中山大学计算机学院

算法设计与分析

本科生实验报告

（2023学年秋季学期）

课程名称：Algorithm design and analysis

教学班级	专业(方向)	学号	姓名
计科二班	计算机科学与技术	21307185	张礼贤

实验题目与要求

- 哨兵布置问题。一个博物馆由排成m*n个矩阵阵列的陈列室组成，需要在陈列室中设立哨位，每个哨位上的哨兵除了可以监视自己所在陈列室外，还可以监视他上、下、左、右四个陈列室，试基于分支限界法给出一个最佳哨位安排方法，使得所有陈列室都在监视之下，但使用的哨兵最少。
- 数据输入：由文件input.txt给出输入数据。第1行有2个正整数m和n。
- 结果输出：将计算出的哨兵人数及其最佳哨位安排输出到文件output.txt。文件的第1行是哨兵人数，接下来的m行中每行n个数， 0表示无哨位， 1表示哨位。
- 输入示例： m=n=4.

算法设计

使用分支限界法解决博物馆哨兵布置问题的算法。将哨兵（机器人）放置在博物馆以监视房间，同时尽量减少使用的哨兵数量，并剪枝那些可以保证是次优的分支。

算法原理：

1. **状态表示：**
    - 状态由一个Node结构表示，包含有关哨兵当前放置、正在监视的房间以及算法在搜索空间中的当前位置的信息。
2. **优先队列：**
    - 使用优先队列（store）以有效地探索搜索空间。优先级由配置中被监视房间的数量确定。
3. **分支和剪枝：**
    - **通过在不同位置放置哨兵来探索不同的配置：**
        - 在解决方案空间中，每个房间都考虑在放置哨兵的情况下的不同配置。算法遍历房间的二维数组，按照从上到下、从左到右的顺序进行。对于任何给定的格子，假设其前面的格子的哨兵放置是最优的，那么在当前格子，只需考虑将哨兵放置在下方、右方以及当前位置的三种情况。这种方式有助于减少搜索树的分支数，从而提高算法的效率。
    - **优先队列帮助首先探索具有较少被监视房间的配置：**
        - 使用优先队列(store)是为了保证在探索解空间时，首先考虑具有较少被监视房间的配置。队列中的元素按照每个配置中被监视房间的数量进行排序，小顶堆的性质确保了每次弹出的配置都是当前队列中监视房间最少的。这样的策略有助于算法更快地找到潜在的最优解。
    - **通过根据被监视房间的数量舍弃保证是次优的配置来进行剪枝：**
        - 在算法的执行过程中，通过观察被监视房间的数量，可以判断当前配置是否有可能是最优解。如果当前配置的被监视房间数量已经超过了先前发现的最优解，那么可以放心地舍弃这个分支，因为在这个分支上找到更优解的可能性很小。这种剪枝策略帮助算法更快地搜索到全局最优解，减少不必要的计算。
        - 初始的时候可以设置阈值为 m * n / 3 + 2，这里假设机器人只能监视三个格子，即不能上下监视，并为了宽松条件，加上一个2作为容错。

算法步骤：

1. **初始化：**
    - init函数初始化状态，设置哨兵的初始放置和被监视的房间。
2. **放置哨兵：**
    - set_node函数通过在不同位置（下方、当前、右方）放置哨兵创建新的配置。它更新状态并将新的配置添加到优先队列。
3. **主循环：**
    - 主循环持续直到优先队列为空。
    - 在每次迭代中，算法从优先队列（store）中检索具有最少被监视房间的配置。
    - 如果(当前获得的哨兵数量最优解-当前已有的哨兵数目) * 5 < 剩余的房间数目，则一定不是最优解，可以跳过进行剪枝
    - 如果整个博物馆都被监视，则如果当前配置使用的哨兵更少，就更新结果。
4. **输出：**
    - 算法输出使用的总哨兵数（res）以及配置矩阵，其中1表示哨兵的位置。

代码实现

```
#include <iostream>
#include <vector>
#include <queue>
#include <cstring>
#include <fstream>

using namespace std;

// m*n的房间
int m,n;
// 自身和上下左右5个方位
int ds[5][2]={
    {0, 0},
    {0, 1},
    {0,-1},
    {1, 0},
    {-1,0}
};
//最优结果 ans为机器人个数 ans_arr为每个机器人的位置

const int MAX_M = 20;
const int MAX_N = 20;

struct Node{
    // 机器人位置
    int board[MAX_M][MAX_N];
    // 被监视的房间位置
    int watched[MAX_M][MAX_N];
    // (i,j)为当前遍历到的坐标
    int cur_i,cur_j;
    // 机器人数    被监视的房间个数
    int count_s,count_r;
};

// 优先队列的比较函数重写
struct cmp
{
    bool operator() (Node a, Node b)
    {
        // 比较哪个快照被监视房间多（小顶堆）
        return a.count_r > b.count_r;
```

```
    }
};

priority_queue<Node, vector<Node>, cmp> store;

Node init_node(Node node)
{
    // 初始化board数组全为0
    memset(node.board,0,sizeof(node.board));
    // 初始化watched数组全为0
    memset(node.watched,0,sizeof(node.watched));
    // 当前点在i=1,j=1
    node.cur_i=1;node.cur_j=1;
    // 当前的机器人数；当前的监控房间数
    node.count_s=0;node.count_r=0;

    // 在博物馆的上下扩充两行
    for(int i=0;i<=m+1;i++)
        node.watched[i][0]=node.watched[i][m+1]=1;
    // 在博物馆的左右扩充两列
    for(int i=0;i<=n+1;i++)
        node.watched[0][i]=node.watched[n+1][i]=1;
    return node;
}

void set_node(Node p,int x,int y)
{
    Node node;
    node=init_node(node);
    node.cur_i=p.cur_i;
    node.cur_j=p.cur_j;
    node.count_r=p.count_r;
    memcpy(node.board, p.board, sizeof(p.board));
    memcpy(node.watched, p.watched, sizeof(p.watched));

    // 在（x,y)点新增机器人，机器人数量要+1
    node.board[x][y]=1;
    node.count_s=p.count_s+1;

    // 对这个新增机器人的上下左右和自身标记被监控
    for(int d=0;d<5;d++)
    {
        // pos_x,pos_y表示机器人上下左右位置，我们标记这些位
```

```
置的房间被监控
            int pos_x=x+ds[d][0];
            int pos_y=y+ds[d][1];
            node.watched[pos_x][pos_y]++;

            // 标记一个房间，count_r就加一。
            // 一定要等于1，因为有的房间会被重复监控，那就是2了
            if(node.watched[pos_x][pos_y]==1)
            {
                node.count_r++;
            }
        }

        // 如果行数不越界 且 当前点被监控了
        while(node.cur_i<=m && node.watched[node.cur_i]
[node.cur_j])
        {
            // 当前点的列右移一个单位
            node.cur_j++;
            // 如果右移之后越界了，就换行
            if(node.cur_j>n)
                node.cur_i++,node.cur_j=1;
        }

        // 把当前快照存到优先队列里，会调用cmp排序，保证最顶端的是
最优的快照
        store.push(node);
        return;
}

int main()
{
    int res = m * n /3 + 2;
    int ans[MAX_M][MAX_N];

    ifstream InputFile("input.txt");
    if(!InputFile.is_open())
    {
        cout<<"Unable to open the input file!"<<endl;
        return 1;
    }

    InputFile >> m >> n;
```

```
    InputFile.close();
    // 机器人最多的数量
    res=m*n/3+2;
    // 初始化
    Node node;
    node=init_node(node);
    // 快照放入队列
    store.push(node);
    // 如果队列不空
    while(!store.empty())
    {
        // 返回队列第一个
        Node p=store.top();
        // 队列把第一个弹走
        store.pop();

        // 如果剩余的房间数大于当前最优解减去已放置的哨兵数量乘以5，则当前节点一定不会引出最优解，可以剪枝
        if((res - p.count_s) * 5 < m * n - p.count_r)continue;

        // 如果房间没有全被监控，则分别在当前遍历点的下方、本身、右方放置机器人
        if(p.count_r < m*n)
        {
            // 1、在下方放置
            // 判断条件就是下方有位置可放，不能在最后一行）
            if(p.cur_i<m)
                set_node(p,p.cur_i+1,p.cur_j);

            // 2、在本身放置。
            // 第一个判断条件是在它已经没有下方和右方的点的情况下，只能选择自身
            // 第二个判断条件是它的右边没有被监控
            if((p.cur_i==m && p.cur_j==n) || p.watched[p.cur_i][p.cur_j+1]==0)
                set_node(p,p.cur_i,p.cur_j);

            // 3、在右方放置
            // 第一个判断条件是遍历点右边是没被监控的点
            // 第二个判断条件是遍历点右边的右边是没有监控的点
            if(p.cur_j<n && (p.watched[p.cur_i][p.cur_j+1]==0 || p.watched[p.cur_i][p.cur_j+2]==0))
```

```
                set_node(p,p.cur_i,p.cur_j+1);
        }
        // 如果房间全被监控了
        else
        {
            // 如果已安置的机器人数是目前最少的，更新结果ans
            if(p.count_s<res)
            {
                res=p.count_s;
                // 把这种安置方法保存到结果数组ans_arr里面
                memcpy(ans, p.board, sizeof(p.board));
            }
        }

    }


    ofstream OutputFile("output.txt",ios::app);
    if(!OutputFile.is_open())
    {
        cout << "Unable to open the output file!" <<endl;
        return 1;
    }

    OutputFile <<"The total sentinels setted are: "  <<
res <<endl;
    OutputFile << "The result matrix is (1 for the
postion of sentinel): "<<endl;

    for(int i=1;i<=m;i++)
    {
        for(int j=1;j<=n;j++)
        {
            OutputFile<<ans[i][j] << ' ';
        }
        OutputFile<<endl;
    }
}
```

算法分析

实验结果分析

实验结果展示详情请见output.txt文件，经过验证，其结果是正确的

复杂度分析

以下是对代码的时空复杂度分析：

- **时间复杂度分析：**
  - init_node 函数中，对 board 和 watched 数组进行了初始化，时间复杂度为 $O(m * n)$。
  - set_node 函数中的循环，在放置机器人时，对相邻房间进行了标记，时间复杂度取决于机器人放置的位置，最坏情况下可能是 $O(m * n)$。
  - 主循环 while(!store.empty()) 的时间复杂度取决于队列中节点的数量。由于每次放置机器人都会生成多个子节点，因此队列的大小可能会非常大。在最坏情况下，时间复杂度可能达到指数级别，即 $O(3^{(m*n)/3})$。
- **空间复杂度分析：**
  - 每个节点的存储空间包括 board 和 watched 两个二维数组，大小均为 $O(m * n)$，以及几个基本类型的变量。因此，每个节点的空间复杂度为 $O(m * n)$。
  - 优先队列 store 中最多同时存储所有可能的节点，因此空间复杂度也达到指数级别。