

**课 程 实 验 报 告**

**课程名称： 并行编程原理与实践**

**专业班级： CS1602**

**学 号： U201614545**

**姓 名： 谭胜克**

**指导教师： 金海**

**报告日期： 2019/7/22**

**计算机科学与技术学院**

**目录**

[1 回溯法解决Akari问题 1](#_Toc15383323)

[1.1实验目的与要求 1](#_Toc15383324)

[1.2问题描述 1](#_Toc15383325)

[1.3算法描述 2](#_Toc15383326)

[1.4实验方案 3](#_Toc15383327)

[1.5实验结果与分析 4](#_Toc15383328)

[2 并行回溯法求解Akari问题 5](#_Toc15383329)

[2.1实验目的与要求 5](#_Toc15383330)

[2.2算法描述 5](#_Toc15383331)

[2.3 实验方案 5](#_Toc15383332)

[2.4 实验结果与分析 5](#_Toc15383333)

[3 改进的并行回溯法求解Akari问题 6](#_Toc15383334)

[3.1实验目的与要求 6](#_Toc15383335)

[3.2算法描述 6](#_Toc15383336)

[3.3实验方案 6](#_Toc15383337)

[3.4实验结果与分析 6](#_Toc15383338)

[4 实验小结 7](#_Toc15383339)

[**附件：源代码** 8](#_Toc15383340)

# 1 回溯法解决Akari问题

## 1.1实验目的与要求

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问题描述

* Akari问题描述

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，同属于Nikoli谜题的除Akari之外还有Sudoku(数独)和Kakuro(数谜)等。

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

* 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
* 编程要求

编程实现使用回溯法解决Akari问题。要求首先对问题进行划分，并根据每一步构造解空间，进而设计合理的数据结构与程序结构对问题进行求解，并在此基础上对算法以及程序逻辑进行优化和改进，最后进行问题的分析、讨论和展望。

* 格式说明

输入数据由若干文件组成，每个文件描述一个Akari问题的初始状态，编写程序读入此文件，根据初始状态求解。有的有一个解，有的有多个解，我们保证有解。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。

## 1.3算法描述

1. 初始构想

在一开始接触Akari问题时，看到回溯法，首先就想到了8皇后问题的求解方式：从第一行第一列开始摆放皇后，逐行逐列进行尝试。若是在尝试过程中能够判断出当前方案下不可能得出解，则进行回溯，回到上一次分支开始的地方。重新进入下一个分支。

在实际执行时发现Akari问题如果采用逐行逐列进行尝试的回溯法，这个过程远比8皇后问题复杂。其主要难点就在于如何进行裁支工作，由于存在黑色方块的干扰，使得裁支难以进行，而必须等到整个矩阵遍历结束之后才能够判断。可以想象，这与遍历法解决相差无几，而且在执行过程中的判断算法还非常复杂。思考过后放弃了这个方案。

1. 选定方案

放弃了初始的想法之后，一直没有想到什么更好的方案去解决问题。某次登录网站认真查看了题目给出的编程要求之后，茅塞顿开。

将问题分为对黑块的处理和对白块的处理。对黑块的处理就是根据黑块中的数字，在其周围四个方向（上、下、左、右）填上相应个数的灯泡，那么在不同数字的黑块周围灯泡的摆放就存在不同数量的方案，我们应该做的就是穷举这些方案，若是在某个方案的某一步发现没有必要进行，则进行回溯。所以在每次分支前都需要将原矩阵进行保存，在回溯时就能够通过原始矩阵尝试下一种摆放方案。

而对于白块的处理则是放在所有的黑块处理完成之后，当所有的黑色方块都能够按照其中的数字，在其周围摆着灯泡，这时候仍然有可能存在未被一些死角，没有被灯光所覆盖。所以应该尝试着在这些空白位置放置灯泡。当没有空白位置，但是矩阵并不符合最终的要求时，本次方案失败，开始回溯，重新回到黑块解决阶段。

1. 开始实施

黑块处理阶段首要任务是如何将黑块提取出来并对黑块进行管理，这里用的是链表结构对黑块进行存储，每一个链表结点存储黑块中的数字，黑块的行、列，以及下一个黑块指针。链表结构如图1.1所示。按照黑色方块的数字顺序分别对矩阵进行便利，构造黑色方块链表。



图1.1 链表结点结构

接下来就是开始穷举方案，尝试在黑色方块周围填入灯泡。比如黑色方中的数字为4，那么只存在一种摆放方案，即上下左右均放置；若是黑色方块中的数字为2，则存在6只种灯泡的放置方案。上左、上下、上右、左下、左右、下右的格子中放置“灯泡”。其它数字的方块依次类推。在每一中方案开始之前，都将当前矩阵进行一次保存，就是将复制过后的数组传入递归中进行处理，使得在回溯时能够恢复到当前状态。当黑块链表处理完成，就进入到剩余白块的处理阶段。

白块的处理相对简单，找出当前所有的空白方块，按顺序往上摆放灯泡，期间每放置一盏灯泡就判断一次是否完成，未完成则继续，完成则返回。若是不存在下一个白块，并且没有完成，则说明此种方案不可行，需要回溯到黑块处理。

在黑块处理中，用到了两个功能函数：一个是放置灯泡的函数：put\_blub；另一个是放置断点标签的函数：put\_dot。put\_blub在传入的行、列位置放置一盏灯泡，成功返回1，失败返回0，同时若成功放置一盏灯泡，则将其周围能够点亮的方块点亮。put\_dot主要是标记出那些不应该存在灯泡的方块，这样在接下来的白块处理阶段就可以省略很多时间。

最终还需要一个检验函数，判断我们当前的状态下结果是否是正确的。按照算法思路以及题目要求，这里的判断函数首先要判断矩阵中是否存在空白位置，也就是没有被灯光覆盖到的当块，或者是存在断点；另一个，要判断所有黑色方块周围的灯泡数量数否与其中的数字相符。至于灯泡不能摆放在同一列、同一行是不用进行检测的，这一步的检测在灯泡放置阶段已经进行，不可能出现如此问题。

## 1.4实验方案

1. 在本地编写调试代码
2. 修改代码用于平台测试

在本地测试时，用的是从文件获取输入矩阵的方式进行输入，而平台给出的调用函数给的是二维的vector，所以要将相应的输入方式进行修改。

1. 进行检测

第一次检测时，7个测试集并没有完全通过，而是仅仅通过了4个，剩余3个测试集，分别是4、6、7未能通过测试。4、6测试集通过算法给出的提示为无解，而7的提示是黑色方块周围的灯泡个数不符合。

1. 修改错误测试集7

首先认真观察了第7项测试集给出的结果矩阵，发现确实存在一个黑色方块周围的灯泡数量与其中的数字不和。但是按照黑块的处理方式并不应该出现这种情况，最后我想了想，既然是结果不符合，那么修改检测函数就可以，于是在检测函数中加入了黑块数量检测的内容，再次测试，通过了第7项测试集。

1. 修改错误测试集4、6

4号、6号测试集给出的提示都为无解。这就让我很难定位错误的存在。想了好久想到会不会是黑块之间相互影响了，就是我在构造黑色方块链表的时候采用的是一次遍历的方法，只要遇到黑色方块就将其插入链表尾部，而没有区分黑块数字。那么很有可能在处理某一个方块时，将其周围的某一位置打上了断点，使得接下来的某一黑块不能按照其中的数字摆放灯泡，从而导致了无解。想到了这一点，我对黑块链表的构造函数稍加修改，改为了多次遍历，按照数字顺序分别插入队列。再次检测，问题解决，通过了第4、第6项测试，完美通关。

## 1.5实验结果与分析

如图1.2所示，为Akari问题在平台的测试结果。7项测试均成功通过，说明算法能够正确解决Akari问题。



图1.1 链表结点结构

# 2 并行回溯法求解Akari问题

## 2.1实验目的与要求

## 2.2算法描述

## 2.3 实验方案

## 2.4 实验结果与分析

# 3 改进的并行回溯法求解Akari问题

## 3.1实验目的与要求

## 3.2算法描述

## 3.3实验方案

## 3.4实验结果与分析

# 4 实验小结

**附件：源代码**

1. 串行回溯法

# include <bits/stdc++.h>

# include "akari.h"

using namespace std;

namespace aka{

//请在命名空间内编写代码，否则后果自负

#include<stdio.h>

#include<stdlib.h>

#include<assert.h>

#include<time.h>

#define EMPTY -2 //空白位置 -2 0

#define DOT -3 //无法放置 -3 1

#define COVER -4 //被点亮 -4 2

#define BARRIER -1 //无数字黑块 -1 3

#define BARRIER\_0 0 //数字0黑块 0 4

#define BARRIER\_1 1 // 1 5

#define BARRIER\_2 2 // 2 6

#define BARRIER\_3 3 // 3 7

#define BARRIER\_4 4 // 4 8

#define BLUB 5 //电灯 5 9

typedef int Elem\_type;

/\*黑色方块结点 \*/

typedef struct Node{

int number; //黑色方块中的数字0～4

size\_t row;//size\_t是unsigned int

size\_t col;

struct Node\* next;

}Num\_barrier, \*Barrier\_list;

int m;//row num

int n;//column num

Elem\_type\* g\_broad;//the global broad

void read\_dimension(int\*, int\*);

void read\_broad(Elem\_type\*);

void print\_broad(Elem\_type\*);

Barrier\_list create\_barrier\_list(Elem\_type\*);

void copy\_broad(Elem\_type\*, const Elem\_type\*);

int check\_complete(Elem\_type\*);

int solve\_puzzle(Barrier\_list, Elem\_type\*);

int put\_blub(int, int, Elem\_type\*);

int put\_dot(int, int, Elem\_type\*);

int handle\_empty(int, Elem\_type\*);

int next\_empty(int, Elem\_type\*);

vector<vector<int> > solveAkari(vector<vector<int> > & g)

{

// 请在此函数内返回最后求得的结果

//read\_dimension(&m, &n);//读取行数和列数

m = g.size();

n = g[0].size();

g\_broad = (Elem\_type\*) malloc(sizeof(Elem\_type) \* m \* n);//动态分配内存给矩阵

//read\_broad(g\_broad);//读取文件中的数据存入变量g\_broad中

for(int i = 0; i < m; i++){

for(int j = 0; j < n; j++){

g\_broad[i\*m+j] = g[i][j];

}

}

Barrier\_list barriers = create\_barrier\_list(g\_broad);

int solved = solve\_puzzle(barriers, g\_broad);

if(!solved) printf("no solution!\n");

Barrier\_list p;

for(p = barriers; p;) {

barriers = barriers->next;

free(p);

p = barriers;

}

g.clear();

vector<int> temp;

for(int i = 0; i < m; i++){

temp.clear();

for(int j = 0; j < n; j++){

temp.push\_back(g\_broad[i\*m+j]);

}

g.push\_back(temp);

}

//free(g\_broad);

return g;

}

/\*从文件读取数据：行、列 \*/

void read\_dimension(int\* row, int\* col) {

//fscanf(fp, "%d %d", row, col);

scanf("%d %d", row, col);

}

/\*从文件读取初始数据 \*/

void read\_broad(Elem\_type\* broad) {

int i;

for(i = 0; i < m\*n; i++)

//fscanf(fp, "%d", &broad[i]);

scanf("%d", &broad[i]);

}

/\*输出矩阵 \*/

void print\_broad(Elem\_type\* broad) {

int r, c;

for(r = 0; r < m; r++) {

for(c = 0; c < n; c++)

printf("%d ", broad[r\*n + c]);

printf("\n");

}

}

Barrier\_list create\_barrier\_list(Elem\_type\* broad) {//创建黑色方块链表

int i;

Barrier\_list header, p;

header = p = NULL;

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_0){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 0;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_4){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 4;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_3){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 3;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_1){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 1;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

for(i = 0; i < m\*n; i++){

if(broad[i] == BARRIER\_2){

Barrier\_list q = (Barrier\_list)malloc(sizeof(Num\_barrier));

q->row = i / n;

q->col = i % n;

q->number = 2;

(header == NULL) ? (header = p = q) : (p->next = q, p=p->next);

}

}

if(p != NULL) p->next = NULL;

return header;

}

int solve\_puzzle(Num\_barrier\* barrier, Elem\_type\* broad) {

if(NULL != barrier) {//first phase：处理黑块

Num\_barrier\* p;

Elem\_type\* cp\_broad;

int r, c;

int handle = 0;

cp\_broad = (Elem\_type\*)malloc(sizeof(Elem\_type) \* m \* n);

copy\_broad(cp\_broad, broad);

p = barrier;

r = p->row;

c = p->col;

switch(p->number){

case 4: //一种可能

if(put\_blub(r-1, c, cp\_broad)

&&put\_blub(r, c-1, cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_blub(r, c+1, cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

break;

case 3: //四种可能

if(put\_blub(r-1, c, cp\_broad)

&&put\_blub(r, c-1,cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_dot(r, c+1,cp\_broad)) {

handle = solve\_puzzle(p->next, cp\_broad);

}

//restore the state of broad

copy\_broad(cp\_broad, broad);

if(!handle && put\_dot(r-1, c, cp\_broad)

&&put\_blub(r, c-1,cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_blub(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

//

copy\_broad(cp\_broad, broad);

if(!handle && put\_blub(r-1, c, cp\_broad)

&&put\_dot(r, c-1,cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_blub(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

//

copy\_broad(cp\_broad, broad);

if(!handle && put\_blub(r-1, c, cp\_broad)

&&put\_blub(r, c-1,cp\_broad)

&&put\_dot(r+1, c, cp\_broad)

&&put\_blub(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next,cp\_broad);

break;

case 2: //6种可能

if(put\_blub(r-1, c, cp\_broad)

&&put\_blub(r, c-1,cp\_broad)

&&put\_dot(r+1, c, cp\_broad)

&&put\_dot(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad, broad);

if(!handle && put\_dot(r-1, c, cp\_broad)

&&put\_blub(r, c-1,cp\_broad)

&&put\_dot(r+1, c, cp\_broad)

&&put\_blub(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad, broad);

if(!handle && put\_blub(r-1, c, cp\_broad)

&&put\_dot(r, c-1,cp\_broad)

&&put\_dot(r+1, c, cp\_broad)

&&put\_blub(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad,broad);

if(!handle && put\_dot(r-1, c, cp\_broad)

&&put\_dot(r, c-1,cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_blub(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad, broad);

if(!handle && put\_dot(r-1, c, cp\_broad)

&&put\_blub(r, c-1,cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_dot(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad, broad);

if(!handle && put\_blub(r-1, c, cp\_broad)

&&put\_dot(r, c-1,cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_dot(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

break;

case 1 : //四种可能

if(put\_blub(r-1, c, cp\_broad)

&&put\_dot(r, c-1,cp\_broad)

&&put\_dot(r+1, c, cp\_broad)

&&put\_dot(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad, broad);

if(!handle && put\_dot(r-1, c, cp\_broad)

&&put\_blub(r, c-1,cp\_broad)

&&put\_dot(r+1, c, cp\_broad)

&&put\_dot(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad, broad);

if(!handle && put\_dot(r-1, c, cp\_broad)

&&put\_dot(r, c-1,cp\_broad)

&&put\_blub(r+1, c, cp\_broad)

&&put\_dot(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

copy\_broad(cp\_broad, broad);

if(!handle && put\_dot(r-1, c, cp\_broad)

&&put\_dot(r, c-1,cp\_broad)

&&put\_dot(r+1, c, cp\_broad)

&&put\_blub(r, c+1,cp\_broad))

handle = solve\_puzzle(p->next, cp\_broad);

break;

case 0:

if(put\_dot(r-1, c,cp\_broad) //上方

&& put\_dot(r, c-1, cp\_broad) //左边

&& put\_dot(r+1, c, cp\_broad) //下边

&& put\_dot(r, c+1, cp\_broad)) //右边

handle = solve\_puzzle(p->next, cp\_broad);

break;

default:

break;

}

//free(cp\_broad);

cp\_broad = NULL;

return handle;

} else {//second phase：处理白块

return handle\_empty(next\_empty(-1, broad), broad);

}

}

/\*\* name:next\_empty

\* function:获取下一个未被灯光覆盖的方块

\* input：int cur：当前坐标， int\* broad：处理过的数据数组

\* return：i：下一个方块坐标； -1:不存在未被覆盖的方块

\*/

int next\_empty(int cur, Elem\_type\* broad){

int i;

for(i = cur+1; i < m \*n; i++)

if(broad[i] == EMPTY)

return i;

return -1;

}

/\*\*name：handle\_empty

\* function：处理未被点亮的空白方块

\* input：int cur：空白方块坐标

\* int\* broad：处理完成黑色方块的数据数组

\* output：处理结果 1:处理完成，得到正确结果

\* 0:处理失败，不存在解

\*/

int handle\_empty(int cur, Elem\_type\* broad) {

Elem\_type\* cp\_broad;

int handle = 0;

if(check\_complete(broad)){ //完全覆盖，得出正确结果

copy\_broad(g\_broad, broad);

return 1;

}

if(cur == -1) //没有空白位置

return 0;

cp\_broad = (Elem\_type\*)malloc(sizeof(Elem\_type) \* m \* n);

copy\_broad(cp\_broad, broad);

put\_blub(cur / n, cur % n, cp\_broad);//在cur所在的位置放一个灯，cur/n和cur%n是为了得到坐标

handle = handle\_empty(next\_empty(cur, cp\_broad), cp\_broad);

if(!handle){

//retore the previous state of the broad

copy\_broad(cp\_broad, broad);

handle = handle\_empty(next\_empty(cur, cp\_broad), cp\_broad);

}

free(cp\_broad);

return handle;

}

/\*

\*Copy the src broad to des

\*/

void copy\_broad(Elem\_type\* des, const Elem\_type\* src) {

int i;

for(i = 0; i < m\*n; i++)

des[i] = src[i];

}

/\*\* name：check\_complete

\* function：最终效果检查，是否所有的空白位置都被灯光覆盖

\* input：int\* broad：处理过后的数据数组

\* return：1:不存在未被照亮的方块 0:存在未被照亮的方块

\*/

int check\_complete(Elem\_type\* broad){

int i;

for(i = 0; i < m\*n; i++)

if(broad[i] == EMPTY || broad[i] == DOT) //存在empty or dot broad[i] < COVER

return 0;

int ps[4][2] = {-1, 0, 1, 0, 0, -1, 0, 1};

for (int i = 0; i < m\*n; i++) {

if(broad[i] >= 0 && broad[i]<=4){

int cnt = 0;

int x = i/n;

int y = i%n;

for (int k = 0; k < 4; ++k) {

int dx = x + ps[k][0], dy = y + ps[k][1];

int tmp = dx\*m + dy;

if (dx >= 0 && dx < n && dy >= 0 && dy < m && broad[tmp] == 5) ++cnt;

}

if (cnt != broad[i]) {

return 0;

}

}

}

return 1;

}

/\*\*name：put\_blub

\* function：在坐标（row，col）处放置灯泡，并且点亮相应方块

\* input：int row：x坐标

\* int col：y坐标

\* int\* broad：处理过的数据数组

\* return：放置结果 1:成功 0:失败

\*/

int put\_blub(int row, int col, Elem\_type\* broad) {

int r, c;

//坐标是否合法，非法无需处理

if((row < 0 && row >= m)

|| (col < 0 && col >= n))

return 0;

//judge can put? ：为cover也不能放置，因为该坐标的行或列上一定存在电灯

if(broad[row \* n + col] != EMPTY && broad[row\*n+col] != BLUB)

return 0;

else if(broad[row\*n+col] == BLUB)

return 1;

broad[row \* n + col] = BLUB;

// illuminate four directions

for(r = row -1 , c = col; r >= 0 && broad[r\*n+c] < BARRIER; r--)

broad[r\*n + c] = COVER;

for(r = row, c = col - 1; c >= 0 && broad[r\* n+c] < BARRIER; c--)

broad[r\*n + c] = COVER;

for(r = row + 1, c = col; r < m && broad[r\*n + c] < BARRIER; r++)

broad[r\*n + c] = COVER;

for(r = row, c = col +1; c < n && broad[r\*n + c] < BARRIER; c++)

broad[r\*n + c] = COVER;

return 1;

}

/\*\*name：put\_dot

\* function：设置标记点，标记该位置不能放置灯泡，优化后续的白块处理效率

\* input：int row：x坐标

\* int col：y坐标

\* int\* broad：处理过后的数据数组

\* return：能否放置标记 1:放置成功 0:放置失败

\*/

int put\_dot(int row, int col, Elem\_type\* broad) {

if((row >= 0 && row < m)

&& (col >= 0 && col < n)){ //判断位置是否在矩阵内

if(broad[row\*n + col] == BLUB)

return 0;

if(broad[row\*n + col] == EMPTY) {

broad[row\*n + col] = DOT;

}

}

return 1;

}

}