算法分析第五次作业

Frog-leap 问题

答：

只有在空位左右各两只的总共四只青蛙能够进行跳跃操作。因此解空间是一个四叉树。

剪枝：不向背对的方向跳；不跳到已经被占据的位置 or 跳出界限；如果后续无法移动，停止当前分支。

终止条件：左右各三只青蛙都已交换位置。

1. 从根结点出发，采用深度优先搜索策略。

2. 选择一个子结点，代表满足跳跃条件的一只青蛙跳跃。

3. 更新状态，跳数 Jump 加 1。

4.如果当前跳数 Jump 已经超过先前计算的最小跳数 minJump ，剪枝。

5. 递归选择当前子结点的子结点，以此类推，直到状态变成目标状态记录最小跳数 minJump 或无法再完成跳跃时终止。

5-6 题答案：

答：

1.

x 表示前缀的点， π 表示除前缀外的点，则总费用：

$$cost = \sum_{j=2}^{i} a(xj-1, xj) + a\big(xi, \pi(i+1)\big) + \sum_{j=i+1}^{n} a(\pi(j), \pi(j \bmod n + 1))$$

其中 j mod n+1 是为了处理回路的闭合情况。所以：

$$cost \geq \sum_{j=2}^{i} a(x_{j-1}, x_j) + \min(x_i) + \sum_{j=i+1}^{n} \min \pi(j) = \sum_{j=2}^{i} a(x_{j-1}, x_j) + \sum_{j=i}^{n} \min(x_j)$$

2.

设计上界函数为:

$$u = \sum_{j=2}^{i} a(xj-1, xj) + \sum_{j=i}^{n} \min\ (\mathrm{xj})$$

如果搜索到u（当前费用加上剩下所有出发的最小边之和）比 mincost 还要大，则剪枝。

算法相比教材需要先计算每个顶点出发的最小边费用。

算法实现题

5-1 题答案：

答：

递归回溯开启：调用 backtrack 函数启动递归回溯。

终止条件：在 backtrack 中，若 target 为 0，将 state 数组存到 res，记录子集大小到 resColSizes，然后返回结束当前递归。

遍历与剪枝：从 start 遍历 choices 数组（即排序后的 nums）。和超目标剪枝：若 target - choices[i] < 0，跳过 choices[i]。避免重复剪枝：从 start 遍历防重复子集和重复选同一元素；i > start 且 choices[i] == choices[i - 1]时，跳过 choices[i]。

选择、递归与撤销：不满足剪枝时，把 choices[i]加入 state，更新 stateSize、target 和 start，递归调用 backtrack。递归返回后，stateSize 减 1 撤销选择，尝试其他组合。

```
/* 回溯算法：子集和 II */
void backtrack(int target, int *choices, int choicesSize, int start)
{
    // 子集和等于 target 时，记录解
    if (target == 0) {
        for (int i = 0; i < stateSize; i++) {
            res[resSize][i] = state[i];
        }
        resColSizes[resSize++] = stateSize;
        return;
    }
    // 遍历所有选择
    // 剪枝二：从 start 开始遍历，避免生成重复子集
    // 剪枝三：从 start 开始遍历，避免重复选择同一元素
    for (int i = start; i < choicesSize; i++) {
        // 剪枝一：若子集和超过 target ，则直接跳过
        if (target - choices[i] < 0) {
            continue;
        }
        // 剪枝四：如果该元素与左边元素相等，说明该搜索分支重复，直接跳过
        if (i > start && choices[i] == choices[i - 1]) {
            continue;
```

```
        }
        // 尝试：做出选择，更新 target, start
        state[stateSize] = choices[i];
        stateSize++;
        // 进行下一轮选择
        backtrack(target - choices[i], choices, choicesSize, i + 1);
        // 回退：撤销选择，恢复到之前的状态
        stateSize--;
    }
}

/* 求解子集和 II */
void subsetSumII(int *nums, int numsSize, int target) {
    // 对 nums 进行排序
    qsort(nums, numsSize, sizeof(int), cmp);
    // 开始回溯
    backtrack(target, nums, numsSize, 0);
}
```

5-3 题答案：

答：

采用回溯法：每个商品可以从 m 个供货商获得，则问题的解空间树是一棵 m 叉树，该树属于子集树而非排列树。

剪枝条件：此问题中的限制条件为价格上限 c，最小重量设 bestw，当前价格 cc<c 且当前重量 cw<bestw 时，执行树的深度遍历，else 执行回溯。

更新最优值以及解：最优解是 bestw，解集合是 x[i]。递归出口处更新一种最优解，以及对应解集合。

代码：

```
// 定义最大部件数
#define MAXN 110
// 回溯函数
function Backtrack(int t) {
    if (t > n) {
        bestw = cw;
        for (int i = 1; i <= n; i++) {
            bestx[i] = x[i];
        }
    } else {
        for (int i = 1; i <= m; i++) {
            if (cc + c[t][i] <= d && cw + w[t][i] < bestw) {
                x[t] = i;
                cc = cc + c[t][i];
                cw = cw + w[t][i];
```

```
                Backtrack(t + 1);
                cc = cc - c[t][i];
                cw = cw - w[t][i];
            }
        }
    }
}
function main() {
    read(n, m, d);
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            read(w[i][j]);
        }
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            read(c[i][j]);
        }
    }
    // 调用回溯函数
    Backtrack(1);
}
```

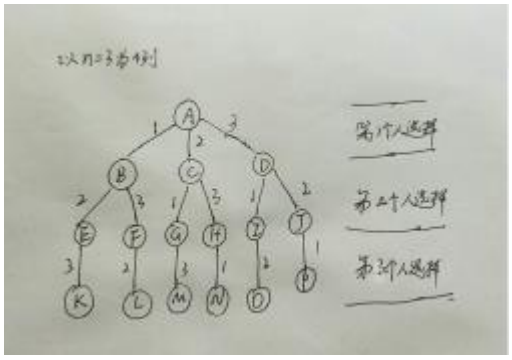
5-13 题答案：

答：

1.解空间

解空间为{x1,x2,x3,……,xn}，其中 xi=1,2,3,4•••,n，表示第 i 个人安排的工作号。

2.解空间树：以 n=3 为例，解空间树如下：



3.剪枝

算法中设置了一个数组，用来存放工作的完成状态，当工作完成时设为 1，否则设为 0。

```
// 回溯函数
void Backstrack(int i, int c) {
```

```
    if (c > scost) {
        return;
    }
    if (i == n) { // 当最后一个人也分配好工作后判断总费用
        if (c < scost) {
            scost = c;
        }
        return;
    }
    for (int j = 0; j < n; j++) {
        if (isC[j] == 0) { // 判断第 j 个工作是否已经完成
            isC[j] = 1;
            Backstrack(i + 1, c + cost[i][j]);
            isC[j] = 0; // 回溯法
        }
    }
}
```

5-16 题答案：

答：

定义问题的解空间。是所有可能的运算顺序。对于表达式 3+4*5-6，可能的运算顺序包括 3+4、3+4*5、3+4*5-6 等。

定义约束条件：保证运算的顺序是正确的，例如先乘后加，先括号内再括号外等等。

定义搜索方式。深度优先搜索。在搜索时，我们需要记录已经搜索过的运算顺序，避免重复搜索。搜索时需要从左往右依次进行运算，当发现一个括号时，需要递归地进行搜索，直到括号内的运算结束。

定义解的处理方式。找到一个解时，将其记录。当发现当前搜索路径不符合约束条件时需要进行回溯，回溯到上一个符合约束条件的搜索路径。

代码：

```
// 检查当前的数字和运算符组合是否能得到目标数字 m
int found() {
    int x = num[0];
    for (int i = 0; i < k; i++) {
        switch (oper[i]) {
            case 0:
                x += num[i + 1];
                break;
            case 1:
                x -= num[i + 1];
                break;
            case 2:
                x *= num[i + 1];
```

```
                break;
            case 3:
                if (num[i + 1] != 0) {
                    x /= num[i + 1];
                } else {
                    return 0;
                }
                break;
        }
    }
    return (x == m);
}

// 递归搜索可能的数字和运算符组合
int search(int dep) {
    if (dep > k) {
        if (found()) {
            return 1;
        } else {
            return 0;
        }
    }
    for (int i = 0; i < n; i++) {
        if (flag[i] == 0) {
            num[dep] = a[i];
            flag[i] = 1;
            for (int j = 0; j < 4; j++) {
                oper[dep] = j;
                if (search(dep + 1)) {
                    return 1;
                }
            }
            flag[i] = 0;
        }
    }
    return 0;
}
```

5-20 题答案：

答：

是求解最大独立集的问题。将部落中两两之间的敌对关系看为在一个无向图中两个点的连线，没有连线的两个点即是没有敌对关系的两个人。

用树表示解空间。设当前扩展节点 Z 位于解空间的第 i 层。在进入左子树前，必须确认从顶点 i 到已经选入的顶点集中每一个顶点都没有边连接他们任何两个， 在进入右子树之前，必须确认有足够多的而可选择定点使得算法有可能在右子树找到更大的独立集。

部落人数 n，敌对关系的对数 m，做有 n 个顶点的图，顶点之间的连线关系按照 m 个敌对关系来确定，有敌对关系的两个部落战士，就把他们连起来，这样问题就转化为了寻找这有 n 个顶点无向图最大独立集。我们用邻接矩阵表示图 G。整型数组 v 返回所找到的最大的独立集。v[i]=1 当且仅当顶点 i 属于找到的最大独立集。

```
function backTrack(int i) {
    if (i > n - 1) {
        for (int b = 0; b < n; b++) {
            bestx[b] = x[b];
        }
        bestn = cn;
        return;
    }
    // 临界条件：与任一已入选护卫队的队员无仇
    boolean ok = true;
    for (int j = 0; j < i; j++) {
        if (x[j] == 1 && a[i][j] == 1) {
            ok = false;
            break;
        }
    }
    if (ok) {
        x[i] = 1;
        cn++;
        backTrack(i + 1);
```

```
            cn--;
        }
        if ((cn + (n - i - 1)) > bestn) {
            x[i] = 0;
            backTrack(i + 1);
        }
    }
    // 主函数
    function main() {
        read(n);
        cn = 0;
        bestn = 0;
        int d;
        read(d);
        int m[MAX_D][2];
        for (int i = 0; i < d; i++) {
            read(m[i][0]);
            read(m[i][1]);
        }
        // 构造邻接矩阵
        for (int i = 0; i < d; i++) {
            a[m[i][0] - 1][m[i][1] - 1] = 1;
            a[m[i][1] - 1][m[i][0] - 1] = 1;
        }
        // 调用回溯函数
        backTrack(0);
    }
```