0,1,1,1},{1,0,1,1},{1,1,0,1},{1,1,1,0}};
```

```
int solution2[6][4] =
{{1,1,0,0},{1,0,1,0},{0,1,1,0},{0,1,0,1},{0,0,1,1},{1,0,0,1}};
int solution1[4][4] = {{0,0,1,0},{0,0,0,1},{0,1,0,0},{1,0,0,0}};
int solution0[1][4] = {{0,0,0,0}};
```

2) 并行粒度

并行可以在每一层都进行分裂并行，也可以仅仅在第一层并行(加上depth=1限定)，实验过程中发现，仅仅在第一层进行并行效果较好。

代码 3.10 并行搜索过程

```
    switch (case.val)
    {
        case 4:
            if (light_up(g, x-1, y, flag) && light_up(g, x+1, y, flag) &&
                light_up(g, x, y-1, flag) && light_up(g, x, y+1, flag))
            {
                data->re = startPthread(g, blacks, depth+1, flag+1);
                return NULL;
            }
            data->re = false;
            return NULL;
            break;
        case 3:
            switch(n)
            {
                case 0:
                    if (light_up(g, x, y-1, flag) && light_up(g, x, y+1,
flag) && light_up(g, x+1, y, flag))
                    {
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                        return NULL;
                    }
                    break;
...
```

3.6 实验结果分析

表 3.1 实验结果

串行	7.463 秒
每一层分裂	33.162 秒
仅第一层分裂	3.902 秒

可以看到，如果每一层进行分裂的情况下，时间反而会比串行时间长，由于越到解空间下层，分裂的收益越低，所以仅仅在第一层收益反而效率更高。

实际过程中，可以找到分裂的收益平衡点，当分裂收益=不分裂收益的时候，作为零界点，其上部分解空间进行分裂，而下部分不分裂，从而获得并行加速效果的最大化。

3.7 串行算法复杂度分析

对于m*m的棋盘，其中n个黑格：

黑色格子的串行遍历搜索为O(n)，而在存在剪枝的情况下，复杂度小于$O(m^n)$

具体来说，假定平均n个黑格的val为p的话，复杂度为小于$O(p^n)$。由于存在回溯，无法给出准确的解空间。

3.8 分析并行实现的正确性

并行过程中，设置全局的解标志，当搜索到全局解标志为真的时候，停止搜索，且每一个副本的数据单独存储拷贝，不存在干扰，故并行过程中能够实现相对独立，搜索过程中，方案成组绑定，不存在解空间的交叉，完美的树形结构，故并行算法结果最后正确，但是由于并行的遍历顺序可能和串行不一样，串行是严格的DFS遍历，并行是多个树形结构的枝干向下，故可能先便利到“右边”的解，多解情况下可能和串行结果不一致。

由于题目保证一定有解，故有解就返回，但是如果是无解的情况，并行搜索会搜索到最大深度然后返回无解，结果仍然正确。

3.9 实验总结

3.9.1 规范化

实验过程中，akria是经典的OJ题，故解法比较经典，其中解空间的搜索上有单个搜索(DFS)搜索和成组搜索(每次完成一个解空间)的方法，最初考虑实现的便捷，采用单个搜索，之后考虑到并行的效率，采用成组搜索，最终的解空间不会有重叠。

解空间的并行上，采用每层分裂和仅第一层分裂进行比较，最终第一层分裂的效果比较好。

实际过程中，可以找到分裂的收益平衡点，当分裂收益=不分裂收益的时候，作为零界点，其上部分解空间进行分裂，而下部分不分裂，从而获得并行加速效果的最大化。

3.10 代码

3.10.1 串行

```
#include <bits/stdc++.h>
// # include "akari.h"

using namespace std;

namespace aka
```

```
{

typedef struct point
{
        int x;
        int y;
        int val;
} point;

point black[100];

int cover[20][20];
int type[20][20];
int map[20][20];
int dx[4] = {1,-1,0,0};
int dy[4] = {0,0,1,-1};

bool ok;

int ans;

int n=0;
int m=0;
int b=0;

bool cmp(point A, point B)
{
        return A.val < B.val;
}

bool in_range(int x, int y)
{
        return (x >= 1 && y >= 1 && x <= n && y <= m);
}


int on(int x, int y)// put in (x,y)
{
        int ret=1; //light num
        int i = 0;
        int j = 0;
        for(int k=0;k<4;k++)
        {
                i=x+dx[k];
```

```
                j=y+dy[k];
                while (in_range(i,j)&& type[i][j] != 2)
                {
                        cover[i][j]++;
                        if(cover[i][j] == 1)
                        {
                                ret ++;
                        }
                        i+=dx[k];
                        j+=dy[k];
                }
        }
        return ret;
}

int off(int x,int y)
{
        int ret=1; //light num
        int i = 0;
        int j = 0;
        for(int k=0;k<4;k++)
        {
                i=x+dx[k];
                j=y+dy[k];
                while (in_range(i,j) &&type[i][j] != 2)
                {
                        cover[i][j]--;
                        i+=dx[k];
                        j+=dy[k];
                }
        }
        return ret;
}


int check_overlap()
{
        int i,j;
    int tmp;
    int ret = 2;
    for(i = 0;i < b;i++) // check black
    {
        if(black[i].val == -1)
            continue;
```

```
        tmp = 0;
        for(j = 0;j < 4;j ++)//check black neibor
        {
            int tx = black[i].x + dx[j];
            int ty = black[i].y + dy[j];
                        if(!in_range(tx,ty))
                        {
                                continue;
                        }
            if(type[tx][ty] == 5)
                tmp ++;
        }
        if(tmp > black[i].val)
            return 0;
        if(tmp != black[i].val)
            ret = 1;
    }
    return ret;
}

void dfs_for_white(int x,int y,int count,int depth)
{
        if(ok)
        {
                return;
        }
        if(count == m*n-b) //light all
        {
                ok=true;
                for(int i=1;i<=n;i++)
                {
                        for(int j=1;j<=m;j++)
                        {
                                map[i][j]=type[i][j];
                        }
                }
        }
        for(int i = x;i <= n;i ++)
    {
        for(int j = 1;j <= m;j ++)
        {
            if(type[i][j] == 0 && cover[i][j] == 0)
            {
                type[i][j] = 5;
```

```
                if(check_overlap() != 2) //overlap
                {
                    type[i][j] = 0;
                    continue;
                }
                int light_num = on(i,j);
                dfs_for_white(i,j,count + light_num,depth + 1);
                off(i,j);
                type[i][j] = 0;
            }
        }
    }
        return;
}

void dfs_for_black(int id,int count,int depth)
{
        if(ok)
        {
                return;
        }
        if(id == b)
    {
        if(check_overlap() == 2)// this situ should not
        {
                        // printf("AS");
            dfs_for_white(1,1,count,depth);
        }
        return;
    }
        int i=0;
        int j=0;
        for(i=0;i<4;i++) // chekc neibor
        {
                if(type[black[id].x + dx[i]][black[id].y + dy[i]] == 5)
            j++;
        }
        if(j > black[id].val)
    {
        return;
        }
        else
    {
        if(j == black[id].val)
```

```
            dfs_for_black(id + 1,count,depth);
        else
        {
            for(i = 0;i < 4;i ++)
            {
                int tx = black[id].x + dx[i];
                int ty = black[id].y + dy[i];
                if(tx < 1 || ty < 1 || tx > n || ty > m)
                    continue;
                if(type[tx][ty] == 0 && cover[tx][ty] == 0)
                {
                    type[tx][ty] = 5;
                    if(check_overlap() == 0)
                    {
                        type[tx][ty] = 0;
                        continue;
                    }
                    int tp = on(tx,ty);
                    dfs_for_black(id,count + tp,depth + 1);
                    off(tx,ty);
                    type[tx][ty] = 0;
                }
            }
        }
    }
}




vector<vector<int> > solveAkari(vector<vector<int> > &g)
{
        n=g.size();
        m=g[0].size();
        b=0;
        for(int i=0;i<g.size();i++) // add black to struct
        {
                for(int j=0;j<g[i].size();j++)
                {
                        if(g[i][j]>=-1)
                        {
                                black[b].x=i+1;
                                black[b].y=j+1;
```

```
                                black[b++].val=g[i][j];
                                type[i+1][j+1]=2;
                        }
                }
        }
        sort(black,black+b,cmp);
        ok=false;
        int i;
        for(i=0;i<b;i++)
        {
                if(black[i].val>0)
                {
                        break;
                }
        }
        while(1)
        {
                if(ok||ans>m*n-b)
                {
                        break;
                }
                dfs_for_black(i,0,0);
                if(ok)
                {
                        break;
                }
                ans++;
        }
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++){
                if(map[i][j]==5)
            {
                g[i - 1][j - 1] = 5;
            }
                }
    return g;
}

} // namespace aka
```

3.10.2 并行

```
// 文件由若干行组成，第一行为两个整数 n，m，代表棋盘的行数和列数。

// 之后的 n 行每行有 m 个整数表示棋盘的每个格子的状态，若它为 -2，则表示是白格

子，若它为 -1，则表示是没有数字的黑格子，

// 若它为 0-4，则表示是数字 0-4 的黑格子。若你想把灯泡放在白色格子上面，则需要

将 -2 改为 5，因为 5 表示有灯泡的格子。

#include <bits/stdc++.h>
#include "akari.h"
using namespace std;

//save the result
vector<vector<int> > result;
int get_result = 0;


namespace aka{


int dx[4] = {1,-1,0,0};
int dy[4] = {0,0,1,-1};
int step[4][2] = {{-1,0},{1,0},{0,-1},{0,1}};
int solution4[1][4] = {{1,1,1,1}};
int solution3[4][4] = {{0,1,1,1},{1,0,1,1},{1,1,0,1},{1,1,1,0}};
int solution2[6][4] =
{{1,1,0,0},{1,0,1,0},{0,1,1,0},{0,1,0,1},{0,0,1,1},{1,0,0,1}};
int solution1[4][4] = {{0,0,1,0},{0,0,0,1},{0,1,0,0},{1,0,0,0}};
int solution0[1][4] = {{0,0,0,0}};


struct point
{
    int x, y;
    int val;
};
//    vector<struct point> blacks;

bool cmp(const struct point &A, const struct point &B)
{
```

```
    return A.val < B.val;
}

void print_g(vector<vector<int> > g)
{
    printf("\n");
    for (int i=0; i<g.size(); i++)
    {
        for (int j=0; j<g.at(i).size(); j++)
        {
            printf("%2d ", g.at(i).at(j));
        }
        printf("\n");
    }
}

// can add light beside the point
bool can_add_light(vector<vector<int> > g, int x, int y)
{
    if (x < 0 || y < 0 || x>=g.size() || y>=g.at(0).size())// beside the wall
        return true;
    int max_count = g.at(x).at(y);
    if (max_count<0)
        return true;
    int now_count = 0;
    if (y+1<g.at(x).size() && g.at(x).at(y+1)>=5)
        now_count++;
    if (x+1<g.size() && g.at(x+1).at(y)>=5)
        now_count++;
    if (y-1>=0 && g.at(x).at(y-1)>=5)
        now_count++;
    if (x-1>=0 && g.at(x-1).at(y)>=5)
        now_count++;
    return ++now_count <= max_count;
}


bool in_range(vector<vector<int> > & g,int x, int y)
{
        return !(x < 0 || y < 0 || x>=g.size() || y>=g.at(0).size());
}
```

```
//判断(x,y)能否放一个灯泡，可以则放置
bool light_up(vector<vector<int> > & g, int x, int y, int flag)
{
    if(!in_range(g,x,y))
    {
        return false;
    }
    else if (g.at(x).at(y) >= 5) //already light
    {
        return true;
    }
    else if (g.at(x).at(y) > -2) //wall
    {
        return false;
    }
    else
    {
        //判断灯泡的上下左右格子能否容纳新增加的灯泡
        if (can_add_light(g, x, y+1)
        && can_add_light(g, x, y-1)
        && can_add_light(g, x-1, y)
        && can_add_light(g, x+1, y))
        {
            //判断是否有灯泡互相照亮

            //上
            for (int i=x-1; i>=0; i--)
                if (g.at(i).at(y)>=-1 && g.at(i).at(y)<=4)//碰到墙
                    break;
                else if (g.at(i).at(y) >= 5)
                    return false;

            //下
            for (int i=x+1; i<g.size(); i++)
                if (g.at(i).at(y)>=-1 && g.at(i).at(y)<=4)//碰到墙
                    break;
                else if (g.at(i).at(y) >= 5)
                    return false;

            //左
```

```
            for (int i=y-1; i>=0; i--)
                if (g.at(x).at(i)>=-1 && g.at(x).at(i)<=4)//碰到墙
                    break;
                else if (g.at(x).at(i) >= 5)
                    return false;
            //右
            for (int i=y+1; i<g.at(x).size(); i++)
                if (g.at(x).at(i)>=-1 && g.at(x).at(i)<=4)//碰到墙
                    break;
                else if (g.at(x).at(i) >= 5)
                    return false;
            g.at(x).at(y) = flag;
            return true;
        }
        return false;
    }
    return false;
}

//if all whites are light on
bool is_ok(vector<vector<int> > g, vector<struct point> whites)
{
    for (int i = 0; i < whites.size(); i++)
    {
        if (g.at(whites[i].x).at(whites[i].y) == -2)
        {
            return false;
        }
    }
    return true;
}


void light_on_line(vector<vector<int> > &g,int x, int y)
{
    for (int k=x+1; k<g.size(); k++)
    {
        if (g.at(k).at(y)>=-1 &&  g.at(k).at(y)<=4)
            break;
        else
            g.at(k).at(y) = -3;
```

```
    }
    for (int k=x-1; k>=0; k--)
    {
        if (g.at(k).at(y)>=-1 &&  g.at(k).at(y)<=4)
            break;
        else
            g.at(k).at(y) = -3;
    }
    for (int k=y+1; k<g.at(x).size(); k++)
    {
        if (g.at(x).at(k)>=-1 &&  g.at(x).at(k)<=4)
            break;
        else
            g.at(x).at(k) = -3;
    }
    for (int k=y-1; k>=0; k--)
    {
        if (g.at(x).at(k)>=-1 &&  g.at(x).at(k)<=4)
            break;
        else
            g.at(x).at(k) = -3;
    }
}

void light_init(vector<vector<int> > &g)
{
    for (int i=0; i<g.size(); i++)
    {
        for (int j=0; j<g[i].size(); j++)
        {
            if (g.at(i).at(j) >= 5) //already light here
            {
                for (int k=i+1; k<g.size(); k++)
                {
                    if (g.at(k).at(j)>=-1 &&  g.at(k).at(j)<=4)
                        break;
                    else
                        g.at(k).at(j) = -3;
                }
                for (int k=i-1; k>=0; k--)
                {
                    if (g.at(k).at(j)>=-1 &&  g.at(k).at(j)<=4)
                        break;
                    else
```

```
                        g.at(k).at(j) = -3;
                }
                for (int k=j+1; k<g.at(j).size(); k++)
                {
                    if (g.at(i).at(k)>=-1 &&  g.at(i).at(k)<=4)
                        break;
                    else
                        g.at(i).at(k) = -3;
                }
                for (int k=j-1; k>=0; k--)
                {
                    if (g.at(i).at(k)>=-1 &&  g.at(i).at(k)<=4)
                        break;
                    else
                        g.at(i).at(k) = -3;
                }
            }
        }
    }
}


void light_off_line(vector<vector<int> > &g,vector<struct point> whites)
{
    // off all
    for (int i = 0; i < whites.size(); i++)
    {
        if (g.at(whites[i].x).at(whites[i].y) == -3)
        {
            g.at(whites[i].x).at(whites[i].y) = -2;
        }
    }
    // light again
    light_init(g);
}

// the n-th white
bool back_tracking2(vector<vector<int> > g, vector<struct point> whites, int
n)
{
    if (n == whites.size())
    {
        if (is_ok(g, whites))
        {
```

```
            result.resize(g.size());// copy result
            result.assign(g.begin(),g.end());
            get_result = 1;
            return true;
        }
        else// get the final state
        {
            return false;
        }
    }
    struct point tmp;
    tmp = whites[n];
    int val = g.at(tmp.x).at(tmp.y);
    if (val == -2) //white point
    {
        if (light_up(g, tmp.x, tmp.y, 5)) // can put light
        {
            g.at(tmp.x).at(tmp.y) = 5;
            light_on_line(g,tmp.x, tmp.y);
            if (back_tracking2(g, whites, n+1))
            {
                return true;
            }
            else
            {
                g.at(tmp.x).at(tmp.y) = -2; // back_tracking here
                light_off_line(g,whites);
                return back_tracking2(g, whites, n + 1);
            }
        }
    }
    return back_tracking2(g, whites, n+1);
}


void white_init(vector<vector<int> > g,vector<struct point> &whites)
{
    struct point tmp;
    for (int i=0; i<g.size(); i++)
    {
        for (int j=0; j<g[i].size(); j++)
        {
            if (g.at(i).at(j) == -2)
            {
```

```
                tmp.x = i;
                tmp.y = j;
                whites.push_back(tmp);
            }
        }
    }
}

bool light_all(vector<vector<int> > g)
{
    light_init(g);

    vector<struct point> whites;
    white_init(g,whites);

    return back_tracking2(g, whites, 0);
}

typedef struct Data{
    vector<vector<int> > g;
    vector<struct point> blacks;
    int n;
    int depth;
    bool re;
    int flag;
}Data;

bool startPthread(vector<vector<int> > g, vector<struct point> blacks, int
depth, int flag);

void * back_tracking0(void * d)
{
    Data *data = (Data *)d;
    vector<vector<int> > g = data->g;
    int n = data->n;
    int depth = data->depth;
    vector<struct point> blacks(data->blacks);
    if (depth == blacks.size())
    {
        cout << "error";
        data->re = false;
        return NULL;
    }
    struct point tmp = blacks[depth];
```

```
    int x = tmp.x;
    int y = tmp.y;
    int flag = data->flag;
    int total = 0;
    switch (tmp.val)
    {
        case 4:
            if (light_up(g, x-1, y, flag) && light_up(g, x+1, y, flag) &&
                light_up(g, x, y-1, flag) && light_up(g, x, y+1, flag))
            {
                data->re = startPthread(g, blacks, depth+1, flag+1);
                return NULL;
            }
            data->re = false;
            return NULL;
            break;
        case 3:
            switch(n)
            {
                case 0:
                    if (light_up(g, x, y-1, flag) && light_up(g, x, y+1,
flag) && light_up(g, x+1, y, flag))
                    {
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                        return NULL;
                    }
                    break;
                case 1:
                    if (light_up(g, x+1, y, flag) && light_up(g, x, y+1,
flag) && light_up(g, x-1, y, flag))
                    {
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                    }
                    break;
                case 2:
                    if (light_up(g, x, y+1, flag) && light_up(g, x-1, y,
flag) && light_up(g, x, y-1, flag))
                    {
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                    }
                    break;
```

```
                case 3:
                    if (light_up(g, x-1, y, flag) && light_up(g, x, y-1,
flag) && light_up(g, x+1, y, flag))
                    {
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                    }
                    break;
            }
            data->re = false;
            return NULL;
            break;
        case 2:
            switch(n)
            {
                case 0:
                    if (light_up(g, x-1, y, flag) && light_up(g, x, y-1,
flag))
                    {
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                        if (data->re)
                            return NULL;
                    }
                    break;
                case 1:

                    if (light_up(g, x-1, y, flag) && light_up(g, x+1, y,
flag))
                    {
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                        if (data->re)
                            return NULL;
                    }
                    break;
                case 2:

                    if (light_up(g, x-1, y, flag) && light_up(g, x, y+1,
flag))
                    {
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                        if (data->re)
```

```
                            return NULL;
                    }
                    break;
                case 3:

                    if (light_up(g, x, y-1, flag) && light_up(g, x+1, y,
flag))
                    {
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                        if (data->re)
                            return NULL;
                    }
                    break;
                case 4:

                    if (light_up(g, x, y-1, flag) && light_up(g, x, y+1,
flag))
                    {
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                        if (data->re)
                            return NULL;
                    }
                    break;
                case 5:

                    if (light_up(g, x+1, y, flag) && light_up(g, x, y+1,
flag))
                    {
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                        if (data->re)
                            return NULL;
                    }
                    break;
            }
            data->re = false;
            return NULL;
            break;
        case 1:
            switch(n)
            {
                case 0:
```

```
                    if (light_up(g, x-1, y, flag))
                    {
                        // g.at(x-1).at(y) = flag;
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                        if (data->re)
                            return NULL;
                    }
                    break;
                case 1:

                    if (light_up(g, x+1, y, flag))
                    {
                        // g.at(x+1).at(y) = flag;
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                        if (data->re)
                            return NULL;
                    }
                    break;
                case 2:

                    if (light_up(g, x, y-1, flag))
                    {
                        // g.at(x).at(y-1) = flag;
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                        if (data->re)
                            return NULL;
                    }
                    break;
                case 3:

                    if (light_up(g, x, y+1, flag))
                    {
                        // g.at(x).at(y+1) = flag;
                        data->re = startPthread(g, blacks, depth+1,
flag+1);
                        if (data->re)
                            return NULL;
                    }
                    break;
            }
```

```
            data->re = false;
            return NULL;
            break;
    }
    return NULL;
}


bool startPthread(vector<vector<int> > g, vector<struct point> blacks, int
depth, int flag)
{
    pthread_t* thread_list;
    Data* data;
    if (depth == blacks.size()) // finial state
    {
        return light_all(g); //solve single thread
    }

    struct point black_now = blacks[depth];


    int n = g.at(black_now.x).at(black_now.y);
    int thread_num = 0;

    switch(n)
    {
        //case_number
        case 4:
            thread_num = 1;
            break;
        case 3:
            thread_num = 4;
            break;
        case 2:
            thread_num = 6;
            break;
        case 1:
            thread_num = 4;
            break;
    }

    thread_list = (pthread_t *)malloc(sizeof(pthread_t)*thread_num);
    data = (Data*)malloc(sizeof(Data)*thread_num);
```

```
    //init thread
    memset(data, 0, sizeof(Data)*thread_num);
    for (int i=0; i<thread_num; i++)
    {
        for (int j=0; j<g.size(); j++)
        {
            vector <int> tmp_int;
            for (int k=0; k<g.at(j).size(); k++)
            {
                tmp_int.push_back(g.at(j).at(k));
            }
            data[i].g.push_back(tmp_int);
        }

        data[i].blacks.resize(blacks.size());
        for (int j=0; j<blacks.size(); j++)
        {
            data[i].blacks[j] = blacks[j];
        }
        data[i].n = i;
        data[i].re = false;
        data[i].depth = depth;
        data[i].flag = flag;


        int ret = pthread_create(&thread_list[i], NULL, back_tracking0,
(void *)&data[i]);
        if (ret != 0)
        {
            printf("error");
        }
    }


    for (int i=0; i<thread_num; i++)
    {
        pthread_join(thread_list[i], NULL);
    }

    for (int i=0; i<thread_num; i++)
    {
        if (data[i].re)
        {
               free(thread_list);
```

```
                free(data);
            return true;
        }
    }
    free(thread_list);
    free(data);
    return false;
}



bool back_tracking(vector<vector<int> > g, vector<struct point> blacks)
{
    return startPthread(g, blacks, 0, 5);
}



vector<vector<int> > solveAkari(vector<vector<int> > & g)
{
    struct point b;
    vector <struct point> blacks;
    for (int i=0; i<g.size(); i++)
    {
        int z = g.at(i).size();
        for (int j=0; j<g.at(i).size(); j++)
        {
            if (g[i][j]>0 && g[i][j]<=4)
            {
                b.x = i;
                b.y = j;
                b.val = g[i][j];
                blacks.push_back(b);
            }
        }
    }
    sort(blacks.begin(), blacks.end(), cmp);

    back_tracking(g, blacks);
    //recover light to white
    for (int i=0; i<result.size(); i++)
    {
        for (int j=0; j<result.at(i).size(); j++)
        {
```

```
            if (result.at(i).at(j) == -3)
                result.at(i).at(j) = -2;
            else if (result.at(i).at(j) >= 5)
                result.at(i).at(j) = 5;
        }
    }
    return result;
}

}
```