浅谈搜索顺序对搜索算法效率的影响

福州大学计算机系02级4班　　陈研

2003年11月28日

摘 要

搜索是人工智能中的一种基本方法，它被广泛的应用于NP问题的求解中。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。因此搜索算法的优化就成了算法设计中尤为重要的一个环节。本文通过几个例子总结了搜索中的常用优化技巧，同时进一步提出了一些经典问题的改进算法。

1　引言

搜索是人工智能中的一种基本方法，它被广泛的应用于NP问题的求解中。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。因此搜索算法的优化就成了算法设计中尤为重要的一个环节。说到搜索算法的优化，很自然的会想到构造强有力的剪枝函数。但有些问题的剪枝函数并不容易找到，或是判断的耗费太大，优化的效果并不明显。这时往往就需要我们从搜索的顺序上来考虑问题，通过对搜索顺序的调整，增强剪枝函数的约束力。常见的搜索顺序调整策略包括：

1. 启发式搜索策略
2. 合理组织输入数据，使其有序化
3. 确定枚举顺序
4. 适当表达所求问题，消除搜索中的冗余

其中第1点，启发式搜索策略属于人工智能的研究范畴，且相关的文献很多，故不在本文的讨论范围之内。

2 数据有序化

数据有序化，就是将杂乱的数据，通过简单的分类和排序，变成有序的数据，从而加快搜索的速度。

在许多问题中，剪枝函数的平均效率很高，但对于一些特定的输入数据却基本不能发挥作用。例如许多组合最优化问题，算法效率的高低取决于初始解的优劣程度。较优的初始解能使估界函数更准确，从而能够减去大部分明显无法达到最优解的分支，减小搜索的总量，提高程序运行效率。而较差的初始解却很难达到剪枝函数的上界，从而使剪枝函数基本不发挥作用。为了解决搜索算法对数据的这种依赖性，我们可以通过调整输入数据的顺序，来消除低效的情况。

例2.1 (图的最大团问题) 给定无向图 $G = (V, E)$，如果 $U \subseteq V$，且对任意 $u, v \in U$ 有 $(u, v) \in E$，则称 U 是 G 的**完全子图**。G 的完全子图 U 是 G 的**团**当且仅当 U 不包含在 G 的更大的完全子图中。G 的**最大团**是指 G 中所含顶点数最多的团。

定义 $N(v_i)$ 表示与 v_i 相邻的顶点集，容易得到以下搜索算法：

```
CLIQUE(U, size)
 1  if |U| = 0
 2    then if size > max
 3           then max ← size
 4                save the solution of the new record;
 5         return
 6  while U ≠ 0
 7    do if size + |U| ≤ max
 8         then return
 9       i ← min{j|v_j ∈ U}
10       U ← U − {v_i}
11       CLIQUE(U ∩ N(v_i), size + 1)
12  return
```

初始时，$max = 0$，调用函数CLIQUE$(V, 0)$ 即可得到结果。

算法的第7行是一个简单的上界估计函数。经试验发现，对于不同的输入数据，算法的效率差别是很大的。主要的原因是，我们在使用估界函数的时候，最希望的就是能较早地得到一个逼近最优解的较优解，从而使上界函数发生更强的效果，去除那些明显无法达到最优解的节点。基于这一思想，我们可以尝试改变图的顶点顺序，使算法能较快得到较优的初始解。这里可以借助贪心算法求近似解。考虑到顶点度大的点属于团的可能性较大，我们可以对顶点按照度从大到小排序。经过这样的有序化处理

后，算法的效率有了较大提高[1]，对输入数据的依赖性也明显减小。

数据的有序化实际上给算法加入了一定程度的**启发性**，使搜索向着有利的方向进行，从而提高求解效率。与A*算法相比，数据有序化的时间只花费在预处理上，无需每次扩展节点都评价一次，节省了大量时间，同时它的编程实现也比A*算法要容易。

对于一些难以找到序关系的问题，我们可以通过对输入数据的顺序随机化，使剪枝函数在平均的意义下发挥较好的作用，这也是 $Sherwood$ 算法的基本思想。

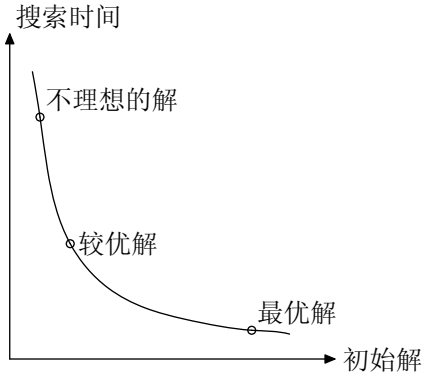
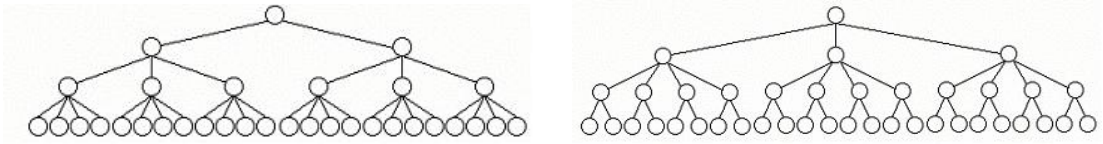


图 1: 初始解对搜索效率的影响

3 确定枚举顺序

对于许多问题而言，在搜索时解向量 $\{a_1, a_2, \cdots, a_n\}$ 的顺序是任意的。在其它条件相当的前提下，我们让可取值最少的 a_i 优先搜索，可以较快得到答案。图2中关于同一问题的2棵不同解空间树，可以体会到这种策略的潜力。



(a) 取值范围小的先搜索　　(b) 取值范围大的先搜索

图 2: 同一问题的2棵不同解空间树

图2(a)中，如果从第1层剪去1棵子树，则从所有应当考虑的3元组中一次消去12个3元组。对于图2(b)，虽然同样从第1层剪去1棵子树，却只从应当考虑的3元组中消去8个3元组。前者的效果明显比后者好。我们来看一个例子：

例3.1 (间隔排列问题) $2N$ 个木块，每个木块标上1到N的自然数中的一个，每个数字会出现在两个木块上。把这些木块排成一排，要求是：标号为i的两个木块之间要间隔i个其它木块。

这道题的每个取值对后续搜索的元素的约束力是不同的。如果我们对每个木块按从小到大的顺序枚举，每确定一个木块的值，对后续搜索几乎不

[1]这个算法的效率还可以进一步提高，见后文。

能起到约束作用。但如果按从大到小的顺序枚举，根据“标号为i的两个木块之间要间隔i个其它木块”的条件，可以很快的去处那些不合理的取值。因此约束力大的元素先搜索，可以使剪枝函数发挥更大的作用。

N	从大到小搜索	从小到大搜索
11	$0.00sec$	$0.00sec$
15	$0.00sec$	$0.10sec$
20	$0.00sec$	$28.14sec$
32	$0.00sec$	$> 50sec$
40	$0.00sec$	$> 1min$

上面的例子还比较简单，我们来看一个不太明显的例子：

例3.2 (最大整数因子团问题) 给定由$n(n \leq 60)$个连续自然数组成的序列$S = \{1, \cdots, n\}$。从当前序列中选择一整数，它在当前序列中至少有2个因子[2]。从当前序列中删去所选整数的所有因子。重复这个过程，直到空序列或当前序列中所有数字在序列中仅有一个因子。该过程的得分是你所选择的数字的总和。求按上述规则所能达到的最大得分。

定义状态：bool st[n]; //其中st[i]==1表示该数以被使用过; st[i]==0表示该数未被使用

定义集合$S_k = \{i | a_i \in S, a_i$在数列中有2个以上因子$\}$。容易找到一个简单的上界函数UPBOUND(level)：

$$\text{CURRENTSCORE} + \sum_{i \in S_{level}} \leq \text{MAXSCORE} \tag{1}$$

本题的状态空间相当庞大，且难以找到精确的上界函数，如果没有选择合适的搜索顺序，对于$n = 30$的情况都要 $10sec$ 左右的时间。注意到本题要求是最大得分，我们每次都应该选择尽量大的数，因此本题要从大到小搜索，这样可以让上界函数的剪枝条件更强。这样做还有一个额外的好处：如果两次搜索达到了相同的状态st，由于我们是从大到小搜索的，可以断定后遇到的这个方案一定不会优于之前达到该状态的方案，因此可以剪枝。这样我们可以对状态建立Hash表，去处状态中的冗余。事实证明这个方法带来的改进是巨大的，在$1.00sec$之内可以做出 $n = 54$ 的解。事实上，这个问题状态空间的冗余是很大，实验证明重复的状态大约是有效状态的4倍。能够消除冗余的根本原因就是我们调整了搜索的顺序。

该问题还可以加入一些小的改进。例如，当自然数i满足: $i*2 \leq n, i*3 > n$时，i显然是要选择的，可以事先预处理掉，而不要参与搜索了。加入这个改进之后，对于$n = 60$的情况可以在$10sec$之内求出。但n的规模进一步增大，就很难出解了，也许可以应用在搜索过程中的一个矛盾的状态进一步提高效率，但没有实现，希望能与大家共同探讨。

[2]该整数本身可以算作一个因子，例如，整数60的所有因子是：$60, 30, 20, 15, 12, 10, 6, 5, 4, 3, 2, 1$。它们构成整数60在当前序列中的因子团

4 问题的表达

有时候适当改变问题的描述方式，将会极大的提高程序的效率。

一种改变方式是人为的对问题加入某些限制，减小搜索的总量。例如求解一些组合问题的时候，问题的解向量 $\{a_1, a_2, \cdots, a_n\}$ 是无序的，如果人为的将向量有序化，约定 $a_i < a_{i+1}, 0 < i < n$ ，则每次递归生成的解都是无重复的，减少了不必要的搜索。更一般的问题：求在给定的置换群下本质不同的所有解。如果只求本质不同的方案总数，可以通过 $Pólya$ 定理[3]求出。但题目要求给出具体方案，这只能用搜索了。按照数据有序化的思想，我们的可以用最小表示法(用等价类中最小[4]的那组数据代表整个等价类)，用搜索算法生成所有的最小表示，他们必然是本质不同的解。这种方法无需判重，节省了空间。

4.1 记忆式搜索

另一种常见的方式是将问题的解分解为若干子问题的解，并从子问题的解中得到原问题的解。这种思想就是动态规划的思想。

例4.1 (均分宝石问题) 有价值分别为1..6的大理石各a[1..6]块，现要将它们分成两部分，使得两部分价值和相等，问是否可以实现。

令$S = \sum(i * a[i])$，若S为奇数，则不可能实现，否则令$Mid = S/2$，则问题转化为能否从给定的宝石中选取部分宝石，使其价值和为Mid。这实际上是判断母函数$(1+x+\cdots+x^{a_1})(1+x^2+\cdots+x^{2a_2})\cdots(1+x^6+\cdots+x^{6a_6})$中是否有$x^{sum/2}$这项。我们可以划分子问题，用动态规划求解。

$m[i,j], 0 \le i \le 6, 0 \le j \le Mid$，表示能否从价值为1..$i$的大理石中选出部分大理石，使其价值和为$j$，若能，则用true表示，否则用false表示。则状态转移方程为：

$$m[i,j] = m[i,j] \ or \ m[i-1, j-i*k](0 \le k \le a[i]) \tag{2}$$

$$m[i,0] = true; 0 \le i \le 6 \tag{3}$$

若m[i,Mid]=true，$0 \le i \le 6$，则可以实现题目要求，否则不可能实现。

实践发现，本题在i较小时，由于可选取的大理石的价值品种单一，数量也较少，因此值为true的状态也较少，但随着i的增大，大理石价值品种和数量的增多，值为true的状态也急剧增多，使得规划过程的速度减慢，影响了算法的时间效率。为了有效的减少true的状态，通过对问题的进一步分析发现：$a[i] > 12$**时，若**$a[i]$**为偶数则令**$a[i] = 12$**；若为奇数则令**$a[i] = 11$**。如果转化后的问题有解，则原问题也有解。**

[3] $Pólya$ 定理: $L = \frac{1}{|G|}\sum_{i=1}^{|G|} m^{c(g_i)}$

[4]此处的大小关系是对应于一种定义在可行解方案集合上的偏序关系

证明： 1. 设$a[6]=n$，即6的个数为$n(n\geq 8)$

(a) 如果可以分成两堆，我们可以分成两种情况：

i. 两堆里都有6，那么我们可知：把n改为$n-2$，仍然可分。(两堆各减一个6)

ii. 只有一堆里有6，设为左边，那么左边的总和不小于6*8=48。我们观察，$5*6=6*5, 4*3=6*2, 3*2=6, 2*3=6, 1*6=6$，而$5*5+4*2+3*1+2*2+1*5=25+8+3+4+5=45<48$。由**抽屉原理**右边必然存在多于*5*个的*5*或者多于*2*个的*4*或者多于*1*个的*3* 或者多于*2*个的*2*或者多于*5*个的*1*。即右边至少存在一组数的和等于若干个6，比如右边有3个4。这样把左边的2个6与右边的3个4交换，则又出现左右都有6的情况。
根据i，我们还是可以把n改为$n-2$且可分的状态不变。

综合i,ii。我们可以看出只要原来n的个数≥ 8,我们就可以把它改为$n-2$，这样操作一直进行到$n<8$。我们可以得出结论，对于大于等于8的偶数，可以换成6；对于大于8的奇数，可以换成7。换完之后仍然可分。

(b) 如果不能分成两堆，显然改为$n-2$时同样也不能分，那么对于大于等于8的偶数，可以换成6; 对于大于8的奇数，可以换成7。换完之后仍然不可分。

综合(a),(b), 我们得出结论：**把不小于8的偶数改为8，大于8的奇数改为7，原来可分与否的性质不会改变**。

2. 以上是对6的讨论，同样的方法可以推出，

(a) 5的个数$6*4+4*4+3*4+2*4+1*4=64<5*13$ 即5的个数多于12时，偶数换为12，奇数换为11

(b) 4的个数$6*1+5*3+3*3+2*1+1*3=35<4*9$ 即4的个数多于8时，偶数换为8，奇数换为7

(c) 3的个数$5*2+4*2+2*2+1*2=24<3*9$ 即3的个数多于8时，偶数换为8，奇数换为7

(d) 2的个数$5*1+3*1+1*1=9<2*5$ 即2的个数多于4时，偶数换为4，奇数换为3

(e) 1的个数多于5则必然可分(在总数是偶数的前提下)

□

利用这个结论，问题的状态空间几乎与输入数据无关，对于任何规模的问题都可在瞬间求出。

4.2 利用子问题的解作为上界函数

有时候，子问题虽然无法直接推出结果，但是我们可以利用子问题作为新的搜索的很精确的上界函数。我们仍以最大团问题为例。

定义c[i]表示$\{v_i, v_{i+1}, \cdots, v_n\}$的最大团。我们得到以下算法：

```
CLIQUE(U, size)
 1  if |U| = 0
 2    then if size > max
 3           then max ← size
 4                save the solution of the new record;
 5                found ← true
 6         return
 7  while U ≠ 0
 8     do if size + |U| ≤ max
 9          then return
10        i ← min{j|v_j ∈ U}
11        if size + c[i] ≤ max
12          then return
13        U ← U − {v_i}
14        CLIQUE(U ∩ N(v_i), size + 1)
15        if found = true
16          then return
17
18  return
```

初始时，令$max = 0$，$found = false$，从大到小求出$c[i]$的值。$c[1]$便是问题的最终答案。

算法的神奇之处在于倒着搜索。从$c[i+1]$到$c[i]$，如果找到更大的团必然有$c[i] = c[i+1] + 1$否则$c[i] = c[i+1]$，因此max一旦被更新就必然是最大的，不必再往下搜索了。同时$size + c[i] \leq max$也可作为一个很准确的上界函数。

经过这样的优化，该算法对于n=50的图，可以轻松的在1秒内出解。

5 结束语

从上面的讨论中可以看出选择搜索顺序对搜索算法的效率起着重要作用。但搜索顺序的选择仅依靠一些基本的方法和思想是不够的，它需要对问题进行全面、透彻的分析，深入挖掘问题的本质，找出解决问题的突破口。算法的优化永远是“**没有最好，只有更好**。”

由于作者水平所限，加之时间仓促，文中错误之处在所难免，希望老

师、同学们能提出意见，共同研究搜索算法的优化，这也是本文创作的初衷。

6 附录

6.1 宝石均分问题源程序

```
#include <stdio.h>

int s,caseno,pl;
int poly[1200],hash[1200];
int a[7];

int read()
{
  int i;
  s=0;
  for (i=1;i<=6;i++) {
    scanf("%d",&a[i]);
    if (a[i]>12) {if (a[i]%2==0) a[i]=12; else a[i]=11;}
    s+=a[i]*i;
  }
  if (s==0) return 0;
  return 1;
}

int main()
{
  int i,j,k,top,temptop,cur;
  caseno=0;
  while (read()) {
    printf("Collection␣#%d:\n",++caseno);
    if (s%2!=0) {printf("Can't␣be␣divided.\n\n");continue;}
    s/=2;
    top=0;
    memset(hash,0,sizeof(hash));
    poly[0]=0;hash[0]=1;pl=1;
    for (i=1;i<=6;i++) {
      temptop=pl;
      for (j=1;j<=a[i];j++)
        for (k=0;k<temptop;k++) {
          if (hash[s]==1) break;
          if (poly[k]+i*j<=s)
            if (hash[poly[k]+i*j]==0) {
              poly[pl++]=poly[k]+i*j;
              hash[poly[k]+i*j]=1;
            }
```

```
        }
      }
    if (hash[s]==1) printf("Can␣be␣divided.\n");
    else printf("Can't␣be␣divided.\n");
    printf("\n");
  }
  return 0;
}
```

6.2 图的最大团源程序

```
#include <stdio.h>

int n,xl,max,V;
int g[51][51];
int x[51],c[51];
int found;

void search(int x[],int xl,int size)
{
  int i,j,k;
  int bak[51];

  if (xl==0) {
    if (size>max) {max=size;found=1;}
    return;
  }
  if (size+xl<=max) return;
  for (i=V;i<n;i++) if (x[i]==1) {
    x[i]=0;
    if (size+c[i]<=max) return;
    k=0;memset(bak,0,sizeof(bak));
    for (j=V;j<n;j++) if (g[i][j]==1&&x[j]==1) {k++;bak[j]=1;}
    search(bak,k,size+1);
    if (found==1) return;
  }
}

int main()
{
  int i,j;
  while (scanf("%d",&n),n!=0) {
    max=0;
    memset(c,1,sizeof(c));
    for (i=0;i<n;i++)
      for (j=0;j<n;j++) scanf("%d",&g[i][j]);
    for (V=n-1;V>=0;V--) {
```

```
      xl=0;found=0;memset(x,0,sizeof(x));
      for (j=V+1;j<n;j++)
        if (g[V][j]==1) {xl++;x[j]=1;}
      search(x,xl,1);
      c[V]=max;
    }
    printf("%d\n",c[0]);
  }
  return 0;
}
```

6.3 整数因子团问题源程序

```
#include <stdio.h>
#define SIZE 1007

int g[121][20];
int st[121],d[121];
int n,max;
int best[121];
int bestd;
long long hash[SIZE*SIZE][2];
static int first[SIZE][SIZE];
int next[SIZE*SIZE];
int entrycount=1;
int nodecount=0;
int conflict=0;

int insert_hash()
{
  long long a,b;
  int c,d,e,i;
  a=0;b=0;
  for (i=1;i<=n;i+=2) a=(a<<1)+st[i];
  for (i=2;i<=n;i+=2) b=(b<<1)+st[i];
  c=(int)(a%SIZE); d=(int)(b%SIZE);
  e=first[c][d];
  while (e!=0) {
    if (hash[e][0]==a&&hash[e][1]==b) return 0;
    e=next[e];
  }
  hash[entrycount][0]=a;
  hash[entrycount][1]=b;
  next[entrycount]=first[c][d];
  first[c][d]=entrycount;
  entrycount++;
  return 1;
```

```
}

int dead(int i)
{
  int j;
  for (j=1;j<=g[i][0];j++) if (st[g[i][j]]==1) return 0;
  return 1;
}

int upbound(int level)
{
  int i,j,k,t,big;
  for (i=n;i>=2;i--) {
    if (st[i]==1) {
      big=1;
      for (j=1;j<=g[i][0];j++) if (st[g[i][j]]==1) {big=0;break;}
      if (big==1) {
        for (k=i*2;k<=n;k+=i) if (st[k]==1) big=0;
        if (big==1) level--;  //delete the indivisual number
      }
    }
  }
  level=level/2;
  for (t=0,i=n;i>=2;i--) {
    if (st[i]==1)
      for (j=1;j<=g[i][0];j++) if (st[g[i][j]]==1) {t+=i;level--;break;}
    if (level<=0) break;
  }
  return t;
}

void search(int depth,int level,int cur)
{
  int i,j,k,t,count=0,cc,tl=0;
  int nst[20],tail[20];

  for (i=n;i>=2&&tl<20;i--) if (st[i]==1&&dead(i)) {
    count=0;
    for (j=i*2;j<=n;j+=i) if (st[j]==1) {count++;cc=j;}
    if (count==1) {tail[tl++]=cc;break;}
  }
  for (i=0;i<tl;i++) {if (st[tail[i]]==1) {
      cur+=tail[i];st[tail[i]]=0;d[depth++]=-tail[i];level--;break;}
  }

  if (max<cur) {
    max=cur;
    memcpy(best,d,sizeof(d));
    bestd=depth;
```

```
  }
  t=upbound(level)+cur;
  if (t<=max) {goto end;}
  if (insert_hash()) nodecount++; else {conflict++;goto end;}
  if (nodecount>=SIZE*SIZE) {printf("Hash␣Full!\n");return;}

  for (i=n;i>=2;i--) if (st[i]==1) {
    memset(nst,0,sizeof(nst));
    nst[0]=i;
    for (k=0,j=1;j<=g[i][0];j++) if (st[g[i][j]]==1) {
      nst[++k]=g[i][j];
    }
    if (k>=1) {
      for (j=0;j<=k;j++) st[nst[j]]=0;
      d[depth]=i;
      search(depth+1,level-(k+1),cur+i);
      for (j=0;j<=k;j++) st[nst[j]]=1;
    }
  }
end:
  for (i=0;i<tl;i++) {st[tail[i]]=1;}
}

int main()
{
  int i,j,l,t;
  while (scanf("%d",&n),n!=0) {
  entrycount=1;nodecount=0;conflict=0;
  memset(first,0,sizeof(first));
  memset(g,0,sizeof(g));

  st[0]=0;
  for (i=1;i<=n;i++) {
    st[i]=1;
    for (j=2;j<i;j++) if (i%j==0) {
      l=++g[i][0];
      g[i][l]=j;
    }
  }
  max=0;l=n-1;st[1]=0;
  for(i=1;i<=n;i++) if (g[i][0]==0) {
    if (i*2>n) { l--;st[i]=0;}
    if (max<i) max=i;
  }
  d[0]=max;
  for (j=1,i=11;i<=n;i++) if (g[i][0]==0)
    if (i*2<=n&&i*3>n) {l--;st[i*2]=0;max+=i*2;d[j++]=i*2; }
  t=upbound(l)+max;
  search(j,l,max);
```

```
  printf("%d:%d␣(",n,max);
  printf("%d␣",best[0]); for (i=j;i<bestd;i++) printf("%d␣",best[i]);
  for (i=1;i<j;i++) printf("%d␣",best[i]);printf(")\n");
  }
  return 0;
}
```

参考文献

[1] Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest, and Clifford Stein, *Introduction To Algorithms*, Addison-Wasley, Reading, Massachusetts, second edition, 2001, ISBN 7-04-011050-4.

[2] 王晓东. 算法设计与分析. 北京: 清华大学出版社, 2003