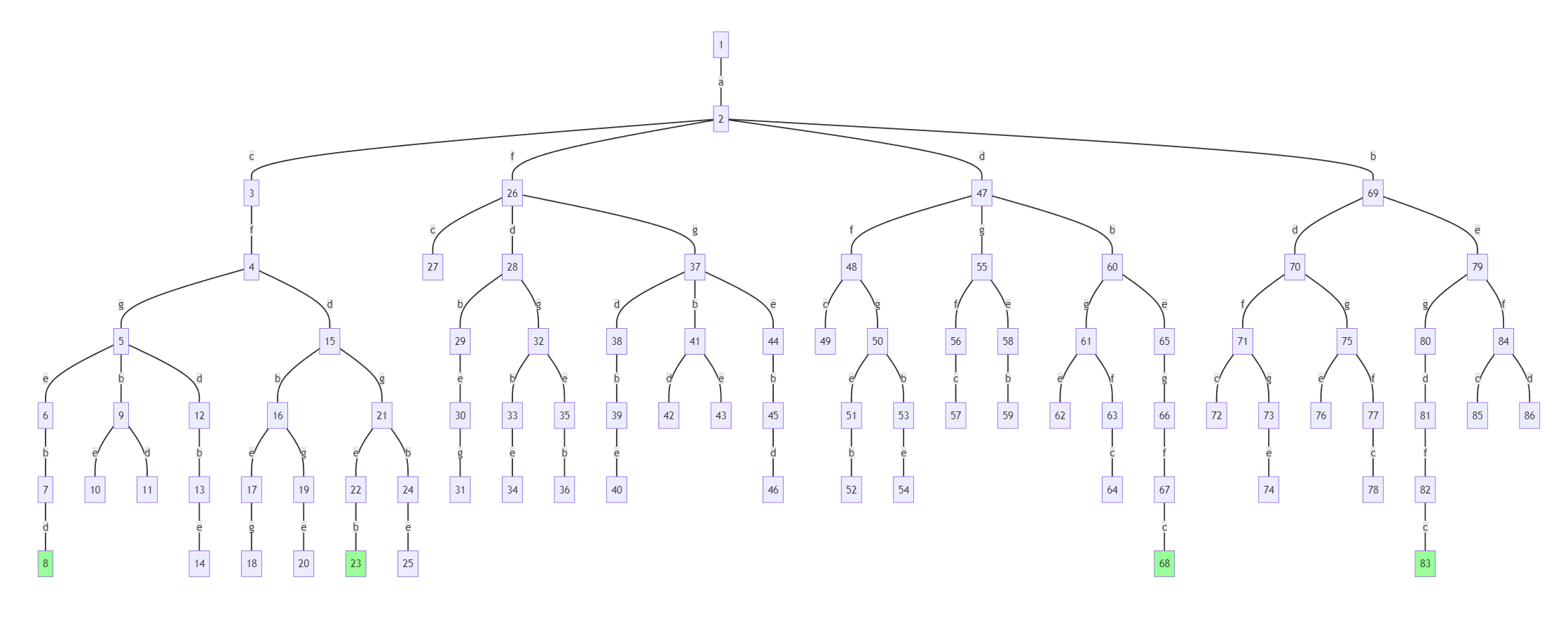
1. 解空间树如下：



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，找到一个解

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 <- 0 to n - 1 do

// 遍历i - note[j]的每种组合

for x in ans[i - note[j]] do

cur <- count(x)

if cur < temp then

temp <- cur

    // 根据最小组合的张数更新状态表

k <- 0

for j <- 0 to n - 1 do

// 遍历i - note[j]的每种组合

for x in ans[i - note[j]] do

// 如果组合最小

if count(x) = temp then

// 增加一张note[j]面额的

y = x

y[note[j]] <- y[note[j]] + 1

// 没有重复

if y not in ans[i] then

ans[i, k] <- y

k <- k + 1

    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

代码如下：

# 面额种类

NOTE = [1, 3, 5]

# 需要兑换的零钱

target = 9

def func(NOTE, target):

    dp = [None] \* (target + 1)

    # dp表建立，每个对应一个集合

    for i in range(target + 1):

        dp[i] = set()

    # 将list转为tuple才有hash，初始化

    dp[1].add(tuple([1, 0, 0]))

    dp[2].add(tuple([2, 0, 0]))

    dp[3].add(tuple([0, 1, 0]))

    dp[4].add(tuple([1, 1, 0]))

    dp[5].add(tuple([0, 0, 1]))

    for i in range(6, target + 1):

        # 一个很大的数

        minSum = 99999999

        # 得到最小组合

        for note in NOTE:

            for x in dp[i - note]:

                tempSum = sum(x)

                minSum = min(minSum, tempSum)

        # 通过最小组合，可能不止一个，找到新的组合

        for note in NOTE:

            for x in dp[i - note]:

                if minSum == sum(x):

                    temp = list(x)

                    temp[NOTE.index(note)] += 1

                    dp[i].add(tuple(temp))

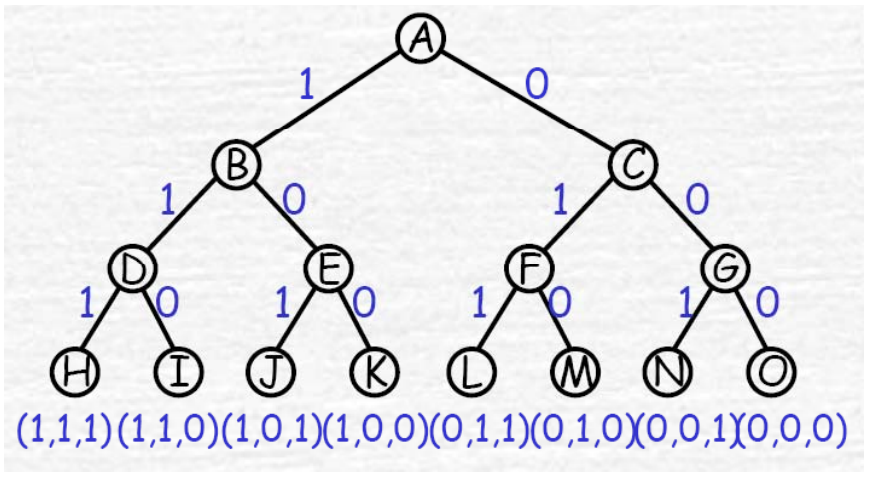
    return dp[target]

print(func(NOTE, target))

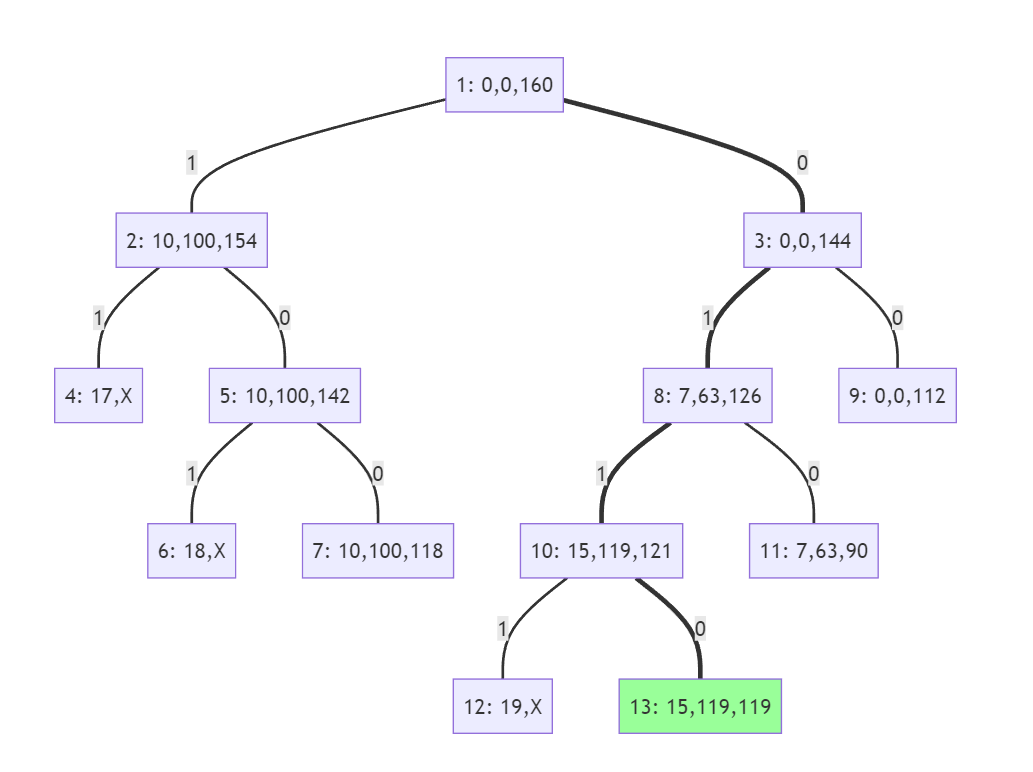
算法过程：

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，我们取n - 1, n - 3, n - 5面额的组合，在此基础上增加一张对应面额，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：

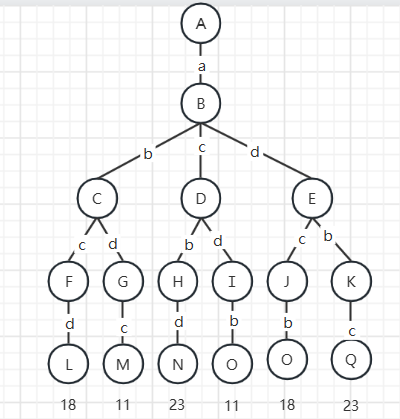


其中方框中的内容为：第i个状态即第i个节点（表示搜索顺序），背包当前重量，背包当前价值以及UP\_BOUND。解空间树的分支为1或0，表示装或不装当前物品。

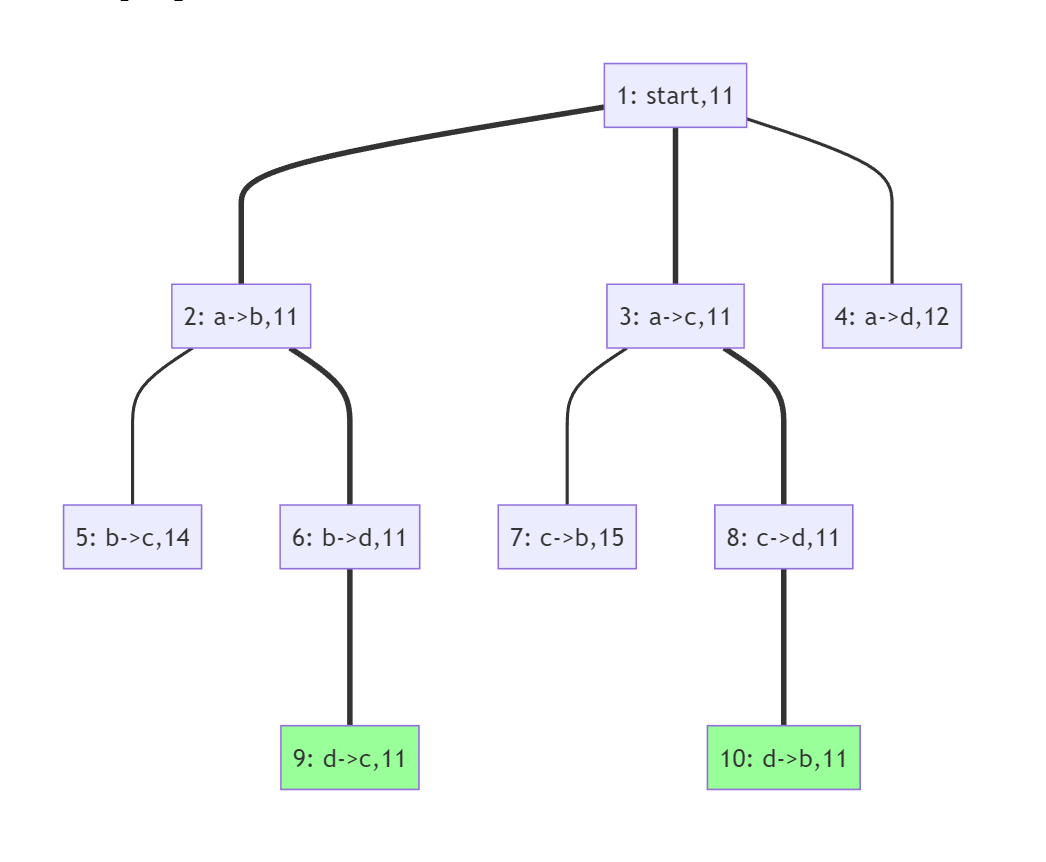
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 扩展结点 | 活结点 | 优先队列 | 可行解 | 解值 |
| 1 | 2，3 | 2，3 |  |  |
| 2 | 4（死节点），5 | 3，5 |  |  |
| 3 | 8，9 | 5，8，9 |  |  |
| 5 | 6（死节点），7 | 8，7，9 |  |  |
| 8 | 10，11 | 10，7，9，11 |  |  |
| 10 | 12（死节点），13 | 13，7，9，11 | 13 | 119 |
| 13 |  |  |  |  |

* 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：



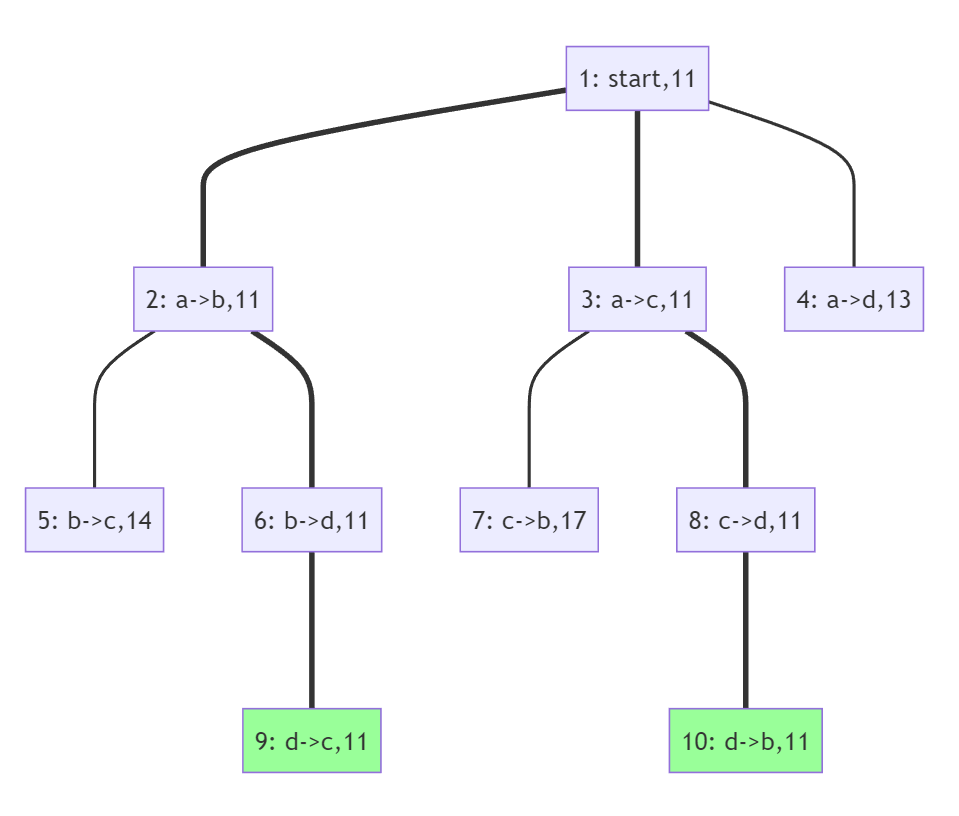
* + 1. LOW\_BOUND设计为邻接矩阵每行尽量取最小两个数，表示一个节点两条边尽量小。并且使用贪心算法计算出“上界”为11



* + 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，尝试遍历周围的六个方向，若该方向上邻居的邻居存在且邻居也存在，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6个方向邻居的邻居的索引offset

NEIGHTBOR数组记录6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

        // 从空位找到六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，尝试遍历周围的六个方向，若该方向上邻居的邻居存在且邻居也存在，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6个方向邻居的邻居的索引offset

NEIGHTBOR数组记录6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

        // 从空位找到六个方向上的邻居的邻居

        for j <- 0 to 5 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]

            // 更新棋盘

            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. 搜索结果如下
2. 代码如下：

import sys

NEI = []

NEI.append([-1, -1, 2, 1, -1, -1])

NEI.append([0, 2, 4, 3, -1, -1])

NEI.append([-1, -1, 5, 4, 1, 0])

NEI.append([1, 4, 7, 6, -1, -1])

NEI.append([2, 5, 8, 7, 3, 1])

NEI.append([-1, -1, 9, 8, 4, 2])

NEI.append([3, 7, 11, 10, -1, -1])

NEI.append([4, 8 ,12, 11, 6, 3])

NEI.append([5, 9, 13, 12, 7, 4])

NEI.append([-1, -1, 14, 13, 8, 5])

NEI.append([6, 11, -1, -1, -1, -1])

NEI.append([7, 12, -1, -1, 10, 6])

NEI.append([8, 13, -1, -1, 11, 7])

NEI.append([9, 14, -1, -1, 12, 8])

NEI.append([-1, -1, -1, -1, 13, 9])

NEINEI = []

NEINEI.append([-1, -1, 5, 3, -1, -1])

NEINEI.append([-1, -1, 8, 6, -1, -1])

NEINEI.append([-1, -1, 9, 7, -1, -1])

NEINEI.append([0, 5, 12, 10, -1, -1])

NEINEI.append([-1, -1, 13, 11, -1, -1])

NEINEI.append([-1, -1, 14, 12, 3, 0])

NEINEI.append([1, 8, -1, -1, -1, -1])

NEINEI.append([2, 9, -1, -1, -1, -1])

NEINEI.append([-1, -1, -1, -1, 6, 1])

NEINEI.append([-1, -1, -1, -1, 7, 2])

NEINEI.append([3, 12, -1, -1, -1, -1])

NEINEI.append([4, 13, -1, -1, -1, -1])

NEINEI.append([5, 14, -1, -1, 10, 3])

NEINEI.append([-1, -1, -1, -1, 11, 4])

NEINEI.append([-1, -1, -1, -1, 12, 5])

board = [True] \* 15

board[12] = False

ans = []

def func(count):

    # 步数超过13

    if count > 13:

        return

    # 只剩一个

    # num = 0

    # idx = -1

    # for i, x in enumerate(board):

    #     if x:

    #         num += 1

    #         idx = i

    # if num  == 1 and idx == 12:

    #     print("success")

    #     print(ans)

    #     sys.exit()

    #     return

    if board.count(True) == 1:

        print("success")

        print(ans)

        sys.exit() # 发现答案数量过多，得到一个答案就结束程序

        return

    print(count, board.count(False))

    # 遍历整个棋盘

    for idx, x in enumerate(board):

        # 空的

        if x == False:

            # 遍历六个方向

            for i in range(6):

                # 有邻居的邻居且有邻居并且都有棒

                if NEINEI[idx][i] != -1 and NEI[idx][i] != 1 and board[NEINEI[idx][i]] == True and board[NEI[idx][i]] == True:

                    board[NEINEI[idx][i]] = False

                    board[NEI[idx][i]] = False

                    board[idx] = True

                    # 记录当前棋盘状态

                    ans.append(board[0:15])

                    func(count + 1)

                    # 退出当前棋盘状态

                    ans.pop()

                    # 回溯

                    board[NEINEI[idx][i]] = True

                    board[NEI[idx][i]] = True

                    board[idx] = False

func(0)

1. 搜索结果如下

1

1 1

1 1 0

1 1 0 1

1 1 1 1 1

1

1 1

1 1 1

1 1 0 0

1 1 1 1 0

1

1 1

1 1 1

0 0 1 0

1 1 1 1 0

1

0 1

0 1 1

1 0 1 0

1 1 1 1 0

1

0 1

1 0 0

1 0 1 0

1 1 1 1 0

1

1 1

0 0 0

0 0 1 0

1 1 1 1 0

1

1 1

0 1 0

0 0 0 0

1 1 1 0 0

0

1 0

0 1 1

0 0 0 0

1 1 1 0 0

0

0 0

0 0 1

0 0 1 0

1 1 1 0 0

0

0 0

0 0 1

0 0 1 0

1 0 0 1 0

0

0 0

0 0 0

0 0 0 0

1 0 1 1 0

0

0 0

0 0 0

0 0 0 0

1 1 0 0 0

0

0 0

0 0 0

0 0 0 0

0 0 1 0 0

1. 经过代码统计发现：若将棋盘位置从0-14编号，从上到下、从左到右，则最后一个插棒只会回到编号为0、4、6、9、12几个位置，并且回到原来位置的方法数最多。

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;

// 判断是否为合适的起点，周围有两条路以上的处于中间，不适合为起点

bool isStart(vector<vector<int>>& grid, int x, int y) {

int cnt = 0;

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

int new\_x = x + dx[i];

int new\_y = y + dy[i];

if (new\_x < 0 || new\_y < 0 || new\_x >= m || new\_y >= n || grid[new\_x][new\_y] == 0)

continue;

++cnt;

}

return cnt <= 2;

}

// 递归函数

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

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) {

// 判断是否为合适的起点

if (grid[i][j] && isStart(grid, i, 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，检查是否为合适的起点。若合适，则进行开采，即进入递归函数。

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

