第 5 章 回溯法

回溯法有“通用解题法”之称。用它可以系统地搜索问题的所有解。回溯法是一个既带有系统性又带有跳跃性的搜索算法。它在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间树。算法搜索至解空间树的任一结点时，先判断该结点是否包含问题的解。如果肯定不包含，则跳过对以该结点为根的子树的搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索。回溯法求问题的所有解时，要回溯到根，且根结点的所有子树都被搜索遍才结束。回溯法求问题的一个解时，只要搜索到问题的一个解就可结束。这种以深度优先方式系统搜索问题解的算法称为回溯法，它适用于求解组合数较大的问题。

5.1 回溯法的算法框架

5.1.1 问题的解空间

用回溯法解问题时，应明确定义问题的解空间。问题的解空间至少应包含问题的一个(最优)解。例如，对于有 n 种可选择物品的 0-1 背包问题，其解空间由长度为 n 的 0-1 向量组成。该解空间包含对变量的所有 0-1 赋值。当 $n=3$ 时，其解空间是：

{(0,0,0),(0,1,0),(0,0,1),(1,0,0),(0,1,1),(1,0,1),(1,1,0),(1,1,1)}

定义了问题的解空间后，还应将解空间很好地组织起来，使得能用回溯法方便地搜索整个解空间。通常将解空间组织成树或图的形式。

例如，对于 $n=3$ 时的 0-1 背包问题，可用完全二叉树表示其解空间，如图 5-1 所示。

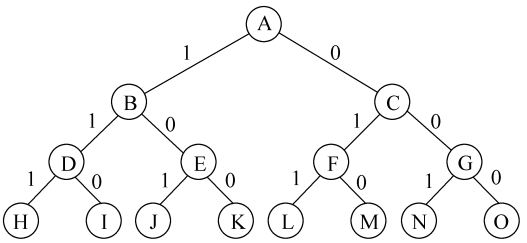

图 5-1 0-1 背包问题的解空间树

解空间树的第 i 层到第 $i+1$ 层边上的标号给出了变量的值。从树根到叶的任一路径表示解空间中的一个元素。例如，从根结点到结点 H 的路径相应于解空间中元素(1,1,1)。

5.1.2 回溯法的基本思想

确定了解空间的组织结构后,回溯法从开始结点(根结点)出发,以深度优先方式搜索整个解空间。这个开始结点成为活结点,同时也成为当前的扩展结点。在当前扩展结点处,搜索向纵深方向移至一个新结点。这个新结点成为新的活结点,并成为当前扩展结点。如果在当前扩展结点处不能再向纵深方向移动,则当前扩展结点就成为死结点。此时,应往回移动(回溯)至最近的活结点处,并使这个活结点成为当前扩展结点。回溯法以这种工作方式递归地在解空间中搜索,直至找到所要求的解或解空间中已无活结点时为止。

例如,对于 $n=3$ 时的 0-1 背包问题,考虑下面的具体实例:$w=[16,15,15]$,$p=[45,25,25]$,$c=30$。从图 5-1 的根结点开始搜索解空间。开始时,根结点是唯一的活结点,也是当前扩展结点。在这个扩展结点处,可以沿纵深方向移至结点 B 或结点 C。假设选择先移至结点 B。此时,结点 A 和结点 B 是活结点,结点 B 成为当前扩展结点。由于选取了 w_1,故在结点 B 处剩余背包容量是 $r=14$,获取的价值为 45。从结点 B 处,可以移至结点 D 或 E。由于移至结点 D 至少需要 $w_2=15$ 的背包容量,而现在仅有的背包容量是 $r=14$,故移至结点 D 导致不可行解。搜索至结点 E 不需要背包容量,因而是可行的。从而选择移至结点 E。此时,E 成为新的扩展结点,结点 A,B 和 E 是活结点。在结点 E 处,$r=14$,获取的价值为 45。从结点 E 处,可以向纵深移至结点 J 或 K。移至结点 J 导致不可行解,而移向结点 K 是可行的,于是移向结点 K,它成为新的扩展结点。由于结点 K 是叶结点,故得到一个可行解。这个解相应的价值为 45。x_i 的取值由根结点到叶结点 K 的路径唯一确定,即 $x=(1,0,0)$。由于在结点 K 处已不能再向纵深扩展,所以结点 K 成为死结点。返回到结点 E 处。此时在结点 E 处也没有可扩展的结点,它也成为死结点。

接下来又返回到结点 B 处。结点 B 同样也成为死结点,从而结点 A 再次成为当前扩展结点。结点 A 还可继续扩展,从而到达结点 C。此时,$r=30$,获取的价值为 0。从结点 C 可移向结点 F 或 G。假设移至结点 F,它成为新的扩展结点。结点 A,C 和 F 是活结点。在结点 F 处,$r=15$,获取的价值为 25。从结点 F,向纵深移至结点 L 处,此时,$r=0$,获取的价值为 50。由于 L 是叶结点,而且是迄今为止找到的获取价值最高的可行解,因此记录这个可行解。结点 L 不可扩展,又返回到结点 F 处。按此方式继续搜索,可搜索遍整个解空间。搜索结束后找到的最好解是相应 0-1 背包问题的最优解。

再看一个用回溯法解旅行售货员问题的例子。

旅行售货员问题的提法是:某售货员要到若干城市去推销商品,已知各城市之间的路程(或旅费)。他要选定一条从驻地出发,经过每个城市一次,最后回到驻地的路线,使总的路程(或总旅费)最短(或最小)。

问题刚提出时,不少人都认为这个问题很简单。后来,在实践中才逐步认识到,这个问题只是叙述简单,易于理解,而其计算复杂性却是问题输入规模的指数函数,属于相当难解的问题之一。事实上,它是 NP 完全问题。这个问题可以用图论语言形式描述。

设 $G=(V,E)$是一个带权图。图中各边的费用(权)为正数。图的一条周游路线是包括 V 中的每个顶点在内的一条回路。周游路线的费用是这条路线上所有边的费用之和。旅行售货员问题要在图 G 中找出费用最小的周游路线。

图 5-2 是一个 4 顶点无向带权图。顶点序列 1,2,4,3,1;1,3,2,4,1 和 1,4,3,2,1 是该图中 3 条不同的周游路线。

旅行售货员问题的解空间可以组织成一棵树,从树的根结点到任一叶结点的路径定义了图 G 的一条周游路线。图 5-3 是当 $n=4$ 时解空间的示例。其中,从根结点 A 到叶结点 L 的路径上边的标号组成一条周游路线 1,2,3,4,1。从根结点到叶结点 O 的路径表示周游路线 1,3,4,2,1。图 G 的每一条周游路线都恰好对应于解空间树中一条从根结点到叶结点的路径。因此,解空间树中叶结点个数为$(n-1)!$。

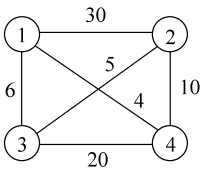

图 5-2 4 顶点带权图

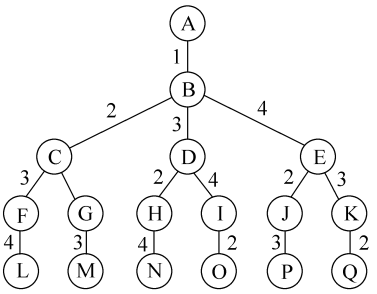

图 5-3 旅行售货员问题的解空间树

对于图 5-2 中的图 G,回溯法找最小费用周游路线时,从解空间树的根结点 A 出发,搜索至 B,C,F,L。在叶结点 L 处记录找到的周游路线 1,2,3,4,1,该周游路线的费用为 59。从叶结点 L 返回至最近活结点 F 处。由于 F 处已没有可扩展结点,算法又返回到结点 C 处。结点 C 成为新扩展结点,由新扩展结点,算法再移至结点 G 后又移至结点 M,得到周游路线 1,2,4,3,1,其费用为 66。这个费用不比已有周游路线 1,2,3,4,1 的费用更小,因此,舍弃该结点。算法又依次返回至结点 G,C,B。从结点 B,算法继续搜索至结点 D,H,N。在叶结点 N 处,相应的周游路线 1,3,2,4,1 的费用为 25,它是当前找到的最好的一条周游路线。从结点 N 算法返回至结点 H,D,然后从结点 D 开始继续向纵深搜索至结点 O。依此方式算法继续搜索遍整个解空间,最终得到最小费用周游路线 1,3,2,4,1。

回溯法搜索解空间树时,通常采用两种策略避免无效搜索,提高回溯法的搜索效率。其一是用约束函数在扩展结点处剪去不满足约束的子树;其二是用限界函数剪去得不到最优解的子树。这两类函数统称为剪枝函数。

例如,解 0-1 背包问题回溯法用剪枝函数剪去导致不可行解的子树。在解旅行售货员问题的回溯法中,如果从根结点到当前扩展结点处的部分周游路线费用已超过当前找到的最好的周游路线费用,则可以断定以该结点为根的子树中不含最优解,因此,可以将该子树剪去。

综上所述,用回溯法解题通常包含以下 3 个步骤:

(1) 针对所给问题,定义问题的解空间。

(2) 确定易于搜索的解空间结构。

(3) 以深度优先方式搜索解空间,并在搜索过程中用剪枝函数避免无效搜索。

5.1.3 递归回溯

回溯法对解空间进行深度优先搜索,因此,在一般情况下可用递归方法实现回溯法。

```
void backtrack (int t)
{
  if (t>n) output(x);
  else
    for (int i=f(n,t);i<=g(n,t);i++)
    {
      x[t]=h(i);
      if (constraint(t)&&bound(t)) backtrack(t+1);
    }
}
```

其中,形式参数 t 表示递归深度,即当前扩展结点在解空间树中的深度。n 用来控制递归深度。当 $t>n$ 时,算法已搜索至叶结点。此时,由 output(x) 记录或输出得到的可行解 x。算法 backtrack 的 for 循环中 $f(n,t)$ 和 $g(n,t)$分别表示在当前扩展结点处未搜索过的子树的起始编号和终止编号。$h(i)$表示在当前扩展结点处 $x[t]$的第 i 个可选值。constraint(t)和 bound(t)是当前扩展结点处的约束函数和限界函数。constraint(t)返回的值为 true 时,在当前扩展结点处 $x[1:t]$取值满足问题的约束条件;否则,不满足问题的约束条件,可剪去相应的子树。bound(t)返回的值为 true 时,在当前扩展结点处 $x[1:t]$取值未使目标函数越界,还需由 backtrack($t+1$)对其相应的子树进一步搜索。否则,当前扩展结点处 $x[1:t]$的取值使目标函数越界,可剪去相应的子树。执行了算法的 for 循环后,已搜索遍当前扩展结点的所有未搜索过的子树。backtrack(t)执行完毕,返回 $t-1$ 层继续执行,对还没有测试过的 $x[t-1]$的值继续搜索。当 $t=1$ 时,若已测试完 $x[1]$的所有可选值,外层调用就全部结束。显然,这一搜索过程按深度优先方式进行。调用一次 backtrack(1)即可完成整个回溯搜索过程。

5.1.4 迭代回溯

采用树的非递归深度优先遍历算法,可将回溯法表示为一个非递归迭代过程。

```
void iterativeBacktrack ()
{
  int t=1;
  while (t>0)
  {
    if (f(n,t)<=g(n,t))
      for (int i=f(n,t);i<=g(n,t);i++)
      {
        x[t]=h(i);
        if (constraint(t)&&bound(t))
        {
          if (solution(t)) output(x);
          else t++;
        }
      }
    else t--;
```

```
    }
}
```

上述迭代回溯算法中，solution(t)判断在当前扩展结点处是否已得到问题的可行解。它返回的值为 true 时，在当前扩展结点处$x[1:t]$是问题的可行解。此时，由 output(x) 记录或输出得到的可行解。它返回的值为 false 时，在当前扩展结点处$x[1:t]$只是问题的部分解，还需向纵深方向继续搜索。算法中 $f(n,t)$和 $g(n,t)$分别表示在当前扩展结点处未搜索过的子树的起始编号和终止编号。$h(i)$表示在当前扩展结点处 $x[t]$的第 i 个可选值。constraint(t)和 bound(t)是当前扩展结点处的约束函数和限界函数。constraint(t)返回的值为 true 时，在当前扩展结点处 $x[1:t]$取值满足问题的约束条件；否则，不满足问题的约束条件，可剪去相应的子树。bound(t)返回的值为 true 时，在当前扩展结点处 $x[1:t]$取值未使目标函数越界，还需对其相应的子树进一步搜索。否则，当前扩展结点处 $x[1:t]$取值使目标函数越界，可剪去相应的子树。算法的 while 循环结束后，完成整个回溯搜索过程。

用回溯法解题的一个显著特征是在搜索过程中动态产生问题的解空间。在任何时刻，算法只保存从根结点到当前扩展结点的路径。如果解空间树中从根结点到叶结点的最长路径的长度为 $h(n)$，则回溯法所需的计算空间通常为 $O(h(n))$。而显式地存储整个解空间则需要 $O(2^{h(n)})$或 $O(h(n)!)$内存空间。

5.1.5 子集树与排列树

图 5-1 和图 5-3 中的两棵解空间树是用回溯法解题时常遇到的两类典型的解空间树。

当所给的问题是从 n 个元素的集合 S 中找出 S 满足某种性质的子集时，相应的解空间树称为子集树。例如，n 个物品的 0-1 背包问题所相应的解空间树是一棵子集树，这类子集树通常有 2^n 个叶结点，其结点总个数为 $2^{n+1}-1$。遍历子集树的算法需 $\Omega(2^n)$计算时间。

当所给问题是确定 n 个元素满足某种性质的排列时，相应的解空间树称为排列树。排列树通常有 $n!$ 个叶结点。因此，遍历排列树需要 $\Omega(n!)$计算时间。图 5-3 中旅行售货员问题的解空间树是一棵排列树。

用回溯法搜索子集树的一般算法可描述如下：

```
void backtrack (int t)
{
  if (t>n) output(x);
    else
      for (int i=0;i<=1;i++)
      {
        x[t]=i;
        if (constraint(t)&&bound(t)) backtrack(t+1);
      }
}
```

用回溯法搜索排列树的算法框架可描述如下：

```
void backtrack (int t)
{
```

```
    if (t>n) output(x);
      else
        for (int i=t;i<=n;i++)
        {
          swap(x[t], x[i]);
          if (constraint(t)&&bound(t)) backtrack(t+1);
          swap(x[t], x[i]);
        }
  }
```

在调用 backtrack(1)执行回溯搜索之前,先将变量数组 x 初始化为单位排列$(1,2,\cdots,n)$。

5.2 装载问题

第4章讨论了最优装载问题的贪心算法。本节讨论最优装载问题的一个变形。

1. 问题描述

有一批共 n 个集装箱要装上两艘载重量分别为 c_1 和 c_2 的轮船,其中,集装箱 i 的重量为 w_i,且 $\sum_{i=1}^{n} w_i \leqslant c_1 + c_2$ 。

装载问题要求确定是否有一个合理的装载方案可将这 n 个集装箱装上这两艘轮船。如果有,找出一种装载方案。

例如,当 $n=3$,$c_1=c_2=50$,且 $w=[10,40,40]$,可将集装箱1和集装箱2装上第一艘轮船,而将集装箱3装上第二艘轮船;如果 $w=[20,40,40]$,则无法将这3个集装箱都装上轮船。

当 $\sum_{i=1}^{n} w_i = c_1 + c_2$ 时,装载问题等价于子集和问题。当 $c_1=c_2$ 且 $\sum_{i=1}^{n} w_i = 2c_1$ 时,装载问题等价于划分问题。

即使限制 w_i,$i=1,\cdots,n$ 为整数,c_1 和 c_2 也是整数。子集和问题与划分问题都是NP难的。由此可知,装载问题也是NP难的。

容易证明,如果一个给定装载问题有解,则采用下面的策略可得到最优装载方案:

(1) 首先将第一艘轮船尽可能装满。

(2) 将剩余的集装箱装上第二艘轮船。

将第一艘轮船尽可能装满等价于选取全体集装箱的一个子集,使该子集中集装箱重量之和最接近 c_1。由此可知,装载问题等价于以下特殊的0-1背包问题。

$$\max \sum_{i=1}^{n} w_i x_i$$

$$\sum_{i=1}^{n} w_i x_i \leqslant c_1$$

$$x_i \in \{0,1\}, \quad 1 \leqslant i \leqslant n$$

当然可以用第3章中讨论过的动态规划算法解这个特殊的0-1背包问题。所需的计算

时间是 $O(\min\{c_1,2^n\})$。下面讨论用回溯法设计解装载问题的 $O(2^n)$ 计算时间算法。在某些情况下该算法优于动态规划算法。

2. 算法设计

用回溯法解装载问题时，用子集树表示其解空间显然是最合适的。可行性约束函数可剪去不满足约束条件 $\sum_{i=1}^{n} w_i x_i \leqslant c_1$ 的子树。在子集树的第 $j+1$ 层结点 Z 处，用 cw 记当前的装载重量，即 $cw = \sum_{i=1}^{j} w_i x_i$，当 $cw > c_1$ 时，以结点 Z 为根的子树中所有结点都不满足约束条件，因而该子树中的解均为不可行解，故可将该子树剪去。

下面的解装载问题的回溯法中，方法 maxLoading 返回不超过 c 的最大子集和，但未给出达到这个最大子集和的相应子集。稍后加以完善。

算法 maxLoading 调用递归方法 backtrack(1)实现回溯搜索。backtrack(i)搜索子集树中第 i 层子树。类 Loading 的数据成员记录子集树中结点信息，以减少传给 backtrack 的参数。cw 记录当前结点相应的装载重量，bestw 记录当前最大装载重量。

在算法 backtrack 中，当 $i>n$ 时，算法搜索至叶结点，其相应的装载重量为 cw。如果 cw>bestw，则表示当前解优于当前最优解，此时应更新 bestw。

当 $i \leqslant n$ 时，当前扩展结点 Z 是子集树的内部结点。该结点有 $x[i]=1$ 和 $x[i]=0$ 两个儿子结点。其左儿子结点表示 $x[i]=1$ 的情形，仅当 $cw+w[i] \leqslant c$ 时进入左子树，对左子树递归搜索。其右儿子结点表示 $x[i]=0$ 的情形。由于可行结点的右儿子结点总是可行的，故进入右子树时不需检查可行性。

算法 backtrack 动态地生成问题的解空间树。在每个结点处算法花费 $O(1)$ 时间。子集树中结点个数为 $O(2^n)$，故 backtrack 所需的计算时间为 $O(2^n)$。另外 backtrack 还需要额外的 $O(n)$ 递归栈空间。

具体算法描述如下：

```
public class Loading
{
  //类数据成员
  static int n;                  //集装箱数
  static int [] w;               //集装箱重量数组
  static int c;                  //第一艘轮船的载重量
  static int cw;                 //当前载重量
  static int bestw;              //当前最优载重量

  public static int maxLoading (int [] ww, int cc)
  {
    //初始化类数据成员
    n=ww.length-1;
    w=ww;
    c=cc;
    cw=0;
    bestw=0;
```

```
        //计算最优载重量
        backtrack(1);
        return bestw;
    }

    //回溯算法
    private static void backtrack (int i)
    {//搜索第 i 层结点
        if (i>n)
        {//到达叶结点
            if (cw>bestw) bestw=cw;
            return;
        }
        //搜索子树
        if (cw+w[i]<=c)
        {//搜索左子树,即 x[i]= 1
            cw+=w[i];
            backtrack(i+1);
            cw-=w[i];
        }
        backtrack(i+1);  //搜索右子树
    }
}
```

3. 上界函数

对于前面描述的算法 backtrack,还可引入一个上界函数,用于剪去不含最优解的子树,从而改进算法在平均情况下的效率。设 Z 是解空间树第 i 层上的当前扩展结点。cw 是当前载重量;bestw 是当前最优载重量;r 是剩余集装箱的重量,即 $r=\sum_{j=i+1}^{n} w_j$。定义上界函数为 cw+r。在以 Z 为根的子树中任一叶结点所相应的载重量均不超过 cw+r。因此,当 cw+r≤bestw 时,可将 Z 的右子树剪去。

在下面的改进算法中,引入类 Loading 的变量 r,用于计算上界函数。引入上界函数后,在达到叶结点时就不必再检查该叶结点是否优于当前最优解,因为上界函数使算法搜索到的每个叶结点都是当前找到的最优解。虽然改进后的算法的计算时间复杂性仍为 $O(2^n)$,但在平均情况下改进后算法检查的结点数较少。

改进后的算法描述如下:

```
public class Loading
{
    //类数据成员
    static int n;              //集装箱数
    static int [] w;           //集装箱重量数组
    static int c;              //第一艘轮船的载重量
    static int cw;             //当前载重量
    static int bestw;          //当前最优载重量
```

```
    static int r;                    //剩余集装箱重量

    public static int maxLoading (int [] ww, int cc)
    {
      //初始化类数据成员
      n=ww.length-1;
      w=ww;
      c=cc;
      cw=0;
      bestw=0;
      r=0;
      //初始化 r
      for (int i=1; i<=n; i++)
        r+=w[i];

      //计算最优载重量
      backtrack(1);
      return bestw;
    }

    //回溯算法
    private static void backtrack (int i)
    {//搜索第 i 层结点
      if (i>n)
      {//到达叶结点
        if (cw>bestw) bestw=cw;
        return;
      }
      //搜索子树
      r-=w[i];
      if (cw+w[i]<=c)
      {//搜索左子树
        cw+=w[i];
        backtrack(i+1);
        cw-=w[i];
      }
      if (cw+r>bestw)  //搜索右子树
        backtrack(i+1);
      r+=w[i];
    }
  }
```

4. 构造最优解

为了构造最优解，必须在算法中记录与当前最优值相应的当前最优解。为此，在类 Loading 中增加两个私有数据成员 x 和 bestx，x 用于记录从根至当前结点的路径，bestx 记

录当前最优解。算法搜索到达叶结点处,就修正 bestx 的值。

进一步改进后的算法描述如下:

```
public class Loading
{
  //类数据成员
  static int n;                    //集装箱数
  static int [] w;                 //集装箱重量数组
  static int c;                    //第一艘轮船的载重量
  static int cw;                   //当前载重量
  static int bestw;                //当前最优载重量
  static int r;                    //剩余集装箱重量
  static int [] x;                 //当前解
  static int [] bestx;             //当前最优解

  public static int maxLoading (int [] ww, int cc, int [] xx)
  {
    //初始化类数据成员
    n=ww.length-1;
    w=ww;
    c=cc;
    cw=0;
    bestw=0;
    x=new int[n+1];
    bestx=xx;

    //初始化 r
    for (int i=1; i<= n; i++)
      r+=w[i];

    //计算最优载重量
    backtrack(1);
    return bestw;
  }

  //回溯算法
  private static void backtrack (int i)
  {//搜索第 i 层结点
    if (i>n)
    {//到达叶结点
      if (cw>bestw)
      {
        for (int j=1; j<=n; j++)
          bestx[j]=x[j];
        bestw=cw;
      }
```