华中科技大学

课 程 实 验 报 告

课程名称： 并行编程原理与实践

专业班级： 计算机科学与技术（校际交流）1601 班

学　　号： U201610136

姓　　名： 朱晓光

指导教师： 陆枫

报告日期： 2019 年 7 月 12 日

计算机科学与技术学院

目　录

2 实验二

2.1 实验目的与要求

本实验中将使用回溯法来对 Akari 问题进行求解，并充分利用计算机并行技术的优势对算法进行改进，从而更迅速地求解问题。

2.2 实验内容

本次实验一共包含以下 3 项实验内容：

1. 使用回溯法解决 Akari 问题；
2. 使用并行回溯法求解 Akari 问题；
3. 使用改进地并行回溯法求解 Akari 问题。

2.3 算法描述

这里将解决 Akari 问题分为一共 2 步进行：

1. 尝试列举出所有带有数字的黑色块附近的灯泡放置方案；
2. 尝试在剩下的未放置灯泡的块中寻找可行的灯泡放置方案。

根据 Akari 游戏规则，可以发现两条隐含的规则，可以帮助我们减少该算法的搜索空间：

1. 标有数字 0 的黑色块的上下左右均不可放置灯泡；
2. 灯泡向上下左右方向射出的光线必定不能遇到其他灯泡，否则该灯泡的放置不合法。

最终采用的搜索算法整体思路如下：

1. 将所有标有数字 0 的黑色块的上下左右块标志为“不可放置且未照亮”；
2. 对所有标有数字 1~4 的黑色块，列举其上下左右块所有可能的灯泡放置方案；
3. 对上一个步骤中所有剩下标志为“未照亮”的块，枚举所有可能的放置方案，返回一个满足游戏规则的解。

该解法的关键步骤为判断是否可以在某位置放置灯泡，下面给出如图 2.1 所示的流程图。

其中在检查同行、同列是否已有灯泡的同时，对每一个遍历检查的块设置了标志“已照亮”，这样在后续放置灯泡的过程中，如果同行、同列已经放置了灯泡，则马上就可以判断此处不能放置灯泡，而不必再遍历到放置灯泡的块，节省了大量时间。同时，在地图上做出此标记也方便了整张地图是否已经完全照亮的判断，仅需检查地图上所有块是否全部为“已照亮”、黑色块、灯泡（或不为“未照亮”、“不可放置且未照亮”）即可。

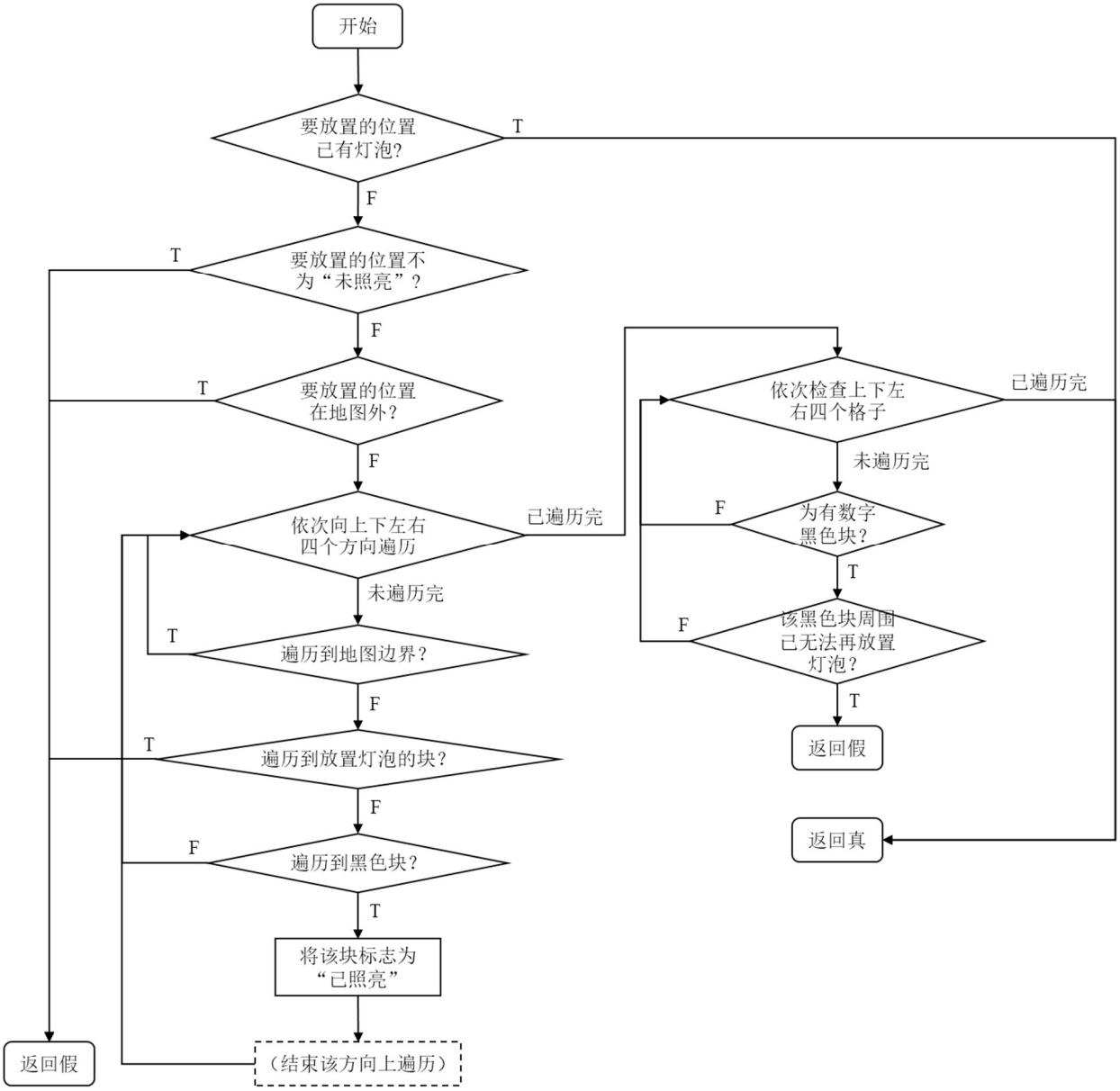

图 2.1 灯泡放置合法性检查过程流程图

在将以上回溯法进行并行化时，选择在遍历标有数字的黑色块周围灯泡放置方案时进行并行化。这里尝试过将遍历空白块也进行并行化，但由于对每个空白块至多只有两个分支，并行化后将导致平台上最后一个测试样例超时，故最终未进行并行化。

改进并行回溯法时，对并行的地方进行了粒度控制，仅当剩余搜索深度大于 7 且当前分支数大于 2 时运行并行算法，否则仍运行串行算法。

2.4 实验结果与分析

在 EduCoder 平台上完成了以上所有实验内容，测试样例均通过。

其中，串行回溯法通过时平台显示耗时 8.276 秒，并行回溯法耗时 9.273 秒，改进的并行回溯法耗时 8.186 秒。

由以上结果可知，对并行算法进行粒度控制是极其重要的优化。

附录

makefile - 编译、测试脚本

可使用“make single”、“make multi”分别运行串行算法与并行算法。

```
.PHONY: main single multi clean

CXX = g++
CXXFLAGS = --std=c++11 -pthread #-DDEBUG_AKARI_CPP

main: multi

main.o: main.cpp

akari.o: akari.cpp makefile
akari: main.o akari.o
    $(CXX) $(CPPFLAGS) $(CXXFLAGS) $^ -o $@
single: akari
    @echo "============= RUN ============="
    @/usr/bin/time -v ./$<

akari-multithreaded.o: akari-multithreaded.cpp makefile
akari-multithreaded: main.o akari-multithreaded.o
    $(CXX) $(CPPFLAGS) $(CXXFLAGS) $^ -o $@
multi: akari-multithreaded
    @echo "============= RUN ============="
    @/usr/bin/time -v ./$<

clean:
    $(RM) ./*.o ./akari ./akari-multithreaded
```

main.cpp - 程序入口

可通过修改 main()函数中 input_map 的定义来选择测试样例。

```
#include <iostream>
#include <vector>
#include "akari.h"
using namespace std;


typedef vector<vector<int> > GameMap;

int main(const int argc, const char **argv)
{
    // GameMap input_map{
```

```
//     {-2, -2, -1,  1, -2, -2, -2},
//     {-2, -2, -2, -2, -2, -2, -2},
//     {-2, -2, -2, -2, -2, -2,  1},
//     { 0, -2, -2, -2, -2, -2,  1},
//     { 2, -2, -2, -2, -2, -2, -2},
//     {-2, -2, -2, -2, -2, -2, -2},
//     {-2, -2, -2,  1, -1, -2, -2}
// };

// GameMap input_map{
//     {-2, -2, -2, -2, -1, -2 ,-2},
//     {-2,  2, -2, -2, -2,  4 ,-2},
//     {-1, -2, -2, -1, -2, -2 ,-2},
//     {-2, -2,  2, -1,  1, -2 ,-2},
//     {-2, -2, -2, -1, -2, -2 , 1},
//     {-2,  2, -2, -2, -2, -1 ,-2},
//     {-2, -2,  2, -2, -2, -2 ,-2}
// };

// GameMap input_map{
//     {-2,  1, -2, -2, -2, -2, -2},
//     {-2, -2,  3, -2, -2, -2,  0},
//     {-2, -2, -2, -2, -2,  1, -2},
//     {-2, -2, -2, -1, -2, -2, -2},
//     {-2,  1, -2, -2, -2, -2, -2},
//     { 0, -2, -2, -2,  2, -2, -2},
//     {-2, -2, -2, -2, -2,  0, -2},
// };

// // 7x7 hard
// GameMap input_map{
//     {-2, -2, -1,  2, -2, -1, -2},
//     { 2, -2, -2, -2, -2, -2, -2},
//     {-2, -2, -2, -2, -2, -2, -1},
//     { 1, -2, -2, -2, -2, -2,  0},
//     { 0, -2, -2, -2, -2, -2, -2},
//     {-2, -2, -2, -2, -2, -2, -1},
//     {-2, -1, -2,  0,  0, -2, -2}
// };

// // 10x10 hard
// GameMap input_map{
//     {-2,  2, -2, -2, -2, -2, -2, -2, -2, -2},
//     {-2, -2, -2, -2,  1, -2,  2, -2, -2, -1},
```

```
//     {-2, -2, -2, -1,  0, -2, -2, -2, -2, -2},
//     {-2, -1, -2,  1, -2, -2,  2,  1, -2, -2},
//     {-2, -2, -2, -2, -2, -2, -2,  1,  2, -2},
//     {-2, -1, -1, -2, -2, -2, -2, -2, -2, -2},
//     {-2, -2,  1, -1, -2, -2,  1, -2,  3, -2},
//     {-2, -2, -2, -2, -2,  1, -1, -2, -2, -2},
//     { 1, -2, -2,  1, -2,  0, -2, -2, -2, -2},
//     {-2, -2, -2, -2, -2, -2, -2, -2,  1, -2}
// };

// // 14x14 easy
// GameMap input_map{
//     {-2, -2, -2, -2, -2, -2, -2, -2,  1, -2, -1,  1, -2, -2},
//     {-2, -2, -2, -2, -2,  3, -1, -2, -2, -2,  1, -2, -2, -2},
//     {-1, -2, -2, -1, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2},
//     {-1,  3, -2, -1, -2, -2, -2,  2, -2, -2, -1,  0, -2, -2},
//     {-2, -2, -2, -2, -1, -2, -2, -1, -2,  1, -2, -2, -2, -2},
//     {-1, -2, -2, -2, -2, -2, -2, -1, -2, -2, -2, -2, -1, -2},
//     {-2, -2, -2, -1,  2, -1, -2, -2, -2, -2, -2, -2,  1, -2},
//     {-2,  1, -2, -2, -2, -2, -2, -2,  0,  0, -1, -2, -2, -2},
//     {-2, -1, -2, -2, -2, -2, -1, -2, -2, -2, -2, -2, -2, -1},
//     {-2, -2, -2, -2, -1, -2,  0, -2, -2, -1, -2, -2, -2, -2},
//     {-2, -2, -1, -1, -2, -2,  1, -2, -2, -2,  2, -2, -1, -1},
//     {-2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -1, -2, -2, -1},
//     {-2, -2, -2,  0, -2, -2, -2, -1,  2, -2, -2, -2, -2, -2},
//     {-2, -2,  1, -1, -2,  0, -2, -2, -2, -2, -2, -2, -2, -2}
// };

// 14x14 hard
GameMap input_map{
    {-2, -2, -1, -2, -1, -1, -2,  1, -2, -2,  2, -2,  1, -2},
    { 2, -2, -2, -2, -2,  0, -2, -2, -2, -2, -2,  2, -2, -2},
    {-2,  2, -2, -2, -2, -2, -2,  0, -2, -1, -2, -2, -2, -1},
    {-1, -2, -2, -2,  2, -2, -1, -2, -2, -2, -2, -2, -2, -2},
    {-2, -2, -1, -2, -2, -2, -2, -2, -2, -2,  1, -2, -2, -1},
    {-2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2,  0, -1},
    { 2, -2,  1, -2, -2, -2, -2, -2, -2, -2,  0, -2, -2, -2},
    {-2, -2, -2, -1, -2, -2, -2, -2, -2, -2, -2,  0, -2,  2},
    {-1, -1, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2, -2},
    {-1, -2, -2, -1, -2, -2, -2, -2, -2, -2, -2, -1, -2, -2},
    {-2, -2, -2, -2, -2, -2, -2,  2, -2,  0, -2, -2, -2, -1},
    {-1, -2, -2, -2, -1, -2, -1, -2, -2, -2, -2, -2,  1, -2},
    {-2, -2,  2, -2, -2, -2, -2, -2,  0, -2, -2, -2, -2,  2},
    {-2,  1, -2,  1, -2, -2, -1, -2,  1, -1, -2, -1, -2, -2}
```

```
    };

    // const int repeat = 25;
    // const int progwidth = 40;
    // std::cout << "Solving Akari x " << repeat << " ..." <<
std::endl;
    // for (int i = 0; i != repeat; ++i) {
    //     std::cout << "[";
    //     for (int j = 0; j != progwidth; ++j) {
    //         if (j < i * progwidth / repeat) { std::cout << "="; }
    //         else if (j == i * progwidth / repeat) { std::cout <<
">"; }
    //         else { std::cout << " "; }
    //     }
    //     std::printf("] %.2f%%\r", (double)i / (double)repeat *
100.0);
    //     std::cout.flush();
    //     aka::solveAkari(input_map);
    // }
    // for (int i = -10; i != progwidth; ++i) {
    //     std::printf(" ");
    // }
    // std::cout << "\rDone" << std::endl;

    std::cout << "Solving Akari..." << std::endl;
    aka::printMap(aka::solveAkari(input_map));

    return 0;
}
```

akari.h

```
#if !defined(LINKED_LIST_H_LIELJE7398CNHD_INCLUDE_)
#define LINKED_LIST_H_LIELJE7398CNHD_INCLUDE_
// # include <bits/stdc++.h>
#include <vector>
using namespace std;
namespace aka{
vector<vector<int> > solveAkari(vector<vector<int> > & g);
void printMap(const vector<vector<int> > &g);
}

#endif
```

akari.cpp - 串行算法实现

```cpp
// #include <bits/stdc++.h>
#include <iostream>
#include <vector>
#include <tuple>
#include <forward_list>
#include <algorithm>
#include "akari.h"
using namespace std;

// #define DEBUG_AKARI_CPP

namespace aka{
//请在命名空间内编写代码，否则后果自负

typedef vector<vector<int> > GameMap;
typedef tuple<int, int> Coord;

enum CellType
{
    mark_noplaceable    = -4,
    mark_lit           = -3,
    white              = -2,
    black_nonumber      = -1,
    light_bulb         = 5
};

inline
const Coord getMapShape(const GameMap &map)
{
    return std::make_tuple(map.size(), map.at(0).size());
}

void printMap(const GameMap &map)
{
    cout << "+";
    for (auto &_: map.at(0)) {
        cout << "--+";
    }
    cout << endl;
    for (auto &row : map) {
        cout << "|";
        for (auto &cell : row) {
            switch (cell) {
                case CellType::mark_noplaceable: cout << "X"; break;
```

```
                case CellType::mark_lit: cout << "L"; break;
                case CellType::white: cout << "  "; break;
                case CellType::black_nonumber: cout << "#"; break;
                case 0: cout << "0"; break;
                case 1: cout << "1"; break;
                case 2: cout << "2"; break;
                case 3: cout << "3"; break;
                case 4: cout << "4"; break;
                case CellType::light_bulb: cout << "?"; break;
                default: cout << " " << cell; break;
            }
            cout << "|";
        }
        cout << "\n+";
        for (auto &_: row) {
            cout << "--+";
        }
        cout << endl;
    }
}

static inline
const vector<Coord> _getCrossAround(const int x, const int y)
{
    return vector<Coord>{
        std::make_tuple(x - 1, y), std::make_tuple(x + 1, y),
        std::make_tuple(x, y - 1), std::make_tuple(x, y + 1)
    };
}

static
void _prune_black_zero(GameMap &map)
{
    int h, w; std::tie(h, w) = getMapShape(map);
    for (int x = 0; x != h - 1; ++x) {
        for (int y = 0; y != w - 1; ++y) {
            if (map.at(x).at(y + 1) == 0 || map.at(x + 1).at(y) == 0) {
                if (map.at(x).at(y) == CellType::white) {
                    map.at(x).at(y) = CellType::mark_noplaceable;
                }
            }
            if (map.at(x).at(y) == 0) {
                if (map.at(x).at(y + 1) == CellType::white) {
                    map.at(x).at(y + 1) = CellType::mark_noplaceable;
```

```
            }
            if (map.at(x + 1).at(y) == CellType::white) {
                map.at(x + 1).at(y) = CellType::mark_noplaceable;
            }
        }
    }
}

static
bool _checkPlaceable(GameMap &map, const int x, const int y, bool
leave_markup=false)
{
    // Check if cell is placeable
    if (map.at(x).at(y) == CellType::light_bulb) { return true; }
    if (map.at(x).at(y) != CellType::white) { return false; }
    int h, w; std::tie(h, w) = getMapShape(map);
    // Check row for existing light bulb
    for (int iy = y - 1; iy != -1; --iy) {
        auto &cell = map.at(x).at(iy);
        if (cell == CellType::light_bulb) { return false; }
        if (cell >= -1 && cell <= 4) { break; }
        if (leave_markup) { cell = CellType::mark_lit; }
    }
    for (int iy = y + 1; iy != w; ++iy) {
        auto &cell = map.at(x).at(iy);
        if (cell == CellType::light_bulb) { return false; }
        if (cell >= -1 && cell <= 4) { break; }
        if (leave_markup) { cell = CellType::mark_lit; }
    }
    for (int ix = x - 1; ix != -1; --ix) {
        auto &cell = map.at(ix).at(y);
        if (cell == CellType::light_bulb) { return false; }
        if (cell >= -1 && cell <= 4) { break; }
        if (leave_markup) { cell = CellType::mark_lit; }
    }
    for (int ix = x + 1; ix != h; ++ix) {
        auto &cell = map.at(ix).at(y);
        if (cell == CellType::light_bulb) { return false; }
        if (cell >= -1 && cell <= 4) { break; }
        if (leave_markup) { cell = CellType::mark_lit; }
    }
    // Check numbered black cell constraint
    for (const auto &ixy : _getCrossAround(x, y)) {
```

```
        int ix, iy; std::tie(ix, iy) = ixy;
        // Continue if out of map
        if (ix < 0 || ix >= h || iy < 0 || iy >= w) { continue; }
        int cell = map.at(ix).at(iy);
        // Continue if not a numbered black cell
        if (cell < 0 || cell > 4) { continue; }
        if (cell == 0) { return false; }
        int remain_slot_cnt = cell;
        if (ix - 1 >= 0 && map.at(ix - 1).at(iy) ==
CellType::light_bulb) { remain_slot_cnt--; }
        if (ix + 1 <  h && map.at(ix + 1).at(iy) ==
CellType::light_bulb) { remain_slot_cnt--; }
        if (iy - 1 >= 0 && map.at(ix).at(iy - 1) ==
CellType::light_bulb) { remain_slot_cnt--; }
        if (iy + 1 <  w && map.at(ix).at(iy + 1) ==
CellType::light_bulb) { remain_slot_cnt--; }
        if (remain_slot_cnt < 1) {
            return false;
        }
    }
    return true;
}

static
GameMap * _placeLightBulb(const GameMap &map, const int x, const int
y,
    bool place_markup=false)
{
    int h, w; std::tie(h, w) = getMapShape(map);
    if (x < 0 || x >= h || y < 0 || y >= w) { return nullptr; }
    GameMap *retmap = new GameMap(map);     // NOTE: Deep copy.
    // Reject if is a illegal move
    if (!_checkPlaceable(*retmap, x, y, place_markup)) { delete
retmap; return nullptr; }
    retmap->at(x).at(y) = CellType::light_bulb;
    return retmap;
}

static
GameMap * _placeMultipleLightBulb(const GameMap &map,
forward_list<Coord> coords)
{
    GameMap *oldmap = new GameMap(map);
    GameMap *newmap = nullptr;
```

```
    int h, w; std::tie(h, w) = getMapShape(map);
    for (const auto &coord : coords) {
        int x, y; std::tie(x, y) = coord;
        if (x < 0 || x >= h || y < 0 || y >= w) { delete oldmap;
return nullptr; }
        newmap = _placeLightBulb(*oldmap, x, y, true);
        delete oldmap;
        if (newmap == nullptr) { return nullptr; }
        oldmap = newmap;
    }
    return newmap;
}

static
forward_list<GameMap *> _solveNumberedCell(const GameMap &map,
    const forward_list<Coord> &numbered_cells)
{
    int h, w; std::tie(h, w) = getMapShape(map);
    // Return case
    if (numbered_cells.empty()) { return forward_list<GameMap *>{new
GameMap(map)}; }
    const auto current_cell = numbered_cells.front();
    int x, y; std::tie(x, y) = current_cell;
    forward_list<Coord> next_numbered_cells(numbered_cells);
    next_numbered_cells.pop_front();
    // Generate branches, place light bulbs
    const forward_list<forward_list<Coord> > *coords_list = nullptr;
    const forward_list<forward_list<Coord> > _case_1_coords_list{
        {std::make_tuple(x - 1, y)},
        {std::make_tuple(x + 1, y)},
        {std::make_tuple(x, y - 1)},
        {std::make_tuple(x, y + 1)}
    };
    const forward_list<forward_list<Coord> > _case_2_coords_list{
        // Corners x 4
        {std::make_tuple(x - 1, y), std::make_tuple(x, y - 1)},
        {std::make_tuple(x - 1, y), std::make_tuple(x, y + 1)},
        {std::make_tuple(x + 1, y), std::make_tuple(x, y - 1)},
        {std::make_tuple(x + 1, y), std::make_tuple(x, y + 1)},
        // Across x 2
        {std::make_tuple(x - 1, y), std::make_tuple(x + 1, y)},
        {std::make_tuple(x, y - 1), std::make_tuple(x, y + 1)},
    };
    const forward_list<forward_list<Coord> > _case_3_coords_list{
```

```
        {std::make_tuple(x - 1, y), std::make_tuple(x, y - 1),
std::make_tuple(x, y + 1)},
        {std::make_tuple(x, y + 1), std::make_tuple(x - 1, y),
std::make_tuple(x + 1, y)},
        {std::make_tuple(x + 1, y), std::make_tuple(x, y - 1),
std::make_tuple(x, y + 1)},
        {std::make_tuple(x, y - 1), std::make_tuple(x - 1, y),
std::make_tuple(x + 1, y)},
    };
    const forward_list<forward_list<Coord> > _case_4_coords_list{{
        std::make_tuple(x - 1, y), std::make_tuple(x + 1, y),
        std::make_tuple(x, y - 1), std::make_tuple(x, y + 1)
    }};
    switch (map.at(x).at(y)) {
        case 1: { coords_list = &_case_1_coords_list; break; }
        case 2: { coords_list = &_case_2_coords_list; break; }
        case 3: { coords_list = &_case_3_coords_list; break; }
        case 4: { coords_list = &_case_4_coords_list; break; }
        default: throw;
    }
    forward_list<GameMap *> sub_branches;
    GameMap *retmap = nullptr;
    for (auto &coords : *coords_list) {
        if ((retmap = _placeMultipleLightBulb(map, coords)) !=
nullptr) {
            sub_branches.push_front(retmap);
        }
    }
    // Recurse into branches
    forward_list<GameMap *> ret_branches;
    for (auto &branch : sub_branches) {
        ret_branches.splice_after(ret_branches.cbefore_begin(),
                _solveNumberedCell(*branch, next_numbered_cells));
        delete branch;
    }
    return ret_branches;
}

static
bool _isSolution_WithMarkup(const GameMap &map)
{
    for (auto &row : map) {
        for (auto &cell : row) {
            if (cell == CellType::white || cell ==
```

```
CellType::mark_noplaceable) {
                return false;
            }
        }
    }
    return true;
}

static
forward_list<GameMap *> _solveWhiteCell(const GameMap &map, const
forward_list<Coord> &white_cells)
{
    // NOTE: Using brute force enumeration algorithm.
    // TODO: Come back with a better algorithm.
    // Success case
    if (_isSolution_WithMarkup(map)) { return forward_list<GameMap
*>{new GameMap(map)}; }
    // Fail case
    if (white_cells.empty()) { return forward_list<GameMap *>(); }
    // Recrusion
    const auto current_cell = white_cells.front();
    int x, y; std::tie(x, y) = current_cell;
    forward_list<Coord> next_white_cells(white_cells);
    next_white_cells.pop_front();
    // Branch - not place
    auto retlist = _solveWhiteCell(map, next_white_cells);
    if (!retlist.empty()) { return retlist; }
    // Branch - place
    GameMap *placed_map = _placeLightBulb(map, x, y, true);
    if (placed_map != nullptr) {
        retlist.splice_after(retlist.cbefore_begin(),
                _solveWhiteCell(*placed_map, next_white_cells));
        delete placed_map;
    }
    return retlist;
}

static
GameMap _solveAkari(const GameMap &g)
{
#if defined(DEBUG_AKARI_CPP)
    std::cout << "Input map..." << std::endl;
    printMap(g);
#endif
```

```
    GameMap *map = new GameMap(g);
    int h, w; std::tie(h, w) = getMapShape(*map);
    // Prune
    _prune_black_zero(*map);
#if defined(DEBUG_AKARI_CPP)
    std::cout << "After pruning..." << std::endl;
    printMap(*map);
#endif
    // Get numbered cells
    forward_list<Coord> numbered_cells;
    for (int x = 0; x != h; ++x) {
        for (int y = 0; y != w; ++y) {
            auto cell = map->at(x).at(y);
            if (cell >= 1 && cell <= 4) {
                numbered_cells.push_front(std::make_tuple(x, y));
            }
        }
    }
    // Recursions for numbered black cells
    auto retmaps = _solveNumberedCell(*map, numbered_cells);
    delete map;
#if defined(DEBUG_AKARI_CPP)
    std::cout << "After placing for all numbered black cells, have
following "
              << "branches..." << std::endl;
    for (const auto &retmap : retmaps) {
      printMap(*retmap);
    }
#endif
    GameMap *solution = nullptr;
    // Recurse into each branch
    for (const auto &retmap : retmaps) {
        if (solution != nullptr) { break; }
        // Get white cells
        forward_list<Coord> white_cells;
        for (int x = 0; x != h; ++x) {
            for (int y = 0; y != w; ++y) {
                if (retmap->at(x).at(y) == CellType::white) {
                    white_cells.push_front(std::make_tuple(x, y));
                }
            }
        }
        // Recursion for white cells
        auto solutions = _solveWhiteCell(*retmap, white_cells);
```

```
        if (!solutions.empty()) { solution = new
GameMap(*solutions.front()); }
        for (const auto &sol : solutions) { delete sol; }
    }
    for (const auto &retmap : retmaps) { delete retmap; }
    if (solution == nullptr) {
        std::cerr << "_solveAkari(): No valid solution!" << std::endl;
        throw "_solveAkari(): No valid solution!";
    }
#if defined(DEBUG_AKARI_CPP)
    std::cout << "Final solution..." << std::endl;
    printMap(*solution);
#endif
    return GameMap(*solution);
}

GameMap solveAkari(GameMap & g)
{
    // 请在此函数内返回最后求得的结果
    auto solution = _solveAkari(g);

    // Remove all markups
    for (auto &row : solution) {
        for (auto &cell : row) {
            if (cell < CellType::white) { cell = CellType::white; }
        }
    }

#if defined(DEBUG_AKARI_CPP)
    std::cout << "Return solution..." << std::endl;
    printMap(solution);
#endif
    return solution;
}

}
```

akari-multithreaded.cpp - 并行算法实现

```
// #include <bits/stdc++.h>
#include <iostream>
#include <vector>
#include <tuple>
#include <forward_list>
#include <list>
```

```cpp
#include <unordered_map>
#include <algorithm>

#include <thread>
#include <future>

#include "akari.h"
using namespace std;

// #define DEBUG_AKARI_CPP

namespace aka{
//请在命名空间内编写代码，否则后果自负

typedef vector<vector<int> > GameMap;
typedef tuple<int, int> Coord;

enum CellType
{
    mark_noplaceable    = -4,
    mark_lit           = -3,
    white              = -2,
    black_nonumber      = -1,
    light_bulb         = 5
};

inline
const Coord getMapShape(const GameMap &map)
{
    return std::make_tuple(map.size(), map.at(0).size());
}

void printMap(const GameMap &map)
{
    cout << "+";
    for (auto &_: map.at(0)) {
        cout << "--+";
    }
    cout << endl;
    for (auto &row : map) {
        cout << "|";
        for (auto &cell : row) {
            switch (cell) {
                case CellType::mark_noplaceable: cout << "X"; break;
```

```
                case CellType::mark_lit: cout << "L"; break;
                case CellType::white: cout << "  "; break;
                case CellType::black_nonumber: cout << "#"; break;
                case 0: cout << "0"; break;
                case 1: cout << "1"; break;
                case 2: cout << "2"; break;
                case 3: cout << "3"; break;
                case 4: cout << "4"; break;
                case CellType::light_bulb: cout << "?"; break;
                default: cout << " " << cell; break;
            }
            cout << "|";
        }
        cout << "\n+";
        for (auto &_: row) {
            cout << "--+";
        }
        cout << endl;
    }
}

static inline
const vector<Coord> _getCrossAround(const int x, const int y)
{
    return vector<Coord>{
        std::make_tuple(x - 1, y), std::make_tuple(x + 1, y),
        std::make_tuple(x, y - 1), std::make_tuple(x, y + 1)
    };
}

static
void _prune_black_zero(GameMap &map)
{
    int h, w; std::tie(h, w) = getMapShape(map);
    for (int x = 0; x != h - 1; ++x) {
        for (int y = 0; y != w - 1; ++y) {
            if (map.at(x).at(y + 1) == 0 || map.at(x + 1).at(y) == 0) {
                if (map.at(x).at(y) == CellType::white) {
                    map.at(x).at(y) = CellType::mark_noplaceable;
                }
            }
            if (map.at(x).at(y) == 0) {
                if (map.at(x).at(y + 1) == CellType::white) {
                    map.at(x).at(y + 1) = CellType::mark_noplaceable;
```

```
            }
            if (map.at(x + 1).at(y) == CellType::white) {
                map.at(x + 1).at(y) = CellType::mark_noplaceable;
            }
        }
    }
}

static
bool _checkPlaceable(GameMap &map, const int x, const int y, bool
leave_markup=false)
{
    // Check if cell is placeable
    if (map.at(x).at(y) == CellType::light_bulb) { return true; }
    if (map.at(x).at(y) != CellType::white) { return false; }
    int h, w; std::tie(h, w) = getMapShape(map);
    // Check row for existing light bulb
    for (int iy = y - 1; iy != -1; --iy) {
        auto &cell = map.at(x).at(iy);
        if (cell == CellType::light_bulb) { return false; }
        if (cell >= -1 && cell <= 4) { break; }
        if (leave_markup) { cell = CellType::mark_lit; }
    }
    for (int iy = y + 1; iy != w; ++iy) {
        auto &cell = map.at(x).at(iy);
        if (cell == CellType::light_bulb) { return false; }
        if (cell >= -1 && cell <= 4) { break; }
        if (leave_markup) { cell = CellType::mark_lit; }
    }
    for (int ix = x - 1; ix != -1; --ix) {
        auto &cell = map.at(ix).at(y);
        if (cell == CellType::light_bulb) { return false; }
        if (cell >= -1 && cell <= 4) { break; }
        if (leave_markup) { cell = CellType::mark_lit; }
    }
    for (int ix = x + 1; ix != h; ++ix) {
        auto &cell = map.at(ix).at(y);
        if (cell == CellType::light_bulb) { return false; }
        if (cell >= -1 && cell <= 4) { break; }
        if (leave_markup) { cell = CellType::mark_lit; }
    }
    // Check numbered black cell constraint
    for (const auto &ixy : _getCrossAround(x, y)) {
```

```
        int ix, iy; std::tie(ix, iy) = ixy;
        // Continue if out of map
        if (ix < 0 || ix >= h || iy < 0 || iy >= w) { continue; }
        int cell = map.at(ix).at(iy);
        // Continue if not a numbered black cell
        if (cell < 0 || cell > 4) { continue; }
        if (cell == 0) { return false; }
        int remain_slot_cnt = cell;
        if (ix - 1 >= 0 && map.at(ix - 1).at(iy) ==
CellType::light_bulb) { remain_slot_cnt--; }
        if (ix + 1 <  h && map.at(ix + 1).at(iy) ==
CellType::light_bulb) { remain_slot_cnt--; }
        if (iy - 1 >= 0 && map.at(ix).at(iy - 1) ==
CellType::light_bulb) { remain_slot_cnt--; }
        if (iy + 1 <  w && map.at(ix).at(iy + 1) ==
CellType::light_bulb) { remain_slot_cnt--; }
        if (remain_slot_cnt < 1) {
            return false;
        }
    }
    return true;
}

static
GameMap * _placeLightBulb(const GameMap &map, const int x, const int
y,
    bool place_markup=false)
{
    int h, w; std::tie(h, w) = getMapShape(map);
    if (x < 0 || x >= h || y < 0 || y >= w) { return nullptr; }
    GameMap *retmap = new GameMap(map);
    // Reject if is an illegal move
    if (!_checkPlaceable(*retmap, x, y, place_markup)) { delete
retmap; return nullptr; }
    retmap->at(x).at(y) = CellType::light_bulb;
    return retmap;
}

static
GameMap * _placeMultipleLightBulb(const GameMap &map,
forward_list<Coord> coords)
{
    GameMap *oldmap = new GameMap(map);
    GameMap *newmap = nullptr;
```

```
    int h, w; std::tie(h, w) = getMapShape(map);
    for (const auto &coord : coords) {
        int x, y; std::tie(x, y) = coord;
        if (x < 0 || x >= h || y < 0 || y >= w) { delete oldmap;
return nullptr; }
        newmap = _placeLightBulb(*oldmap, x, y, true);
        delete oldmap;
        if (newmap == nullptr) { return nullptr; }
        oldmap = newmap;
    }
    return newmap;
}

forward_list<GameMap *> _solveNumberedCell(const GameMap &map,
    const forward_list<Coord> &numbered_cells, int granularity=-1)
{
    int h, w; std::tie(h, w) = getMapShape(map);
    // Return case
    if (numbered_cells.empty()) { return forward_list<GameMap *>{new
GameMap(map)}; }
    const auto current_cell = numbered_cells.front();
    int x, y; std::tie(x, y) = current_cell;
    forward_list<Coord> next_numbered_cells(numbered_cells);
    next_numbered_cells.pop_front();
    // Generate branches, place light bulbs
    const forward_list<forward_list<Coord> > *coords_list = nullptr;
    const forward_list<forward_list<Coord> > _case_1_coords_list{
        {std::make_tuple(x - 1, y)},
        {std::make_tuple(x + 1, y)},
        {std::make_tuple(x, y - 1)},
        {std::make_tuple(x, y + 1)}
    };
    const forward_list<forward_list<Coord> > _case_2_coords_list{
        // Corners x 4
        {std::make_tuple(x - 1, y), std::make_tuple(x, y - 1)},
        {std::make_tuple(x - 1, y), std::make_tuple(x, y + 1)},
        {std::make_tuple(x + 1, y), std::make_tuple(x, y - 1)},
        {std::make_tuple(x + 1, y), std::make_tuple(x, y + 1)},
        // Across x 2
        {std::make_tuple(x - 1, y), std::make_tuple(x + 1, y)},
        {std::make_tuple(x, y - 1), std::make_tuple(x, y + 1)},
    };
    const forward_list<forward_list<Coord> > _case_3_coords_list{
        {std::make_tuple(x - 1, y), std::make_tuple(x, y - 1),
```

```
std::make_tuple(x, y + 1)},
       {std::make_tuple(x, y + 1), std::make_tuple(x - 1, y),
std::make_tuple(x + 1, y)},
       {std::make_tuple(x + 1, y), std::make_tuple(x, y - 1),
std::make_tuple(x, y + 1)},
       {std::make_tuple(x, y - 1), std::make_tuple(x - 1, y),
std::make_tuple(x + 1, y)},
   };
   const forward_list<forward_list<Coord> > _case_4_coords_list{{
      std::make_tuple(x - 1, y), std::make_tuple(x + 1, y),
      std::make_tuple(x, y - 1), std::make_tuple(x, y + 1)
   }};
   switch (map.at(x).at(y)) {
      case 1: { coords_list = &_case_1_coords_list; break; }
      case 2: { coords_list = &_case_2_coords_list; break; }
      case 3: { coords_list = &_case_3_coords_list; break; }
      case 4: { coords_list = &_case_4_coords_list; break; }
      default: throw;
   }
   forward_list<GameMap *> sub_branches;
   int sub_branches_len = 0;
   GameMap *retmap = nullptr;
   for (auto &coords : *coords_list) {
      if ((retmap = _placeMultipleLightBulb(map, coords)) !=
nullptr) {
         sub_branches.push_front(retmap);
         sub_branches_len++;
      }
   }
   // Recurse into branches
   forward_list<GameMap *> ret_branches;
   if (granularity < 0) {
      granularity = 0;
      for (const auto &_ : numbered_cells) { granularity++; }
   }
   // Do in multithread
   if (granularity > 7 && sub_branches_len > 2) {
   // if (true) {
      list<future<forward_list<GameMap *> > > futures;
      for (auto &branch : sub_branches) {
         futures.push_front(async(std::launch::async,
            [](const tuple<const GameMap *, const
forward_list<Coord> *, int> &args) {
               auto ret = _solveNumberedCell(*std::get<0>(args),
```

```
*std::get<1>(args), std::get<2>(args));
                delete std::get<0>(args);
                return ret;
            },
            std::make_tuple(branch, &next_numbered_cells,
granularity - 1)
        ));
      }
      for (; !futures.empty(); futures.pop_back()) {
         ret_branches.splice_after(ret_branches.cbefore_begin(),
futures.back().get());
      }
      return ret_branches;
   }
   // Do in serial
   for (auto &branch : sub_branches) {
      ret_branches.splice_after(ret_branches.cbefore_begin(),
            _solveNumberedCell(*branch, next_numbered_cells,
granularity - 1));
      delete branch;
   }
   return ret_branches;
}

static
bool _isSolution_WithMarkup(const GameMap &map)
{
   for (auto &row : map) {
      for (auto &cell : row) {
         if (cell == CellType::white || cell ==
CellType::mark_noplaceable) {
            return false;
         }
      }
   }
   return true;
}

forward_list<GameMap *> _solveWhiteCell(const GameMap &map, const
forward_list<Coord> &white_cells,
   int granularity=-1)
{
   // Success case
   if (_isSolution_WithMarkup(map)) { return forward_list<GameMap
```

```
*>{new GameMap(map)}; }
    // Fail case
    if (white_cells.empty()) { return forward_list<GameMap *>(); }
    // Recrusion
    const auto current_cell = white_cells.front();
    int x, y; std::tie(x, y) = current_cell;
    forward_list<Coord> next_white_cells(white_cells);
    next_white_cells.pop_front();
    if (granularity < 0) {
        granularity = 0;
        for (const auto &_ : white_cells) { ++granularity; }
    }
    // if (granularity > 7) {
    if (false) {
        // Muliti-threaded version (really not worthy)
        auto future_noplace = async(std::launch::async,
            [granularity](const tuple<const GameMap *, const
forward_list<Coord> *> &args) {
                auto ret = _solveWhiteCell(*std::get<0>(args),
*std::get<1>(args), granularity - 1);
                return ret;
            },
            std::make_tuple(&map, &next_white_cells)
        );
        GameMap *placed_map = _placeLightBulb(map, x, y, true);
        if (placed_map == nullptr) { return future_noplace.get(); }
        auto future_place = async(std::launch::async,
            [granularity](const tuple<const GameMap *, const
forward_list<Coord> *> &args) {
                auto ret = _solveWhiteCell(*std::get<0>(args),
*std::get<1>(args), granularity - 1);
                delete std::get<0>(args);
                return ret;
            },
            std::make_tuple(placed_map, &next_white_cells)
        );
        auto retlist = future_noplace.get();
        retlist.splice_after(retlist.cbefore_begin(),
future_place.get());
        return retlist;
    } else {
        // Single-threaded solution
        // Branch - not place
        auto retlist = _solveWhiteCell(map, next_white_cells,
```

```
granularity - 1);
       if (!retlist.empty()) { return retlist; }
       // Branch - place
       GameMap *placed_map = _placeLightBulb(map, x, y, true);
       if (placed_map != nullptr) {
          retlist.splice_after(retlist.cbefore_begin(),
                 _solveWhiteCell(*placed_map, next_white_cells,
granularity - 1));
          delete placed_map;
       }
       return retlist;
   }
   throw;
}

static
GameMap _solveAkari(const GameMap &g)
{
#if defined(DEBUG_AKARI_CPP)
   std::cout << "Input map..." << std::endl;
   printMap(g);
#endif
   GameMap *map = new GameMap(g);
   int h, w; std::tie(h, w) = getMapShape(*map);
   // Prune
   _prune_black_zero(*map);
#if defined(DEBUG_AKARI_CPP)
   std::cout << "After pruning..." << std::endl;
   printMap(*map);
#endif
   // Get numbered cells
   forward_list<Coord> numbered_cells;
   for (int x = 0; x != h; ++x) {
      for (int y = 0; y != w; ++y) {
         auto cell = map->at(x).at(y);
         if (cell >= 1 && cell <= 4) {
            numbered_cells.push_front(std::make_tuple(x, y));
         }
      }
   }
   // Recursions for numbered black cells
   auto retmaps = _solveNumberedCell(*map, numbered_cells);
   delete map;
#if defined(DEBUG_AKARI_CPP)
```

```
   std::cout << "After placing for all numbered black cells, have
following "
          << "branches..." << std::endl;
   int num_retmaps = 0;
   for (const auto &retmap : retmaps) {
     printMap(*retmap);
     ++num_retmaps;
   }
   std::cout << "total retmaps " << num_retmaps << std::endl;
#endif
   GameMap *solution = nullptr;
   // Recurse into each branch
   // TODO: Parallelize
   list<future<GameMap> > white_cell_futures;
   for (const auto &retmap : retmaps) {
      if (solution != nullptr) { break; }
      // Get white cells
      forward_list<Coord> white_cells;
      for (int x = 0; x != h; ++x) {
         for (int y = 0; y != w; ++y) {
            if (retmap->at(x).at(y) == CellType::white) {
               white_cells.push_front(std::make_tuple(x, y));
            }
         }
      }
      // Launch recursion for white cells
      white_cell_futures.push_back(async(std::launch::async,
         [](const tuple<const GameMap *, const forward_list<Coord> >
&args) {
            auto solutions = _solveWhiteCell(*std::get<0>(args),
std::get<1>(args));
            delete std::get<0>(args);
            GameMap solution = GameMap();
            if (!solutions.empty()) {
               solution = GameMap(*solutions.front());
            }
            for (const auto &sol : solutions) {
               delete sol;
            }
            return solution;   // on fail, returns an empty GameMap
         },
         std::make_tuple(retmap, white_cells)
      ));
   }
```

```
    while (!white_cell_futures.empty()) {
        auto ret = white_cell_futures.front().get();
        white_cell_futures.pop_front();
        if (!ret.empty()) {
            return ret;
        }
    }
    if (solution == nullptr) {
        std::cerr << "_solveAkari(): No valid solution!" << std::endl;
        throw "_solveAkari(): No valid solution!";
    }
#if defined(DEBUG_AKARI_CPP)
    std::cout << "Final solution..." << std::endl;
    printMap(*solution);
#endif
    auto retsolution = GameMap(*solution);
    delete solution;
    return retsolution;
}

GameMap solveAkari(GameMap & g)
{
    // 请在此函数内返回最后求得的结果
    auto solution = _solveAkari(g);

    // Remove all markups
    for (auto &row : solution) {
        for (auto &cell : row) {
            if (cell < CellType::white) { cell = CellType::white; }
        }
    }

#if defined(DEBUG_AKARI_CPP)
    std::cout << "Return solution..." << std::endl;
    printMap(solution);
#endif
    return solution;
}

}
```