目 录

1 实验一 使用回溯法解决 Akari 问题

1.1 实验目的与要求

- 了解 Akari 问题，探索解决办法
- 掌握回溯法原理
- 基于回溯法解决 Akari 问题
- 调试并行程序并分析结果

1.2 问题描述

Akari 问题又叫 Light Up 问题，源于日本逻辑解谜游戏 Nikoli，google 一下即可找到网站进行在线解密。如果想快速了解这个游戏，可以自己尝试一下不同难度的谜题。

游戏规则是在一张棋盘中，有白色格子，黑色格子。白色格子代表空格，可以放置灯泡，灯泡向上下左右四个方向发射光。黑色格子分为没有数字和有数字的，没有数字的代表障碍，不能放置灯泡，但是可以挡住灯泡的光，有数字的代表此黑格周围应该有几个灯泡。并且两个灯泡不能出现同一行同一列。

最终只要棋盘白色格子全部被照亮即可。

1.3 算法描述

本题要求采用回溯法求解 Akari 问题。回溯法是一种迭代法，首先将解分为若干步，然后选择一种继续构造解空间，在解空间树上进行 DFS，直到找到第一个解为止，若遍历完全部解，仍然没有找到，则说明没有解。

下面是回溯法的一般算法。

```
1.  function BT(c) {
2.    if (DeadEnd(c)) return;
```

```
3.    if (Solution(c)) Output(c);
4.    else
5.      foreach (s = next moves from c) {
6.        BT(s);
7.      }
8.  }
```

首先需要划分问题。

将一步定义为在一个有数字的黑色方格周围放置规定数量的灯泡。

要完成这个，就需要定义解决黑色方格的顺序，创建一个 list，用来保存需要处理的黑色方格。

首先第一次遍历，找到数字为 0 的黑色方格，一个个加入 list，因为这种情况下只有一个解，就是四周都不能放灯泡，在我们的解法里，就给它四周放置✕。

第二次遍历，找到数字为 4 的黑色方格，加入 list。同理，这种情况也是只有一种解，就是给四周全部放上灯泡。

第三次遍历，找到数字为 3 的黑色方格，加入 list。因为这种情况下解法有 4 种，较少。

第四次遍历，找到数字为 1 的黑色方格，加入 list。因为这种情况下解法有 4 种，较少。

第五次遍历，找到数字为 2 的黑色方格，加入 list。因为这种情况下解法有 6 种，最多。所以将它放在最后，避免一开始就分了过多的情况，做很多无用功。

如图 1-1 创建黑色方格列表流程图。

这样就做好了第一步，为接下来的回溯做好了工作。

我们还需要为棋盘上的格子设置宏定义，方便后续工作。比如把空格设置为 0，✕设为 1，灯光设为 2，无数字黑格设为 3，有 0-4 数字的分别设为 4-8，灯泡设为 9.

接着就进入到解决 akari 问题的核心了。首先我们需要把棋盘复制一下，便于后面回溯到错误解后恢复原来的棋盘。从 list 第一个开始，根据黑色方块的数字，决定放置策略。比如，如果数字为 4，就代表在这个方块的四周都需要放上灯泡，

这时候就需要 put_lightbulb 这个函数了。

这个函数的作用就是放置灯泡，如果超过边界，就返回 0，代表这种放置不正确。如果放置的格子不是空且不是灯泡，那么说明不能放置，返回 0.丹如果放置的位置是灯泡，那么说明其他步已经在此放过了，返回 1.否则，就在要放置的位置放上灯泡（9）。接下来，就向四个方向发射光，填满（格子置为 2）。注意遇到黑格时被挡住，而上面的宏定义黑格为 3，空格 0，✕1，灯光 2，都比 3 小，因此将这个作为判断条件。

当然的，上面提到需要放置✕，因此也就需要对应的函数 put_redx。如果需要放置的位置在棋盘内，那么就做以下工作：如果放置的地方已经放了灯泡，说明解错误，返回 0；如果放置的地方是空，则将位置设为 2.返回 1。如果要放置的位置不在棋盘内，也返回 1.

put_lightbulb 和 put_redx 两个函数的流程图如图 1-2 put_lightbulb 流程图图 1-3 put_redx 流程图。

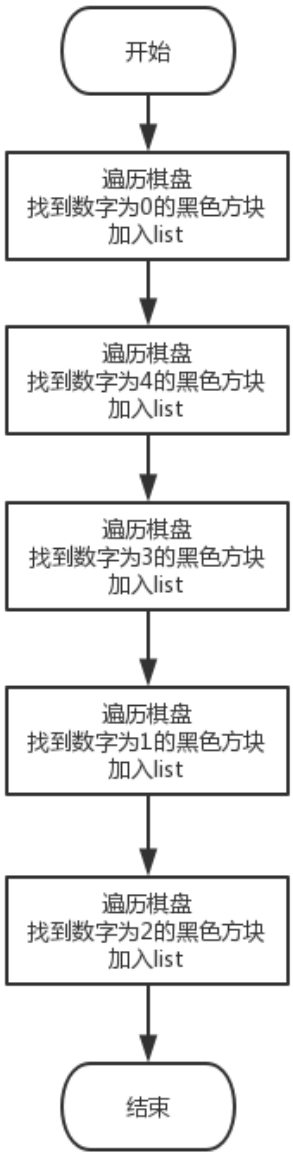

图 1-1 创建黑色方格列表流程图

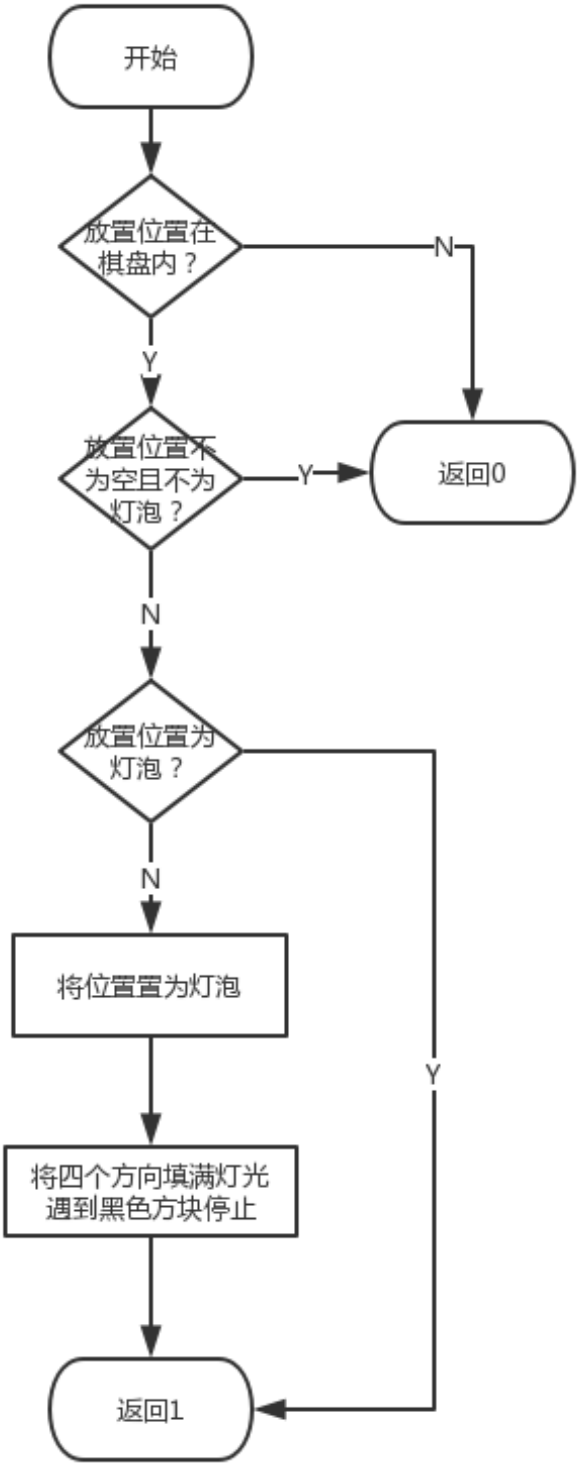

图 1-2 put_lightbulb 流程图

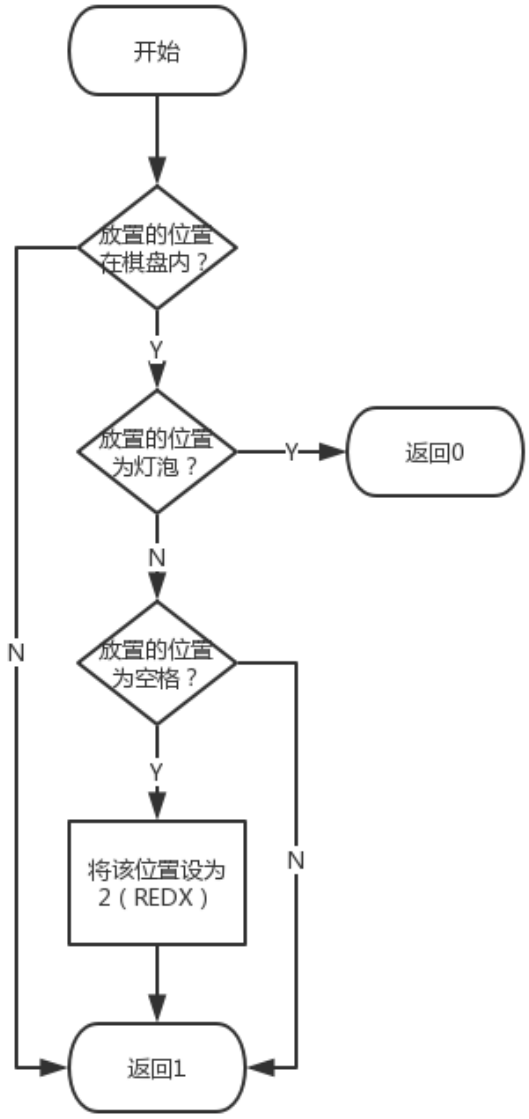

图 1-3 put_redx 流程图

上面说到黑色方块为 4 的情况，下面来说一下为 3，2，1，0 的情况。类似的，如果为 3，就要挑一个位置放✕，另外 3 个位置放灯泡。因此就有四种情况。值得注意的是，每种情况之前都需要恢复棋盘为上一步棋盘，直到所有情况试完。类似，如果为 2，就需要考虑 C_4^2=6 种情况。为 1 有 4 种情况，为 0 只有一种。如果黑色方块全部处理完后，就需要考虑填剩下的空格了。

solve_akari_puzzle 流程图如图 1-4。

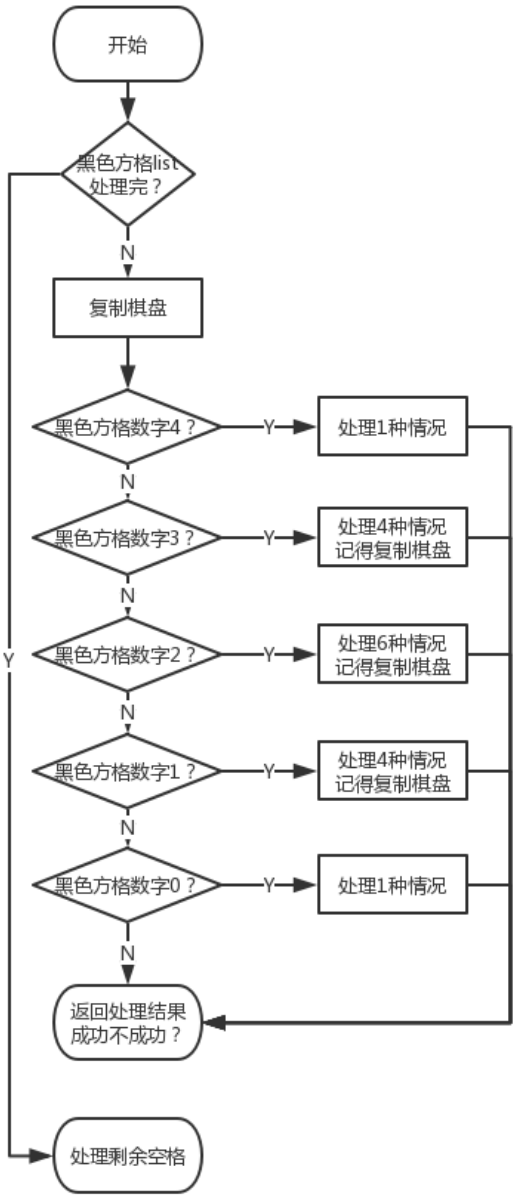

图 1-4 solve_akari_puzzle 流程图

到这里主要算法部分已经讲述完了，接下来就是处理剩余空格部分的算法了。

首先检查是否已经达成了最终解，方法就是看棋盘内有没有小于灯光的值（最终解的棋盘最小值一定是灯光），如果有说明没有达到一个解，继续执行后面的代码。否则直接返回 1。找到第一个空格，放置灯泡，然后继续调用这个函数，输入参数为下一个空格的位置。如果解错误，恢复棋盘，继续找下一个空位。

这样直到最后，如果找到了一个解，就会返回 1，输出解。如果输出为 0，说

明遍历了所有的情况还没有找到解，这种情况就是该 puzzle 无解，返回对应的信息就好。

1.4 实验方案

1.4.1 开发与运行环境

本实验在测试平台上测试，也在我的本机上通过了测试。这里给出本机的环境。

系统版本：ubuntu1804.2

编译环境：g++ 7.4.0

CPU：Intel® Core™ i5-7200U CPU @ 2.50GHz 2.70GHz

RAM: 8GB

1.4.2 实验过程

根据算法描述编写程序，先通过本机测试，观察输出结果，如果大致正确就放到平台测试。注意观察时间。

如果平台测试不通过，需要考虑输出是否符合要求。

1.5 实验结果与分析

最后在平台上通过了所有的测试集。结果如图：

图 1-5 串行 akari 测试结果图

可以看到代码是正确的。

对于平台的测试时间，经过多次测试，发现平台的每次测试时间的差距过大，因此这里只能说作为一个参考，真正的分析要到后面的思考题中去解答，在这里就只是贴上了通关截图。

2 实验二 使用并行回溯法解决 Akari 问题

2.1 实验目的与要求

- 掌握串行程序并行化的方法
- 掌握在 linux 环境下使用 C++进行并行程序设计
- 了解并行程序设计与并行算法设计的基本分析方法与途径
- 掌握 pthread 并行化方法
- 调试并行程序并分析结果

2.2 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。问题分解就是采用分治的思想将问题分为几个子问题，解除数据相关就是要使用一定的策略来使私有数据不互相影响（如增加冗余或锁）。

在之前串行 akari 的实验中，已经很明确地描述了问题，并且也详细描述了如何使用回溯法来解决这个问题。在这次实验，主要是讨论如何并行的问题，由于大体框架不变，对于其他实现细节不再赘述。

在回溯法中，回溯路线的选择对正确解的产生没有影响，因此一层节点的计算是可以并行的，并且不同层的节点也可并行。同时每个解空间的节点数据不会影响，因此无数据相关。

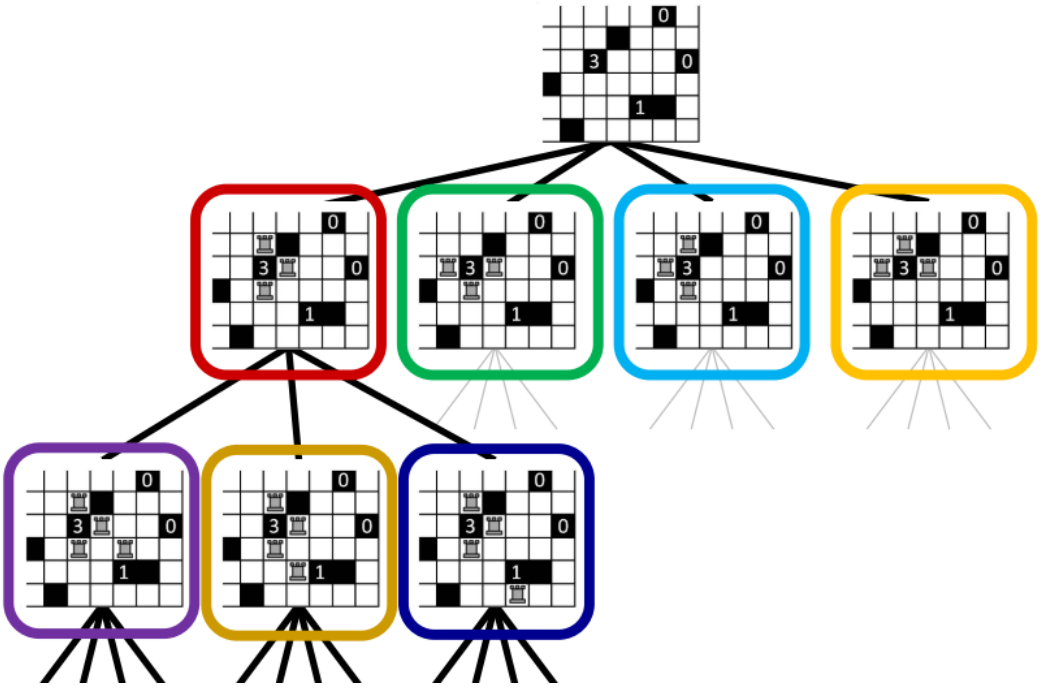

图 2-1 akari 问题分解

如图，不同颜色的矩形可以并行。

任意两个节点可以并行，同样地，在这里是初次尝试，因此选择了比较容易实现的对每个节点都新建一个 thread 处理。

在这里，由于是基于串行的并行，因此大部分函数都不需要改变，比如 put_lightbulb，put_redx， create_blackblock_list 等等。在这一部分就重点说一下并行的部分。

首先我们调用 create_blackblock_list 获得黑色方格 list，然后进入 solve_akari_puzzle 函数，在这里我们需要首先判断一下是不是已经找到了一个可行解，在这里用全局变量 handle 判断，如果是，则直接结束查找过程。如果继续的话，会产生很多不必要的开销，这样的话会降低并行的效率。

然后就开始对 list 中的每一个黑色方块按照顺序进行处理，这里和串行不同的是，这里对每一个放置情况都用一个 thread 进行处理，这样就完成了并行的操作。之后需要等待所有线程处理完成后（pthread_join），释放线程。说到了线程，就需要一个线程处理函数。

线程处理函数完成的具体操作就是根据不同情况来放置灯泡或者✕。这里就需要传递放置方向信息了，在这里用了四个宏定义，分别为0001，0010，0100，

1000.代表四个不同的方向，之后在线程处理函数中，对每一个方向进行判断，如果在某个方向上 bit 位为 1，那么就放置灯泡（调用 put_lightbulb），否则就放置✕.如果放置失败，直接结束这种情况。如果放置成功，则需要进一步考虑。如果在成功的情况下已经找到了一个可行解，那么就直接结束.否则创建一个线程,将当前情况作为参数，solve_akari_puzzle 作为线程处理函数。这样就进入了下一层。

由于 solve_akari_puzzle 函数流程也基本一样，所以在这里只给出线程处理函数 mupltiplesolve 流程图。

如图 2-2 多线程处理函数流程图。

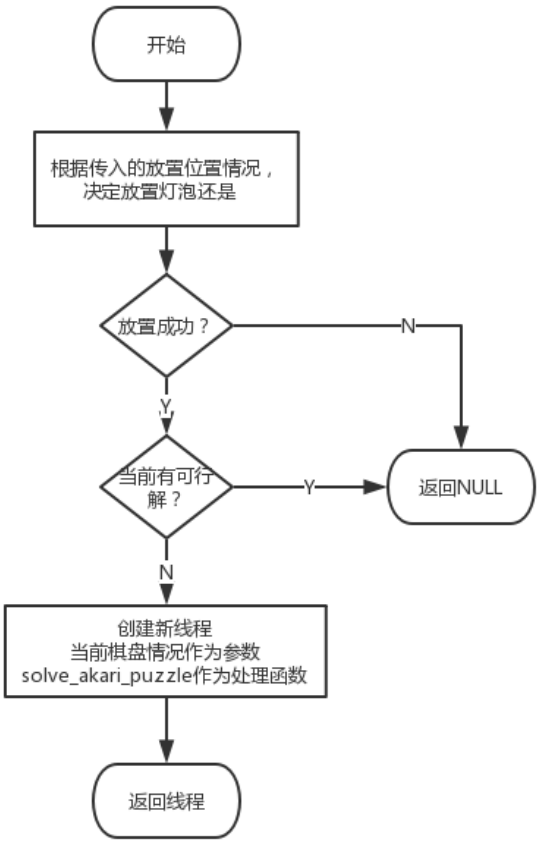

图 2-2 多线程处理函数流程图

当处理完了之后黑格之后，也需要处理剩余的空格。在这里也可以让填充剩

余的空格进行并行。

首先需要检查棋盘是否被填满了（也就是达到一个可行解），这里需要对线程进行上锁，否则可能会改变棋盘的状态。

如果没有检测到有可行解（handle 为 0），则将 handle 置为 1.之后释放锁。在之后如果检测到 handle 为 1，则直接结束。否则就需要创建线程来在空位放置灯泡，之后就将新的需要的参数作为线程参数，fill_empty 为线程处理函数。这样就达到了并行的效果。

如图 2-3 填充空格并行流程图。

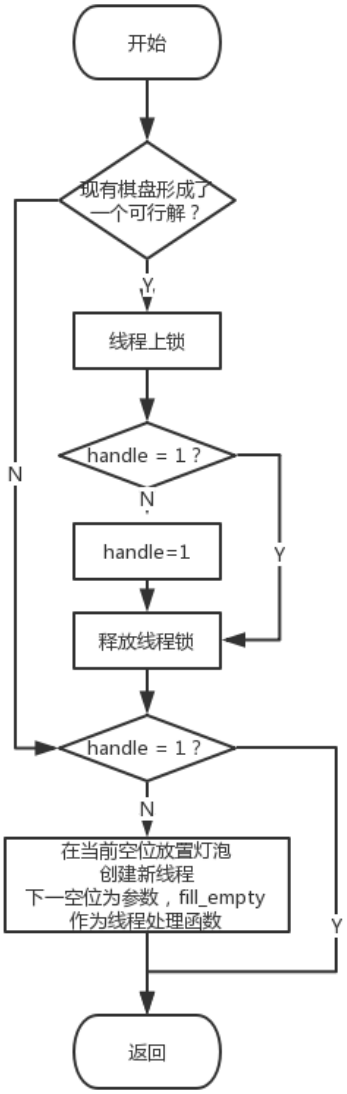

图 2-3 填充空格并行流程图

2.3 实验方案

2.3.1 开发与运行环境

本实验在测试平台上测试，也在我的本机上通过了测试。这里给出本机的环境。

系统版本：ubuntu1804.2

编译环境：g++ 7.4.0

CPU：Intel® Core™ i5-7200U CPU @ 2.50GHz 2.70GHz

RAM: 8GB

2.3.2 实验过程

根据算法描述编写程序，由于本次与使用 pthread 时代码变化不大，因此直接在原有代码上进行修改即可。gcc 编译时也不需要加上-pthread。

测试时先通过本机测试，观察输出结果，如果大致正确就放到平台测试。注意观察时间。

如果平台测试不通过，需要考虑输出是否符合要求。

2.4 实验结果与分析

最后在平台上通过了所有的测试集。结果如图：

图 2-4 并行 akari 测试结果图

可以看到代码是正确的。

同时，与实验一类似，由于平台测试时间具有不确定性，并且与平台在线人数有关系，不能正确反映程序真正的执行时间，因此在本实验报告里只证明程序的正确性，加速比分析之类的在思考题报告中反映。

同样地，由于存在将输入矩阵进行转换的过程，也存在后面为了满足输出要求进行矩阵转换的过程，因此这里的时间不仅仅包含解决 akari 问题的时间。

在这里并行粒度是全粒度，加速度比的测量将在思考题中呈现。

3 实验三 改进并行回溯法解决 Akari 问题

3.1 实验目的与要求

- 掌握优化并行程序的方法
- 了解并行粒度与性能之间的关系，掌握通过并行粒度分析程序性能的方法
- 了解并行程序设计与并行算法设计的基本分析方法与途径

3.2 算法描述

多线程应用的并行任务工作量的大小会对并行的性能有很大影响，这个任务量就称为粒度。

逻辑上，并行任务量越多，那么 CPU 的空闲时间就会较少，效率就会增加。但是分割任务，创建线程，将任务分配给线程，以及线程之间的数据通信这些开销又会降低效率。为了避免开销较大，使得整体效率降低，可以确定合适的粒度来保证最佳效率。尽管大量的小任务看起来是可以极大提高效率，但是线程之间的通信等开销会使得它的效率不如适量的中型任务。

因此，这里有以下几种方案：

1. 每个解空间的节点分裂均产生新任务

这个方案就是为每一个孩子节点创建线程进行处理。如图 3-1

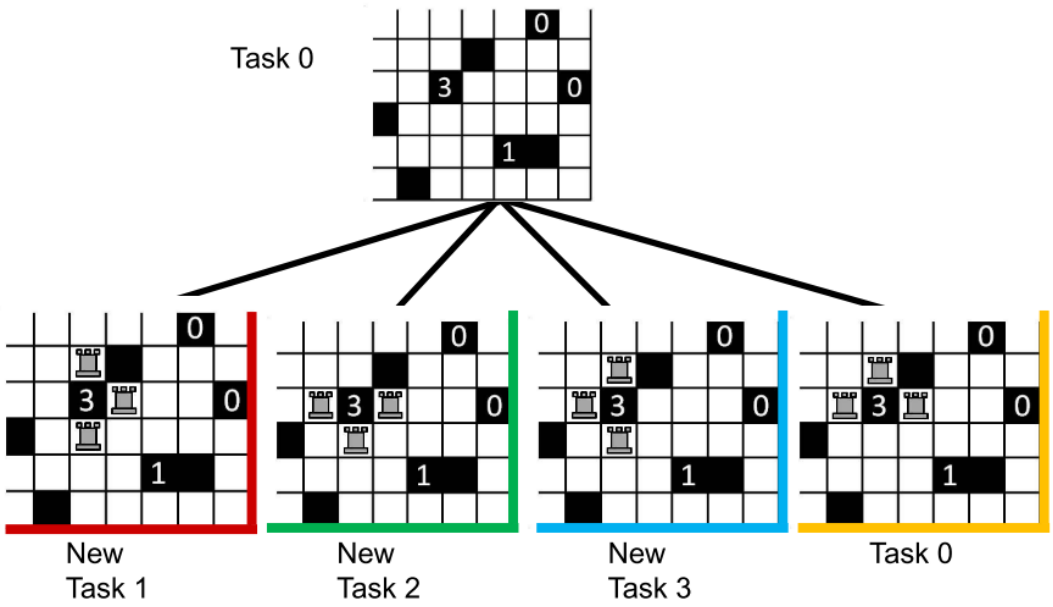

图 3-1 方案 1

这样做优点很明显，就是设计很简单，并且容易实现。每个 task 几乎都是一样的工作量，十分平衡。

但同样的，缺点也十分明显。第一个就是太多的 task 被创建了，这样使得创建开销巨大，降低效率。第二个就是每个 task 分配的任务量太少了，导致没有发挥足够的效率，不足以抵消创建的开销。第三个就是，每次创建新线程都得将当前的棋盘复制一遍，这样及其消耗内存。

2. 当且仅当解空间树节点第一次分裂的时候产生新任务

这种方案仅在第一次分裂时产生新任务，如图 3-2。

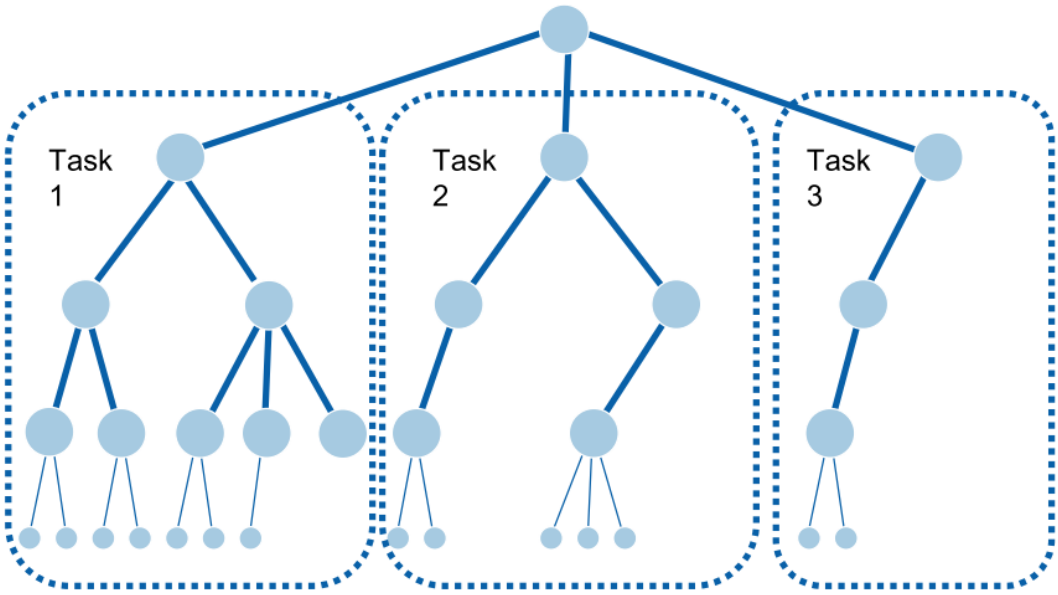

图 3-2 方案 2

这样做的优点就是创建 task 的时间开销小了，没有过大的开销限制效率。并且内存使用也比较小。

但是，这样很明显地，没有充分利用并行的优点。第一子树的变化太大，每个 task 的执行时间可能大不相同。也就是说每个 task 的负载太不相同了，严重降低了执行效率。第二它的扩展性较差，只在第一次分裂产生新任务，对于不同的情况，表现波动大，不是一种合适的方案。

3. 当且仅当解空间树节点在数字“3”的时候分裂时产生新任务

这种方案其实就是上面两种方案的折中。

如图 3-3：

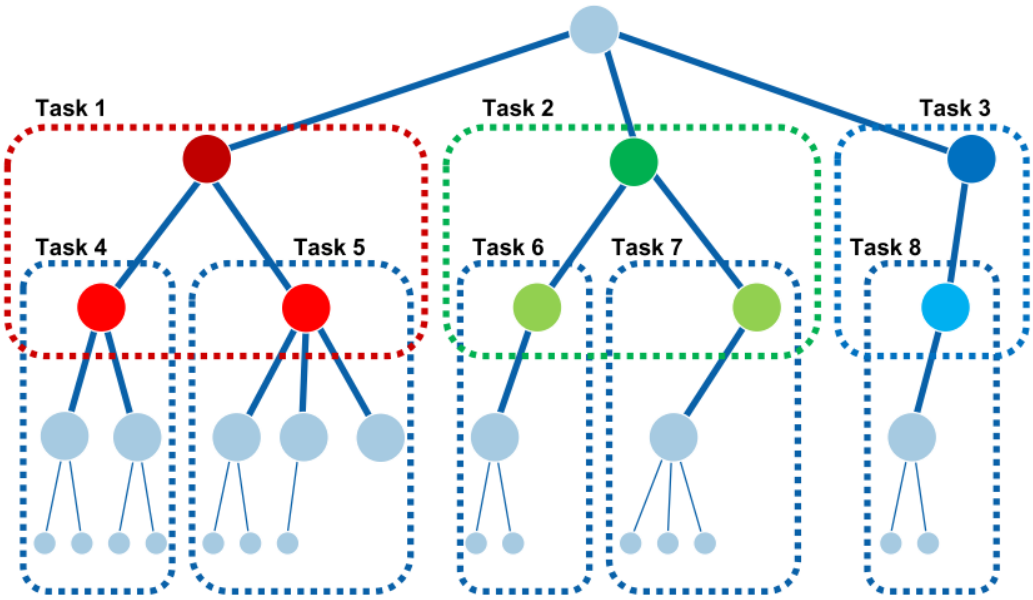

图 3-3 方案 3

这样做集合了 1 和 2 的优点，第一它产生任务的时间缩短了，并且每个 task 的负载也比 2 平衡的多。

但是缺点是实现比较困难，并且这种方案有较大的随机性，过于依赖树的特性。但是如果能确定在合适的时候分裂任务的话，应该能获得最大的效率。

但是如果仅当数字为 3 时才产生新任务的话，这样所实现的效果还不如每次都分裂产生新任务来的好。

最终经过粗略推导，方案 2 的效率肯定是最低的。那么剩下的就在方案 1 和方案 3 中间抉择了。对于不同的测试集，两种方案的效率可能会略有差别，但是总体上，应该是方案 1 是这三种中最好的。

3.3 实验方案

3.3.1 开发与运行环境

本实验在测试平台上测试，也在我的本机上通过了测试。这里给出本机的环境。

系统版本：ubuntu1804.2

编译环境：g++ 7.4.0

CPU：Intel® Core™ i5-7200U CPU @ 2.50GHz 2.70GHz

RAM: 8GB

3.3.2 实验过程

根据算法描述编写程序，先通过本机测试，观察输出结果，如果大致正确就放到平台测试。注意观察时间。

如果平台测试不通过，需要考虑输出是否符合要求。

3.4 实验结果与分析

最后在平台上通过了所有的测试集。结果如图：

图 3-4 改进并行回溯法测试结果图

可以看到代码是正确的。

同理，由于平台测试时间具有不确定性，并且与平台在线人数有关系，不能正确反映程序真正的执行时间，因此在本实验报告里只证明程序的正确性，加速比分析之类的在思考题报告中反映。

附录

1. 串行

```
# 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>

#define EMPTY 0
#define REDX 1
#define YELLOWLIGHT 2
#define BLACKBLOCK 3
#define BLACKBLOCK0 4
#define BLACKBLOCK1 5
#define BLACKBLOCK2 6
#define BLACKBLOCK3 7
#define BLACKBLOCK4 8
#define LIGHTBULB 9

typedef struct Node{
    int number;
    size_t row;
    size_t col;
    struct Node* next;
```

```
}Blackblocknode, *BlackblockList;

int m;//row
int n;//col

int* global_checkerboard;//global chess board

BlackblockList create_blackblock_list(int*);

void cp_board(int*, const int*);

int final_check(int*);

int solve_akari_puzzle(BlackblockList, int*);
int put_lightbulb(int, int, int*);
int put_redx(int, int, int*);

int fill_empty(int, int*);
int next_empty(int, int*);

vector<vector<int> > solveAkari(vector<vector<int> > & g)
{
    // 请在此函数内返回最后求得的结果
    //get n & m
    int len = 0;
    m = (int)g.size();
    n = (int)g[0].size();
    //cp g ans
    vector<vector<int> > ans(m);
    for(int i=0; i<m; ++i){
```

```
57.         ans[i].resize(n);
58.     }
59.     global_checkerboard = (int*) malloc(sizeof(int) * m * n);
60.     //change default array
61.     for(int r = 0; r < m; r++) {
62.         for(int c = 0; c < n; c++){
63.             ans[r][c] = g[r][c];
64.             switch(g[r][c]){
65.             case -2:
66.                 global_checkerboard[r*n + c] = 0;
67.                 break;
68.             case -1:
69.                 global_checkerboard[r*n + c] = 3;
70.                 break;
71.             case 0:
72.                 global_checkerboard[r*n + c] = 4;
73.                 break;
74.             case 1:
75.                 global_checkerboard[r*n + c] = 5;
76.                 break;
77.             case 2:
78.                 global_checkerboard[r*n + c] = 6;
79.                 break;
80.             case 3:
81.                 global_checkerboard[r*n + c] = 7;
82.                 break;
83.             case 4:
84.                 global_checkerboard[r*n + c] = 8;
85.                 break;
86.             default:
```

```
87.                 global_checkerboard[r*n + c] = 0;
88.                 break;
89.             }
90.         }
91.     }
92.
93.     //create list of blackblock 0,1,2,3,4
94.     BlackblockList blackblocks = create_blackblock_list(global_checkerboard)
    ;
95.     int solved = solve_akari_puzzle(blackblocks, global_checkerboard);
96.     if(solved) {
97.         //solved
98.         for(int r = 0; r < m; r++) {
99.             for(int c = 0; c < n; c++){
100.                 if(global_checkerboard[r*n + c] == 9){
101.                     ans[r][c] = 5;//replace LIGHTBULB
102.                 }
103.             }
104.         }
105.     }else{
106.         //no solution
107.     }
108.
109.     free(global_checkerboard);
110.
111.     //free blackblocks
112.     BlackblockList p;
113.     for(p = blackblocks; p ;) {
114.         blackblocks = blackblocks->next;
115.         free(p);
```

```
116.         p = blackblocks;
117.     }
118.
119.     //return vector<vector<int> >;
120.     return ans;
121. }
122.
123. BlackblockList create_blackblock_list(int* board) {
124.     int i;
125.     BlackblockList header, p;
126.     header = p = NULL;
127.     //first add block0 because it only has one situation
128.     for(i = 0; i < m * n; ++i){
129.         if(board[i] == BLACKBLOCK0){
130.             BlackblockList q = (BlackblockList)malloc(sizeof(Blackblocknode
    ));
131.             q->row = i / n;
132.             q->col = i % n;
133.             q->number = 0;//q's label is 0
134.             (header == NULL) ? (header = p = q) : (p->next = q, p = p->next
    );
135.         }
136.     }
137.
138.     //second add block4 because it only has one situation
139.     for(i = 0; i < m * n; ++i){
140.         if(board[i] == BLACKBLOCK4){
141.             BlackblockList q = (BlackblockList)malloc(sizeof(Blackblocknode
    ));
142.             q->row = i / n;
```

```
143.             q->col = i % n;
144.             q->number = 4;
145.             (header == NULL) ? (header = p = q) : (p->next = q, p = p->next
    );
146.         }
147.     }
148.
149.     //third add block3 because it has four situation & has more lightbulbs
150.     for(i = 0; i < m * n; ++i){
151.         if(board[i] == BLACKBLOCK3){
152.             BlackblockList q = (BlackblockList)malloc(sizeof(Blackblocknode
    ));
153.             q->row = i / n;
154.             q->col = i % n;
155.             q->number = 3;
156.             (header == NULL) ? (header = p = q) : (p->next = q, p = p->next
    );
157.         }
158.     }
159.
160.     //fourth add block1 because it has four situation
161.     for(i = 0; i < m * n; ++i){
162.         if(board[i] == BLACKBLOCK1){
163.             BlackblockList q = (BlackblockList)malloc(sizeof(Blackblocknode
    ));
164.             q->row = i / n;
165.             q->col = i % n;
166.             q->number = 1;
```

```
167.             (header == NULL) ? (header = p = q) : (p->next = q, p=p->next);
168.         }
169.     }
170.
171.     //fifth add block2 because it has six situation
172.     for(i = 0; i < m * n; ++i){
173.         if(board[i] == BLACKBLOCK2){
174.             BlackblockList q = (BlackblockList)malloc(sizeof(Blackblocknode
    ));
175.             q->row = i / n;
176.             q->col = i % n;
177.             q->number = 2;
178.             (header == NULL) ? (header = p = q) : (p->next = q, p=p->next);
179.         }
180.     }
181.
182.     if(p != NULL) p->next = NULL;
183.     return header;
184.
185. }
186.
187.
188. int solve_akari_puzzle(Blackblocknode* blackblocknow, int* board) {
189.     //solve akari
190.     if(NULL != blackblocknow) {
191.         //first put lightbulbs aroud blackblocks
192.         int r, c;
193.         int handle = 0;
```

```
194.         Blackblocknode* p;
195.         int* copied_board;
196.         copied_board = (int*)malloc(sizeof(int) * m * n);
197.         cp_board(copied_board, board);
198.
199.         p = blackblocknow;
200.         r = p->row;
201.         c = p->col;
202.         switch(p->number){
203.         case 4:
204.             if(put_lightbulb(r-1, c, copied_board)&&put_lightbulb(r, c-
    1, copied_board)
205.                 &&put_lightbulb(r+1, c, copied_board)&&put_lightbulb(r, c+1
    , copied_board))
206.                 handle = solve_akari_puzzle(p->next, copied_board);
207.             break;
208.         case 3:
209.             if(put_lightbulb(r-1, c, copied_board)&&put_lightbulb(r, c-
    1,copied_board)
210.                 &&put_lightbulb(r+1, c, copied_board)&&put_redx(r, c+1,copi
    ed_board))
211.                 handle = solve_akari_puzzle(p->next, copied_board);
212.             //restore the state of board
213.             cp_board(copied_board, board);
214.             if(!handle && put_redx(r-
    1, c, copied_board)&&put_lightbulb(r, c-1,copied_board)
215.                 &&put_lightbulb(r+1, c, copied_board)&&put_lightbulb(r, c+1
    ,copied_board))
216.                 handle = solve_akari_puzzle(p->next, copied_board);
217.             //
```

```
218.             cp_board(copied_board, board);
219.             if(!handle && put_lightbulb(r-
    1, c, copied_board)&&put_redx(r, c-1,copied_board)
220.                 &&put_lightbulb(r+1, c, copied_board)&&put_lightbulb(r, c+1
    ,copied_board))
221.                 handle = solve_akari_puzzle(p->next, copied_board);
222.             //
223.             cp_board(copied_board, board);
224.             if(!handle && put_lightbulb(r-
    1, c, copied_board)&&put_lightbulb(r, c-1,copied_board)
225.                 &&put_redx(r+1, c, copied_board)&&put_lightbulb(r, c+1,copi
    ed_board))
226.                 handle = solve_akari_puzzle(p->next,copied_board);
227.             break;
228.         case 2:
229.             if(put_lightbulb(r-1, c, copied_board)&&put_lightbulb(r, c-
    1,copied_board)
230.                 &&put_redx(r+1, c, copied_board)&&put_redx(r, c+1,copied_bo
    ard))
231.                 handle = solve_akari_puzzle(p->next, copied_board);
232.             cp_board(copied_board, board);
233.             if(!handle && put_redx(r-
    1, c, copied_board)&&put_lightbulb(r, c-1,copied_board)
234.                 &&put_redx(r+1, c, copied_board)&&put_lightbulb(r, c+1,copi
    ed_board))
235.                 handle = solve_akari_puzzle(p->next, copied_board);
236.             cp_board(copied_board, board);
237.             if(!handle && put_lightbulb(r-
    1, c, copied_board)&&put_redx(r, c-1,copied_board)
```

```
238.                 &&put_redx(r+1, c, copied_board)&&put_lightbulb(r, c+1,copi
    ed_board))
239.                 handle = solve_akari_puzzle(p->next, copied_board);
240.             cp_board(copied_board,board);
241.             if(!handle && put_redx(r-1, c, copied_board)&&put_redx(r, c-
    1,copied_board)
242.                 &&put_lightbulb(r+1, c, copied_board)&&put_lightbulb(r, c+1
    ,copied_board))
243.                 handle = solve_akari_puzzle(p->next, copied_board);
244.             cp_board(copied_board, board);
245.             if(!handle && put_redx(r-
    1, c, copied_board)&&put_lightbulb(r, c-1,copied_board)
246.                 &&put_lightbulb(r+1, c, copied_board)&&put_redx(r, c+1,copi
    ed_board))
247.                 handle = solve_akari_puzzle(p->next, copied_board);
248.             cp_board(copied_board, board);
249.             if(!handle && put_lightbulb(r-
    1, c, copied_board)&&put_redx(r, c-1,copied_board)
250.                 &&put_lightbulb(r+1, c, copied_board)&&put_redx(r, c+1,copi
    ed_board))
251.                 handle = solve_akari_puzzle(p->next, copied_board);
252.             break;
253.         case 1 :
254.             //printf("r : %d\n", r);
255.             //printf("1\n");
256.             if(put_lightbulb(r-1, c, copied_board)&&put_redx(r, c-
    1,copied_board)
257.                 &&put_redx(r+1, c, copied_board)&&put_redx(r, c+1,copied_bo
    ard))
258.                 handle = solve_akari_puzzle(p->next, copied_board);
```

```
259.             cp_board(copied_board, board);
260.             //printf("2\n");
261.             if(!handle && put_redx(r-
    1, c, copied_board)&&put_lightbulb(r, c-1,copied_board)
262.                 &&put_redx(r+1, c, copied_board)&&put_redx(r, c+1,copied_bo
    ard))
263.                 handle = solve_akari_puzzle(p->next, copied_board);
264.             cp_board(copied_board, board);
265.             //printf("3\n");
266.             if(!handle && put_redx(r-1, c, copied_board)&&put_redx(r, c-
    1,copied_board)
267.                 &&put_lightbulb(r+1, c, copied_board)&&put_redx(r, c+1,copi
    ed_board))
268.                 handle = solve_akari_puzzle(p->next, copied_board);
269.             cp_board(copied_board, board);
270.             //printf("4\n");
271.             if(!handle && put_redx(r-1, c, copied_board)&&put_redx(r, c-
    1,copied_board)
272.                 &&put_redx(r+1, c, copied_board)&&put_lightbulb(r, c+1,copi
    ed_board))
273.                 handle = solve_akari_puzzle(p->next, copied_board);
274.             break;
275.         case 0:
276.             if(put_redx(r-1, c,copied_board)&& put_redx(r, c-
    1, copied_board)
277.                 && put_redx(r+1, c, copied_board)&& put_redx(r, c+1, copied
    _board))
278.                 handle = solve_akari_puzzle(p->next, copied_board);
279.             break;
280.         default:
```

```
281.             break;
282.         }
283.
284.         free(copied_board);
285.         copied_board = NULL;
286.         return handle;
287.     } else {
288.         return fill_empty(next_empty(-1, board), board);
289.     }
290. }
291.
292. int fill_empty(int cur_pos, int* board) {
293.     //handle left blank spaces
294.     int* copied_board;
295.     int handle = 0;
296.     if(final_check(board)){
297.         cp_board(global_checkerboard, board);
298.         return 1;
299.     }
300.     if(cur_pos == -1)
301.         return 0;
302.     copied_board = (int*)malloc(sizeof(int) * m * n);
303.     cp_board(copied_board, board);
304.     put_lightbulb(cur_pos / n, cur_pos % n, copied_board);
305.     handle = fill_empty(next_empty(cur_pos, copied_board), copied_board);
306.     if(!handle){
307.         //retore
308.         cp_board(copied_board, board);
309.         handle =  fill_empty(next_empty(cur_pos, copied_board), copied_boar
    d);
```

```
310.     }
311.     free(copied_board);
312.     return handle;
313. }
314.
315.
316. int final_check(int* board){
317.     int i;
318.     for(i = 0; i < m*n; i++)
319.         if(board[i] < YELLOWLIGHT)
320.             return 0;
321.     return 1;
322. }
323.
324. int put_lightbulb(int row, int col, int* board) {
325.
326.     int r, c;
327.     //starts with zero
328.     if(row < 0 || row >= m || col < 0 || col >= n)
329.         return 0;
330.     //judge
331.     if(board[row * n + col] != EMPTY && board[row*n+col] != LIGHTBULB)
332.         return 0;
333.     else if(board[row*n+col] == LIGHTBULB)
334.         return 1;
335.     board[row * n + col] = LIGHTBULB;
336.     //fill light
337.     for(r = row -1 , c = col; r >= 0 && board[r*n + c] < BLACKBLOCK; r--)
338.         board[r*n + c] = YELLOWLIGHT;
339.     for(r = row + 1, c = col; r < m && board[r*n + c] < BLACKBLOCK; r++)
```

```
340.         board[r*n + c] = YELLOWLIGHT;
341.     for(r = row, c = col - 1; c >= 0 && board[r*n + c] < BLACKBLOCK; c--)
342.         board[r*n + c] = YELLOWLIGHT;
343.     for(r = row, c = col +1; c < n && board[r*n + c] < BLACKBLOCK; c++)
344.         board[r*n + c] = YELLOWLIGHT;
345.     return 1;
346. 
347. }
348. 
349. int put_redx(int row, int col, int* board) {
350.     if((row >= 0 && row < m) && (col >= 0 && col < n)){
351.         if(board[row*n + col] == LIGHTBULB)
352.             return 0;
353.         if(board[row*n + col] == EMPTY) {
354.             board[row*n + col] = REDX;
355.         }
356.     }
357.     return 1;
358. }
359. 
360. 
361. void cp_board(int* dst, const int* src) {
362.     int i;
363.     for(i = 0; i < m*n; i++)
364.         dst[i] = src[i];
365. }
366. 
367. 
368. int next_empty(int cur_pos, int* board){
369.     int i;
```

```
370.     for(i = cur_pos+1; i < m *n; i++)
371.         if(board[i] == EMPTY)
372.             return i;
373.     return -1;
374. }
375.
376. }
```

2. pthread

```
1.  # include <bits/stdc++.h>
2.  #include<pthread.h>
3.  # include "akari.h"
4.  using namespace std;
5.
6.  namespace aka{
7.  //请在命名空间内编写代码，否则后果自负
8.  #include<stdio.h>
9.  #include<stdlib.h>
10. #include<assert.h>
11.
12. #define EMPTY 0
13. #define REDX 1
14. #define YELLOWLIGHT 2
15. #define BLACKBLOCK 3
16. #define BLACKBLOCK0 4
17. #define BLACKBLOCK1 5
18. #define BLACKBLOCK2 6
```

```
19. #define BLACKBLOCK3 7
20. #define BLACKBLOCK4 8
21. #define LIGHTBULB 9
22. 
23. 
24. #define D_UP  (1<<0)
25. #define D_RIGHT   (1<<1)
26. #define D_DOWN  (1<<2)
27. #define D_LEFT   (1<<3)
28. 
29. typedef struct Node{
30.     int number;
31.     size_t row;
32.     size_t col;
33.     struct Node* next;
34. }Blackblocknode, *BlackblockList;
35. 
36. int m;//row
37. int n;//col
38. 
39. int* global_checkerboard;//global chess board
40. 
41. BlackblockList create_blackblock_list(int*);
42. 
43. void cp_board(int*, const int*);
44. 
45. int final_check(int*);
46. 
47. void* solve_akari_puzzle(void*);
48. int put_lightbulb(int, int, int*);
```

```
49. int put_redx(int, int, int*);
50. 
51. void* fill_empty(void*);
52. int next_empty(int, int*);
53. 
54. pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
55. int handle = 0;
56. 
57. typedef struct {
58.     int* boardp;
59.     BlackblockList blackblock;
60. } blackblockParam;
61. 
62. typedef struct {
63.     int* boardp;
64.     int pos;
65. } fillemptyParam;
66. 
67. BlackblockList create_blackblock_list(int* board) {
68.     //create_blackblock_list
69.     int i;
70.     BlackblockList headp, p;
71.     headp = p = NULL;
72.     for(i = 0; i < m*n; i++){
73.         if(board[i] == BLACKBLOCK0){
74.             BlackblockList q = (BlackblockList)malloc(sizeof(Blackblocknode)
    );
75.             q->row = i / n;
76.             q->col = i % n;
77.             q->number = 0;
```

```
78.             //(headp == NULL) ? (headp = p = q) : (p->next = q, p=p->next);
79.             if(headp == NULL){
80.                 headp = p = q;
81.             }else{
82.                 p->next = q;
83.                 p = p->next;
84.             }
85.         }
86.     }
87.
88.     for(i = 0; i < m*n; i++){
89.         if(board[i] == BLACKBLOCK4){
90.             BlackblockList q = (BlackblockList)malloc(sizeof(Blackblocknode)
    );
91.             q->row = i / n;
92.             q->col = i % n;
93.             q->number = 4;
94.             // (headp == NULL) ? (headp = p = q) : (p->next = q, p=p->next);
95.             if(headp == NULL){
96.                 headp = p = q;
97.             }else{
98.                 p->next = q;
99.                 p = p->next;
100.              }
101.         }
102.     }
103.     for(i = 0; i < m*n; i++){
104.         if(board[i] == BLACKBLOCK3){
```

```
105.             BlackblockList q = (BlackblockList)malloc(sizeof(Blackblocknode
    ));
106.             q->row = i / n;
107.             q->col = i % n;
108.             q->number = 3;
109.             // (headp == NULL) ? (headp = p = q) : (p->next = q, p=p->next)
    ;
110.             if(headp == NULL){
111.                 headp = p = q;
112.             }else{
113.                 p->next = q;
114.                 p = p->next;
115.             }
116.         }
117.     }
118.     for(i = 0; i < m*n; i++){
119.         if(board[i] == BLACKBLOCK1){
120.             BlackblockList q = (BlackblockList)malloc(sizeof(Blackblocknode
    ));
121.             q->row = i / n;
122.             q->col = i % n;
123.             q->number = 1;
124.             // (headp == NULL) ? (headp = p = q) : (p->next = q, p=p->next)
    ;
125.             if(headp == NULL){
126.                 headp = p = q;
127.             }else{
128.                 p->next = q;
129.                 p = p->next;
130.             }
```

```
131.         }
132.     }
133.     for(i = 0; i < m*n; i++){
134.         if(board[i] == BLACKBLOCK2){
135.             BlackblockList q = (BlackblockList)malloc(sizeof(Blackblocknode
    ));
136.             q->row = i / n;
137.             q->col = i % n;
138.             q->number = 2;
139.             // (headp == NULL) ? (headp = p = q) : (p->next = q, p=p->next)
    ;
140.             if(headp == NULL){
141.                 headp = p = q;
142.             }else{
143.                 p->next = q;
144.                 p = p->next;
145.             }
146.         }
147.     }
148.     if(p != NULL) p->next = NULL;
149.     return headp;
150. }
151.
152.
153. pthread_t* mupltiplesolve(int flag, int* board, BlackblockList blackblock)
    {
154.     int* copied_board = (int*) malloc(sizeof(int) * m*n);
155.     cp_board(copied_board,board);
156.     int i, success = 0;
157.     int r = blackblock->row;
```

```
158.     int c = blackblock->col;
159.     for(i = 0; i < 4; i++) {
160.         int direction  = (1<<i);
161.         int type = (flag & direction);
162.         switch(direction) {
163.         case D_UP:
164.             success = (type != 0) ? put_lightbulb(r-
    1, c, copied_board):put_redx(r-1, c, copied_board);
165.             break;
166.         case D_RIGHT:
167.             success = (type != 0) ? put_lightbulb(r, c-
    1, copied_board):put_redx(r, c-1, copied_board);
168.             break;
169.         case D_DOWN:
170.             success = (type != 0) ? put_lightbulb(r+1, c, copied_board):put
    _redx(r+1, c, copied_board);
171.             break;
172.         case D_LEFT:
173.             success = (type != 0) ? put_lightbulb(r, c+1, copied_board):put
    _redx(r, c+1, copied_board);
174.             break;
175.         }
176.         if(!success)
177.             break;
178.     }
179.
180.     if(!success) {
181.         free(copied_board);
182.         return NULL;
183.     }
```

```
184.     else {
185.         //if success
186.         if(handle) {
187.             //find a solution
188.             free(copied_board);
189.             return NULL;
190.         }
191.         blackblockParam* bbpara = (blackblockParam*) malloc(sizeof(blackblo
    ckParam));
192.         pthread_t* nthread = (pthread_t*) malloc(sizeof(pthread_t));
193.         bbpara->blackblock = blackblock->next;
194.         bbpara->boardp = copied_board;
195.         pthread_create(nthread, NULL, solve_akari_puzzle, (void*)bbpara);
196.         return nthread;
197.     }
198. }
199.
200. void* solve_akari_puzzle(void* bbpara) {
201.     blackblockParam* params = (blackblockParam*)bbpara;
202.     BlackblockList blackblock = params->blackblock;
203.     int* board = params->boardp;
204.
205.     if(handle) {
206.         //find a solution
207.         free(board);
208.         free(params);
209.         return NULL;
210.     }
211.
212.     if(NULL != blackblock) {
```

```
213.         //first put lightbulbs aroud blackblocks
214.         int i;
215.         pthread_t** threads_for_akari;
216.         threads_for_akari = (pthread_t**) calloc(6, sizeof(pthread_t*));
217.         switch(blackblock->number){
218.         case 4:
219.             threads_for_akari[0] = mupltiplesolve(D_UP|D_RIGHT|D_DOWN|D_LEF
    T, board, blackblock);
220.             break;
221.         case 3:
222.             threads_for_akari[0] = mupltiplesolve(D_UP|D_DOWN|D_RIGHT, boar
    d, blackblock);
223.             threads_for_akari[1] = mupltiplesolve(D_DOWN|D_LEFT|D_RIGHT, bo
    ard, blackblock);
224.             threads_for_akari[2] = mupltiplesolve(D_LEFT|D_UP|D_RIGHT, boar
    d, blackblock);
225.             threads_for_akari[3] = mupltiplesolve(D_LEFT|D_UP|D_DOWN, board
    , blackblock);
226.             break;
227.         case 2:
228.             threads_for_akari[0] = mupltiplesolve(D_LEFT|D_UP, board, black
    block);
229.             threads_for_akari[1] = mupltiplesolve(D_LEFT|D_DOWN, board, bla
    ckblock);
230.             threads_for_akari[2] = mupltiplesolve(D_LEFT|D_RIGHT, board, bl
    ackblock);
231.             threads_for_akari[3] = mupltiplesolve(D_UP|D_DOWN, board, black
    block);
232.             threads_for_akari[4] = mupltiplesolve(D_UP|D_RIGHT, board, blac
    kblock);
```

```
233.             threads_for_akari[5] = mupltiplesolve(D_DOWN|D_RIGHT, board, bl
    ackblock);
234.             break;
235.         case 1:
236.             threads_for_akari[0] = mupltiplesolve(D_LEFT, board, blackblock
    );
237.             threads_for_akari[1] = mupltiplesolve(D_UP, board, blackblock);

238.             threads_for_akari[2] = mupltiplesolve(D_DOWN, board, blackblock
    );
239.             threads_for_akari[3] = mupltiplesolve(D_RIGHT, board, blackbloc
    k);
240.             break;
241.         case 0:
242.             threads_for_akari[0] = mupltiplesolve(0, board, blackblock);
243.             break;
244.         default:
245.             break;
246.         }
247.         for(i = 0; i < 6;i++)
248.             if(threads_for_akari[i] != NULL) {
249.                 pthread_join(*threads_for_akari[i], NULL);
250.
251.             }
252.
253.         for(i = 0; i < 6;i++)
254.             if(threads_for_akari[i] != NULL)
255.                 free(threads_for_akari[i]);
256.
257.         free(threads_for_akari);
```

```
258.         free(board);
259.         free(params);
260.         return NULL;
261.     } else {
262.         //fill left empty space
263.         fillemptyParam* fillemptypara = (fillemptyParam*) malloc(sizeof(fil
    lemptyParam));
264.         fillemptypara->pos = next_empty(-1, board);
265.         fillemptypara->boardp = board;
266.         fill_empty((void*) fillemptypara);
267.         return NULL;
268.     }
269. }
270.
271. void *fill_empty(void* params) {
272.
273.     fillemptyParam* fillemptypara = (fillemptyParam*) params;
274.     fillemptyParam* fillemptypara2;
275.     int* board, *copied_board;
276.     pthread_t* thread_handle;
277.     int empty_pos;
278.     board = (int*)fillemptypara->boardp;
279.     empty_pos = fillemptypara->pos;
280.     if(final_check(board)){
281.         pthread_mutex_lock(&lock);
282.         if(!handle) {
283.             cp_board(global_checkerboard, board);
284.             handle= 1;
285.         }
286.         pthread_mutex_unlock(&lock);
```

```
287.         free(fillemptypara->boardp);
288.         free(fillemptypara);
289.         return NULL;
290.     }
291.     if(empty_pos == -1) {
292.         free(fillemptypara->boardp);
293.         free(fillemptypara);
294.         return NULL;
295.     }
296.
297.     if(handle) {
298.         //find a solution
299.         free(fillemptypara->boardp);
300.         free(fillemptypara);
301.         return NULL;
302.     }
303.
304.     thread_handle = (pthread_t*) malloc(sizeof(pthread_t));
305.     copied_board = (int*)malloc(sizeof(int) * m * n);
306.     cp_board(copied_board, board);
307.     put_lightbulb(empty_pos / n, empty_pos % n, copied_board);
308.     fillemptypara2 = (fillemptyParam*) malloc(sizeof(fillemptyParam));
309.     fillemptypara2->pos = next_empty(empty_pos, copied_board);
310.     fillemptypara2->boardp = copied_board;
311.     //create new thread
312.     pthread_create(thread_handle, NULL, fill_empty, (void *) fillemptypara2
    );
313.     //fill next empty pos
314.     fillemptypara->pos = next_empty(empty_pos, board);
315.     fill_empty((void*) fillemptypara);
```

```
316.     pthread_join(*thread_handle, NULL);
317.     free(thread_handle);
318.     return NULL;
319. }
320.
321.
322. int final_check(int* board){
323.     int i;
324.     for(i = 0; i < m*n; i++)
325.         if(board[i] < YELLOWLIGHT)
326.             return 0;
327.     return 1;
328. }
329.
330. int put_lightbulb(int row, int col, int* board) {
331.
332.     int r, c;
333.     //starts with zero
334.     if(row < 0 || row >= m || col < 0 || col >= n)
335.         return 0;
336.     //judge
337.     if(board[row*n + col] != EMPTY && board[row*n+col] != LIGHTBULB)
338.         return 0;
339.     else if(board[row*n+col] == LIGHTBULB)
340.         return 1;
341.     board[row*n + col] = LIGHTBULB;
342.     //fill light
343.     for(r = row -1 , c = col; r >= 0 && board[r*n + c] < BLACKBLOCK; r--)
344.         board[r*n + c] = YELLOWLIGHT;
345.     for(r = row + 1, c = col; r < m && board[r*n + c] < BLACKBLOCK; r++)
```

```
346.         board[r*n + c] = YELLOWLIGHT;
347.     for(r = row, c = col - 1; c >= 0 && board[r*n + c] < BLACKBLOCK; c--)
348.         board[r*n + c] = YELLOWLIGHT;
349.     for(r = row, c = col +1; c < n && board[r*n + c] < BLACKBLOCK; c++)
350.         board[r*n + c] = YELLOWLIGHT;
351.     return 1;
352. 
353. }
354. 
355. int put_redx(int row, int col, int* board) {
356.     if((row >= 0 && row < m) && (col >= 0 && col < n)){
357.         if(board[row*n + col] == LIGHTBULB)
358.             return 0;
359.         if(board[row*n + col] == EMPTY) {
360.             board[row*n + col] = REDX;
361.         }
362.     }
363.     return 1;
364. }
365. 
366. 
367. void cp_board(int* dst, const int* src) {
368.     int i;
369.     for(i = 0; i < m*n; i++)
370.         dst[i] = src[i];
371. }
372. 
373. 
374. int next_empty(int cur_pos, int* board){
375.     int i;
```

```
376.     for(i = cur_pos+1; i < m *n; i++)
377.         if(board[i] == EMPTY)
378.             return i;
379.     return -1;
380. }
381.
382. vector<vector<int> > solveAkari(vector<vector<int> > & g)
383. {
384.     // 请在此函数内返回最后求得的结果
385.     int len = 0;
386.     m = (int)g.size();
387.     n = (int)g[0].size();
388.     //cp g ans
389.     vector<vector<int> > ans(m);
390.     for(int i=0; i<m; ++i){
391.         ans[i].resize(n);
392.     }
393.     global_checkerboard = (int*) malloc(sizeof(int) * m * n);
394.     //change default array
395.     for(int r = 0; r < m; r++) {
396.         for(int c = 0; c < n; c++){
397.             ans[r][c] = g[r][c];
398.             switch(g[r][c]){
399.             case -2:
400.                 global_checkerboard[r*n + c] = 0;
401.                 break;
402.             case -1:
403.                 global_checkerboard[r*n + c] = 3;
404.                 break;
405.             case 0:
```

```
406.                 global_checkerboard[r*n + c] = 4;
407.                 break;
408.             case 1:
409.                 global_checkerboard[r*n + c] = 5;
410.                 break;
411.             case 2:
412.                 global_checkerboard[r*n + c] = 6;
413.                 break;
414.             case 3:
415.                 global_checkerboard[r*n + c] = 7;
416.                 break;
417.             case 4:
418.                 global_checkerboard[r*n + c] = 8;
419.                 break;
420.             default:
421.                 global_checkerboard[r*n + c] = 0;
422.                 break;
423.             }
424.         }
425.     }
426.     BlackblockList barriers = create_blackblock_list(global_checkerboard);
427.
428.     blackblockParam* params = (blackblockParam*) malloc(sizeof(blackblockPa
    ram));
429.     int* copied_board =(int*) malloc(sizeof(int)*m*n);
430.     cp_board(copied_board, global_checkerboard);
431.     params->boardp = copied_board;
432.     params->blackblock = barriers;
433.
```

```
434.     solve_akari_puzzle((void*) params);
435. 
436.     if(handle) {
437.         for(int r = 0; r < m; r++) {
438.             for(int c = 0; c < n; c++){
439.                 if(global_checkerboard[r*n + c] == 9){
440.                     ans[r][c] = 5;//replace LIGHTBULB
441.                 }
442.             }
443.         }
444.     }else
445.         printf("No solution!\n");
446. 
447.     free(global_checkerboard);
448. 
449.     BlackblockList p;
450.     for(p = barriers; p;) {
451.         barriers = barriers->next;
452.         free(p);
453.         p = barriers;
454.     }
455. 
456.     return ans;
457. 
458. }
459. 
460. }
```