浅谈记忆化搜索

江苏省常州高级中学 吴景岳

【摘要】

搜索和动态规划是信息学中的两大重要算法。它们各有自己的优点和缺点。针对它们的优缺点，一个新的算法——“记忆化搜索”产生了。它采用了搜索的形式与动态规划的思想，扬长避短，在解决某些题目时，有非常出色的表现。它在信息学竞赛中也有举足轻重的地位，NOI2001 的 cannon 与 NOI2002 的 dragon 都使用到了这个算法。这篇论文着重分析了搜索、动态规划和记忆化搜索之间的联系和区别，以及各自的优缺点，并通过几个例子使得大家对记忆化搜索有一个初步的了解。

【关键字】

重叠子问题 拓扑关系 形式+思想

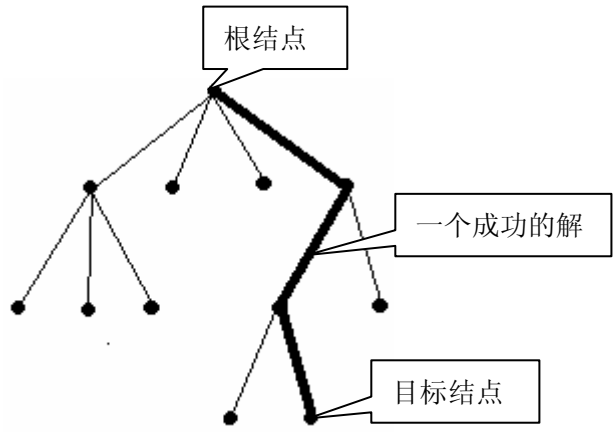
【目录】

【正文】

一、 搜索

1. 搜索树

一道搜索题目拿到手，我们往往要弄清楚这样一些问题：以什么作为状态？这些状态之间又有什么样的关系？其实，在这样的思考过程中，我们已经不知不觉地将一个具体的问题抽象成了一个图论的模型——树。



状态对应着顶点，状态之间的关系（或者说从一个状态到另一个状态的形成过程）对应着边。这样的一棵树就叫做***搜索树***。初始状态对应着根结点，目标状态对应着目标结点。我们的任务就是找到一条从根结点到目标结点的路径——一个成功的解。

2. 例子——words

[问题描述]

Io 和 Ao 在玩一个单词游戏。他们轮流说出一个仅包含元音字母的单词，并且后一个单词的第一个字母必须与前一个单词的最后一个字母一致。游戏可以从任何一个单词开始。

任何单词禁止说两遍，游戏中只能使用给定词典中含有的单词。

游戏的复杂度定义为游戏中所使用的单词的长度总和。

编写程序求出使用一本给定的词典来玩这个游戏所能达到的游戏最大可能复杂度。

数据规模限制：单词总数不超过 16，单词长度不超过 100。

[算法分析]

这是省集训队冬令营的一道题目，当时大多数同学都用了搜索。让我们看看他们是如何搜索的。

状态：

现在说到哪一个单词+已经说过的单词集合

我们将状态表示为(I,S)，表示说到了单词 I，已经说过的单词集合为 S。

状态之间的关系：

(I,S)->(j,S1) (S1=S+j)的条件为单词 J 不在集合 S 中，并且单词 J 可以接在单词 I 后面。

但是，这样搜索效率是不会高的。所以，当时很多人都是因为超时，最后三个点没有出来。那么，搜索效率低下的根本原因是什么呢？

3. 重叠子问题

在一些复杂的搜索问题中，搜索树是极其庞大的，结点数量非常多，结点之间的关系也非常复杂，这是导致搜索算法效率低下的重要原因。其实，在有的情况下，状态的总数并没有多少，有可能很多结点表示的都是同一个状态，我们把这样的状态叫做***“重叠子问题”***。由于搜索算法本身性质的制约，它不能把这些重叠子问题记录下来，只能不知疲倦地继续访问下去，不断遇到这些讨厌的重叠子问题。显然，如果能不把时间放在处理重叠子问题上，算法的效率将发生一个质的飞跃。那么有没有这样一种可以避免重复地处理重叠子问题的算法呢？这就是——

二、 动态规划

1. 基本原理

动态规划是大家极为熟悉的一种算法，实际上一道动态规划的问题也可以转化为求解搜索树的问题。同样是搜索树，动态规划却比搜索算法高效很多，这是因为它把每个状态的最优值都记录了下来，于是成功地避免了重复地处理重叠子问题。不过，动态规划的使用是要满足两个基本条件的：

- 最优子结构

  用动态规划解决一个问题的第一步是刻划一个最优解的结构。如果一个问题的最优解中包含了子问题的最优解，我们就说这个问题具有***最优子结构***。

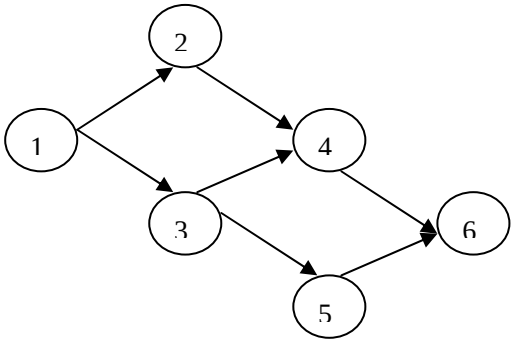
- 无后效性

  动态规划的过程是一个状态转移的过程：不断地从一个已知的状态出发，推出一个新的状态。但是，如果新的状态又能够推出前面的状态（影响前面的状态），那么这样的状态划分就具有了后效性。动态规划能够成功的一个必要条件就是无后效性。

2. 拓扑关系

我们同样地对动态规划问题进行抽象：状态->结点，状态之间的关系->边。这样，一个动态规划的问题就可以抽象成为一个图。因为动态规划是满足无后效性的，这样的图必将是一个有向无环图。在动态规划中，状态之间有着严格的递推关系，我们必须确定哪些状态必须先推出，哪些应当后推出。这就称为***拓扑关系***。将点（状态）按照拓扑关系排序，就叫做***拓扑排序***。拓扑排序保证：除初始状态以外，每个状态都可以由它前面的状态推出。

举一个例子：下图是一个有向无环图，求从顶点 1 到顶点 6 的最长路径。（规定边的方向从左到右）



我们将从起点（顶点 1）开始到某个顶点的最长路径作为状态，用一维数组 opt 记录。显然，opt[1]=0，这是初始状态，即动态规划的边界条件。于是，我们很容易地写出状态转移方程式：opt[j]=max{opt[k]+a[k,j]}（k 到 j 有一条长度为 a[k,j]的边）。虽然有了完整的状态转移方程式，但是我们还不知道动态规划的顺序。所以，我们还需要先进行一下拓扑排序，按照排序的顺序推下去，opt[6]就是问题的解。

可以看出，动态规划相比搜索之所以高效，是因为它将所有的状态都保存了下来。当遇到重复子问题时，它就不会像搜索那样把这个状态的最优值再算一遍，只要把那个状态的最优值调出来就可以了。例如，当计算 opt[4]和 opt[5]的时候，我们都用到了 opt[3]的值。因为我们将它保存下来了，所以就没有必要再去搜索了。

但是，动态规划仍然是有缺点的。一个很突出的缺点就是要进行拓扑排序。这道题的拓扑关系是很简单的，但有些题的拓扑关系是很复杂的。对于这些题目，如果也进行拓扑排序，工作量巨大。遇到这种情况，我们又该怎么办呢？——

三、 记忆化搜索

1. 概念

讲了那么多有关搜索和动态规划的东西，终于进入正题了——记忆化搜索。什么是记忆化搜索呢？前面我们分析了搜索和动态规划各自的优缺点：搜索的低效在于没有能够很好地处理重叠子问题；动态规划虽然比较好地处理了重叠子问题，但是在有些拓扑关系比较复杂的题目面前，又显得无奈。记忆化搜索正是在这样的情况下产生的，它采用搜索的形式和动态规划中递推的思想将这两种方法有机地综合在一起，扬长避短，简单实用，在信息学中有着重要的作用。用一个公式简单地说：**记忆化搜索=搜索的形式+动态规划的思想**。

既然它具有搜索的形式，那么我们也按照讨论搜索算法的方式讨论这个新事物：

2. 记忆化深度优先搜索

（1） 程序框架

```
const
  nonsense=-1; {这个值可以随意定义，表示一个状态的最优值未知}

var
…… {全局变量定义}

function work(s:statetype):longint;
var
…… {局部变量定义}
begin
  if opt[s]<>nonsense then
  begin
    work:=opt[s];
```

```
    exit;
  end; {如果状态 s 的最优值已经知道，直接返回这个值}
  枚举所有可以推出状态 S 的状态 S1
  begin
    temp:=由 S1 推出的 S 的最优值 {有关 work(s1)的函数} ;
    if (opt[s]=nonsense) or (temp 优于 opt[s])
      then opt[s]:=temp;
  end;
  work:=opt[s]; {用动态规划的思想推出状态 s 的最优值}
end;
```

记忆化深度优先搜索采用了一般的深度优先搜索的递归结构，但是在搜索到一个未知的状态时它总把它记录下来，为以后的搜索节省了大量的时间。可见，记忆化搜索的实质是动态规划，效率也和动态规划接近；形式是搜索，简单易行，也不需要进行什么拓扑排序了。

（2） 例子——words

我们继续讨论这个老题目 words（问题描述见 1.1.3），不过这次我们不再讨论如何搜索了，而是讨论如何记忆化搜索。先回顾一下状态和状态之间的关系：

状态：

到某一时刻为止的最后一个单词+已经说过的单词集合

我们将状态表示为(I,S)，表示说到了单词 I，已经说过的单词集合为 S。

状态之间的关系：

(I,S)->(j,S1)的条件为单词 J 不在集合 S 中，并且单词 J 可以接在单词 I 后面（S1=S+j)。

按照这样的状态表示方法，状态总数最多为 2^20。在 Free Pascal 下是完全可以存得下的。

经过这样的划分，状态转移方程式显而易见：

opt[j,s1]=max{opt[I,s]+length[j]}　单词 J 不在集合 S 中，并且单词 J 可以接在单词 I 后面（S=S1-j)。

初始条件：opt[I,{I}]=length[I]

不过值得注意的是集合的表示方法：将一个集合与一个二进制数对应，再将二进制数与十进制数对应。为了方便操作，单词的编号也可以从 1…N，改成 0…N-1。比如：集合[1，4，5]->集合[0，3，4]->11001(2)->2^0+2^3+2^4=25。这样操作不但节省了空间，而且在进行集合操作时可以用位操作，又节省了时间。

我们再来考虑算法的实现，如果用一般的动态规划，就需要先进行拓扑排序，很显然这样做是不可取的，因为状态之间的关系十分复杂。于是，我们将目光投向了记忆化搜索。记忆化搜索是不需要拓扑排序的，这正好避免了我们的问题——复杂的拓扑关系。按照推导出的状态转移方程式和前面展示的框架，相信大家都可以轻松地写出源程序了。（源程序见附录 1）

在这个解法中，虽然我们想到了如何表示状态以及状态之间的关系，但是如果没有记忆化搜索的参与，我们对于如此复杂的拓扑关系仍是无所适从。所以，记忆化搜索的参与可以使得这类具有复杂拓扑关系的动态规划问题得到了较为圆满的解决。

3. 记忆化宽度优先搜索

（1） 程序框架

- 边界条件进入队列
- 找到一个没有扩展过的且肯定最优的状态，没有则退出
- 扩展该状态，并修改扩展到的状态的最优值
- 回第二步

记忆化宽度优先搜索实际上是把一般的动态规划倒过来了。动态规划在计算每一个状态的最优值时，总是寻找可以推出该状态的已知状态，然后从中选择一个最优的决策。而记忆化宽度优先搜索是不断地从已知状态出发，看看能推出哪些其它的状态，并修改其它状态的最优值。

这个算法有两大优点：

- 与记忆化深度优先搜索相似，避免了对错综复杂的拓扑关系的分析
- 化倒推为顺推，更加符合人类的思维过程，而且在顺推比倒推简单时，可以使问题简单化

下面我们通过一个例子，具体问题具体分析。

（2） 例子——cannon

[问题描述]

司令部的将军们打算在 N*M 的网格地图上部署他们的炮兵部队。一个 N*M 的地图由 N 行 M 列组成，地图的每一格可能是山地（用“H” 表示），也可能是平原（用“P”表示），如下图。在每一格平原地形上最多可以布置一支炮兵部队（山地上不能够部署炮兵部队）；一支炮兵部队在地图上的攻击范围如图中黑色区域所示：

P	P	H	P	H	H	P	P
P	H	P	H	P	H	P	P
P	P	P	H	H	H	P	H
H	P	H	P	P	P	P	H
H	P	P	P	P	H	P	H
H	P	P	H	P	H	H	P
H	H	H	P	P	P	P	H

如果在地图中的灰色所标识的平原上部署一支炮兵部队，则图中的黑色的网格表示它能够攻击到的区域：沿横向左右各两格，沿纵向上下各两格。图上其它白色网格均攻击不到。从图上可见炮兵的攻击范围不受地形的影响。

现在，将军们规划如何部署炮兵部队，在防止误伤的前提下（保证任何两支炮兵部队之间不能互相攻击，即任何一支炮兵部队都不在其他支炮兵部队的攻击范围内），在整个地图区域内最多能够摆放多少我军的炮兵部队。

数据限制：N<=100 M<=10

[算法分析]

这一题目若用图论的模型来做，就是一个最大独立集问题，而最大独立集问题确实一个

NP 问题。并且我们也做过这种题目的原型题目，大部分都必须通过搜索来解决。故我们需要考察这个问题的特殊性。

由数据限制可知$m_{max}<<n_{max}$。注意这是唯一一个与原型题目不同的地方。可是如何使用呢？我们先考虑一种极端情况，例如m=1 的情况。这个时候，我们可以用如下的动态规划方程来解出结果：

$$c[0]=0$$

$$c[i]=\begin{cases}\max\limits_{1\le i\le n}\{c[i-3]+1,c[i-1]\} & \text{第i行第1个格子为平地}\\ c[i-1] & \text{第i行第1个格子为山地}\end{cases}$$

其中 *c[i]*的含义是在前 *i* 行各自中可以放置的最多炮兵数，*c[n]*即为解答。

这种方法是否可以扩展到 *m* 列呢？

我们这时用 *c[i]*表示 *m* 列时在前 *i* 行格子中可以放置的最多炮兵数。然后把每一行可能放置炮兵的方法求出来，然后就可以只考虑行与行之间放置炮兵的关系了。不过这样我们还是无法写出规划方程。观察这个时候情况与 *m*=1 式情况的不同：就是对于每以列我们多需要保存各自的 *c[i-3]*和 *c[i-1]*。有鉴于此，我们需要增设一个参数 *p* 来满足动态规划的需要。这里 *p* 应是一个 *m* 位的数组，并且每一位都有三种选择，第 *k* 列的三种选择分别代表该列取 *c[i-1]*、*c[i-2]*、*c[i-3]*时所对应的最值。

这样，我们有：

$$c[0,p]=0$$

$$c[i,p]=\max\{\max_{j\in J}\{c[i-1,p\oplus j]+|j|\},\max_{q\in p^-}\{c[i,q]\}\}$$

最后的答案就是$\max\limits_{j\in J}\{c[n,j]\}$。

我们虽然写出了状态转移方程式，但是如果用这样的方程式进行动态规划，实现起来是非常烦琐的，尤其涉及到了参数 *p* 的倒推，颇为复杂。于是我们又将目光投向了记忆化搜索——化倒推为顺推。把算法进行一些修改：c[0,p]为边界状态；以阵地的行作为阶段进行搜索，从 c[I,p]推出所有 c[I+1,p]，再从 c[I+1,p]推出 c[I+2,p]，……，直到 c[n,p]。这样，使用记忆化搜索使得问题得到了圆满的解决！（源程序见附录 2）

小结一下：首先，在“m<=10”的启发下，找到了动态规划的算法，这是主要的一个步骤；其次，我们选择了记忆化搜索，使得算法的实现变得容易了许多，这也是不可或缺的一步。总而言之，记忆化宽度优先搜索将宽度优先搜索的结构和动态规划的思想有机地结合在一起，为我们的解题提供了一个新的途径。

4. 缺点分析

以上所说的几乎都是记忆化搜索的优势，但是我们也无法忽略记忆化搜索的缺点。

因为记忆化搜索利用的是搜索算法的结构，尤其是记忆化深度优先搜索用了递归的过程，导致算法时间复杂度的系数较大。我写了两个计算斐波那契数列第 20000 项的程序。第一个用的是一般的递推（也可以看作是一般的动态规划），另外一个是用的记忆化搜索。由于两个程序的运行时间都很短，我们将他们运行 10000 遍，发现两个程序的运行时间有明显的差别。在 PentiumIII 1000MHz 的机器上，第一个程序用时 6.57sec，而第二个程序用时 24.77sec。这两个算法应该具有相同的时间复杂度（O（n）），但是由于记忆化搜索算法的系数较大，导致第二个程序比第一个慢了一个常数倍。这看上去影响不大，但是现在竞赛的时

限越来越苛刻，我们也应当小心谨慎，即在可以比较简单的写出一般的动态规划程序的时候，不要乱用记忆化搜索。

四、 总结

“记忆化搜索=搜索的形式+动态规划的思想”，这个公式确切地反映了记忆化搜索的实质。既具有动态规划的高效性，也具有搜索算法的易实现性，是它的优点；效率有所降低（比递推式的动态规划慢一个常数倍）是它的缺点。总体来说，记忆化搜索仍然是利大于弊的。我们要从正反两方面认识它，以便扬长避短，充分应用。

有的人把它看作搜索的一种优化或者动态规划的一种实现方法，这种说法未尝不可，但是既然它有着自己的独特结构，那么不妨把它看作一种新的算法。其实就这个算法本身而言，也是值得我们思考的。近几年来，信息学奥赛题的难度不断增加，这里的难度主要是指算法的不确定性，不稳定性和综合性，也就是说一道题目的解法可能并不是我们通常碰到的经典算法，而是从来没有见过的新的算法。这就要求选手们创造性地去应答没有遇到过的挑战。在提出记忆化搜索之前，我们首先对两种经典算法进行了深刻的***分析***，然后在加以***综合***。由此可见，记忆化搜索是搜索和动态规划两种经典算法的结合物，是创造性思维的典范。掌握了记忆化搜索不但可以解决一类题目，而且可以学会解决问题的一种方法——***分析+综合***。

【参考资料】

《实用算法的分析与程序设计》 电子工业出版社 编著：吴文虎 王建德

《Introduction to Algorithms》 The MIT Press 编著：Thomas H.Cormen
Charles E.Leiserson
Ronald L.Rivest
Clifford Stein

附录 1：

```
program Words;

const
  maxn=16;
  pow:array                    [0..16]                    of
longint=(1,2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768,65536);
  {2 的次幂}
  maxtotal=65535;

var
  w:array [0..maxn-1] of string; {单词}
  a:array [0..maxn-1,0..maxn-1] of boolean; {单词之间的关系}
  c:array [0..maxn-1,0..maxtotal] of longint; {最优值}
  len:array [0..maxn-1] of longint; {单词长度}
  n,i,j,ans,temp,total:longint;

  function work(i,j:longint):longint; {计算（单词 I，集合 J）状态的最优值}
  var
    j1,k,temp:longint;
  begin
    if c[i,j]>0 then
    begin
      work:=c[i,j];
      exit;
    end;
    j1:=j-pow[i];
    for k:=0 to n-1 do
      if a[i,k] and (pow[k] and j<>0) then
      begin
        temp:=work(k,j1)+len[i];
        if temp>c[i,j] then c[i,j]:=temp;
      end;
    work:=c[i,j];
  end;

begin
  assign(input,'words.in');
  assign(output,'words.out');
  reset(input);
  readln(n);
  for i:=0 to n-1 do
  begin
```

```
    readln(w[i]);
    len[i]:=length(w[i]);
  end;
  close(input);{读入数据}

  fillchar(a,sizeof(a),0);
  for i:=0 to n-1 do
    for j:=0 to n-1 do
      if (i<>j) and (w[i,len[i]]=w[j,1]) then a[i,j]:=true; {建立单词间的关系}

  total:=pow[n]-1;
  fillchar(c,sizeof(c),0);
  for i:=0 to n-1 do c[i,pow[i]]:=len[i]; {初始条件——只有一个单词的情况}

  ans:=0;
  for i:=0 to n-1 do
    for j:=0 to total do
      if (pow[i] and j)<>0 then
      begin
        {搜索每一个可能的状态}
        temp:=work(i,j);
        if temp>ans then ans:=temp;
      end;

  rewrite(output);
  writeln(ans);
  close(output); {输出结果}
end.
```

附录 2：

```
{
NOI 2001
Day 2 Task 1 Cannon
By Wu Jingyue
}

program cannon;

const
  maxm=100;
  maxn=10;
  pow:array [0..10] of longint=(1,3,9,27,81,243,729,2187,6561,19683,59049);
  {3 的次幂}
```

```
  maxtotal=59048;

type
  statetype=array [1..maxn] of longint; {状态类型}

var
  a:array [1..maxm,1..maxn] of boolean; {地图}
  c,c1:array [0..maxtotal] of longint; {最优值}
  t:array [1..maxn] of boolean; {搜索堆栈}
  state,state1:statetype;
  m,n,i,j,total,s,ans:longint;
  ch:char;

  procedure numtoarr(k:longint; var a:statetype);
  {将一个十进制的数转化为三进制的数}
  var
    j:longint;
  begin
    for j:=1 to n do
      a[j]:=(k div pow[j-1]) mod 3;
  end;

  procedure arrtonum(a:statetype; var k:longint);
  {将一个三进制的数转化为一个十进制的数}
  var
    j:longint;
  begin
    k:=0;
    for j:=1 to n do
      inc(k,a[j]*pow[j-1]);
  end;

  procedure search(top,now:longint);
  {对每一层的炮兵的放法进行搜索}
  var
    j,j1,temp:longint;
  begin
    if top>n then
    begin
      for j:=1 to n do
        if t[j]
          then state1[j]:=2
          else if state[j]=0
                 then state1[j]:=0
```

```
                else state1[j]:=state[j]-1;
      arrtonum(state1,j1);
      temp:=c[i]+now;
      if temp>c1[j1] then c1[j1]:=temp;
      exit;
    end;
    t[top]:=false; search(top+1,now);
    if a[s,top] and (state[top]=0) then
    begin
      t[top]:=true;
      if top+1<=n then t[top+1]:=false;
      if top+2<=n then t[top+2]:=false;
      search(top+3,now+1);
    end;
  end;

begin
  assign(input,'cannon.in');
  assign(output,'cannon.out');
  reset(input);
  readln(m,n);
  for i:=1 to m do
  begin
    for j:=1 to n do
    begin
      read(ch);
      if ch='H' then a[i,j]:=false else a[i,j]:=true;
    end;
    readln;
  end;
  close(input); {读入数据}

  total:=pow[n]-1;
  for j:=1 to total do c[j]:=-1; c[0]:=0; {初始条件}
  for s:=1 to m do
  begin
    {按行记忆化搜索}
    for i:=0 to total do c1[i]:=-1;
    for i:=0 to total do
      if c[i]<>-1 then
      begin
        {状态可达}
        numtoarr(i,state);
        search(1,0);
```

```
      end;
    c:=c1;
  end;

  ans:=0;
  for j:=0 to total do
    if c[j]>ans then ans:=c[j]; {最优值}
  rewrite(output);
  writeln(ans);
  close(output); {输出结果}
end.
```