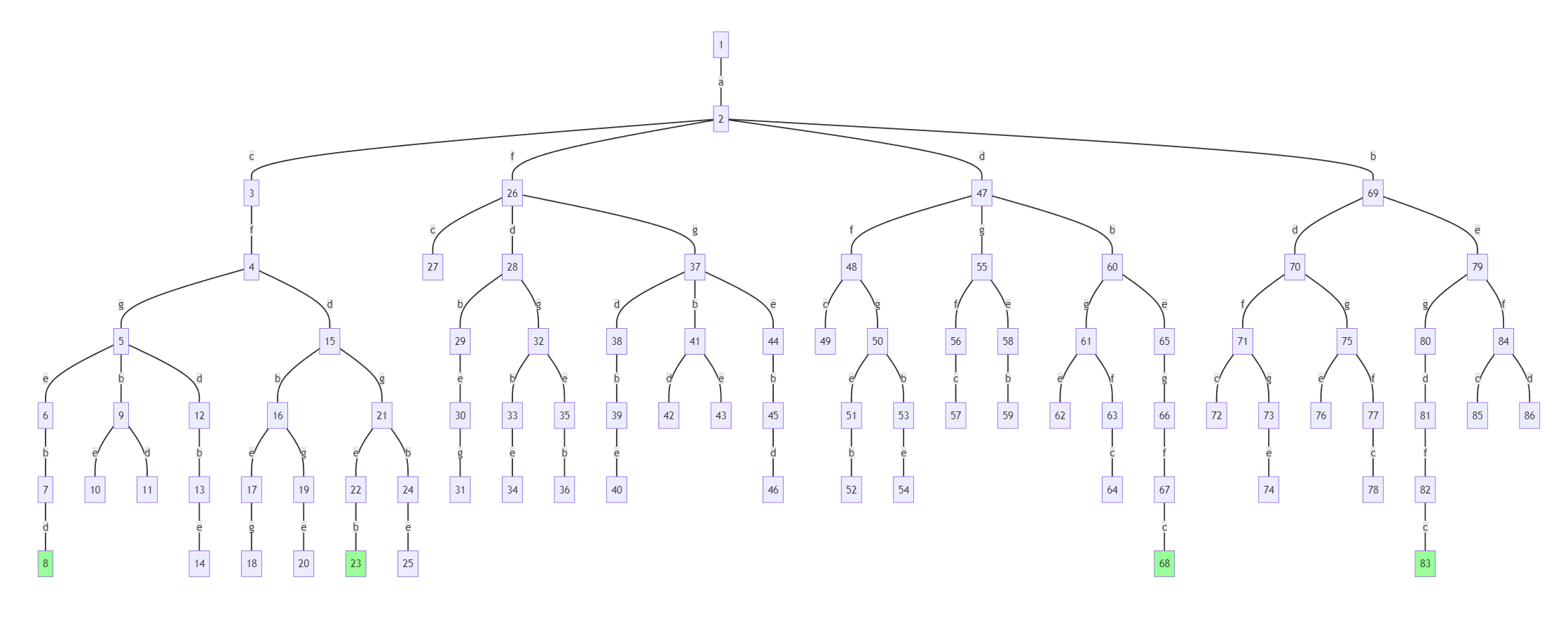
1. 解空间树如下：



其中方框中的状态表示状态变化的顺序，绿色标记即为可行解

acfdgeba/acgebda/adbegfca/abegdfca

搜索过程为：从状态图1到状态图2依次往下搜索，并且设置标记表示走过了，直到没路可走，就回退到上一步检查还没走过的路，并且清除标记。以此类推完成整个回溯。

例如：

1. >c->f->g->e->b->d

回溯到g->b->e

回溯到g->b->d

回溯到g->d->b->e

等等

1. 动态规划求解找零问题，有无限张1元、2元和3元

设置状态表如下：

|  |  |
| --- | --- |
| 面额 | 组合 |
| 1 | 1元\*1 |
| 2 | 1元\*2 |
| 3 | 3元\*1 |
| 4 | 1元\*1+3元\*1 |
| 5 | 5元\*1 |
| 6 | 1元\*1+5元\*1， 3元\*2 |
| 7 | 1元\*2+5元\*1， 1元\*1+3元\*2 |
| 8 | 3元\*1+5元\*1 |
| 9 | 1元\*1+3元\*1+5元\*1， 3元\*3 |

故解为：1张1元与一张3元与一张5元、3张3元

伪代码：

Algorithm note\_change(notes[0..n-1], target)

    // 用动态规划求解找零问题所有解

    // 输入：找零面额notes[0..n-1]，升序排列，找零目标target

    // 输出：面额为target的找零结果组合的集合

    // 首先构建状态表，初始化1至5元的最少张数组合

    ans[1] = [[0, 1, 0, 0, 0, 0]]

    ans[2] = [[0, 2, 0, 0, 0, 0]]

    ans[3] = [[0, 0, 0, 1, 0, 0]]

    ans[4] = [[0, 1, 0, 1, 0, 0]]

    ans[5] = [[0, 0, 0, 0, 0, 1]]

    // 从目前最大面额的下一个开始，直到目标面额

    for i <- note[n - 1] + 1 to target do

        temp <- ∞

        k <- 0

    // 找到最小组合张数temp，并记录组合的索引

    for j <- 1 to ⌊i / 2⌋ do

        cur <- count(ans[j]) + count(and[i - j])

        if cur < temp then

        temp <- cur

        idx[k] <- j

        k <- k + 1

    // 根据记录的索引更新状态表

    for j <- 0 to k do

        temp <- add(ans[idx[j]], ans[idx[i - j]])

        // 去重

        if temp not in ans[i]

            ans[i]中加入temp

    return ans[target]

Algorithm count(a[0..n - 1])

    // 输入：某一面额需要的面额组合

    // 输出：组合面额张数

    ans <- 0

    for i <- 0 to n - 1 do

        ans <- ans + a[i]

    return ans

Algorithm add(a[0..n - 1], b[0..n - 1])

    // 输入：a[0..n - 1]，b[0..n - 1]，两个面额需要的面额组合

    // 输出：所输入的两种面额组合的加合

    for i <- 0 to n - 1 do

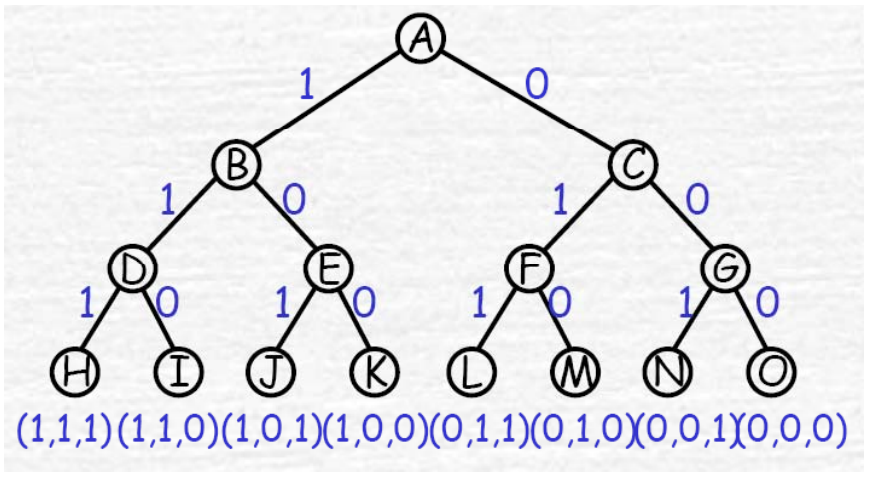
        ans[i] = a[i] + b[i]

    return ans

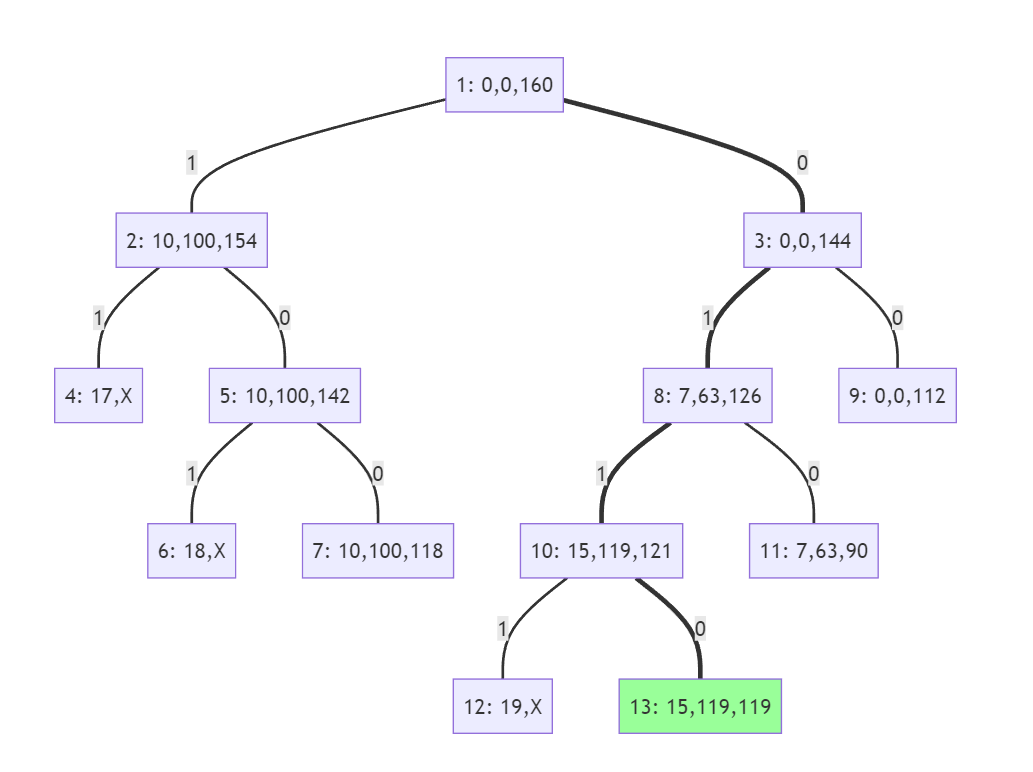
算法过程：

设计ans[i]记录面额为i所需要的最少的张数的组合，利用二维数组记录组合，ans[i][j][k]表示第j中组合的面额为k需要多少张，考虑到所给出的面额1，3，5，做出初始化ans[1..5]，对于面额n，在总金额为n - dj的一堆面额中加上面额为dj的一堆面额，从所有组合中取最小的组合，注意该过程中需要取最小张数和去重。

* 1. 解空间树如下：



* 1. 搜索过程：

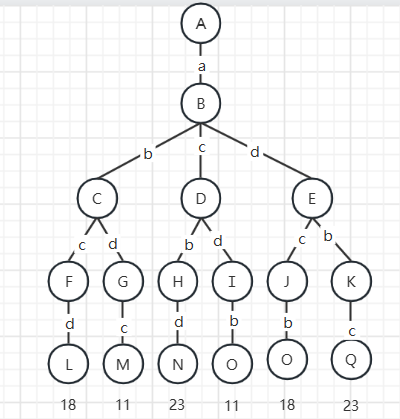


其中方框中的内容为：第i个状态即第i个节点（表示搜索顺序），背包当前重量，背包当前价值以及UP\_BOUND。解空间树的分支为1或0，表示装或不装当前物品。

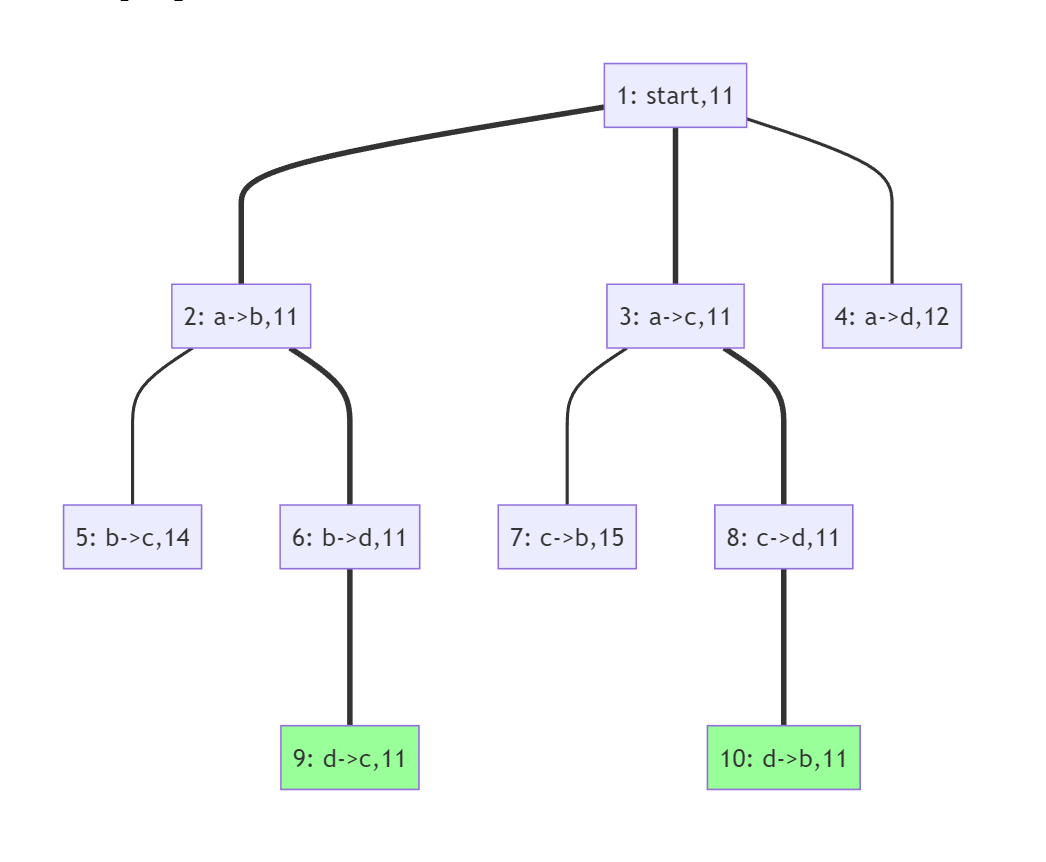
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 扩展结点 | 活结点 | 优先队列 | 可行解 | 解值 |
| 1 | 2，3 | 2，3 |  |  |
| 2 | 4（死节点），5 | 3，5 |  |  |
| 3 | 8，9 | 5，8，9 |  |  |
| 5 | 6（死节点），7 | 8，7，9 |  |  |
| 8 | 10，11 | 10，7，9，11 |  |  |
| 10 | 12（死节点），13 | 13，7，9，11 | 13 | 119 |
| 13 |  |  |  |  |

* 1. 结果为选择第2，3个物品，解值为119

1. TSP
   1. 解空间树如下：



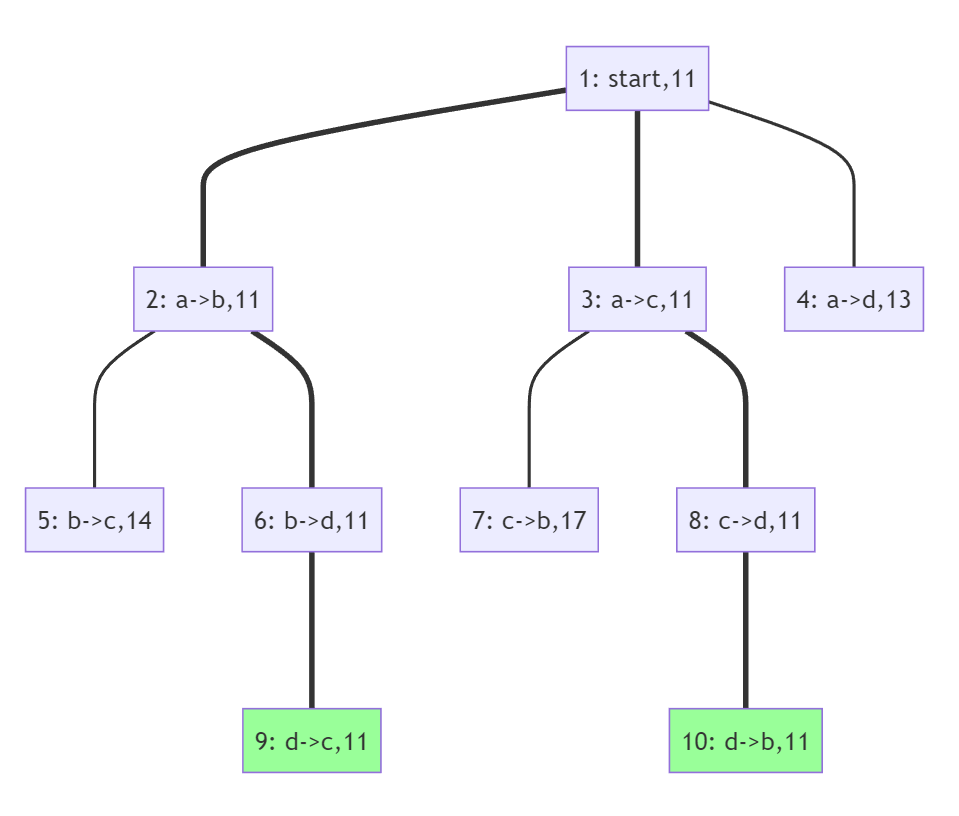
* + 1. LOW\_BOUND设计为邻接矩阵每行尽量取最小两个数，表示一个节点两条边尽量小。并且使用贪心算法计算出“上界”为11



* + 1. 搜索过程

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 扩展结点 | 活结点 | 优先队列 | 可行解 | 解值 |
| 1 | 2，3，4（剪枝） | 2，3 |  |  |
| 2 | 5（剪枝），6 | 3，6 |  |  |
| 3 | 7（剪枝），8 | 6，8 |  |  |
| 6 | 9 | 8，9 | 9 | 11 |
| 8 | 10 | 9，10 | 10 | 11 |
| 9 |  | 10 |  |  |
| 10 |  |  |  |  |

* + 1. 最优解为a->b->d->c 或 a->c->d->b，解值均为11
  1. 1. LOW\_BOUND设计为在边集中选择还未选择的最小的边的加合。并且使用贪心算法计算出“上界”为11



* + 1. 搜索过程同上
    2. 最优解同上

1. 插棒
2. 剩下插棒位置不限
3. 主要思路

对于棋盘上当前的空位，尝试遍历周围的五个方向，若该方向上邻居的邻居存在，则可以将插棒移动至当前空位，并将被跳过的邻居移走，其中是否为空位可以用一个布尔型数组表示true表示有插棒，false则表示空位。

然后进入递归函数重复上述步骤。其中可以利用剪枝函数判断若当前步数已经大于13则可以停止。递归函数结束的条件是棋盘上只剩下一个插棒。

退出递归函数的时候，将之前的false重新置为true，以便下一次的回溯。

count为全局变量，初始值为0，记录移动步数

board记录棋盘是否为空

DIRECTION数组记录5个方向邻居的邻居的索引offset

NEIGHTBOR数组记录5个方向上邻居的索引offset

1. 伪代码

Algorithm triangle\_problem(board[0...14])

    // 剪枝，当前步数大于13就不用继续了

    if count > 13 then

        return

    // 只剩下一个插棒结束递归

    num <- 0

    for i <- 0 to 14 do

        if board[i] = true then

            num <- num + 1

    if num = 1 then

        record()

        return

    // 找出空位

    for i <- 0 to 14 do

        if board[i] = true then

            continue

        // 从空位找到五个方向上的邻居的邻居

        for j <- 0 to 4 do

            // 邻居的邻居

            new\_empty = i + DIRECTION[j]

            // 出界或空位就跳过

            if new\_empty < 0 or new\_empty > 14 or board[new\_empty] = true then

                continue

            // 得到i的邻居

            neighbor = i + NEIGHBOR[j]

            // 更新期盼

            board[i] <- true

            board[new\_empty] <- false

            board[neighbor] <- false

            // 增加当前步数

            count <- count + 1

            // 进入递归函数

            triangle\_problem(board)

            // 回溯

            board[i] <- false

            board[new\_empty] <- true

            board[neighbor] <- true

            // 减少当前步数

            count <- count - 1

1. 剩下插棒的位置在最初空位上
2. 主要思路

对于棋盘上当前的空位，尝试遍历周围的五个方向，若该方向上邻居的邻居存在，则可以将插棒移动至当前空位，并将被跳过的邻居移走，其中是否为空位可以用一个布尔型数组表示true表示有插棒，false则表示空位。

然后进入递归函数重复上述步骤。其中可以利用剪枝函数判断若当前步数已经大于13则可以停止。递归函数结束的条件是棋盘上只剩下一个插棒，并且该插棒位置与初始空位相同我们才将问题的解记录下来。

退出递归函数的时候，将之前的false重新置为true，以便下一次的回溯。

count为全局变量，初始值为0，记录移动步数

board记录棋盘是否为空

DIRECTION数组记录5个方向邻居的邻居的索引offset

NEIGHTBOR数组记录5个方向上邻居的索引offset

1. 伪代码，其中count为全局变量，初始值为0，移动步数

Algorithm triangle\_problem(board[0...14])

    // 剪枝，当前步数大于13就不用继续了

    if count > 13 then

        return

    // 只剩下一个插棒结束递归

    num <- 0

    idx <- -1

    for i <- 0 to 14 do

        if board[i] = true then

            num <- num + 1

            idx <- i

    if num = 1 then

        // 初始位置才记录

        if idx = INIT\_LOCATION then

            record()

        return

    // 找出空位

    for i <- 0 to 14 do

        if board[i] = true then

            continue

        // 从空位找到五个方向上的邻居的邻居

        for j <- 0 to 4 do

            // 邻居的邻居

            new\_empty = i + DIRECTION[j]

            // 出界或空位就跳过

            if new\_empty < 0 or new\_empty > 14 or board[new\_empty] = true then

                continue

            // 得到i的邻居

            neighbor = i + NEIGHBOR[j]

            // 更新期盼

            board[i] <- true

            board[new\_empty] <- false

            board[neighbor] <- false

            // 增加当前步数

            count <- count + 1

            // 进入递归函数

            triangle\_problem(board)

            // 回溯

            board[i] <- false

            board[new\_empty] <- true

            board[neighbor] <- true

            // 减少当前步数

            count <- count - 1

编程题，金矿

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

    int dy[4] = {-1, 1, 0, 0};

    int m, n;

    int ans = 0;

    int temp = 0;

    void help(vector<vector<int>>& grid, vector<vector<bool>>& vis, int x, int y) {

        ans = max(ans, temp);

        // 出界

        if (x < 0 || y < 0 || x >= m || y >= n) {

            return;

        }

        // 没东西或已经访问过

        if (grid[x][y] == 0 || vis[x][y]) {

            return;

        }

        // 当前收获

        temp += grid[x][y];

        // 访问过

        vis[x][y] = true;

        // 遍历四个方向

        for (int i = 0; i < 4; ++i) {

            int new\_x = x + dx[i];

            int new\_y = y + dy[i];

            help(grid, vis, new\_x, new\_y);

        }

        vis[x][y] = false;

        temp -= grid[x][y];

    }

    int getMaximumGold(vector<vector<int>>& grid) {

        m = grid.size();

        n = grid[0].size();

        for (int i = 0; i < m; ++i) {

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

                vector<vector<bool>> vis(m, vector<bool>(n, false));

                help(grid, vis, i, j);

            }

        }

        return ans;

    }

算法思路

利用一个变量temp记录当前开采总量，ans记录解值，一个首先枚举网格内所有点作为起点，进行开采。

利用回溯与递归的方法。递归结束条件为出界或开采到0个金矿或访问过的地方。

递归体中首先将ans更新为temp和ans中的最大值，然后将temp增加当前开采的金矿，并将当前位置设置为访问过vis[x][y] = true，遍历上下左右四个方向进入递归函数，跳出递归函数之后，将该位置设置为没访问过vis[x][y] = false，以便下一次访问，temp减去当前开采的金矿。

最后即可得到解值。

复杂度分析：

空间复杂度：每轮递归都需要一个和原来grid大小相同的vis数组记录是否访问过，因此复杂度为O(mn)

时间复杂度：共有mn个起点，每个起点对应一轮递归，注意到仅仅一开始进入递归时可能有四个方向进入函数，而之后最多三个方向，因此复杂度为O(mn \* 3mn)

运行截图

