算法专题：回溯和分支定界

前言：

解决问题的时候，可以列出所有的**候选解**，然后依次检查每一个，检查完后就可以得到所需要的解。

对候选解进行系统检查的常用两种方法是**回溯**和**分支定界**。

推荐文章：回溯法算法复习伪码—自用-CSDN博客

回溯：

1.定义：

回溯是一种系统的搜索问题解答的方法。

首先定义一个**解空间**，下一步组织解空间以便于它能容易的被搜索（树或者图），然后再按照**深度优先**的方法进行搜索。

从开始结点开始，深搜，如果某节点无法移动到 一个新节点，该结点就变为死节点，当找到答案或者回溯尽了所有的活结点时，搜索结束。

回溯通俗来讲指的是一条路走不通了再往回走的意思，应为为**traceback**。

注：

1. 优化问题和存在性问题
2. 排列树和子集树

2.解题思想

在对解空间进行搜索时，盲目搜索的时间和空间是很大的。因此产生两种回溯算法：

1）使用限界函数**阻止**不可能获得解答的结点的**扩张**。

2）通过**不移动**到不可能包含比当前最优解还要好的解的右子树。

3)

- 任何搜索算法都可以用建立在**解空间上的状态搜索树**加以描述。
- 状态搜索树是我们**尝试选择元组的各个分量时产生的树结构。**
- 搜索算法并非事先将状态空间树存在计算机内再进行遍历，而是**通过展开状态空间树来找所求的解。**
- 展开的过程中使用**启发式**的限界方法（减去状态空间树上的某些分支）使搜索算法**只展开状态空间树的一部分**，从而降低搜索算法的时间和空间复杂度。

4）**有关状态空间树**：

- 每个搜索算法都在系统的展开状态空间树。

- 状态空间树的几种结点的描述：

活结点：已展开了部分子节点，但所有子节点尚未全部展开的结点。

死结点：被限界或已展开了所有子节点的结点。

E-结点：当前正在展开子节点的活结点。

5）状态空间树的展开方法：

- 深度优先展开方法。
- 回溯法（加上限界的深度优先展开）。
- 分支限界法：结点变成E节点后，展开所有子节点，自己变成死结点。同时需要一个结构维持已展开但是还没成为E结点的那些结点。

3.应用：

*0/1背包问题

这个背包问题比货箱装船的第一艘船的承载量的最大思想严谨多了。

这里是采用一种对于未来的预测，也就是保存bestcw，同时对未来将要遍历的节点做出判断，假设背包中的物品可划分时会造成的最好最好结果（也就是先将所有密度最大的装入），如果这个结果比bestcw要小，那完全没有继续遍历该子树的必要。

因此可以得出1）需要将背包中的物品按照密度从大到小排序，便于计算。2）每个节点最重要的要保存的数值是当前获益cp（当前最好容量），未来可能的最大容量bound（剩余容量）从而画出递归树进行判断。

伪代码如下：

```
    按照密度对物品排序
    bestp <- -∞
    设x=(x(1),x(1),x(3).....x(k))为当前E节点(bound>bestp成立)

    展开左子节点
    if cw+w(k) <= c
        装入物品k
         cw <- cw+w(k),cp <- cp+p(k)
         x(k)=1
    else 展开右子节点

    if(bound<=bestp)停止产生右子树
    else x(k) <- 0,并令(x(1),x(1),x(3).....x(k))为E节点
    //类似货船装箱问题
```

C++代码：

```
#include<bits/stdc++.h>
using namespace std;

struct Item {
    int w;//重量
    int v;//价值
```

```cpp
    double density;//密度
    int index;
};

//仿函数，按照密度从大到小排序
bool compare(Item a, Item b) {
    return a.density > b.density;
}

vector<int> bestItems; // 用于保存最佳物品组合的向量

int knapsack(vector<Item>& items, int c) {
    sort(items.begin(), items.end(), compare);

    int bestv = 0;
    vector<int> currentItems(items.size(), 0); // 用于保存当前物品组合的向量

    function<void(int, int, int)> backtrack = [&](int i, int cw, int cv) {
        //递归出口
        if (i >= (int)items.size()) {
            if (cv > bestv) {
                bestv = cv;
                bestItems = currentItems; // 当找到更好的解决方案时，更新最佳物品组合
            }
            return;
        }

        if (cw + items[i].w <= c) {
            currentItems[i] = items[i].index; // 添加当前物品到当前物品组合
            backtrack(i + 1, cw + items[i].w, cv + items[i].v);
        }

        if (cv + (c - cw) * items[i].density > bestv) {
            currentItems.erase(remove(currentItems.begin(), currentItems.end(), items[i].index), currentItems.end());
            backtrack(i + 1, cw, cv);
        }
    };

    backtrack(0, 0, 0);
    return bestv;
}

int main() {
    vector<Item> items = {{23, 92, 92 / 23, 0}, {31, 57, 57 / 31, 1}, {29, 49, 49 / 29, 2}, {44, 68, 68 / 44, 3}, {53, 60, 60 / 23, 4}, {38, 43, 43 / 38, 5}, {63, 67, 67 / 63, 6}, {85, 84, 84 / 85, 7}, {89, 87, 87 / 89, 8}, {82, 72, 72 / 82, 9}};
    int c = 165;
    cout << knapsack(items, c) << endl;

    // 打印最佳物品组合
    for (int i = 0; i < (int)items.size(); i++) {
        if (find(bestItems.begin(), bestItems.end(), i) != bestItems.end())
            cout << 1;
```

```
        else
            cout << 0;
    }

    cout << endl;

    return 0;
}
```

*货箱装船问题

> 有n个货箱，w数组中有各个货箱的重量，有两艘船，两艘船的承载量用c1，c2表示。请问是否有一种可以将n个货箱全部装入的方案，若有请找出该方案。

分析：由于每个货箱有装和不装两种状态，如果采用此种方法则复杂度到了指数级别，所以只能采取其它方案。

思路一：由于这个问题和01背包很类似，可以先采用动态规划尽可能的装满第一艘船，再判断剩下是否能全部装进第二艘船即可。元组法时间复杂度为O(min{c1,2^n});

思路二：当然，这个专题探讨的是回溯，我们来看看回溯思想的解法。

第一种回溯算法

从上到下遍历解空间，同时阻止现有容量超过了节点的扩张。

代码如下：

```
template<class T>
class Loading {
        friend MaxLoading(T [], T, int);
    private:
        void maxLoading(int i);
        int n;//货箱数目
        T *w;//货箱重量数组
        T c, cw, bestcw; //第一艘船的容量，当前容量，目前装载最优的容量。
};

template<class T>
void Loading<T>::maxLoading(int i) {
    //在第i层时
    if (i > n) {
        //处于叶子节点
        //bestcw是前面的叶子节点得到的最大值
        if (cw > bestcw)bestcw = cw;
        return;
    }

    //检查子树
    //当加上这个节点时不会超过最大，那加不加都可以
    /****显示算法思想的步骤****/
    if (cw + w[i] <= c) {
        cw += w[i]; //加当前节点时
```

```
        maxLoading(i + 1);
        cw -= w[i]; //不加当前节点时
        maxLoading(i + 1);
    }//（感慨，递归真好用）
}

template<class T>
T MaxLoading(T w[], T c, int n) {
    Loading<T> X;
    X.w = w, X.c = c, X.n = n, X.cw = 0, X.bestcw = 0; //初始化

    X.maxLoading(1);
    return X.bestcw;
}
```

第二种回溯算法

对第一种进行优化，加上不移动到不包含比当前最优解还好的解的右子树。

由于是广度搜索，该优化主要发生在i层和i-1层，在i层找到第一个bestcw时，后面的如果解比它差就不需要再进行下去；

同时设置r为剩余货箱容量，如果当前加上剩余仍然比不上最好，那也没必要遍历该子树，关键部分代码修改如下：

```
template<class T>
void Loading<T>::maxLoading(int i) {
    //在第i层时
    if (i > n) {
        //处于叶子节点
        //bestcw是前面的叶子节点得到的最大值
        if (cw > bestcw)bestcw = cw;
        return;
    }

    //检查子树
    r -= w[i];

    //当加上这个节点时不会超过最大，那加不加都可以
    if (cw + w[i] <= c) {
        cw += w[i]; //加当前节点时
        maxLoading(i + 1);
        cw -= w[i]; //不加当前节点时
    }//（感慨，递归真好用）
    //要判断不加当前节点那你以后的最好值能不能大于当前最优解
    if (cw + r > bestcw)maxLoading(i + 1);
    r += w[i];
}
```

当然还可以再进一步优化添加数组保存可行时的路径，代码就不更新了。

*子集和数问题

> 已知n+1个正数，wi（1<=i<=n）和M，要求找出w的所有子集，使子集的元素之和等于M。

思考：0/1背包问题？？？就是要把背包装满。但这个题目说用回溯，那策略也不太能和01背包的回溯套用，毕竟在回溯上解题点不同，但是要是动态规划应该是可以套用着试试。

n限界函数的一种简单选择是:

**当且仅 $\sum_{i=1}^{k} W(i)X(i) + \sum_{i=k+1}^{n} W(i) < M$ 时, B(X(1),...,X(k))=true, 并停止展开节点(X(1),...,X(k)).

**已将W(i)按非降次序排列, 则可以进一步限界如下: 如果 $\sum_{i=1}^{k} W(i)X(i) \neq M \quad and \quad \sum_{i=1}^{k} W(i)X(i) + W(k+1) > M$ 则继续展开X(1),...,X(k)不可能得到答案结点.

**限界函数B(X(1),...,X(k))=ture当且仅当 $\sum_{i=1}^{k} W(i)X(i) + \sum_{i=k+1}^{n} W(i) < M$

$\sum_{i=1}^{k} W(i)X(i) + W(k+1) > M$ 停止产生子节点(X(1),...,X(k))及其子树.

```
令S=w(1)X(1)+…+w(k-1)X(k-1)
     r=w(k)+…+w(n),假定S+r≥M(不满足该条件的节点已被限界掉)
如果S=M，则找到了一个和数为M的子集．回溯找其它解．
展开左子节点：
如果S+W(k)+W(k+1)>M则停止展开左子节点，
    r←r-w(k)，并展开右子节点；
否则,X(k)←1 ，
             S←S+w(k)，
             r←r-w(k)，k←k+1 (令(x(1),…,x(k))为E节点)；
展开右子节点：
如果S+r<M或S+w(k+1)>M则停止展开右子节点并回溯；
否则,X(k)←0，
             r←r-w(k),k←k+1 (令(x(1),…,x(k))为E节点)；
回溯：k ←k-1  (回到父节点)．
```

*n皇后问题

> n皇后问题的要求在于要求每行每列都要有一个皇后，并且这些皇后不能在同一直线或者同一斜线上。

我们尝试画一下它的状态空间树，

要时刻记住，回溯的关键在于减少递归树的分支（无论是阻止扩容还是不移动到无法超过最优解，这些只是主要的思想手段），因此在化状态空间树时，有以下几种策略可以减少递归树的分支：

~~有n行n列，因此具有对称性，每行不需要遍历所有格子，选取其中的一半即可。

~~违反题目要求的，比如不在同行同列，不能往下展开。

```cpp
//递归回溯
void Queen::Backtrack(int t)
{
    if(t>n) sum++;
    else
        for(int i=1;i<=n;i++){
            x[t]=i;//第t行放在第i列
            if(place(t))Backtrack(t+1);
    }
}

bool Queen::place(int k)
{
    for(int j=1;j<k;j++)
        if(abs(k-j)==abs(x[j]-x[k])||x[j]==x[k])//不能在同一列同斜线
            return false;
    return true;
}
```

```cpp
//迭代回溯
void Queen::Backtrack()
{
    x[1]=0;
    int k=1;
    while(k>0){
        x[k]+=1;//第k行的放到下一列。
        while(x[k]<=n&&!place(k))x[k]+=1;//x[k]不能放置，则放到下一列，直到可以放置

        if(x[k]<=n)//放在n列范围内
            if(k==n)sum++; else k++,x[k]=0;
        else
            k--;//第k行无法放置，回溯
    }
}
```

*旅行商问题

> 给定一个n 节点的网络, 称一条包含网络中n 个节点的环路为一条周游路线.
>
> 旅行商问题要求找出一条最小成本的周游路线.

如果先画出它的状态分布树的话，这个树是很奇怪的，如果是4个节点，原本的六条路径在树中被重复了很多次。这个树也是被称为**置换树。**

限界条件：

限界(1)：**剪去不可行。**i-1级节点x[i-1]和它的子节点x[i]之间有边相连(!=NoEdge)

限界(2)：**剪去非最优。**设bestc为当前得到的最优解的成本值；路径x[1:i]的长度<bestc

则搜索继续向下进行；否则施行限界。

因此就可以减少状态分布数的时间和空间复杂度。

```cpp
template<class T>
T AdjacencyWDigraph<T>::TSP(int V[]) {
    //用回溯算法解决旅行商问题
    //返回最优旅游路径的耗费，最优路径存 入v[1:n]
    //初始化
    x = new int [n + 1]; //x是排列
    for (int i = 1; i <= n; i++)
        x[i] = i;
    bestc = NoEdge;
    bestx = v; //使用数组v来存储最优路径
    cc = 0;
    //搜索x[2:n]的各种排列
    tSP(2);
    delete [] x;
    return bestc;
}

void AdjacencyWDigraph<T>::tSP(int i) {
    //旅行商问题的回溯算法
    if (i == n)  //位于一个叶子的父节点,通过增加两条边来完成旅行
        if (a[x[n - 1]][x[n]] = NoEdge && a[x[n]][1] != NoEdge && (cc + a[x[n - 1]][x[n]] + a{x[n]][1] < bestc || bestc == NoEdge)) {
            //找到更优的旅行路径
            for (int j = 1; j <= n; j++)
                bestx[j] = x[j];
            bestc = cc + a[x[n - 1]][x[n]] + a[x[n]][1];
        }

        //尝试子树
    for (int j = i; j <= n; j++)
    //能移动到子树x[j]吗?
        if (a[x[i - 1]][x[j]] != NoEdge && (cc + a[x[i - 1]][x[i]] < bestc || bestc == NoEdge)) {
            //能搜索该子树
            Swap(x[i], x[j]);
            cc += a[x[i - 1]][x[i]];
            tSP(i + 1);
            cc -= a[x[i - 1]][x[i]];
            Swap(x[i], x[j]);
        }
}
```

*最大完备子集

> - 完全图：图的每个顶点之间都有边
> - 完全子图的尺寸是图U中顶点的数量
> - 如果一个完全子图U不被包含在G的一个更大的完全子图中，称它是图G的一个集团。
> - 最大集团是指具有最大尺寸的集团，也称最大团。
> - U是图G的最大团同时也是补图G0的最大独立集

最大团问题属于优化问题，也是子集树。

解题：

限界1：**剪去不可行。**检查根节点到状态空间树的某一状态节点的路径上对应的顶点子集是否构成一个完全子图。

限界2：**剪去非最优。**如果剩余未考虑的顶点数加上团中顶点数不大于当前解的顶点数，则不再展开。

（好普通的回溯。。。）

```
void AdjacencyGraph::maxClique(int i){
    if(i>n){//找到一个完备子集
        for(int j=1;j<=n;j++)bestx[j]=x[j];
        bestn=cn;
        return;
    }

    int OK=1;
    for(int j=0;j<i;j++)//看是否与前面所有点都有边
        if(x[j]&&a[i][j]==noEdge)
            OK=0,break;

    if(OK){
        x[i]=1;
        cn++;
        maxClique(i+1);
        x[i]=0;
        cn--;
    }//有边就加上去

    if(cn+n-i>bestn){
        x[i]=0;//没有边就要考虑是否满足限界2
        maxClique(i+1);
    }
}
```

4.回溯的一般方法（伪代码）

算法说明：（回溯=状态树+限定条件）

每个解用数组 X(1:n)来表示.

假定 X(1),...,X(k-1)的值已确定, T(X(1),...,X(k- 1))代表x(k)的所有可能的取值.

限界函数B(X(1),..., X(k))判断那些X(k)的取值不能导致问题的解,从而停止展开该子节点

```
    BACKTRACK:
        k<---1
        while k>0 do
         {
            if (X(1),…,X(k-1))还有没展开的子节点
               {取X(k)属于T(X(1),…,X(k-1));
                  if B(X(1),…,X(k))=false
                      {if (X(1),…,X(k))是一答案节点
                                      则输出 (X(1),…,X(k));
```

```
                        k<---k+1}//继续展开子节点//
            else k<---k-1 //回溯到父节点//
          }

    RBACKTRACK(k)
    //进入该子程序时解X(1: n)的前k-1个分量
    X(1),…,X(k-1)已取值//
    for X(k)属于T(X(1),…,X(k-1)) and
    B(X(1),…,X(k))=false
    {
    if (X(1),…,X(k))是答案节点{
          输出(X(1),…,X(k));
           RBACKTRACK(k+1)}
    }
```

分支定界：

1.定义

分支定界同样是一种系统的搜索解空间的办法。

二者关于搜索上思想的不同可利用数据结构中的广度优先搜索和深度优先搜索区分。回溯遍历结点的时候是到了叶子节点或者不能扩张的结点再往回走，分支定界会将该层所有的点放入队列中，然后依次将当层的点遍历完，这就是广度的来源。

2.解题思想

为了实现我们分支定界的思想，常有以下两种思路：
1）先进先出（FIFO）。即从活结点表中取出节点的顺序和加入节点的顺序相同，因此活节点表的性质和队列相同。

2）最大耗费或者最大收益法。在该模式中，每个节点都有一个对应的耗费或者收益，如果查找一个具有最少耗费的解，活结点表可以用最小堆建立，下一个E节点就是具有最小耗费的活结点；如果是搜索一个有最大收益的解，则可用最大堆建立思路同理。（第二种的表述有点抽象，后面再结合具体题目看看。）

LC-检索

如果活节点表中每个节点以c(x)为权值,每次从活节点表中取出最小权值节点作为E-节点,则算法能很快找到优化解.

但在展开x 前不可能知道c(x)的值. 但是有可能从历史信息获得c(x)的某一下界ĉ(x).

以c(x)的下界估值ĉ(x)做为活节点表中节点的权值, 每次取出有最小ĉ(x)的节点进行展开;

要求设计的ĉ(x)满足: ĉ(x)=cost(x),当x为可行**叶节点**时;

LC-分支-限界算法

```
E=T; U←∞;
置活节点表为空;
while(true)
{
    for E 的每个子节点x
    If x 是叶节点 then U←min{U,cost(x)};
    if ĉ(x)<U  then {Add(x), parent(x)=E;}
    If 活节点表空 then 算法结束;
    delete(E);
    if ĉ(E)≥U 算法结束;
}
```

基于优先级队列的分枝-限界算法

```
  E=T;
  置活节点表为空;
  while(true)
{
  if E是可行叶节点 return;
  for E 的每个子节点x
       {Add(x); parent(x)=E}
       delete(E);
}

/*算法正确性证明:
当算法结束在第4行时 ĉ(E)=c(E);
对活节点表中每个节点x, 都有:  ĉ(x)≥ĉ(E)=c(E)
但c(x)≥ĉ(x)≥c(E), 且c(x)为状态空间树上以x为根的子树的最小成本值.
又因为活节点表覆盖了状态空间树所有叶节点, 所以c(E)是最小成本值, 而E是最优解.*/
```

3.应用:

（只要理解思想就差不了多少，应用例子就不举这么多了。）

*0/1背包问题

（子集树）

限界条件:

1）选择该物品后的容量不会超过背包容量。

2）由于如果想确定每个节点收益时，不到叶子节点是无法确定最终收益，不能随便去掉子树。但如果换个角度思考，当我有最大收益时有的损失是最少的，（损失也就是指不选择某个物品时的损失），当损失确定时，如果到达某个节点是不选择的物品过多，损失过大，那么一定无法超过当前的最大收益。因此引入当前最小损失这个变量。

*最小罚款额作业调度

> n个作业,1台处理机,每个作业i对应一个三元组(pi,di,ti)。
>
> pi - 罚款额； di - 截止期； ti - 需要的处理机时间。
>
> 求可行的作业子集J,使得罚款额Σpj最小,其中j为不在J中的作业.
>
> 假设给定4个作业为（5，1，1），（10，3，2），（6，2，1），（3，1，1）。

该题思路很灵活。

先用子集树 ：

由于每个作业选择的先后会造成是否有罚款额两种结果，所以我们认为先选的是不产生罚款额的。

分支限界重要的在于两点：

1）状态空间树。——子集树

2）限界条件。

限界条件一般包括两个，1是题干中给出的，也就是在已经有选择的情况下我再选择这个作业如果不产生罚款额就可以是选择，否则只能是不选择。2是隐形的，由于不选择某项会产生罚款额，因此记录每个节点的罚款额，如果罚款额已经比当前最大罚款额还要大时，就直接出队列。

Consequently，该题思路为：

从第一个物品选择与否开始，**宽度优先搜索**，将符合限定条件1也就是选择可以**不产生罚款额的节点**和不选择时的节点加入优先队列中，**优先队列**中节点的排序是按照罚款额由小到大的，按照以上思路持续遍历，当得到一个当前罚款额时，可以经过比较去掉队列中的某些节点，直到所有节点被遍历完，结束。

排列树：

这个排列并不是列出作业做的先后顺序，而是是否能完成做这个事情并且不产生罚款额，和子集树中限界条件1的理念是一致的。

限界2就是当前情况的罚款额不能比当前最小罚款额还要大了。

*旅行商问题

> 该类问题的实质是在一个带权完全无向图中，找一个权值最小的哈密顿回路。

贴个链接回溯法和分支限界法 | Xinhecuican's Blog，这篇文章讲的很好理解。

好的我不写了，各位看链接，链接里面是关于规约矩阵的做法。然后利用规约矩阵得到一个状态空间树。

规约矩阵：
行规约矩阵就是每行减去每行最小的数，列规约矩阵就是每列减去那一列最小的数。

当矩阵的每行每列都有0时就得到了原矩阵的规约矩阵。原矩阵每行每列减去的值的绝对值加起来的和就是归约数。

得到原矩阵的规约矩阵之后，比较出发点到其余各点的距离（由出发点到点的距离+原矩阵变成规约矩阵的规约数+去掉出发点行，到达点列，（到达点，出发点）位置的数变为无穷后的矩阵变成规约矩阵的规约数），得到到达某个点最小的距离。

然后该点变成出发点，矩阵降一维重复以上的步骤。

所有重复后从上到下也可以形成一个状态空间树。

附：

1.内容参考

佟鑫宇老师 天津大学智能与计算学部 2023秋 算法设计与分析 ppt

2.

考试考察重点：

回溯法原理，货箱装船问题，0/1背包问题，旅行商问题

分支限界法活结点扩充法，旅行商问题。

3.

全文同样包含个人的主观理解，如有错误，欢迎访问原文链接指正。