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 Project

1.1 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。

1.2 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

1.3 算法描述

1.3.1 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。

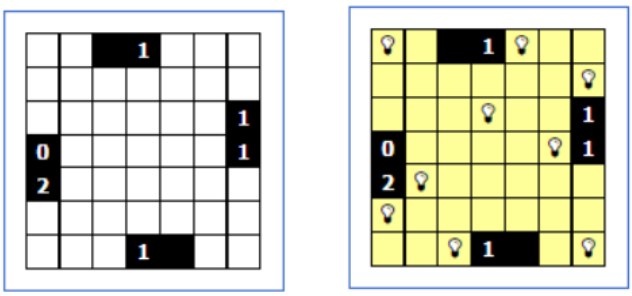

图 1.1 Akari 的初始状态与解

1.3.2 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.

1.3.3 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。

1.4 串行实验方案

1.4.1 数据结构定义

代码 1.1 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。

1.4.2 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

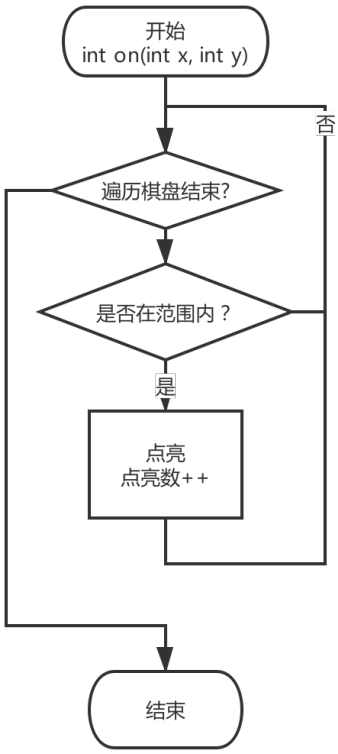

图 1.2 点亮流程图

代码 1.2 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 1.3 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;
}
```

1.4.3 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 1.4 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 1.5 边界判定

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

1.4.4 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 1.6 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 1.7 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

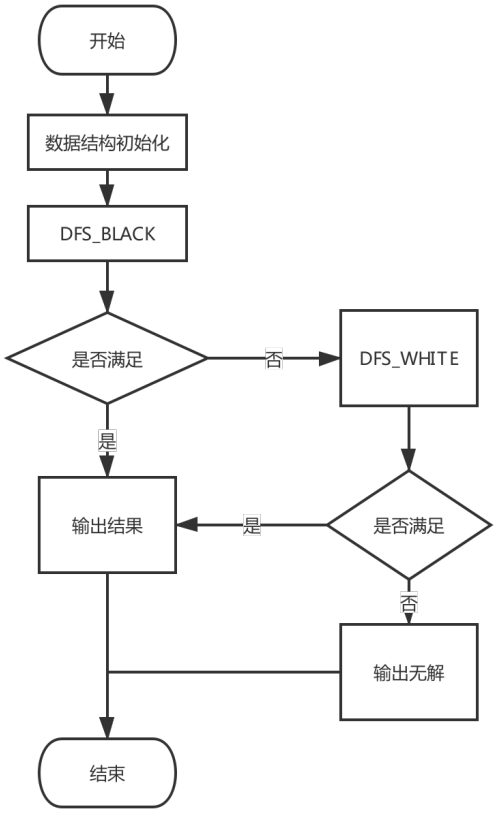

图 1.3 整体操作流程图

代码 1.8 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;
}
```

1.5 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。

1.5.1 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。

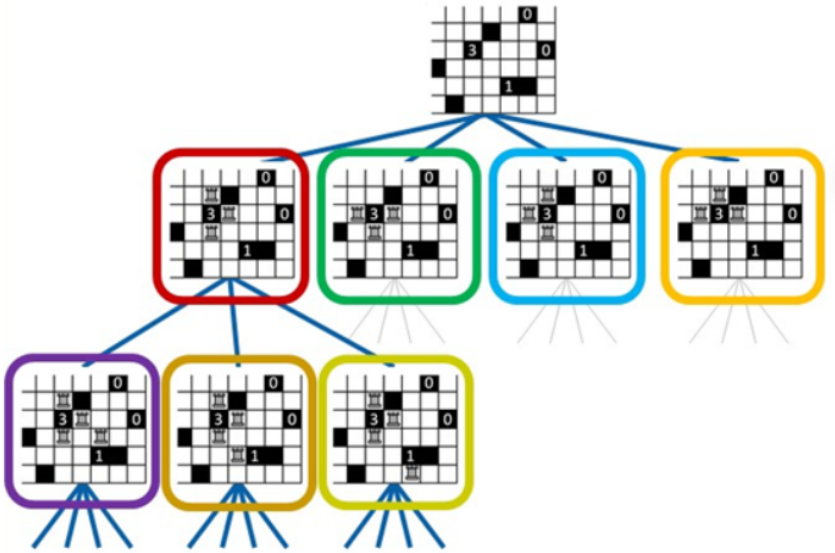

图 1.4 并行的 Akari 回溯算法

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

1.5.2 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 1.9 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 1.10 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;
...
```

1.6 实验结果分析

表 1.1 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。

1.7 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。

1.8 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。

1.9 实验总结

1.9.1 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。

1.10 代码

1.10.1 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
```

1.10.2 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;
}

}
```