搜索方法中的剪枝优化

南开中学 齐鑫

【关键字】搜索、优化、剪枝

【摘要】本文讨论了搜索方法中最常见的一种优化技巧——剪枝而且主要以剪枝判断方法的设计为核心。文章首先借助搜索树，形象的阐明了什么是剪枝；然后分析了设计剪枝判断方法的三个原则：正确、准确、高效，本文将常见的设计剪枝判断的思路分成可行性剪枝和最优性剪枝两大类，并结合上述三个原则分别以一道竞赛题为例作了说明；文章最后对剪枝方法作了一些总结。

1、 引子

搜索是人工智能中的一种基本方法，也是信息学竞赛选手所必须熟练掌握的一种方法。我们在建立一个搜索算法的时候，首要的问题不外乎两个：

1. 建立算法结构。

2. 选择适当的数据结构。

然而众所周知的是，搜索方法的时间复杂度大多是指数级的，简单的不加优化的搜索，其时间效率往往低的不能忍受，更是难以应付信息学竞赛严格的运行时间限制。

本文所讨论的主要内容就是在建立算法的结构之后，对程序进行优化的一种基本方法——剪枝。

首先应当明确的是，“剪枝”的含义是什么。我们知道，搜索的进程可以看作是从树根出发，遍历一棵倒置的树——搜索树的过程。而所谓剪枝，顾名思义，就是通过某种判断，避免一些不必要的遍历过程，形象的说，就是剪去了搜索树中的某些“枝条”，故称剪枝。

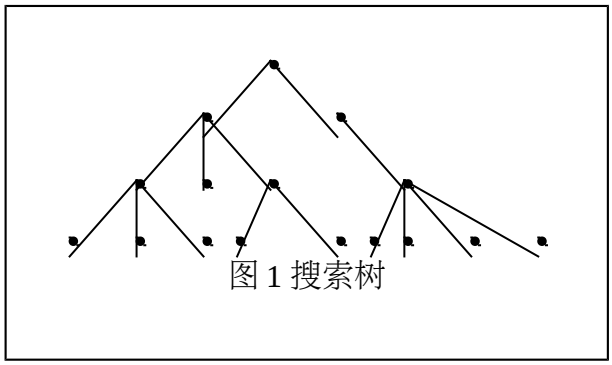

图 1 搜索树

我们在编写搜索程序的时候，一般都要考虑到剪枝。显而易见，应用剪枝优化的核心问题是设计剪枝判断方法，即确定哪些枝条应当舍弃，哪些枝条应当保留的方法。设计出好的剪枝判断方法，往往能够使程序的运行时间大大缩短；否则，也可能适得其反。那么，我们就应当首先分析一下设计剪枝判断方法的时候，需要遵循的一些原则。

2、 剪枝的原则

原则之一：正确性。

我们知道，剪枝方法之所以能够优化程序的执行效率，正如前文所述，是因为它能够“剪去”搜索树中的一些“枝条”。然而，如果在剪枝的时候，将“长有”我们所需要的解的枝条也剪掉了，那么，一切优化也就都失去了意义。所以，对剪枝的第一个要求就是正确性，即必须保证不丢失正确的结果，这是剪枝优化的前提。

为了满足这个原则，我们就应当利用“必要条件”来进行剪枝判断。也就是说，通过解所必须具备的特征、必须满足的条件等方面来考察待判断的枝条能否被剪枝。这样，就可以保证所剪掉的枝条一定不是正解所在的枝条。当然，由必要条件的定义，我们知道，没有被剪枝不意味着一定可以得到正解（否则，也就不必搜索了）。

原则之二：准确性。

在保证了正确性的基础上，对剪枝判断的第二个要求就是准确性，即能够尽可能多的剪去不能通向正解的枝条。剪枝方法只有在具有了较高的准确性的时候，才能真正收到优化的效果。因此，准确性可以说是剪枝优化的生命。

当然，为了提高剪枝判断的准确性，我们就必须对题目的特点进行全面而细致的分析，力求发现题目的本质，从而设计出优秀的剪枝判断方案。

原则之三：高效性。

一般说来，设计好剪枝判断方法之后，我们对搜索树的每个枝条都要执行一次判断操作。然而，由于是利用出解的“必要条件”进行判断，所以，必然有很多不含正解的枝条没有被剪枝。这些情况下的剪枝判断操作，对于程序的效率的提高无疑是具有副作用的。为了尽量减少剪枝判断的副作用，我们除了要下功夫改善判断的准确性外，经常还需要提高判断操作本身的时间效率。

然而这就带来了一个矛盾：我们为了加强优化的效果，就必须提高剪枝判断的准确性，因此，常常不得不提高判断操作的复杂度，

也就同时降低了剪枝判断的时间效率；但是，如果剪枝判断的时间消耗过多，就有可能减小、甚至完全抵消提高判断准确性所能带来的优化效果，这恐怕也是得不偿失。很多情况下，能否较好的解决这个矛盾，往往成为搜索算法优化的关键。

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。

当然，我们在应用剪枝优化的时候，仅有上述的原则是不够的，还需要具体研究一些设计剪枝判断方法的思路。我们可以把常用的剪枝判断大致分成以下两类：

1. 可行性剪枝。

2. 最优性剪枝（上下界剪枝）。

下面，我们就结合上述的三个原则，分别对这两种剪枝判断方法进行一些讨论。

三、可行性剪枝

我们已经知道，搜索过程可以看作是对一棵树的遍历。在很多情况下，并不是搜索树中的所有枝条都能通向我们需要的结果，很多的枝条实际上只是一些死胡同。如果我们能够在刚刚进入这样的死胡同的时候，就能够判断出来并立即剪枝，程序的效率往往会得到提高。而所谓可行性剪枝，正是基于这样一种考虑。

下面我们举一个例子——Betsy 的旅行(USACO)。

题目简述：一个正方形的小镇被分成 N^2 个小方格，Betsy 要从左上角的方格到达左下角的方格，并且经过每个方格恰好一次。编程对于给定的 N，计算出 Betsy 能采用的所有的旅行路线的数目。

我们用深度优先的回溯方法来解决这个问题：Betsy 从左上角出发，每一步可以从一个格子移动到相邻的没有到过的格子中，遇到死胡同则回溯，当移动了 N^2-1 步并达到左下角时，即得到了一条新的路径，继续回溯搜索，直至遍历完所有道路。

但是，仅仅依照上述算法框架编程，时间效率极低，对 N=6 的情况无法很好的解决，所以，优化势在必行。对本题优化的关键就在于当搜索到某一个步骤时，能够提前判断出在后面的搜索过程中是否一定会遇到死胡同，而可行性剪枝正可以在这里派上用场。

我们首先从“必要条件”，即合法的解所应当具备的特征的角度分析剪枝的方法，主要有两个方向：

1. 对于一条合法的路径，除出发点和目标格子外，每一个中间格子都必然有“一进一出”的过程。所以在搜索过程中，必须保证每个尚未经过的格子都与至少两个尚未经过的格子相邻（除非当时 Betsy 就在它旁边）。这里，我们是从微观的角度分析问题。

2. 在第一个条件的基础上，我们还可以从宏观的角度分析，进一步提高剪枝判断的准确性。显然，在一个合法的移动方案的任何时刻，都不可能有孤立的区域存在。虽然孤立区域中的每一个格子也可能都有至少两个相邻的空的格子，但它们作为一个整体，Betsy 已经不能达到。我们也应当及时判断出这种情况，并避免之。

以上两个剪枝判断条件都是正确的，其准确度也比较高。但是，仅仅满足这两点还不够，剪枝判断的操作过程还必须力求高效。假如我们在每次剪枝判断时，都简单的对 N^2 个格子进行一遍扫描，其效率的低下可想而知。因此，我们必须尽可能的简化判断的过程。

实际上，由于 Betsy 的每一次移动，只会影响到附近的格子，

所以每次执行剪枝判断时，应当只对她附近的格子进行检查：

对于第一个剪枝条件，我们可以设一个整型标志数组，分别保存与每个格子相邻的没被经过的格子的数目，Betsy 每次移动到一个新位置，都只会使与之相邻的至多 4 个格子的标志值发生变化，只要检查它们的标志值即可。

而对于第二个剪枝条件，处理就稍稍麻烦一些。但我们仍然可以使用局部分析的方法，即只通过对 Betsy 附近的格子进行判断，就确定是否应当剪枝，下图简要说明了剪枝的原理：

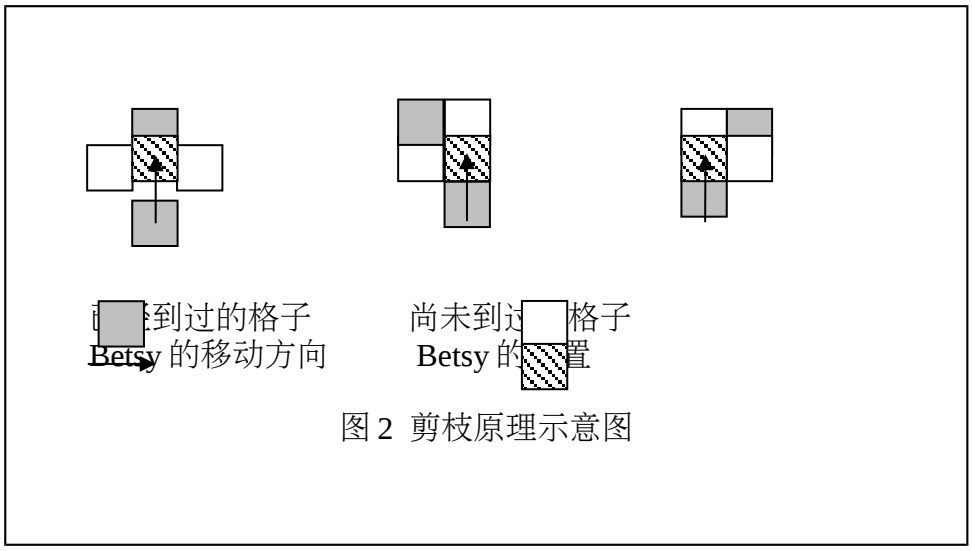

图 2 剪枝原理示意图

上图给出了可以剪枝的三种情况。由于 Betsy 到过的所有格子都一定是四连通的，所以每种情况下的两个白色的格子之间必定是不连通的，它们当中必然至少有一个是属于某个孤立区域的，都一定可以剪枝。

经过上述的优化，程序的时间效率有了很大的提高（参见附录）。

一般说来，可行性剪枝多用于路径搜索类的问题。除本例外，如 Prime Circle (ACM Asian Regional 96)等问题，也都可以使用这种剪枝方法。

在应用可行性剪枝的时候，首先要多角度全面分析问题的特点（本题就是从微观和宏观两个角度设计剪枝方法），找到尽可能多的可以剪枝的情况；同时，还必须注意提高剪枝的时间效率，所以我们使用了“局部判断”的方法，特别是在处理第二个剪枝条件时，更是通过局部判断来体现整体性质（是否有孤立区域），这一技巧不仅在设计剪枝方法的时候能够发挥作用，在其他方面也有着极为

广泛的应用。

3、 最优性剪枝

在我们平时遇到的问题中，有一大类是所谓最优化问题，即所要求的结果是最优解。如果我们使用搜索方法来解决这类问题，那么，最优性剪枝是一定要考虑到的。

为了表述的统一，首先要作一些说明：我们知道，解的优劣一般是通过一个评价函数来评判的。这里定义一个抽象的评价函数——“优度”，它的值越大，对应的解也就越优（对于具体的问题，我们可以认为“优度”代表正的收益或负的代价等）。

然后，我们再来回顾一下搜索最优解的过程：一般情况下，我们需要保存一个“当前最优解”，实际上就是保存解的优度的一个下界。在遍历到搜索树的叶子节点的时候，我们就能得到一个新的解，当然也就得到了它的评价函数值，与保存的优度的下界作比较，如果新解的优度值更大，则这个优度值就成为新的下界。搜索结束后，所保存的解就是最优解。

那么，最优性剪枝又是如何进行的呢？当我们处在搜索树的枝条上时，可以通过某种方法估算出该枝条上的所有解的评价函数的上界，即所谓估价函数 h。显然，h 大于当前保存的优度的下界，是该枝条上存在最优解的必要条件，否则就一定可以剪枝。所以，最优性剪枝也可以称为“上下界剪枝”。同时，我们也可以看到，最优性剪枝的核心问题就是估价函数的建立。

下面举一个应用最优性剪枝的典型例题——最少乘法次数。

题目简述：由 x 开始，通过最少的乘法次数得出 x^n，其中 n 为输入数据。（参见参考书目 1）

因为两式相乘等于方幂相加，所以本题可以等效的表示为：构造一个数列 $\{a_i\}$，满足

$$a_i=\begin{cases}1 & (i=1)\\ a_j+a_k\ (1<=j,k<i) & (i>1)\end{cases}$$

要求 $a_t=n$，并且使 t 最小。

我们选择回溯法作为本程序的主体结构：当搜索到第 i 层时，a_i 的取值范围在 $a_{i-1}+1$ 到 $a_{i-1}*2$ 之间，为了尽快接近目标 n，应当从 $a_{i-1}*2$ 开始从大到小为 a_i 取值，当然，选取的数都不能大于 n。当搜索到 n 出现时，就得到了一个解，与当前保存的解比较取优。最终搜索结束之后，即得到最终的最优解。

如果按上述结构直接进行搜索，效率显然很低。因此，我们可以

考虑使用最优性剪枝进行优化:

优化之一：当然首先要建立估价函数。由于使数列中的最大数加倍无疑是最快的增长方式，所以一种最简单的估价函数为（设数列中当前的最大者是 a_i，即当前搜索深度为 i）：

$$h=\left\lceil \log_2 \frac{n}{a_i} \right\rceil$$

然而，这个估价函数的准确性太差，特别是当 a_i 大于 $\frac{n}{2}$ 时，h 只能等于 1，根本不能发挥剪枝的作用。因此，无论是深度优先的回溯法还是宽度优先的 A*搜索方法，只要还使用这个估价函数，其优化效果都比较有限。

下面，我们再讨论一些进一步的优化手段——

优化之二：着眼于估价函数的“生命”——准确性。我们可以利用加法在奇偶性上的特点的推广，使估价函数在某些情况下的准确性得到一定的提高（具体改进请参见附录）。

优化之三：我们新建立的这个估价函数虽然准确性稍有提高，但时间复杂度也相应提高。为了提高剪枝的效率，我们可以使用一种“逐步细化”的技巧，即先使用一次原先的估价函数（快而不准）进行“过滤”，再使用新的估价函数（稍准但较慢）减少“漏网之鱼”，以求准确性和运行效率的平衡。

优化之四：我们可以在搜索之前将 n 分解质因数，对每个质因数先使用上述搜索方法求解（如某个质因数仍太大，还可以将其减 1，再分解），即相当于将构造数列的过程，划分成若干阶段处理。这样得到的结果虽然不一定是全局的最优解，却可以作为初始设定的优度下界，使后面的全局搜索少走很多弯路。事实上，该估计方法相当准确，在很多情况下都能直接得到最优解，使程序效率得到极大提高。

优化之五：当数列中的当前最大数 a_i 超过 $\frac{n}{2}$ 后，原估价函数就难以发挥作用了。但是，此后的最优方案，实际上就等价于从 a_1 到 a_i 这 i 个数中，选出尽可能少的数（可重复），使它们的和等于 n。这个问题已经可以使用动态规划方法来解决。这样得到的“估价函数”不但完全准确，甚至直接就可以代替后面的搜索过程。这里也体现出了搜索和动态规划的优势互补。

这道题中所体现的最优性剪枝技巧是很多的，它们的优化效果也是非常显著的（优化效果的分析请参见附录）。

由本题，并结合以前的经验，我们可以简单的总结一些在应用最优性剪枝时需注意的问题：

1. 估价函数的设计当然首先得满足正确的原则，即使用“必要条件”来剪枝。在此基础上，就要注意提高估价的准确性。本题的优化之二就是为了这个目的。

2. 与其他剪枝判断操作一样，最优性剪枝的估价函数在提高准确性的同时，也必须注意使计算过程尽量高效的原则。由于剪枝判断在运行时执行极为频繁，所以对其算法进行精雕细琢是相当必要的，有时甚至还可以使用一些非常手法，如前述的“逐步细化”技巧，就是一种寻求时间效率和精确度相平衡的方法。

3. 在使用最优性剪枝时，一个好的初始“优度”下界往往是非常重要的。在搜索开始之前，我们可以使用某种高效方法（如贪心法等）求出一个较优解，作为初始下界，经常可以剪去大量明显不可能的枝条。本题使用划分阶段的搜索方法进行初始定界，就带来了大幅度的优化。

4. 本节所举的是在深度优先搜索中应用最优性剪枝的例子。当然，在广度优先搜索中，也是可以使用最优性剪枝的，也就是我们常说的分枝定界方法。本题也可以使用分枝定界结合 A*算法的方法来解决（用改进的估价函数作为优度上界估计，即 h*函数；前述的动态规划方法可以作为优度下界估计），但时间效率和空间效率都要差一些。不过，在有些问题中，分枝定界和 A*算法的结合以其较好的稳定性，还是有用武之地的。

4、 总结

搜索方法，因其在时间效率方面“先天不足”，所以人们才有针对性的研究出了很多优化技巧。本文所论述的“剪枝”就是最常见的优化方法之一，几乎可以说，只要想使用搜索算法来解决问题，就必须考虑到剪枝优化。

另外需要说明的是，本文所介绍的可行性和最优性两种剪枝判断，其分类只是依据其不同的应用对象，而前面阐述的剪枝方法的三个原则——正确、准确和高效，才是贯穿于始终的灵魂。

本文还介绍了一些剪枝中的常用技巧，如“局部分析”、“逐步细化”等。恰当的使用它们，也能够使程序的效率（主要是时间效率）得到相当的提高。

但是，剪枝方法无论多么巧妙，都不能从本质上降低搜索算法的时间复杂度，这是不争的事实。因此，我们在动手设计一个搜索算法之前，不妨先考虑一下是否存在着更为有效的方法。

而且，在信息学竞赛中，还有一个编程复杂度的问题。我们对一个搜索算法使用了很多优化技巧，虽然可能使程序的时间效率得到一定的提高，但却往往要消耗大量的编程时间，很容易造成“拣了芝麻，丢了西瓜”的结果。

总之，我们在实际设计程序的过程中，不能钻牛角尖，而更应当充分发挥思维的灵活性，坚持“具体问题具体分析”的思想方法。

【参考书目】

1. **《青少年国际信息学（计算机）奥林匹克竞赛指导——人工智能搜索与程序设计》刘福生 王建德 编著**
2. 《Informatics Olympic》

【附录】

1、“Betsy 的旅行”的测试情况：（单位：秒）

N 值	路径数	程序运行时间			
		无剪枝	第一种剪枝	第二种剪枝	两种剪枝
2	1	<0.01	<0.01	<0.01	<0.01
3	2	<0.01	<0.01	<0.01	<0.01
4	8	<0.01	<0.01	<0.01	<0.01
5	86	0.17	<0.01	<0.01	<0.01
6	1770	41.25	0.44	0.33	0.11
7	88418	Very long	28.06	26.86	8.51

（测试环境：Pentium II 266MHz/64MB）

可见，使用前述的可行性剪枝判断方法，能够使程序的时间效率得到相当大的提高。

二、“最少乘法次数”的估价函数的改进：

最初的估价函数的设计思路实际上是在当前数列 $a_1,\cdots,a_i$ 的基础上理想化的构造 $2a_i,4a_i,\cdots,2^p a_i$，当 $2^p a_i < n < 2^{p+1}a_i$ 时，原估价方法认为只需再进行一次加法，即找一个数与 $2^p a_i$ 相加就够了。

然而，这只是保证了能够得到大于等于 n 的数，并不一定恰好得到 n。我们可以作如下分析：

首先，任何一个自然数 i 都可以表示成 $2^k(2m+1)\quad (k,m\in Z^-)$ 的形式，我们可以设 $k(i)$ 表示一个自然数 i 所对应的 k 值。显然，对于任何两个自然数 a 和 b，都有 $k(a+b)\ge \min\{k(a),k(b)\}$（我们由此可以很容易的联想到“奇+奇=偶，偶+偶=偶，奇+偶=奇”的性质）。

然后，我们再研究前述估价时所构造的数列：

$a_1,a_2,\cdots,a_i,2a_i,4a_i,\cdots,2^p a_i$　（其中，$2^p a_i < n < 2^{p+1}a_i$）

在应用新的剪枝判断之前，我们应当先检验 $\lceil \log_2(n/(a_i+a_{i-1}))\rceil \ge p$，这个条件可以保证只有构造上述数列才是最优的。

若存在自然数 $j\,(1\le j\le p)$，使得 $k(2^j a_i) > k(n)$，由 $k(a+b)\ge\min\{k(a),k(b)\}$，

则有 $k(2^t a_i + 2^p a_i)\ge k(2^t a_i)\ge k(2^j a_i) > k(n)\qquad (j\le t\le p)$

$\therefore 2^t a_i + 2^p a_i \ne n$　（$k(a)=k(b)$ 是 $a=b$ 的必要条件）

即 $2^j a_i,\cdots,2^p a_i$ 中的任何一个数与 $2^p a_i$ 相加都不可能得到 n 。

所以，如果 $n-2^p a_i > 2^{j-1} a_i$，则在得到 $2^p a_i$ 后，至少还需要再进行两次加法才有可能得到 n 。

虽然上述改进可以使估价函数在某些情况下的准确性略有提高，但是其本身的时间效率却比较差，这是因为新的估价函数不但包括了原先的估价函数（构造数列），而且还增加了诸如求 $k(a_i)$ 这样的操作。因此，尽管新的估价函数只是原估价函数的改进，而不象在“Betsy 的旅行”中那样是互相补充的两个方面，但在实际的程序中，仍然要连续调用两个估价函数，即使用前文所述的“逐步细化”技巧。

最后需要说明的是，本文所提及的估价函数中所有使用对数运算的部分在实际的程序中都必须改用循环累加的方式来实现。这是因为实数函数在计算机内部是通过级数展开式计算的，实际上也是循环结构，而且更为复杂低效。

当然，实际的程序中，也还有其他的一些细节上的优化。

三、“最少乘法次数”的测试情况及分析：

部分测试数据的运行时间：（单位：秒）

N 值	乘法次数	运行时间						
		程序 1	程序 2	程序 3	程序 4	程序 5	程序 6	程序 7
127	10	0.22	0.22	0.05	0.06	0.06	<0.01	0.16
191	11	3.52	3.90	0.83	0.55	0.55	0.27	1.75
382	11	13.02	13.29	1.05	0.94	0.66	0.33	2.69
511	12	Very long	Very long	0.82	5.66	0.33	0.27	1.81
635	13	46.96	40.37	20.65	19.61	10.06	8.13	Overflow
719	13	45.59	39.11	45.64	13.45	39.11	6.64	27.35
1002	13	11.81	10.65	2.74	3.46	1.10	0.99	6.26
1023	13	Very long	Very long	2.66	Very long	1.04	0.88	6.15
1357	13	9.51	6.98	5.93	5.44	3.40	2.74	13.35
1894	14	44.87	37.24	13.89	15.49	6.37	4.61	23.68

（测试环境：Pentium Pro 233MHz/64MB）

当 n 在 1000 以内的时候，程序 6 的运行时间都不超过 10 秒，其中只有十几个测试数据的运行时间在 5 秒以上。

程序说明：

程序编号	程序特征要点				
	简单的剪枝判断	改进的剪枝判断	初始定界(分段搜索)	动态规划的结合使用	A*算法结合分枝定界
1	✔				
2	✔	✔			
3	✔		✔		
4	✔			✔	
5	✔	✔	✔		
6	✔	✔	✔	✔	
7					✔

测试结果分析：

首先，我们比较一下三个主要的优化点——改进的剪枝判断（逐步细化）、初始定界和结合动态规划——单独应用的优化效果：

1. 单纯加入改进的剪枝判断方法（程序 2），程序时间效率的提高不大，这主要是因为新的估价函数的准确性只是稍有提高，而其本身的时间效率却比较低，这在一定程度上抵消了估价准确性提高所能带来的优化效果。
2. 本题的初始定界方法（程序 3），由于准确性非常高（经常能直接得到最优解），所以能够使程序的效率得到比较明显的改善，在前四个程序中，只有它能够在一分钟以内完成表中所列出的每个测试数据，特别是 n=1023 时，效果尤为显著。但是，如果初始定界的准确性稍有下降，则程序的优化效果就会极不明显（如 n=719）。
3. 加入动态规划后的程序（程序 4），实际上使得搜索过程主要集中在[n/2]以下，在相当程度上避免了对[n/2]到 n 的估价函数几乎没有作用的部分的搜索，所以程序 4 几乎在每个测试点上都比较明显的优于程序 1。

然后，我们再来看看程序 5，由于使用了初始定界，使搜索前就已经有了比较准确的优度下界，所以新的准确性较高的估价函数发挥作用的概率就比较高，因此优化效果相当明显。与程序 3 一样，在初始定界不甚准确的时候（n=719），优化效果大打折扣。

这 7 个程序中最好的当然是程序 6，由于综合了各种优化方法，使它们优势互补，所以获得了最稳定的优化效果。

最后，我们还可以看到，在本题的特定情况下，经过充分优化的深度优先的回溯法搜索无论是在时间上，还是在空间上都明显优于 A*算法（程序 7 使用保护模式编译，除了 A*算法外，还结合使用了分枝定界的方法，否则时空效率

更差）。

【程序】

一、“Betsy 的旅行”的程序（使用两种剪枝）：

```
{$A+,B-,D-,E+,F-,G+,I-,L-,N+,O-,P-,Q-,R-,S-,T-,V+,X+,Y-}
{$M 65520,0,655360}
program Betsy;{IOI'99 集训队 论文例题 1：Betsy 的旅行}
{说明:
  1.为了便于测试计时，本程序采用命令行输入的方式。
  2.为了处理的方便，程序中在地图的最外层补了一圈标志格。
  3.程序中，将逻辑上相对独立的程序段用空行分开。
}
const
 max=7;{本程序所能处理的最大的数据规模}
 delta:array[1..4,1..2]of shortint=((-1,0),(0,1),(1,0),(0,-1));{方向增量}

var
 map:array[0..max+1,0..max+1]of integer;
 {用于标记 Betsy 的移动路线的地图:
    没有到过的位置标记-1,
    最外层的标志格标记 0,
    其它格子标记 Betsy 到达该格子时的移动步数
 }
 left:array[0..max+1,0..max+1]of shortint;
 {标志数组：记录每个格子相邻的四个格子中尚未被 Betsy 经过的格子的数目}
 n:1..max;{地图边长}
 n2:integer;{N*N}
 s:longint;{累加器,记录 Betsy 的移动路线的总数}

procedure init;{读入数据，初始化}
 var
  i:integer;{循环变量}
  temp:integer;{临时变量}
 begin
  val(paramstr(1),n,temp);{从命令行读入 n}
  n2:=n*n;
```

```
  fillchar(map,sizeof(map),255);
  for i:=0 to n+1 do begin
    map[0,i]:=0;
    map[i,0]:=0;
    map[n+1,i]:=0;
    map[i,n+1]:=0;
  end;
  map[1,1]:=1;
  {以上程序段为对地图的初始化}

  fillchar(left,sizeof(left),4);
  for i:=2 to n-1 do begin
    left[1,i]:=3;left[n,i]:=3;
    left[i,1]:=3;left[i,n]:=3;
  end;
  left[1,1]:=2;left[1,n]:=2;left[n,1]:=3;left[n,n]:=2;
  dec(left[1,2]);dec(left[2,1]);
  {以上程序段是对标志数组的初始化}

  s:=0;{累加器清零}
 end;

procedure change(x,y,dt:integer);{给(x,y)相邻的四个格子的 left 标志值加上 dt}
{设标志时，dt 取-1；回溯时，dt 取 1}
 var
  k:integer;{循环变量}
  a,b:integer;{临时坐标变量}
 begin
  for k:=1 to 4 do begin
    a:=x+delta[k,1];b:=y+delta[k,2];
    inc(left[a,b],dt);
  end;
 end;

procedure expand(step,x,y:integer);{搜索主过程：搜索第 step 步，扩展出发位置(x,y)}
 var
```

```
  nx,ny:integer;{Betsy 的新位置}
  dir:byte;{Betsy 的移动方向}

function able(x,y:integer):boolean;{判断 Betsy 是否可以进入(x,y)}
  begin
    able:=not((map[x,y]<>-1)or((step<>n2)and(x=n)and(y=1)));
  end;

function cut1:boolean;{剪枝判断 1}
  var
    i:integer;{循环变量}
    a,b:integer;{临时坐标变量}
  begin
    for i:=1 to 4 do begin
      a:=x+delta[i,1];b:=y+delta[i,2];
      if (map[a,b]=-1)and((a<>nx)or(b<>ny))and(left[a,b]<=1) then begin
        cut1:=true;
        exit;
      end;
    end;
    cut1:=false;
  end;

function cut2:boolean;{剪枝判断 2}
  var
    d1,d2:integer;{相对于当前移动方向的"左右"两个方向}
    fx,fy:integer;{Betsy 由当前位置，沿原方向，再向前移动一步的位置}
  begin
    if (dir=2)or(dir=4) then begin
      d1:=1;d2:=3;
    end
    else begin
      d1:=2;d2:=4;
    end;
    fx:=nx+delta[dir,1];fy:=ny+delta[dir,2];
```

```
    if (map[fx,fy]<>-1)and(map[nx+delta[d1,1],ny+delta[d1,2]]=-1)
    and(map[nx+delta[d2,1],ny+delta[d2,2]]=-1) then begin
      cut2:=true;
      exit;
    end;
    if (map[fx+delta[d1,1],fy+delta[d1,2]]<>-1)and(map[nx+delta[d1,1],ny+delta[d1,2]]=-1)
    and(map[fx,fy]=-1) then begin
      cut2:=true;
      exit;
    end;
    if (map[fx+delta[d2,1],fy+delta[d2,2]]<>-1)and(map[nx+delta[d2,1],ny+delta[d2,2]]=-1)
    and(map[fx,fy]=-1) then begin
      cut2:=true;
      exit;
    end;
    {以上程序段中的三个条件判断分别对应论文剪枝原理图中所列的三种情况}

    cut2:=false;
  end;

 begin{Expand}
  if (step>n2)and(x=n)and(y=1) then begin{搜索到最底层,累加器加 1}
    inc(s);
    exit;
  end;
  for dir:=1 to 4 do begin{循环尝试 4 个移动方向}
    nx:=x+delta[dir,1];ny:=y+delta[dir,2];
    if able(nx,ny) then begin
      if cut1 then continue;{调用剪枝判断 1}
      if cut2 then continue;{调用剪枝判断 2}
      change(nx,ny,-1);
      map[nx,ny]:=step;
      expand(step+1,nx,ny);{递归调用下一层搜索}
      map[nx,ny]:=-1;
      change(nx,ny,1);
    end;
```

```
   end;
 end;

begin{主程序}
 init;{初始化}
 expand(2,1,1);{调用搜索过程}
 writeln('The number of tours is ',s);{输出结果}
end.
```

二、“最少乘法次数”的程序（即附录中的程序 6）：

```
{$A+,B-,D-,E+,F-,G+,I-,L-,N+,O-,P-,Q-,R-,S-,T-,V+,X+,Y-}
{$M 65520,0,655360}
program LeastMultiply;{IOI'99 集训队 论文例题 2：最少乘法次数}
{说明:
   1.为了测试计时的方便,本程序从命令行读入 n 值。
   2.程序结束后，在 output.txt 中给出执行乘法的方式，并给出总的乘法次数。
   3.在搜索过程中，由于与动态规划结合，所以在没有搜索到底层的时候，就可以得到最优解的数列长度(但此时没有得到完整的最优幂次数列)，所以在搜索结束后，程序中调用 formkeep 过程在 keep 中生成完整的构造数列。
   4.由于程序中的搜索过程生成的是最优幂次数列，而没有直接给出乘法的进行方式，所以在输出结果的过程 output 中，对其进行了转换。
      5.为了尽可能的提高程序的时间效率，程序中有几处细节上的优化，请参见程序内的注释。
      6.程序中，逻辑上相对独立的程序段用空行分开。
}

const
 max=20;{数列最大长度}
 maxr=2000;{动态规划计数数组的最大长度(输入的 n 不能超过 maxr 的 2 倍)}
 mp=100;{预处理估价时,可以直接搜索处理的数的范围上限}
 power2:array[0..12]of integer= (1,2,4,8,16,32,64,128,256,512,1024,2048,4096); {2 的方幂}

type
 atype=array[0..max]of integer;{用于记录构造的幂次数列的数组类型，0 号元素记录数列长
```

```
度}

var
 n:integer;{读入的目标数字}
 time:array[0..maxr]of integer;
 {动态规划计数数组，time[i]表示在当前构造数列的基础上，组成数 i 至少需要的加数个数}
 range:integer;{使用动态规划处理的范围：range=[n/2]}
 a:atype;{搜索中记录数列}
 kp:atype;{预处理估界时，记录结果数列的临时数组}
 keep:atype;{记录最优结果数列的数组}
 best:integer;{当前最优解}
 f:text;{输出文件}

procedure init;{初始化}
 var
  temp:integer;{临时变量}
 begin
  val(paramstr(1),n,temp);{从命令行读入 n}
  keep[0]:=1;
  keep[1]:=1;
  best:=maxint;{最优数列长度的初值}
  assign(f,'output.txt');{连接输出文件}
 end;

procedure search(n:integer;var best:integer;var keep:atype);{搜索主过程}
{搜索之前，给出的搜索目标为 n;
      在 best 中存放搜索前已经给出的优度下界;
      在 keep 中存放初始优度下界对应的最优幂次数列。
 搜索结束之后，在 best 中给出的是构造的最优幂次数列的长度,即最少乘法次数加 1;
        在 keep 中给出所构造的最优幂次数列。
}
 var
  kn:integer;{n 所含的 2 的方幂的次数}
  i:integer;{循环变量}
```

```
function getk(num:integer):integer;{求 num 所含的 2 的方幂次数,即论文中所设的 k(num)}
  var
    i:integer;{循环变量}
  begin
    i:=0;
    while not odd(num) do begin
      num:=num shr 1;
      inc(i);
    end;
    getk:=i-1;
  end;

procedure find(step:integer);{递归搜索过程}
  var
    i:integer;{循环变量}
    k:integer;{本层搜索的循环范围上限}

  function ok(num:integer):boolean;{判断数 num 能否在当前被构造出来}
  {为了提高程序的效率，这里利用了动态规划的结果}
    var
      i,j:integer;{循环变量}
    begin
      if num<=range then begin{待判断数 num 在[n/2]以内}
        ok:=(time[num]=2);{直接利用最少需要的加数是否为 2 来判断}
        exit;
      end;
      for i:=step-1 downto 1 do begin
        if a[i]+a[i]<num then break;
        if time[num-a[i]]=1 then begin
        {time[t]=1 表明数 t 在已有数列中出现过，这样可以避免使用循环判断}
          ok:=true;
          exit;
        end;
      end;
      ok:=false;
    end;
```

```
procedure evltime;{动态规划子过程}
  var
    i,j:integer;{循环变量}
    p:integer;{本次动态规划递推的上限}
  begin
    p:=k;
    if p>range then p:=range;
    for i:=a[step-1]+1 to p do begin
      time[i]:=time[i-a[step-1]]+1;{目标函数赋初值}
      for j:=step-2 downto 2 do{决策枚举}
        if time[i-a[j]]+1<time[i] then time[i]:=time[i-a[j]]+1;
    end;
  end;

function h(num:integer):byte;{最初的简单估价函数 h}
  var
    i:integer;{循环计数变量}
  begin
    i:=0;
    while num<n do begin
      num:=num shl 1;
      inc(i);
    end;
    h:=i;
  end;

function cut1:boolean;{最初的剪枝判断，直接调用估价函数 h}
  begin
    if h(a[step])+step>=best then cut1:=true else cut1:=false;
  end;

function cut2:boolean;{使用改进的估价函数的剪枝判断}
  var
    pt:integer;{k(a[step])的值，即 a[step]中所含的 2 的幂的次数}
    rest:integer;{新构造的数列中，满足 k(rest)=k(n)的数}
```

```
  i:integer;{循环计数变量}
 begin
  if h(a[step]+a[step-1])+step+1<best then begin
  {如果新构造数列的第一步选择 a[step]+a[step-1]，而不是 2*a[step]，就必然导致剪枝，
   这是新的估价函数起作用的必要条件。
  }
   cut2:=false;
   exit;
  end;

  pt:=getk(a[step]);{求 k(a[step])}
  if pt>kn then rest:=a[step-1]
  else
   if pt=kn then rest:=a[step]
   else rest:=maxint;
     {给 rest 赋初值，以防新构造的数列中的所有数的 k 值都小于 k(n)}

  i:=0;
  repeat
   a[step+i+1]:=a[step+i] shl 1;
   inc(i);
   if pt+i=kn then rest:=a[step+i];
  until a[step+i]>n;
  dec(i);
  {以上程序段为构造数列的过程}

  if (n-a[step+i]>rest)and(step+i+2>=best) then cut2:=true else cut2:=false;{剪枝判断}
 end;

begin{Find}
 if a[step-1]+a[step-1]>=n then begin
 {数列中的当前最大数已经超过[n/2]，则直接引用动态规划的结果}
  if time[n-a[step-1]]+step-1<best then begin{判断出解}
   best:=time[n-a[step-1]]+step-1;
   keep:=a;
  end;
```

```
      exit;
    end;

    k:=a[step-1]+a[step-1];{计算 a[step]的可选范围的上限}
    evltime;{调用动态规划子过程}

    inc(a[0]);
    for i:=k downto a[step-1]+1 do
      if ok(i) then begin
        if i<=range then time[i]:=1;
        a[step]:=i;
        if cut1 then break;{调用剪枝判断 1}
        if cut2 then continue;{调用剪枝判断 2}
        find(step+1);{递归调用下一层搜索}
      end;
    dec(a[0]);
  end;

 procedure formkeep;{生成最优数列 keep}
 {由于在搜索时，如果 a[step]>[n/2]则直接引用动态规划的结果，所以最优结果数列的最后一
  部分实际上并未得到，所以需要在本过程中将其补充完整。
  这里还需要使用递归和回溯（实际上就是一个小规模的搜索），不过，由于动态规划给出的结
  果表示的是从已有数列中，选出最少的数求和而得到 n，所以对这些数是可以定序的。
 }
   var
    found:boolean;{找到最优数列的标志}

   procedure check(from,left:integer);{回溯法生成最优数列的最后一部分}
      {from 表示当前层内循环的上界(用于定序)，left 表示剩余的需要通过求和而得到的数}
    var
      i:integer;{循环变量}
    begin
      if keep[0]=best then begin
        if left=0 then found:=true;{找到最优数列}
```

```
        exit;
      end;

      inc(keep[0]);
      for i:=from downto 1 do{循环枚举数列中的数}
       if keep[i]<=left then begin
         keep[keep[0]]:=keep[keep[0]-1]+keep[i];
         check(i,left-keep[i]);{调用下一层搜索，但所使用的数不能再超过 keep[i](定序)}
         if found then exit;
       end;
      dec(keep[0]);
    end;

  begin{FromKeep}
   found:=false;
   check(keep[0],n-keep[keep[0]]);{调用搜索过程}
  end;

 begin{Search}
  kn:=getk(n);{计算 k(n)}
  time[0]:=0;
  time[1]:=1;
  a[0]:=1;
  a[1]:=1;
  range:=n div 2;
  {以上程序段为搜索前的初始化}

  find(2);{调用搜索过程}
  formkeep;{搜索结束后，在 keep 中生成完整的构造数列}
 end;

function guess(n:integer):integer;{递归方式实现的预处理估界}
{说明：
   1.子程序 guess 运行结束后，返回的值即为对 n 估价的结果；同时，在 keep 数组中得到对应的数列。
   2.为了能够使估价尽可能准确，guess 中同时考虑了两种分解策略，即使用了回溯结构。
```

*3.*由于每次递归调用 *guess*，其最终结果都必须记入 *keep* 数组，所以每次 *guess* 运行都只操作 *keep* 数组的一部分，同时还要借助于 *kp,kpt* 等临时数组。

}

```
var
  num:integer;{将 n 中的所有 2 的因子都提出后剩下的数}
  pfact:integer;{表示 num 的素因子的变量}
  best:integer;{向下调用时记录返回的估价值}
  b2:integer;{记录 num 中的 2 的因子的数目}
  s:integer;{对 n 的估价步数}
  i:integer;{循环变量}
  sq:integer;{num 的平方根的整数部分，分解素因子时作为上界}
  s1,k2,k:integer;{临时变量}
  kpt:atype;{临时记录最优结果数列}

begin
  num:=n;
  b2:=0;
  while not odd(num) do begin
    num:=num div 2;
    inc(b2);
    inc(keep[0]);
    keep[keep[0]]:=power2[b2];
  end;
  s:=b2+1;
  k2:=keep[0];
  {以上程序段的作用是将 n 中所含的 2 的因子全部提出，剩下的部分即 num}

  if num<=mp then begin{如果 num 足够小(小于等于 mp)，则直接调用搜索过程处理}
    best:=maxint;
    search(num,best,kp);{对 n 调用搜索过程，得到的数列存放在 kp 中}
    inc(s,best-1);
    for i:=2 to best do keep[i+keep[0]-1]:=kp[i];
    inc(keep[0],best-1);
  end
  else begin{使用估价方法处理}
    k:=keep[0];
```

```
    best:=guess(num-1);{递归调用 guess}
    keep[keep[0]+1]:=keep[keep[0]]+1;
    inc(keep[0]);
    inc(s,best);
    {以上程序段为估价的第一种策略：将 num 减 1 后，对 num-1 进行估价}

    sq:=trunc(sqrt(num));
    pfact:=3;
    while (pfact<=sq)and(num mod pfact>0) do inc(pfact,2);
    if pfact<=sq then begin
      kpt:=keep;
      keep[0]:=k2;
      s1:=b2+1;
      {以上程序段为使用第二种策略前的初始化}

      k:=keep[0];
      best:=guess(pfact);
      inc(s1,best-1);
      {以上程序段中递归调用 guess，对质因数 pfact 进行估价}

      k:=keep[0];
      best:=guess(num div pfact);
      for i:=k+1 to keep[0] do keep[i]:=pfact*keep[i];
      inc(s1,best-1);
      {以上程序段对[num/pfact]进行估价}

      if s1<s then s:=s1 else keep:=kpt;{比较两种策略结果的优劣}
    end;
    {以上程序段是估价的第二种策略：将 num 分解出一个较小的质因数后再处理。
     估价的结果存放在 s1 中，使用 kpt 临时存放第一种策略所构造的数列。
    }
  end;

  for i:=k2+1 to keep[0] do keep[i]:=keep[i]*power2[b2];
  guess:=s;{返回估价结果}
end;
```

```
procedure output;{输出结果}
  var
    i,j,k:integer;{循环变量}
  begin
    rewrite(f);
    writeln(f,'X1');
    for i:=2 to best do begin
      for j:=1 to i-1 do begin
        for k:=j to i-1 do
          if keep[j]+keep[k]=keep[i] then break;
        if keep[j]+keep[k]=keep[i] then break;
      end;
      writeln(f,'X',i,'=X',j,'*X',k);
    end;
    writeln(f,'Times of multiplies=',best-1);
    close(f);
  end;

begin{主程序}
  init;{读入数据，初始化}
  best:=guess(n);{调用预处理估界}
  search(n,best,keep);{调用搜索主过程}
  output;{输出结果}
end.
```