谈搜索算法的剪枝优化

许晋炫

【摘要】本文讨论了搜索算法中“剪枝”这一常见的优化技巧。首先由回溯法解决迷宫问题展开论述，介绍了什么是剪枝；而后分析剪枝的三个原则禀正确、准确、高效，并分别就剪枝的两种思路：可行性剪枝及最优性剪枝，结合例题作进一步的阐述；最后对剪枝优化方法进行了一些总结。

【关键字】搜索、优化、剪枝、时间复杂度

引 论

在竞赛中，我们有时会碰到一些题目，它们既不能通过建立数学模型解决，又没有现成算法可以套用，或者非遍历所有状况才可以得出正确结果。这时，我们就必须采用搜索算法来解决问题。

搜索算法按搜索的方式分有两类，一类是深度优先搜索，一类是广度优先搜索。我们知道，深度搜索编程简单，程序简洁易懂，空间需求也比较低，但是这种方法的时间复杂度往往是指数级的，倘若不加优化，其时间效率简直无法忍受；而广度优先搜索虽然时间复杂度比前者低一些，但其庞大的空间需求量又往往让人望而却步。

所以，对程序进行优化，就成为搜索算法编程中最关键的一环。

本文所要讨论的便是搜索算法中优化程序的一种基本方法禀“剪枝”。

什么是剪枝

相信刚开始接触搜索算法的人，都做过类似迷宫这样的题目吧。我们在“走迷宫”的时候，一般回溯法思路是这样的：

1、这个方向有路可走，我没走过

2、往这个方向前进

3、是死胡同，往回走，回到上一个路口

4、重复第一步，直到找着出口

这样的思路很好理解，编程起来也比较容易。但是当迷宫的规模很大时，回溯法的缺点便暴露无遗：搜索耗时极巨，无法忍受。

我们可不可以在向某个方向前进时，先一步判断出这样走会不会走到死胡同里呢？这样一来，搜索的时间不就可以减少了吗？

答案是：可以的。

剪枝的概念，其实就跟走迷宫避开死胡同差不多。若我们把搜索的过程看成是对一棵树的遍历，那么剪枝顾名思义，就是将树中的一些“死胡同”，不能到达我们需要的解的枝条“剪”掉，以减少搜索的时间。

搜索算法，绝大部分需要用到剪枝。然而，不是所有的枝条都可以剪掉，这就需要通过设计出合理的判断方法，以决定某一分支的取舍。在设计判断方法的时候，需要遵循一定的原则。

剪枝的原则

1、正确性

正如上文所述，枝条不是爱剪就能剪的。如果随便剪枝，把带有最优解的那一分支也剪掉了的话，剪枝也就失去了意义。所以，剪枝的前提是一定要保证不丢失正确的结果。

2、准确性

在保证了正确性的基础上，我们应该根据具体问题具体分析，采用合适的判断手段，使不包含最优解的枝条尽可能多的被剪去，以达到程序“最优化”的目的。可以说，剪枝的准确性，是衡量一个优化算法好坏的标准。

3、高效性 设计优化程序的根本目的，是要减少搜索的次数，使程序运行的时间减少。但为了使搜索次数尽可能的减少，我们又必须花工夫设计出一个准确性较高的优化算法，而当算法的准确性升高，其判断的次数必定增多，从而又导致耗时的增多，这便引出了矛盾。

因此，如何在优化与效率之间寻找一个平衡点，使得程序的时间复杂度尽可能降低，同样是非常重要的。

倘若一个剪枝的判断效果非常好，但是它却需要耗费大量的时间来判断、比较，结果整个程序运行起来也跟没有优化过的没什么区别，这样就太得不偿失了。

综上所述，我们可以把剪枝优化的主要原则归结为六个字：正确、准确、高效。

剪枝算法按照其判断思路可大致分成两类：可行性剪枝及最优性剪枝。

下面分别结合例题对这两种方法进行阐述。

可行性剪枝

这个方向可不可以走？走下去会不会碰到死胡同？这就是对某一枝条进行可行性剪枝的简要判断过程。

我们现来看这样一道题。

问题简述：一个规则矩形网络状的城市，城市中心坐标为(0,0)。城市包含M个无法通行的路障(M<=50)，采用如下规则游历城市：第一步走 1 格，第二步走 2 格，依此类推，第N步走n格(N<=20)，除了第一步有四个方向可走，其余各步必须在前一步基础上左转或右转 90 度，最后回到出发点（0,0）。对于给定的N、M，编程求出所有可行的路径。

这道题为ACM竞赛中“黄金图形”一题的简化，曾在GDKOI98 中出为“数学家旅游”一题。

书中的解答采用了简单的回溯法，原因是该题的本身就已经有很强的剪枝判断了。那么我们先来分析一下用回溯法解题的思路：

用x，y两个变量存储当前坐标，每一步对x，y的值进行修改，没有遇到障碍就继续走，走完n步看看有没有回到（0,0），没有的话回溯搜索，直到找完所有路径。

接着，我们来看看这种算法的时间复杂度。

一共走n步，每步要搜索四个方向，假设在最坏的情况下，没有任何障碍物，那么它的时间复杂度应该为：$O(4^n)$。

很明显，这样的算法效率并不会很高，所以我们必须对程序进行剪枝，在未走完n步之前就提早判断出这样的走法是否可行。

当走到第o步时，假设当前坐标为（x_o，y_o），那么离（0,0）的最远距离就应该是Max(x_o, y_o)，而剩下的n-o步可以走的最远距离则是(o+1)+(o+2)+……+n，即$\sum_{i=o+1}^{n} i$。所以，若$\sum_{i=o+1}^{n} i$ <Max(x_o, y_o)的话，就表示就算现在“回头”也没办法到达出发点了，也就是说这条分支即便再搜索下去也找不出解来，这时我们已经可以舍弃这一分支而回溯了。

这样剪枝似乎已经不错了，但是，它的效果只有当数据较大时，才能体现得明显。除了上述的优化，还有没有其他的方法呢？

我们可以这样想，这个城市是规则矩形网络状的，东、南、西、北四个方向都是对称的。打个比方说，与（1,0）这个点对称的可以有（-1,0），（0,1），（0,-1）这三点。那可否设想，当从一个方向出发，寻找到一个解之后，将这个解旋转 90°、180°、270°，不就得出其余三个解了么？这样岂不是节省了 3/4 的搜索次数？

由这个设想出发，我们可以设计出下面的优化：

忽略所有的障碍物，第一步固定走方向a（比如东面），在这个基础上搜索路径，每找到一条路径都将其余三个“对称路径”一起判断，看看有没有经过障碍物，若没有则该路径为解之一。

通过以上分析，我们已经可以编出一个效率较高的搜索程序。请看下表：

“黄金图形”的测试情况（单位：秒）

N值	障碍物数	程序运行时间			
		普通回溯	剪枝一	剪枝二	综合优化
8	2	<0.01	<0.01	<0.01	<0.01
12	4	0.06	0.05	0.06	<0.01
15	0	0.36	0.22	0.11	0.05
16	8	1.54	0.82	0.16	0.11
18	6	5.66	2.85	0.65	0.33
20	0	10.05	5.17	2.58	1.26
21	50	38.56	14.39	5.11	2.42
24	50	210.2	73.11	42.89	20.93

测试结果分析：

1、普通回溯法，在处理比较小的数据时，耗时还是比较低的，但当规模扩大到一定程度时，其时间复杂度呈指数级上升，因此竞赛时应尽量避免使用单纯的不加优化的回溯法。

2、采用第一种剪枝方法，当数据较小时与普通回溯法耗时相当，数据规模逐渐增大时，与回溯法的耗时差距便逐渐拉开，因为剪枝得当，搜索次数比不加优化时至少减小了一半。

3、用对称性来使时间复杂度减少一个指数级，从表中可以明显看出优化后的程序与完全不优化的耗时简直不可同日而语，与前一剪枝方法比较，按照剪枝原则中准确性原则来判断，这种方法比前者要好。

4、综合两种剪枝方法，准确性得到提高，耗时非常少。为了明显比较出各种算法的优劣，我将N值提高到24，结果综合优化的程序只需21秒便出解，耗时为普通回溯法十分之一。

5、这两种剪枝，以及综合的剪枝方法，都遵循了正确性的原则。它们之间的差异主要是在准确性与高效性两点上。可以说，最后一种优化算法综合了前两种，既提高了准确性，又保证了高效性，使得两种剪枝优势互补，取得了非常优秀的效果。竞赛中搜索程序常常使用不只一种的优化方法，所要求达到的就是这种效果。

最优性剪枝

最优性剪枝，又称为上下界剪枝。我们可以回想一下，平时在做一些要求最优解的问题时，搜索到一个解，是不是把这个解保存起来，若下次搜索到的解比这个解更优，就又把更优解保存起来？其实这个较优解在算法中被称为“下界”，与此类似还有“上界”。在搜索中，如果已判断出这一分支的所有子节点都低于下界，或者高于上界，我们就可以将它剪枝。

如何估算出上（下）界呢？我们引入一个概念枭“估价函数”。最优性剪枝算法的核心，就在于设计估价函数上。估价函数在不同的题目中被赋予不同的意义，比如说当前状态与目标状态相差的步数，某一数列的长度等等，都可作为估价函数的一部分。

我们再来看一道题目：

问题简述：在一个N*M的迷宫矩阵中，有X个不可逾越的障碍物，给定x0, y0, x1, y1, 求出由(x0, y0)到(x1, y1)之间的所有路径。

这一道题看起来非常简单，也与上一题有不少相似之处。难道我们不能用简单的回溯法来解决它吗？

我们来分析一下时间复杂度。与上题类似，我们同样假设在最坏情况下，迷宫中没有任何障碍，即是一个坦荡荡的矩形，从(0,0)到(n,m)的话，每一步有四个方向可以走，时间复杂度将达到$O(4^n*4^m)$！假设n=m=20，那么搜索次数将达到2^{42}次！

看来这“枝”是非“剪”不可了。

最初步的剪枝当然就是将走过的方格置为“障碍”，因为若重复通过同一格，最终结果必定不是最优解。

其次，我们可以将每一次搜索出的路径长度与上界比较（初始下界＝∞），不断降低上界，一旦出现路径长超出上界而仍未到达目标点，则放弃该搜索进程。因为就算继续搜索下去，这一条路径也必然比其他路径长，不是最优解。

要完成规模更大的迷宫，仅仅这样剪枝是不够的。我们还必须采取其他的方法，比如先用动态规划求出一个准确的上界，再依据此上界进行搜索等。

当然，对于迷宫这一类题目，搜索算法并不是最好的。我们完全可以用标号法填满一个二维矩阵，再用简单的回溯法进行输出。在这里引用这样的一道题，目的是想让读者更好的理解最优化剪枝的思路及其应用，起到抛砖引玉的作用。

“迷宫问题”的测试情况（单位：秒）

N值	M值	障碍物数	程序运行时间		
			普通回溯	普通定界	动态规划+定界
4	4	1	<0.01	<0.01	<0.01
6	6	2	4.12	<0.01	<0.01
8	8	0	Very long	4.07	0.11
10	10	0	Very long	Very long	1.70
12	12	10	Very long	Very long	5.77
14	14	60	Very long	8.95	0.06

结合本题，我们可以得出最优化剪枝算法所应该注意的一些问题：

1、与可行性剪枝一样，最优性剪枝在保证正确性的同时，同样需要注意提高准确性和高效性，估价函数的计算极为频繁，必须尽量提高其运算效率，不要“拣了芝麻，掉了西瓜”，寻找准确与高效的平衡点，确实重要。

2、在使用最优化剪枝时，一个好的估价函数往往起到“事半功倍”的效果。在搜索之前，我们可以

采用一些高效率的算法，如贪心法、动态规划、标号法等等，求出一个较优解，作为上（下）界，将有解的范围缩小，以尽可能的剪去多余的枝条。如上表中最后一种算法，采用动态规划求出上界，再进行搜索，效率提高了许多。

此外，不但深度优先搜索可以运用最优性剪枝，在广度优先搜索中，同样可以采用这种剪枝方法。比较典型的算法有：分支定界法、A*算法，博弈树等等。不过，这些方法一般空间复杂度都比深度优先搜索要大得多，如何取舍就要依据不同的题目作出相应的判断了。

总 结

在搜索算法中，几乎都需要采用程序优化，以减少时间复杂度。而本文所论述的“剪枝”算法，就是最常见的优化方法之一。

编优化程序的过程中，不论运用的是可行性剪枝还是最优性剪枝，都离不开剪枝的三个原则棗正确、准确、高效，可以说，它们就是剪枝算法的“生命”。

然而，尽管可以采用众多优化算法使得程序的效率有所提高，搜索算法本身的时间复杂度不能从本质上减少是不可改变的事实，我们在考虑对一道题采用搜索算法之前，不妨先仔细想想，有没有其他更好的算法。毕竟，优化程序也是一项“吃力不讨好”的工作，与其耗费大量的时间来优化搜索程序，倒不如多想，多写，尽量以非搜索算法解决问题。

总之，我们在竞赛中应当多动动脑筋，从不同角度来思考问题，采取合适的算法解决问题。

【参考书目】

1、齐鑫论文《搜索方法中的剪枝优化》

2、吴文虎主编《ACM 国际大学生程序设计竞赛试题与解析（一）》

3、吴文虎、王建德编著《实用算法的分析与程序设计》

【程 序】

一、*用“综合优化”解决黄金图形源程序：*{ backtracking method }

```
Const
     input = 'golygon.dat';
     output= 'golygon.out';
     path : Array [1..4, 1..2] of integer =
               ((1, 0), (0, -1), (-1, 0), (0, 1));
     head : Array [1..4] of char =
               ('e', 's', 'w', 'n');
     clogs = 50; { m <= clogs }
     ways = 24; { n <= ways }
Var
     fp : text;
     n, { 最长步数 }
     m, { 障碍物数 }
     o { 路径指针 }
          : byte;
     clog : Array [1..clogs] of   { 储存障碍物坐标 }
     Record x, y : integer End;
     way : Array [0..ways] of byte;{ 储存路径 }
     results : integer; { 路径数 }
Procedure _init; { 初始化 }
```

```
Var i : byte;
Begin
     Assign(fp, input); Reset(fp);
     Readln(fp, n); Readln(fp, m);
     for i := 1 to m do
     Readln(fp,clog[i].x, clog[i].y);
     Close(fp);
     Assign(fp, output); Rewrite(fp)
End;
Procedure _out; { 输出路径 }
Var i, j, t : byte; x, y : integer;
     b : boolean;
Begin
     for j := 1 to 4 do
     Begin
          x := 0; y := 0; b := TRUE;
          for i := 1 to o do
          Begin
               x := x+path[way[i]][1]*i;
               y := y+path[way[i]][2]*i;
               for t := 1 to m do
                    if(x=clog[t].x) and((y-path[way[i]][2]*i<clog[t].y)
                         and(y >= clog[t].y) or (y-path[way[i]][2]*i > clog[t].y)
                         and(y <= clog[t].y)) or (y = clog[t].y)
                         and((x-path[way[i]][1]*i < clog[t].x)
                         and(x >= clog[t].x) or (x-path[way[i]][1]*i > clog[t].x)
                         and(x <= clog[t].x))
                              { 如果通过障碍物 }
                    then Begin b := FALSE; Break End;
               if not b then Break
          End;
          if b then
          Begin
               Inc(results);
               for i := 1 to o do Write(fp, head[way[i]]);
               Writeln(fp)
          End;
          for i := 1 to o do way[i] := way[i] mod 4 + 1 { 旋转路径 }
     End
End;
Function _sum(step : integer) : integer;
Var i, r : integer;
Begin
     r := 0;
```

```
     for i := step+1 to n do Inc(r, i);
     _sum := r
End;
Function _is_able_to_pass (x, y, i, step : integer) : boolean;
Var t : integer;
Begin
     _is_able_to_pass := TRUE;
     if (abs(x) > _sum(step)) or
          (abs(y) > _sum(step)) or      { 不能“回头”了？ }
          (i = way[o])           { 重复走同一个方向了？ }
          or (abs(i-way[o]) = 2){ 折返了？ }
     then _is_able_to_pass := FALSE
End;
Procedure _main(step, x, y : integer);
{ step - 第几步 x, y - 当前坐标 }
Var i : byte;
Begin
     if (x = 0) and (y = 0) and (step = n+1)then
     Begin
          _out; { 若到达（0,0）则输出，回溯 } Exit
     End;
     if step > n then Exit; { 如果超过步数没有回到起点就回溯 }
     for i := 1 to 4 do { 尝试每个方向 }
          if _is_able_to_pass { 如果通行 }(x+path[i][1]*step, y+path[i][2]*step, i, step) then
          Begin
               Inc(o); way[o] := i; { 存下路径 }
               _main(step+1, { 走下一步 }
                    x+path[i][1]*step, y+path[i][2]*step); { 改变坐标 }
               Dec(o) { 恢复原来的路径 }
          End
End;
Procedure _exit;
Begin
     Writeln(fp, 'Found ', results, ' golygon(s).');
     Close(fp)
End;
Begin
     _init;
     o := 1; way[1] := 1;
     _main(2, 1, 0);
     _exit
End.
```

二、用动态规划求上界再搜索的迷宫问题源程序：（为了便于测试，没有输出路径）

```
Const
```

```
     input = 'trip.dat';
     output= 'trip.out';
     path : Array [1..4, 1..2] of shortint =
          ((-1, 0), (1, 0), (0, -1), (0, 1));
     maxn = 30;
     maxm = 30;
Var
     fp : text;
     n, m, x0, y0, x1, y1, all, o, results, xx : longint;
          { all - 路径总数; results - 上界 }
     r : Array [1..maxn, 1..maxm] of shortint;
          { 迷宫，0-可行，-1 -障碍 }
     way : Array [1..1000] of byte;{ 储存路径 }
     l : Array [1..maxn, 1..maxm] of integer; { 动态规划的数组 }
Procedure _init;
Var a, b, x : byte;
Begin
     Assign(fp, input); Reset(fp);
     Readln(fp, n, m, x);
     for x := x downto 1 do
     Begin
          Readln(fp, a, b);
          r[a,b] := -1
     End;
     Readln(fp, x0, y0, x1, y1);
     Close(fp);
     Assign(fp, output); Rewrite(fp)
End;
Procedure _solve; { 动态规划 }
Var i, j, k : integer; b : boolean;
Begin
     l[x0,y0] := 1;
     Repeat
          b:= TRUE;
          for i := 1 to n do
               for j := 1 to m do
                    for k := 1 to 4 do
                         if (l[i,j] > 0) then
                              if (i+path[k][1] > 0)
                              and (i+path[k][1] <= n)
                              and (j+path[k][2] > 0)
                              and (j+path[k][2] <= m) then { 没越界 }
                                   if (r[i+path[k][1],j+path[k][2]] <> -1) then { 不是障碍 }
                                        if (l[i,j]+1 < l[i+path[k][1],j+path[k][2]]) or
```

```
                                        (l[i+path[k][1],j+path[k][2]] = 0) then
                                    Begin
                                        l[i+path[k][1],j+path[k][2]] := l[i,j] + 1;
                                            { 填数组，l[I,j]表示该格的“估价值” }
                                        b := FALSE
                                    End
    Until b; { 一直循环到不再修改数组 }
    results := abs(l[x1,y1] - l[x0,y0]){ 得出上界 }
End;
Procedure _main(x, y : byte);
Var i : byte;
Begin
    if abs(l[x1,y1]-l[x,y]) > results then Exit; { 如果超出上界就立刻回溯 }
    if (x = x1) and (y = y1) then
    Begin  Inc(all); Exit End; { 到达终点，输出 }
    for i := 1 to 4 do
    if (x+path[i][1] > 0) and (x+path[i][1] <= n)
        and (y+path[i][2] > 0) and (y+path[i][2] <= m)
        and (r[x+path[i][1],y+path[i][2]] <> -1)
            { 不越界，而且不遇到障碍 }
        and (abs(l[x1,y1]-l[x+path[i][1],y+path[i][2]]) < abs(l[x1,y1]-l[x,y]))
            {如果那一格的“估价值”比当前这一格好才往那个方向走 }
    then Begin
        Inc(o);
        way[o] := i; r[x,y] := -1; { 存储路径，把走过的格置成障碍 }
        _main(x+path[i][1], y+path[i][2]);{ 搜索下一格 }
        Dec(o); r[x,y] := 0
    End
End;
Procedure _exit;
Var i, j : byte;
Begin
    Writeln(fp, all);
    Close(fp)
End;
Begin
    _init;
    _solve;
    _main(x0, y0);
    _exit
End.
```