第一课 搜索算法的基本理论

课题：搜索原理
目标：
　　知识目标：产生式系统、搜索框架、搜索算法的分类
　　能力目标：对问题进行阐析，分析综合数据库和产生式规则
重点：综合数据库和产生式规则的描述，搜索算法的实质
难点：产生式规则的理解
板书示意：
1） 综合数据库
2） 产生式规则
3） 状态空间图与节点的耗散值
4） 搜索算法的框架

授课过程：

信息学竞赛的试题一般有两种类型：

一、简明的数学模型揭示问题本质。对于这一类试题，我们尽量用解析法求解。

二、对给定的问题建立数学模型，或即使有一定的数学模型，但采用数学方法解决有一定困难。对于这一类试题，我们只好用模拟或搜索求解。

尽管搜索的时间复杂度一般是指数级的，但在缺乏解决问题的有效模型时，搜索却是一种最行之有效的解决问题的基本方法，而且，在使用搜索算法解决问题时，在实现过程中能有很大的优化空间。信息学竞赛中考察搜索算法，其一是考察选手算法运用能力，其二是考察选手的算法优化能力。

下面我们来看看八数码问题。

在 3*3 组成的九宫格棋盘上，摆有八个将牌，每一个将牌都刻有 1—8 中的某一个数码。棋盘中留有一个空格，允许其周围的某一个将牌向空格中移动，如右图所示。这样通过移动将牌就可以不断改变的布局结构，给出一个初始布局（称初始状态）和一个目标布局（称目标状态），问如何移动将牌，才能实现从初始状态到目标状态的转换。

2	8	3
1	6	4
7		5

1	2	3
8		4
7	6	5

图 1 八数码问题的初始和目标状态

对此题，我们很难找到非常简明的数学方法，因此搜索势在必行。在介绍算法之前，我们先介绍一下与搜索有关的基本概念。

一、综合数据库

与问题相关的所有数据元素构成的集合，称为综合数据库。它是用来表述问题状态或有关事实，即它含有所求解问题的信息，其中有些部分可以是不变的，有些部分则可能只与当前问题的解有关。人们可以根据问题的性质，用适当的方法来构造综合数据库的信息。

在描述问题时，可以选择字符串，数组，集合，树，图等数据结构来构建综合数据库，对于八数码问题而言，选用二维数组可以简明直观地描述问题。因此，八数码问题的综合数据库可描述如下：

$\{P_{xy}\}$，其中 1<=x,y<=3，$P_{xy}\in\{0,1,2,3,4,5,6,7,8\}$，且 P_{xy} 互不相等。

因此，从理论上分析，八数码问题的综合数据库共有 9！=362880 种不同的状态。但实际并不能达到这么多，大概只有 9！/2=181440 种。

问题中所有的可能的出现的状态集合就构成了问题的状态空间。问题的状态空间描述了一个问题的求解过程。它一般包含三种类型状态，即初始状态集合 S，中间状态集合 M，目标状态集合 G，所以我们可把状态空间记为三元状态(S,M,G)。对八数码而说，S={唯一的初始状态}，M={初始状态所有可达的非目标状态}，G={唯一的目标状态}。

用 Pascal 语言描述如下：

```
type
    tState = array[1..3,1..3]of byte;
var
    S ,G: tState;  {S 为初始状态，G 为目标状态}
    M : array[0.. 181440] of tState;  {综合数据库}
```

二、产生式规则

构建了综合数据库以后，还需要研究问题的移动规则，称为产生式规则。产生式规则一般可以用如下形式描述：

if 条件 then 规则；

条件描述了规则实施的先决因素，而规则描述问题实施的具体过程，通常为应用该规则所采取的行动或得到的结论。在搜索过程中，一条产生式规则满足了应用条件之后，就可以作用于综合数据库，使综合数据库的状态发生改变，产生新的状态。对于八数码问题而言，把将牌移入空格，实际上等价于将空格向将牌移动，因此移动的规则有四条，具体描述如下：

设 P_{xy} 表示将牌第 x 行第 y 列的数码，m,n 表示数码空格所在的行列值，记 $P_{m,n}=0$,则可以得到如下四条规则：

① if n -1>=1 then begin $P_{m,n}:=P_{m,n-1}$; $P_{m,n-1}:=0$ end;

② if m - 1>=1 then begin $P_{m,n}:=P_{m-1,n}$; $P_{m-1,n}:=0$ end;

③ if n + 1<=3 then begin $P_{m,n}:=P_{m,n+1}$; $P_{m,n+1}:=0$ end;

④ if m + 1<=3 then begin $P_{m,n}:=P_{m+1,n}$; $P_{m+1,n}:=0$ end;

三、状态空间图

在由初始状态扩展到目标状态的进程中，将扩展出来的所有状态构建成一个图来进行保存，这个图称为状态空间图。为了对它有更深入的了解，下面介绍一些图论中的基本概念。

节点：图中的顶点。

弧：描述从一个节点到另一个节点之间的关系，它有方向性，一般用带箭头的边表示。

边：描述两个之间的关系，无方向性。

有向图：图中的节点与节点是用弧连接。

无向图：图中的节点与节点是用边连接。

孩子和父亲节点：一条弧 AB 是从节点 A 指向 B，那么 B 就是 A 的后继（或者叫孩子），A 是 B 的前驱（或者叫父亲）。

有向树：没有回路的连通的有向图。

在树中没有父亲的节点称为根节点。其余节点有且只有一个父亲，树中没有后继的节点称为末端节点或叶节点。

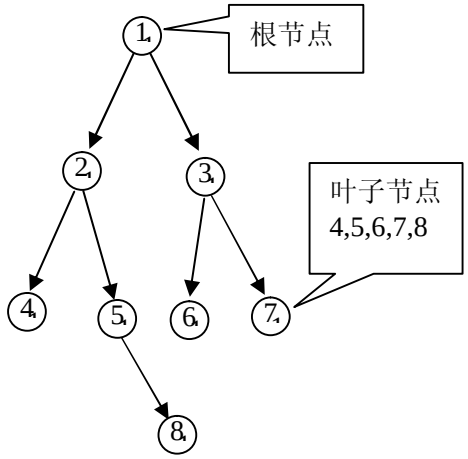


图 2 搜索树扩展

在构筑状态空间的过程中，图中的每一个节点代表综合数据库中的一个状态,每一条弧代表一条产生式规则，如果 C 规则作用于 A 状态得到 B 状态,就从 A 连一条弧 C 指向 B。图搜索的过程实际上就是一棵搜索树(图 2)的生成过程。

四、节点的耗散值

搜索树的度：对一棵搜索树而言，有这样的特征，除叶节点外，所有的节点都有 b 个相同数量后继(b 为产生式规则个数)。我们称 b 为这棵搜索树的度。

节点的深度：根节点的深度为 0，树上任何其它节点的深度是其父节点的深度加 1。

路径：对于一个节点序列 n_0，n_1，...,n_i,...,n_k，对 i= 1 ，2 ，… ,k，若节点 n_{i-1}都有一个后继节点 n_i，则该节点序列称为从节点 n_0到节点 n_k的长度为 k 的一条路径。

路径的耗散值：令 c(i,j)为从节点 n_i 到 n_j 的这段路径（或者弧）的耗散值,一条路径的耗散值就等于连接这条路径各节点间所有弧的耗散值总和。可以用递归公式描述如下：

$$c(n_i,t)= c(n_i, n_j)+ c(n_j, t)$$

其中，$c(n_i,t)$为n_i到 t 这条路径的耗散值。

如果一条路径的耗散值最小，则称这条路径被为最佳路径。通常，我们所求的问题是给定一个初始状态，要求寻找从初始状态到目标状态的最佳路径。常用的方法是：从初始节点出发，按照产生式规则逐步扩展出搜索树，直到找到目标，算出各路径耗散值，取其最小耗散值的路径。这样，我们就从开始的一个状态逐步扩展出含有代价弧的一张图，称这个过程为隐式图扩展过程。采用的策略称为隐式图的搜索。

下面看看隐式图的搜索过程。

五、搜索策略

搜索策略的实质是确定如何选取规则的方式。有两种基本方式：一种是不考虑给定问题所具有的特定知识，系统根据事先确定好某种固定顺序，依次调用规则或随机调用规则，这实际上是盲目搜索的方法。另一种是考虑问题领域可应用的知识，动态地确定规则的排列次序，优先调用较合适的规则使用，这就是通常所说的启发式搜索策略。

对于各种搜索方法，概括起来有

⑴求任一解路的搜索

回溯法(Backtracking)
爬山法(Hill Climbing)
宽度优先法(Breadth-first)
深度优先法(Depth-first)
限定范围的搜索(Beam Search)
好的优先法(Best-first)

⑵求最佳解路的搜索

大英博物馆法(British Museum)
分支界限法(Branch and Bound)
动态规划法(Dynamic Programming)
最佳图的搜索(A*)

⑶与或图的搜索

一般与或图搜索(AO*)
极大极小法(Minimax)
α β剪枝法(Alpha-beta Prunning)
启发式剪枝法(Heuristic Prunning)

只要控制策略一经确定，搜索算法的框架也就确定了。我们解决问题的基本思路是：

分析问题

1.建立初始数据库和目标数据库，以及其它中间数据库的基本资料

2.根据问题提供规则，建立产生式规则

选择控制策略

1. 合理选择产生式规则作用初始数据库，将产生的节点添加到综合数据库中，并建立隐式图。

2.合理选择产生式规则，合理选择综合数据库中好的节点，继续进行扩展，直到找到目标节点。

产生式系统算法描述如下：

PROCEDURE Production-System(算法 1)
1. DATA←初始化数据库
2. Repeat
3. 在规则集中选择某一条可作用于 DATA 的规则 R
4. DATA←R 作用于 DATA 后得到的结果
5. Until DATA 满足结束条件

由上述过程可以看出，每次都是从综合数据库 DATA 中选取一个未被扩展的节点，

按上下左右的顺序选用产生式规则逐步扩展，直到找到目标节点为止。这实际上是一个宽度优先的扩展过程。当然，除了这种扩展方式，我们还可以按照不同的顺序优先选择某些控制策略进行扩展，同样，也可以在综合数据库中，根据每个节点的好坏优先选择被扩展的节点，这是我们下面需要探讨的问题。

上面是产生式系统的算法，下面我们根据图的一般特点，将产生式系统的算法加以扩展，便得到了一般图搜索算法。

一般图搜索算法描述如下：

```
PROCEDURE GRAPH-SEARCH;(算法 2)
1.    G:=G0;                              {其中 G 表示搜索图，G0 是初始图}
2.    Open:=(Source)          ;           {把 source(初始节点)加入 OPEN 表}
3.    Closed:=nil;                        {置 CLOSED 表为空}
4.    Repeat
5.     IF OPEN=nil then Exit(Fail);       {如果 OPEN 为空，输出无解并退出}
6.     n:=FIRST(OPEN);Remove(n,Open);     {取出 OPEN 表首节点,从 Open 删除 n}
7.     Add(n,Closed);                     {把 n 加入到 Closed 表}
8.     If n=Goal then Exit(Success);      {如果 n 为目标，就退出并输出解}
9.     mi:=Expand(n);                     { mi 中不包含其先辈节点 }
10.   标记和修改 mi 的指针                { mi=mj∪mj∪mj }
      .Add(mj,Open)，标记 mj 到 n 的指针;{ mj 为不在 Open 和 Closed 中的节点}
      .计算是否要修改 mk,ml 到 n 的指针;{mk 为在 Open 中的节点}
      .计算是否要修改 mi 到其后继的指针{mi 为在 Open 中的节点}
11.   Add(mi,G);                         {把 mi 加入图 G，并标记修改指针}
12.   对 Open 中的节点按某种原则重新排序;
13.   Until false;
```

以上算法不过是图搜索算法的一个简单框架，对于不同的图搜索算法在实现时，还需要利用到其它一下比较高级的数据结构。上面算法第 10，11 和 13 步在这里介绍的比较笼统，深度优先，宽度优先，A*算法的主要区别也就是在这几步，我们着重加以介绍。

下面以八数码问题为例，看看隐式图的扩展过程:

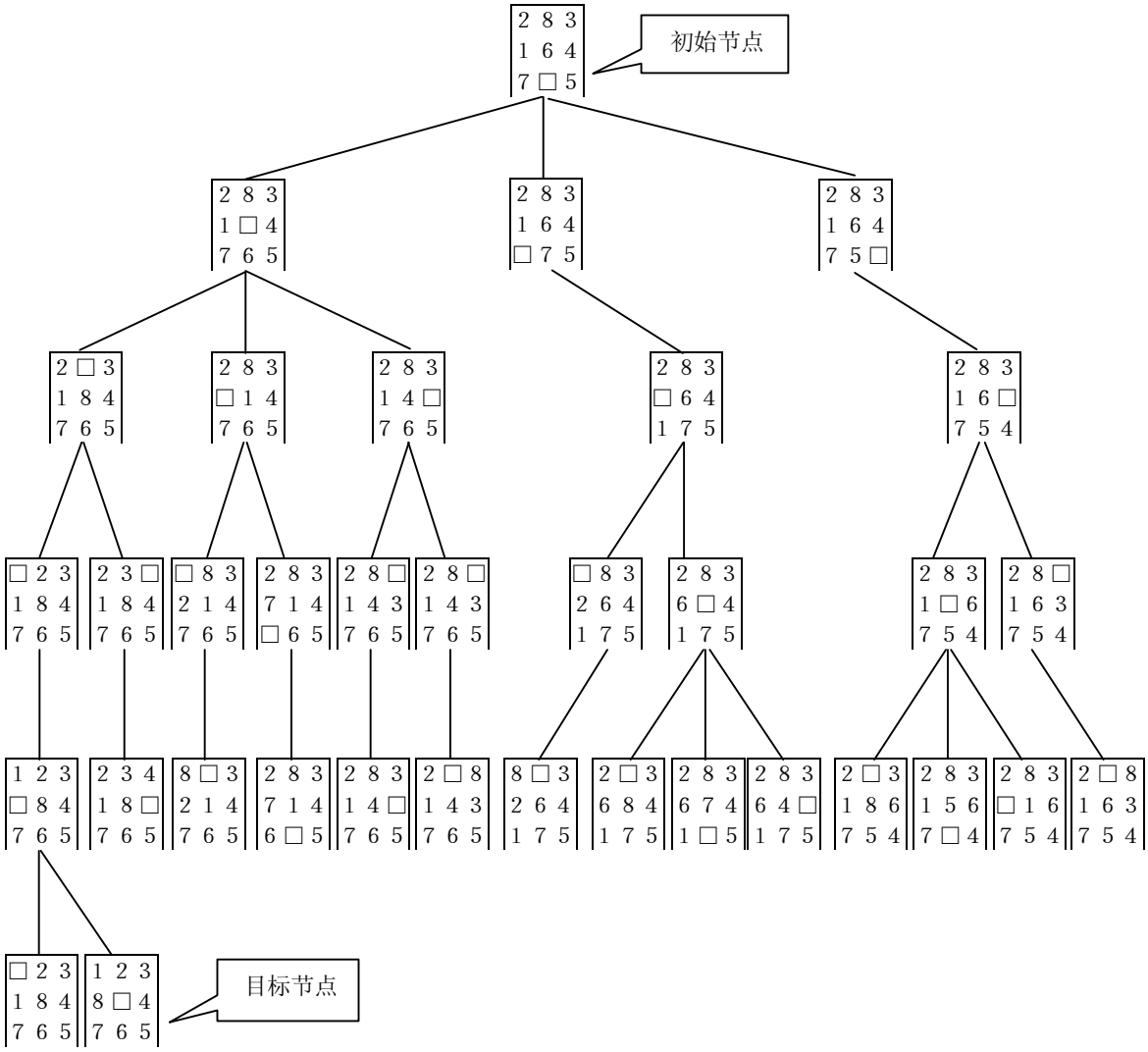


图 3 八数码问题扩展过程

第二课 宽度优先搜索

课题：宽度优先搜索
目标：
　　知识目标：宽度优先搜索的原理与实现
　　能力目标：宽度优先搜索的应用
重点：宽度优先搜索的应用
难点：宽度优先搜索的应用

板书示意：
1. 宽度优先搜索的过程与框架
2. 用宽度优先搜索实现“八数码问题”
3. 宽度优先搜索的应用（聪明的打字员问题）

授课过程：

一．宽度优先搜索的过程

宽度优先搜索算法（又称宽度优先搜索）是最简便的图的搜索算法之一，这一算法也是很多重要的图的算法的原型。Dijkstra 单源最短路径算法和 Prim 最小生成树算法都采用了和宽度优先搜索类似的思想。

宽度优先算法的核心思想是：从初始节点开始，应用算符生成第一层节点，检查目标节点是否在这些后继节点中，若没有，再用产生式规则将所有第一层的节点逐一扩展，得到第二层节点，并逐一检查第二层节点中是否包含目标节点。若没有，再用算符逐一扩展第二层的所有节点……，如此依次扩展，检查下去，直到发现目标节点为止。

二．宽度优先搜索框架

宽度优先算法框架描述如下：

```
PROCEDURE BFS-SEARCH;(算法 3)
1.  G:=G0;
2.  Open:=(Source)          ;
3.  Closed:=nil;
4.  Repeat
5.   IF OPEN=nil then Exit(Fail);
6.   n:=FIRST(OPEN);Remove(n,Open);
7.   Add(n,Closed);
8.   If n=Goal then Exit(Success);
9.   mi:=Expand(n);
10.  对 mi，舍去在 G 中已经出现的节点；
11.  将图中未出现的 mi 加入到 Open 表的表尾
12.  Add(mi,G);
13. Until false;
```

由上述算法可以看出，宽度优先搜索算法与图搜索算法框架的差别仅仅在于 10 步，其中原来的 12 步排序没有必要进行。

10 步的处理为判重操作，因为宽度优先是按照深度小的先扩展，如果 m_i 中出现已经扩展的节点 x，那么 m_i 的当前节点深度肯定大于或者等于原节点（即先出现在 G 中的节点），

所以 x 就没有扩展的必要。

11 步的处理是把当前 m_i 加入到 Open 的表尾。其中，Open 表保存的是待扩展节点，宽度优先必须按照深度小的先扩展，必须先加入 Open 表的先扩展，后加入的后扩展，所以当前只能加入到 Open 表尾。

上面仅仅是宽度优先搜索的一个简单框架，下面我们看看怎样用宽度优先搜索来解决八数码问题。

算法描述如下：

为了较好的实现算法 3 中的 5，6，7，10，11 步我们采用队列的数据结构来构造 Closed 表和 Open 表。

```
1．    List[1]=source;Closed:=0;Open:=1;    {加入初始节点,List 为扩展节点的表}
2．    Repeat
3．      Closed:=Closed+1; N:=List[Closed]; {取出 Closed 对应的节点}
4．      For x:=1 to 4 do [
5．        New:=Move(N,4);                  {对 N 节点使用第 x 条规则，得到 New}
6．        If not_Appear(New,List) then [   {如果 New 没有在 List 中出现}
7．          Open:=Open+1;List[Open]:=New;{加入新节点到 Open}
8．          List[Open].Father:=Closed;       {修改指针}
9．          If List[Open]=Goal then GetOut;
10．       ];
11．     ]
12．  Until Closed<Open;
```

```
program 8no_bfs; {八数码的宽度优先搜索算法}

Const
  Dir : array[1..4,1..2]of integer                {四种移动方向，对应产生式规则}
      = ((1,0),(-1,0),(0,1),(0,-1));
  N=10000;

Type
  T8No = array[1..3,1..3]of integer;

  TList = record
    Father : integer;                             {父指针}
    dep : byte;                                   {深度}
    X0,Y0 : byte;                                 {0 的位置}
    State : T8No;                                 {棋盘状态 }
  end;

Var
  Source,Target : T8No;
  List : array[0..10000] of TList;                {综合数据库 }
  Closed,Open,Best : integer                      { Best 表示最优移动次数}
  Answer : integer;                               {记录解}
  Found : Boolean;                                {解标志}

  procedure GetInfo;                              {读入初始和目标节点}
  var i,j : integer;
  begin
    for i:=1 to 3 do
      for j:=1 to 3 do read(Source[i,j]);
    for i:=1 to 3 do
      for j:=1 to 3 do read(Target[i,j]);
```

```
end;

procedure Initialize;                                    {初始化}
var x,y : integer;
begin
   Found:=false;
   Closed:=0;Open:=1;
   with List[1] do begin
      State:=Source;dep:=0;Father:=0;
      For x:=1 to 3 do
         For y:=1 to 3 do
            if State[x,y]=0 then Begin x0:=x;y0:=y; End;
   end;
end;

Function Same(A,B : T8No):Boolean;              {判断 A,B 状态是否相等 }
Var    i,j : integer;
Begin
   Same:=false;
   For i:=1 to 3 do for j:=1 to 3 do if A[i,j]<>B[i,j] then exit;
   Same:=true;
End;

function Not_Appear(New : tList):boolean;       {判断 New 是否在 List 中出现 }
var i : integer;
begin
   Not_Appear:=false;
   for i:=1 to Open do if Same(New.State,List[i].State) then exit;
   Not_Appear:=true;
end;

procedure Move(N : tList;d : integer;var ok : boolean;var New : tList);
{将第 d 条规则作用于 N 得到 New,OK 是 New 是否可行的标志 }
var x,y : integer;
begin
   X := N.x0 + Dir[d,1];
   Y := N.y0 + Dir[d,2];
   {判断 New 的可行性}
   If not ((X > 0) and ( X < 4 ) and ( Y > 0 ) and ( Y < 4 )) then begin ok:=false;exit end;
   OK:=true;
    New.State:=N.State;       {New=Expand(N,d)}
   New.State[X,Y]:=0;
   New.State[N.x0,N.y0]:=N.State[X,Y];
   New.X0:=X;New.Y0:=Y;
end;

procedure Add(new : tList);                              {插入节点 New}
begin
   If not_Appear(New) then Begin                         {如果 New 没有在 List 出现 }
      Inc(Open);                                         {New 加入 Open 表 }
      List[Open] := New;
    end;
end;

procedure Expand(Index : integer; var N : tList);        {扩展 N 的子节点}
var i : integer;
```

```
        New : tList;
        OK : boolean;
    Begin
      If Same(N.State , Target) then begin                    {如果找到解}
        Found := true;
        Best :=N.Dep;
        Answer:=Index;
        Exit;
      end;
      For i := 1 to 4 do begin                                {依次使用 4 条规则}
        Move(N,i,OK,New);
        if not ok then continue;
        New.Father := Index;
        New.Dep :=N.dep + 1;
        Add(New);
      end;
    end;

    procedure GetOutInfo;                                     {输出}
      procedure Outlook(Index : integer);                     {递归输出每一个解}
      var i,j : integer;
      begin
        if Index=0 then exit;
        Outlook(List[Index].Father);
        with List[Index] do
        for i:=1 to 3 do begin
          for j:=1 to 3 do write(State[i,j],' ');
          writeln;
        end;
        writeln;
      end;
    begin
      Writeln('Total = ',Best);
      Outlook(Answer);
    end;

    procedure Main;                                           {搜索主过程}
    begin
      Repeat
        Inc(Closed);
        Expand(Closed,List[Closed]);                          {扩展 Closed}
      Until (Closed>=Open) or Found;
      If Found then GetOutInfo                                {存在解}
                else Writeln('No answer');                    {无解}
    end;

Begin
    Assign(Input,'input.txt');ReSet(Input);
    Assign(Output,'Output.txt');ReWrite(Output);
    GetInfo;
    Initialize;
    Main;
    Close(Input);Close(Output);
End.
```

由上面的八数码算法的实现，我们可以总结出一个适用于所有宽度优先搜索算法的一般

实现方式。

```
program BFS;
1．    初始化；
2．    建立数据库 data;
3．    初始状态存入数据库;
4．    队首指针 Closed:=0;队尾指针 Open:=1;
5．    Repeat
6．     取下一个 closed 所指的节点;
7．     for r:=1 to  规则数 do [
8．      New=Expand(N,r);
9．      if   Not_Appear(New)   then [
10．       Open:=Open+1;
11．       把新节点加入数据库队尾;
12．       if New=target then  输出并且退出;
13．      ]
14．    ]
15．  Until Closed>=Open;
16．  If Closed>=Open then NoAnswer
17．                  Else GetOutInfo;                    {输出}
```

例 1 聪明的打字员(NOI2001 第一试第三题)

阿兰是某机密部门的打字员，她现在接到一个任务：需要在一天之内输入几百个长度固定为 6 的密码。当然，她希望输入的过程中敲击键盘的总次数越少越好。

不幸的是，出于保密的需要，该部门用于输入密码的键盘是特殊设计的，键盘上没有数字键，而只有以下六个键：Swap0，Swap1，Up，Down，Left，Right，为了说明这 6 个键的作用，我们先定义录入区的 6 个位置的编号，从左至右依次为 1，2，3，4，5，6。下面列出每个键的作用：

Swap0：按 Swap0，光标位置不变，将光标所在位置的数字与录入区的 1 号位置的数字（左起第一个数字）交换。如果光标已经处在录入区的 1 号位置，则按 Swap0 键之后，录入区的数字不变；

Swap1：按 Swap1，光标位置不变，将光标所在位置的数字与录入区的 6 号位置的数字（左起第六个数字）交换。如果光标已经处在录入区的 6 号位置，则按 Swap1 键之后，录入区的数字不变；

Up：按 Up，光标位置不变，将光标所在位置的数字加 1（除非该数字是 9）。例如，如果光标所在位置的数字为 2，按 Up 之后，该处的数字变为 3；如果该处数字为 9，则按 Up 之后，数字不变，光标位置也不变；

Down：按 Down，光标位置不变，将光标所在位置的数字减 1（除非该数字是 0），如果该处数字为 0，则按 Down 之后，数字不变，光标位置也不变；

Left：按 Left，光标左移一个位置，如果光标已经在录入区的 1 号位置（左起第一个位置）上，则光标不动；

Right：按 Right，光标右移一个位置，如果光标已经在录入区的 6 号位置（左起第六个位置）上，则光标不动。

当然，为了使这样的键盘发挥作用，每次录入密码之前，录入区总会随机出现一个长度为 6 的初始密码，而且光标固定出现在 1 号位置上。当巧妙地使用上述六个特殊键之后，可以得到目标密码，这时光标允许停在任何一个位置。

现在，阿兰需要你的帮助，编写一个程序，求出录入一个密码需要的最少的击键次数。

输入文件（clever.in）

文件仅一行，含有两个长度为 6 的数，前者为初始密码，后者为目标密码，两个密码之间用一个空格隔开。

输出文件（clever.out）

文件仅一行，含有一个正整数，为最少需要的击键次数。

输入样例

123456 654321

输出样例

11

样例说明：

初始密码是 123456，光标停在数字 1 上。对应上述最少击键次数的击键序列为：

击键序列：	击键后的录入区
	123456
Swap1	623451
Right	623451
Swap0	263451
Down	253451
Right	253451
Up	254451
Right	254451
Down	254351
Right	254351
Up	254361
Swap0	654321

最少的击键次数为 11。

分析：

我们先考虑问题的规模，确定数据结构：对于此题，如果我们用（S,index）表示问题中的一个状态，S 为密码，index 表示光标位置。那么，（Source，1）为问题的初始状态，（Target，pos）为问题的目标状态。其中 pos 任意。那么综合数据库中可能存在的节点数为：$6*10^6$。

根据问题提供的规则，可以列出如下产生式规则：

设当前状态为(S,index)，下一个状态为(S',index')

① Swap0：

```
if index<>1 then [ S':=S;S'[1]:=S[index];S'[index]:=S[1];Index':=index;]
```

② Swap1：

```
if index<>6 then [ S':=S;S'[6]:=S[index];S'[index]:=S[6];Index':=index;]
```

③ Up：

```
if S[index]<>9 then [ S':=S;S'[index]:=S'[index]+1;Index':=index; ]
```

④ Down：

```
if S[index]<>0 then [ S':=S;S'[index]:=S'[index]-1;Index':=index; ]
```

⑤ Left：

```
if index<>0 then  [ S':=S; Index':=index-1;]
```

⑥ Right：

```
if index<>6 then  [ S':=S; Index':=index+1;]
```

下面我们采用宽度优先扩展的方法，从初始状态出发，按照上述规则扩展搜索树，直到找到目标节点。程序如下：

```
program clever;
  const
    finp          = 'clever.in';
    fout          = 'clever.out';
    maxl           = 6000000;
  type
    statetype     = array[1 .. 6] of shortint;
    Nodetype      = record
                      state : statetype;              {密码}
                      point , step : shortint;        {光标位置，按键次数}
```

```
                    end;
var
  q              : array[0 .. maxl] of Nodetype;        {队列}
  app            : array[1 .. 6 , 0 .. 9 , 0 .. 9 , 0 .. 9 , 0 .. 9 , 0 .. 9] of boolean;
                                                        {判重数组}
  ch             : char;
  final          : statetype;                           {目标状态}
  i , u , v      : integer;
  closed , open : longint;

begin
  assign(input , finp);
  reset(input);
                                                         {读入}
  for i := 1 to 6 do begin
    read(ch);
    while (ch < '0') or (ch > '9') do read(ch);
    q[1].state[i] := ord(ch) - 48;
  end;
  for i := 1 to 6 do begin
    read(ch);
    while (ch < '0') or (ch > '9') do read(ch);
    final[i] := ord(ch) - 48;
  end;
  close(input);
                                                         {初始化}
  closed := 0; open := 1;
  q[1].point := 1;
  fillchar(app , sizeof(app) , false);
  while true do begin
    closed := closed mod maxl + 1;
    with q[closed] do begin
      if comparebyte(state , final , 6) = 0 then begin {如果是目标的话}
        assign(output , fout);
        rewrite(output);
        writeln(step);
        close(output);
        break;
      end;
                                                        {规则 1 }
      if (point > 1) and not app[point-1,state[1],state[2],state[3],state[4],state[5],state[6]]
      then begin
        open := open mod maxl + 1;
        q[open].state := state;
        q[open].step := step + 1;
        q[open].point := point - 1;
        app[point-1,state[1],state[2],state[3],state[4],state[5],state[6]] := true;
      end;
                                                         {规则 2}
      if(point < 6) and not app[point+1,state[1],state[2],state[3],state[4],state[5],state[6]]
      then begin
        open := open mod maxl + 1;
        q[open].state := state;
        q[open].step := step + 1;
        q[open].point := point + 1;
        app[point+1,state[1],state[2],state[3],state[4],state[5],state[6]] := true;
```

```
            end;
                                                    {规则 3}
            if state[point] > 0 then begin
               state[point] := state[point] - 1;
               if not app[point,state[1],state[2],state[3],state[4],state[5],state[6]]
               then begin
                  open := open mod maxl + 1;
                  q[open].state := state;
                  q[open].step := step + 1;
                  q[open].point := point;
                  app[point,state[1],state[2],state[3],state[4],state[5],state[6]] := true;
               end;
               state[point] := state[point] + 1;
            end;
                                                    {规则 4}
            if state[point] < 9 then begin
               state[point] := state[point] + 1;
               if not app[point,state[1],state[2],state[3],state[4],state[5],state[6]]
               then begin
                  open := open mod maxl + 1;
                  q[open].state := state;
                  q[open].step := step + 1;
                  q[open].point := point;
                  app[point,state[1],state[2],state[3],state[4],state[5],state[6]] := true;
               end;
               state[point] := state[point] - 1;
            end;
            u := state[point]; v := state[1];
            state[1] := u; state[point] := v;
                                                    {规则 5}
            if not app[point,state[1],state[2],state[3],state[4],state[5],state[6]]
            then begin
               open := open mod maxl + 1;
               q[open].state := state;
               q[open].step := step + 1;
               q[open].point := point;
               app[point,state[1],state[2],state[3],state[4],state[5],state[6]] := true;
            end;
            state[1] := v; state[point] := state[6]; state[6] := u;
                                                    {规则 6}
            if not app[point,state[1],state[2],state[3],state[4],state[5],state[6]]
            then begin
               open := open mod maxl + 1;
               q[open].state := state;
               q[open].step := step + 1;
               q[open].point := point;
               app[point,state[1],state[2],state[3],state[4],state[5],state[6]] := true;
            end;
         end;
      end;
end.
```

第三课 深度优先搜索

课题：深度优先搜索
目标：
　　知识目标：深度优先搜索的原理与实现
　　能力目标：深度优先搜索的应用
重点：深度优先搜索的应用
难点：深度优先搜索的递归与非递归的实现

板书示意：
1. 深度优先搜索的过程与框架（递归与非递归）
2. 用深度优先搜索实现“八数码问题”
3. 深宽度优先搜索的应用（欧拉回路和生日蛋糕问题）

授课过程：

一．深度优先搜索的过程

深度优先搜索所遵循的搜索策略是尽可能“深”地搜索图。在深度优先搜索中，对于最新发现的节点，如果它还有以此为起点而未搜索的边，就沿此边继续搜索下去。当节点 v 的所有边都已被探寻过，搜索将回溯到发现节点 v 有那条边的始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被发现为止。

下面我们以我们迷宫问题为例来看看深度优先搜索的过程。

在图 4 所示迷宫中，要求找到从左上角走到右下角的一条路径。

假设规则分别为向上，向左，向下，向右。那么的它的深度搜索路径就是：

A**à**B**à**…**à**Y**à**Z**à**a**à**b**à**…**à**y**à**z

可以发现，它有一个明显的特点：能走就走，如果无路可走就退回。

其实这就是深度优先搜索的特点：优先扩展深度大的节点，如果无法扩展的话，就回溯。

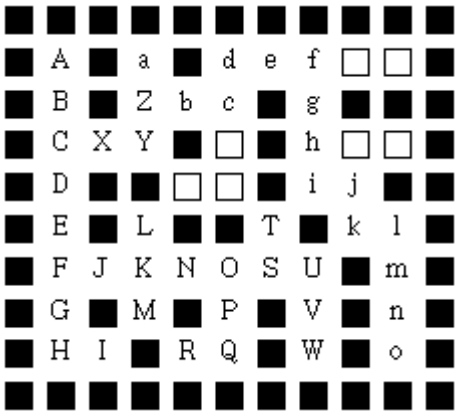


图 4 迷宫问题

二．深度优先搜索框架

```
PROCEDURE DFS-SEARCH;(算法 4)
1.   G:=G0;
2.   Open:=(Source)          ;
3.   Closed:=nil;
4.   Repeat
5.    IF OPEN=nil then Exit(Fail);
6.    n:=FIRST(Open);Remove(n,Open);
7.    Add(n,Closed);
8.    If n=Goal then Exit(Success);
9.    mi:=Expand(n);
10.   对 mi，舍去在图中已经出现的节点
11.   将 mi 加入到 Open 表的表头;
```

```
12.   Add(mi,G);
13.  Until false;
```

由此可以看出，深度优先搜索算法与图搜索框架的差别也仅仅在于 10，11，其中原来的 13 步排序没有必要进行。

10 步的判重的目的是删除与祖先节点重复的节点。

11 步的处理是把当前 m_i 加入到 Open 的表头。其中，Open 表保存的是待扩展节点，这样，深度优先保证让深度大的节点优先扩展。

下面我们看看采用深度优先搜索来解决**八数码问题**。

具体算法：

为了较好的实现算法算法 4 中的插入表头的操作，我们引入栈这个数据结构,其中 Open 表示栈顶。具体伪代码如下：

```
PROCEDURE Recursive;
1.  For x:=1 to 4 do [
2.    New:=Move(List[Open],4);          {对 N 节点使用第 x 条规则，得到 New}
3.    If not_Appear(New,List) then [    {如果 New 没有在 List 中出现}
4.      Open:=Open+1;List[step]:=New;   {插入当前节点}
5.      If List[Step]=Goal then GetOut; {判断是否为目标节点}
6.      Recursive;                      {递归搜索下一层}
7.      Open:=Open-1;                   {Open-1}
8.    ];
9.  ]

PROCEDURE DFS
1.  List[1]:=Source;
2.  Best:=maxint;
3.  Open:=1;
4.  Recursive;                          {递归搜索}
5.  输出
```

```
program No8_DFS;                                    {八数码的深度优先搜索算法}

Const
  Dir : array[1..4,1..2]of integer = ((1,0),(-1,0),(0,1),(0,-1));
  maxN = 15;                                        {可以承受的最大深度}
Type
  T8No = array[1..3,1..3]of integer;
  tList = record
    state : T8No;
    x0,y0 : integer;
  end;
Var
  Source,Target : T8No;
  List,Save : array[0..maxN]of tList;               {综合数据库，最优解路径}
  Open,Best : integer;

  procedure GetInfo;                                {见程序 1}
  Function Same(A,B : T8No):Boolean;                {见程序 1}
  function Not_Appear(New : tList):boolean;         {见程序 1}
  procedure Move(N : tList;d : integer;var ok : boolean;var New : tList);
                                                    {见程序 1}
```

```
  procedure GetOutInfo;                                    {输出}
  var i,x,y : integer;
  begin
     writeln('total = ',best);
     for i:=1 to best+1 do begin
        for x:=1 to 3 do begin
           for y:=1 to 3 do write(Save[i].State[x,y],' ');
           writeln;
        end;
        writeln;
     end;
  end;

  Procedure Recursive;                                     {递归搜索过程}
  Var i : integer;
      New: tList;
      ok : boolean;
  Begin
     If Open-1>=Best then exit;
     If Same(List[Open].state,Target) then begin           {如果找到解，保存当前最优解}
        Best:=Open-1;
        Save:=List;
     end;
     For i:=1 to 4 do begin                                {依次选用规则}
        Move(List[Open],i,OK,New);
        if ok and not_Appear(New) then begin               {如果没有重复}
           inc(open);                                      {插入综合数据库}
           List[Open]:=New;
           Recursive;                                      {继续搜索}
           dec(Open);                                      {退栈}
        End;
     End;
  End;

  procedure Main;                                          {搜索主过程}
  var x,y : integer;
  begin
     List[1].state:=Source;                                {初始化}
     for x:=1 to 3 do
        for y:=1 to 3 do if Source[x,y]=0 then begin
           List[1].x0:=x;
           List[1].y0:=y;
        end;
     Best:=MaxN;
     Open:=1;
     Recursive;                                            {开始搜索}
     If Best=maxint then writeln('No answer')
     Else GetOutInfo;
  end;

Begin
  Assign(Input,'input.txt');ReSet(Input);
  Assign(Output,'Output.txt');ReWrite(Output);
  GetInfo;
  Main;
```

```
  Close(Input);Close(Output);
End.
```

上面的八数码程序利用到了递归来实现，其实深度优先搜索还有一种无需递归的实现方式，下面我们介绍一下深度优先的一般实现方法：递归算法和非递归算法。

递归算法伪代码：

```
procedure DFS_ recursive(N);
1.  if N=target then  更新当前最优值并保存路径;
2.  for r:=1 to  规则数  do [
3.    New:=Expand(N,r)
4.    if  值节点 New 符合条件  then [
5.      产生的子节点 New 压入栈;
6.      DFS_recursive(i+1);
7.      栈顶元素出栈;
8.    ]
9.  ]
```

```
program DFS;
1.  初始化;
2.  DFS_recursive(N);
```

非递归算法伪代码：

```
procedure Backtracking;
1.  dep:=dep-1;
2.  if dep>0 then  取出栈顶元素
3.               else p:=true;
```

```
program DFS;
1.  dep:=0;
2.  Repeat
3.    dep:=dep+1;
4.    j:=0;brk:=false;
5.    Repeat
6.      j:=j+1;
7.      New=Expand(Track[dep],j);
8.      if New  符合条件  then [
9.        产生子节点 New 并将其压栈;
10.       If  子节点 New=target then  更新最优值并出栈
11.                                  else brk:=true;
12.     ]
13.     else if j>=规则数  then Backtracking
14.                            else brk:=false;
15.   Until brk=true
16. Until dep=0;
```

两种方式本质上是等价，但两者也时有区别的。

1. 递归方式实现简单，非递归方式较之比较复杂；
2. 递归方式需要利用栈空间，如果搜索量过大的话，可能造成栈溢出，所以在栈空间无法满足的情况下，选用非递归实现方式较好。

例 2 欧拉回路

给定一个无向图，找出一条经过每条边有且仅有一次的路径，这条路径就叫做欧拉路径。如果这条路径的起点和终点是同一个点的话，这条路径就叫做欧拉回路。

分析：

确定一个图是否存在欧拉回路已经有数学方法。下面我们简单介绍一下：

如果这个图连通且每一个点的度数都为偶数，那么这个图存在一个欧拉环。

虽然很容易确定是否存在欧拉环，但是找出他们却不是一件很容易的事。下面我们看一

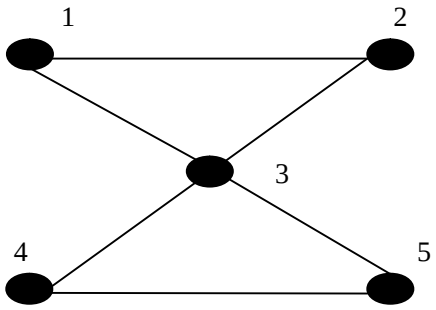


图 5 原图

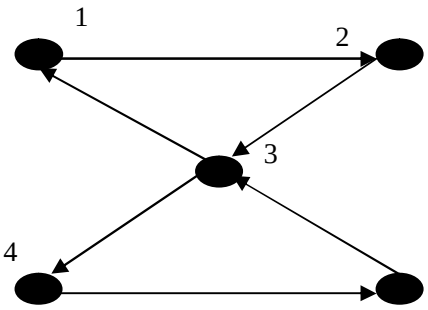


图 6 欧拉回路

个简单的例子：

如图，左图为原图，右图就是我们要求的欧拉回路，我们可以用(v1,v2,v3,v4,v5,v3,v1)表示。仔细分析一下右图的欧拉回路我们可以发现是由两条简单的回路（简单的回路就是每个顶点只经过一次的回路）构成的。由此我们可以得到如下性质：

每个回路（包括简单回路）都可以分解成若干个相连的简单回路。

于是我们的任务也就变成求简单回路了。以下算法也就是基于这个性质的，即递归地寻找一条以该点为端点的回路。

算法描述如下：

```
PROCEDURE find_circuit(node i);              {递归找圈}
{Circuitpos 表示当前的输出序列长度，Circuit 表示输出路径.}
1. if node i 没有邻接点 then [
2.  circuit(circuitpos) = node i;            {加入序列}
3.  circuitpos = circuitpos + 1;
4. ]
5. else
6.  while (node i 有邻接点) do [
7.    任意取出一个邻接点 j;
8.    delete_edges (node j, node i);         {删除边(ij)}
9.    find_circuit (node j);                 {继续递归找圈}
10. ]
11. circuit(circuitpos) = node i             {加入序列}
12. circuitpos = circuitpos + 1
PROCEDURE Euler
1. circuitpos = 0;                           {初始化}
2. find_circuit(node 1);                     {递归求解}
```

这个算法的时间复杂度是 O(e+n),其中 e 是边数，n 是点数。

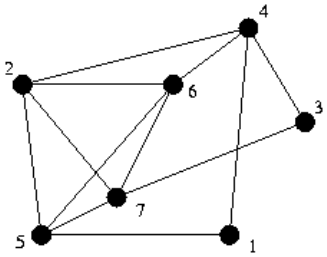
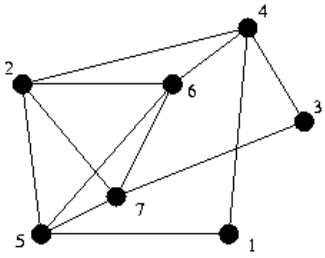


图 7 求欧拉图

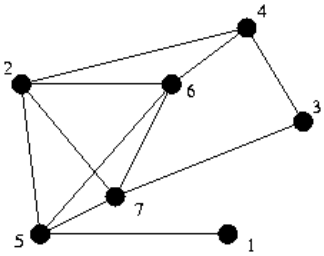
下面我们通过例子，看看搜索过程中栈的变化。图 7 为原图，我们得到的欧拉回路是:

(V1, V5, V7, V6, V4, V3, V7, V2, V6, V5, V2, V4, V1)

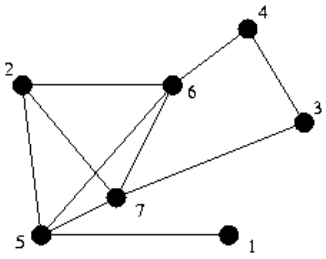
图 8，详尽的描述搜索过程中栈的变化，其中图的顺序是从左到右，从上到下.



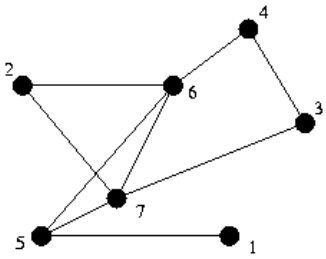
栈: 1
Location: 4
输出:



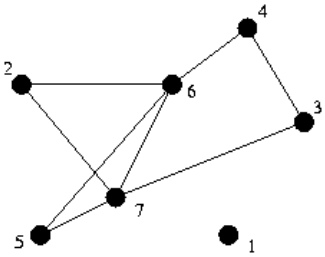
栈: 1
Location: 4
输出:



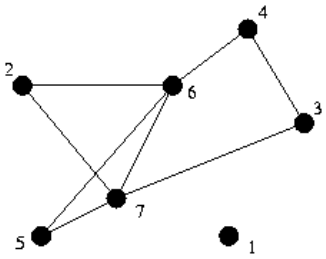
栈: 1 4
Location: 2
输出:



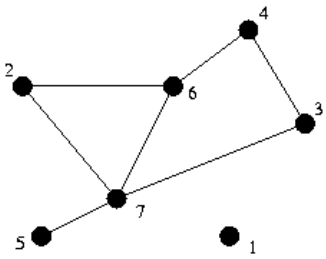
栈: 1 4 2
Location: 5
输出:



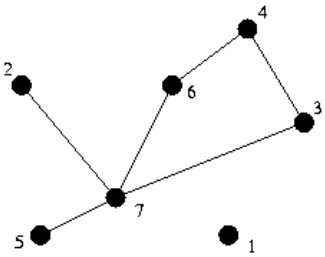
栈: 1 4 2 5
Location: 1
输出:



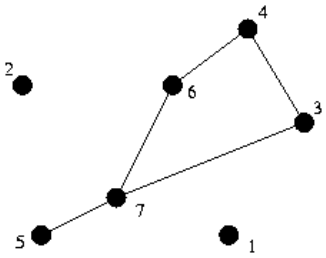
栈: 1 4 2
Location: 5
输出: 1



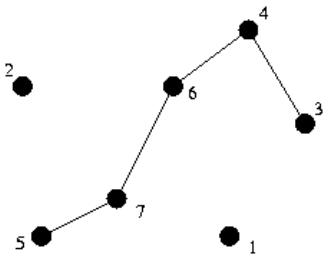
(A)栈: 1 4 2 5
Location: 6
输出: 1



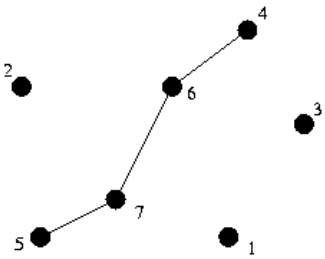
(B)栈: 1 4 2 5 6
Location: 2
输出: 1



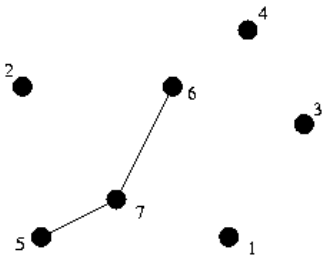
(C)栈: 1 4 2 5 6 2
Location: 7
输出: 1



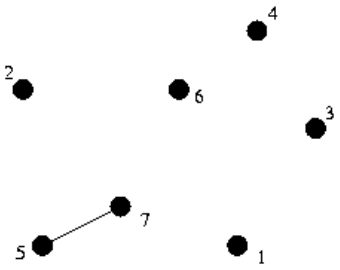
(D)栈: 1 4 2 5 6 2 7
Location: 3
输出: 1



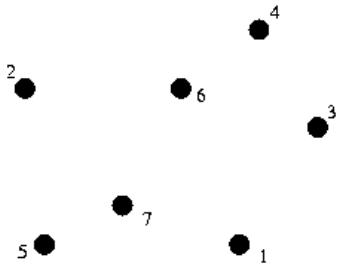
(E)栈: 1 4 2 5 6 2 7 3
Location: 4
输出: 1



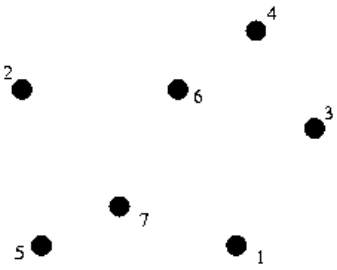
(F)栈: 1 4 2 5 6 2 7 3
4 Location: 6
输出: 1



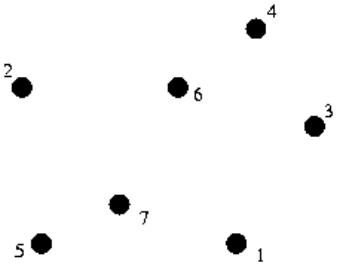
(G)栈: 1 4 2 5 6 2 7 3 4 6
Location: 7
输出: 1



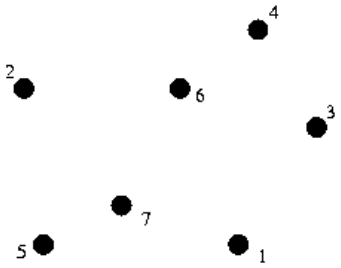
(H)栈: 1 4 2 5 6 2 7 3 4 6 7
Location: 5
输出: 1



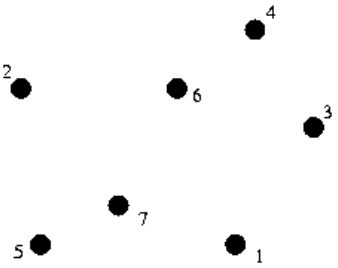
(I)栈: 1 4 2 5 6 2 7 3 4 6 7
Location:7
输出: 1 5



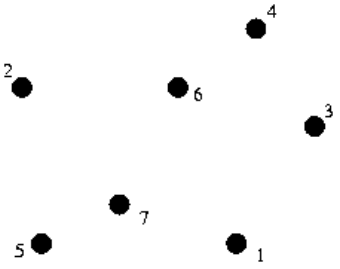
(J)栈: 1 4 2 5 6 2 7 3 4
Location: 6
输出: 1 5 7



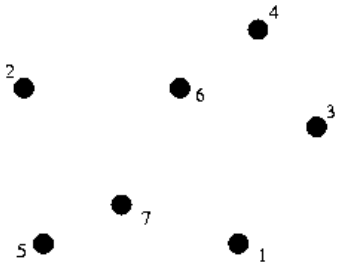
(K)栈: 1 4 2 5 6 2 7 3
Location: 4
输出: 1 5 7 6



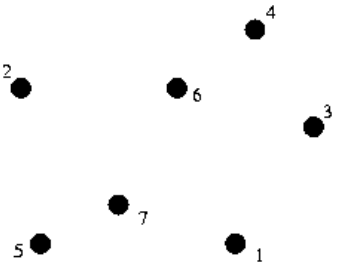
(L)栈: 1 4 2 5 6 2 7
Location: 3
输出: 1 5 7 6 4



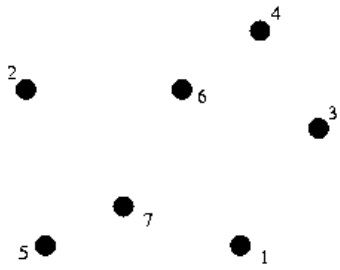
(M)栈: 1 4 2 5 6 2
Location: 7
输出: 1 5 7 6 4 3



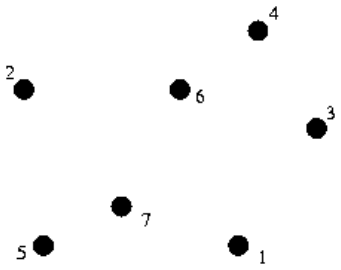
(N)栈: 1 4 2 5 6
Location: 2
输出: 1 5 7 6 4 3 7



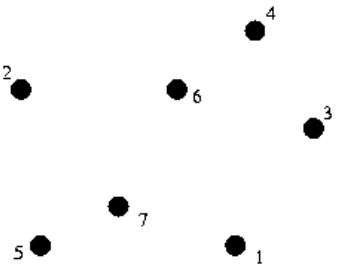
(O)栈: 1 4 2 5
Location: 6
输出: 1 5 7 6 4 3 7 2



(P)栈: 1 4 2
Location: 5
输出: 1 5 7 6 4 3 7 2 6



(Q)栈: 1 4
Location: 2
输出: 1 5 7 6 4 3 7 2 6 5



(R)栈: 1
Location: 4
输出: 1 5 7 6 4 3 7 2 6 5 2

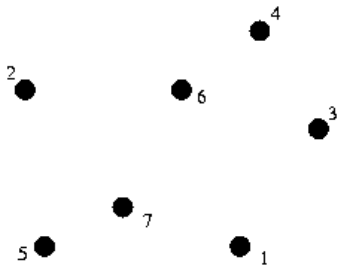


图 8 搜索欧拉回路的搜索过程

A**à**B**à**C**à**D**à**E**à**F**à**
G**à**H**à**I**à**J**à**K**à**L**à**
M**à**N**à**O**à**P**à**Q**à**R**à**S

(S)栈:
Location:
输出: 1 5 7 6 4 3 7 2 6
5 2 4 1

程序清单

下面我们给出两种实现方式，一种是非递归方式，一种是递归方式。

```
program euler_nonrecursion;                          {非递归方式}
const     maxn = 100;
var
   e,_a,_b,nsequence,i,j,n,dep : integer;            {e 表示边数}
                                                     {nsequence 记录输出序列的长度}
                                                     {dep 为栈顶指针}
   ok : boolean;
   map : array[0..maxn,0..maxn]of integer;           {记录无向图}
   stack,sequence : array[0..maxn*maxn]of integer;   {栈，输出序列}
begin
   fillchar(map,sizeof(map),0);

   assign(input,'input.txt');reset(input);           {读入}
   assign(output,'output.txt');rewrite(output);
   read(n,e);
   for i:=1 to e do begin                            {以边的方式读入}
      read(_a,_b);
      inc(map[_a,_b]);
      inc(map[_b,_a]);
   end;
   dep:=1;stack[1]:=1;nsequence:=0;
   while dep>0 do begin
      j:=stack[dep];
      ok:=false;
      for i:=1 to n do if map[j,i]>0 then begin      {是否存在邻接点}
         dec(map[j,i]);
         dec(map[i,j]);
         ok:=true;
         inc(dep);stack[dep]:=i;                     {入栈}
         break;
      end;
      if not ok then begin
         inc(nsequence);
         sequence[nsequence]:=stack[dep];            {加入输出序列}
         dec(dep);
      end;
   end;
   for i:=1 to nsquence do write(sequence[i],' ');   {输出}
```

```
   writeln;
end.
```

```
program euler_recursion;                                    {递归方式,注释同上}
const      maxn = 100;
var
   e,_a,_b,nsequence,i,j,n : integer;
   map : array[0..maxn,0..maxn]of integer;
   sequence : array[0..maxn*maxn]of integer;
   procedure recursion(location : integer);
   var ok : boolean;
        I : integer;
   begin
      Ok:=false
      for i:=1 to n do if map[location,i]>0 then begin
         dec(map[location,i]);dec(map[i,location]);
         ok:=true;
         recursion(i);
      end;
      if not ok then begin
         inc(nsequence);
         sequence[nsequence]:=location;
      end;
   end;
begin
   fillchar(map,sizeof(map),0);
   assign(input,'input.txt');reset(input);
   assign(output,'output.txt');rewrite(output);
   read(n,e);
   for i:=1 to e do begin
      read(_a,_b);
      inc(map[_a,_b]);
      inc(map[_b,_a]);
   end;
   nsequence:=0;
   recursion(1);
   for i:=1 to nsquence do write(sequence[i],' ');
   writeln;
end.
```

例 3 生日蛋糕(Cake)(NOI99 第一试第三题)

7 月 17 日是 Mr.W 的生日，ACM-THU 为此要制作一个体积为 Nπ 的 M 层生日蛋糕，每层都是一个圆柱体。

设从下往上数第 i(1<=i<=M)层蛋糕是半径为 R_i，高度为 H_i 的圆柱。当 i<M 时，要求 $R_i>R_{i+1}$ 且 $H_i>H_{i+1}$。

由于要在蛋糕上抹奶油，为尽可能节约经费，我们希望蛋糕外表面（最下一层的下底面除外）的面积 Q 最小。

令 Q= Sπ

请编程对给出的 N 和 M，找出蛋糕的制作方案（适当的 R_i 和 H_i 的值），使 S 最小。

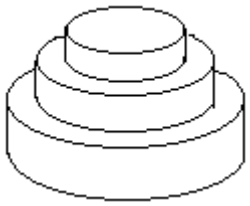


图 9　蛋糕

（除 Q 外，以上所有数据皆为正整数）

输入

有两行，第一行为 N（N<=10000），表示待制作的蛋糕的体积为 Nπ；第二行为 M(M<=20)，表示蛋糕的层数为 M。

输出

仅一行，是一个正整数 S（若无解则 S=0）。

附：圆柱公式

体积 $V=\pi R^2H$

侧面积 $A'=2\pi RH$

底面积 $A=\pi R^2$

分析：

综合数据库：

我们设当前为第 i 层蛋糕，上一层蛋糕的半径和高度为 R,H,当前的面积为 S,余下的体积为 V.

我们可以用(i,R,H,V,S)表示一个状态.

初始状态为$(1,R_1,H_1,N-R*R*H,R*R+2*R*H)$,$R_1,H_1$为第一层的半径和高，R*R*H 表示第一层的体积，R*R+2*R*H 表示底面积加侧面积,即当前总面积.

目标状态$(M,R_M,H_M,0,S_M)$,其中 R_M,H_M 为最后一层的半径和高，S_M 表示总面积．于是我们的目标是找到一条路径从初始节点到任意目标节点，并且 S_M 最小.

产生式规则：

$(i,R_i,H_i,V_i,S_i) \rightarrow (i+1,R_{i+1},H_{i+1},V_{i+1},S_{i+1})$

其中必须满足:

1. $R_i>R_{i+1}$
2. $H_i>H_{i+1}$
3. $V_{i+1}=V_i-R_{i+1}*R_{i+1}*H_{i+1}$
4. $S_{i+1}=S_i+2*R_{i+1}*H_{i+1}$

控制策略

深度优先搜索．下面是采用递归形式的程序.

```
{$R-,Q-,S-,G+}
{$M 32767,0,655360}
Const Inp                = 'input.txt';
      MaxR               = 50;                    {R 最大值}
      MaxH               = 150;                   {H 最大值}
Var
  N,M,Best,R,H,S,V,O     : Integer;               {循环变量或读入量}

  Function Min(A,B:Integer):Integer;              {取 min}
    Begin
      if A>B then Min:=B Else Min:=A
    End;

  Procedure Try(St,LastR,LastH,LastV,LastS:Integer);   {递归过程,说明见伪代码}
    Var R,H,NowS,NowV                : Integer;
    Begin
      if LastS>Best then Exit;
      if St>0 then For R:=LastR Downto St do
        Begin
          O:=Min(LastV div Power2[R],LastH);
          For H:=o Downto St do
            Begin
              NowV:=LastV-Power2[R]*H;NowS:=2*R*H;
              Try(St-1,R-1,H-1,NowV,LastS+NowS);    {递归搜索}
            End;
        End
      Else if LastV=0 then Best:=LastS
```

```
  End;

Begin
  Assign(Input,Inp);Reset(Input);Readln(N,M);Close(Input);
  Best:=MaxInt;
  For R:=M to Trunc(Sqrt(N)) do
    For H:=N div Power2[R] Downto M do
      Begin
        V:=N-Power2[R]*H;S:=2*R*H;
        Try(M-1,R-1,H-1,V,S+Power2[R]);                {递归搜索}
      End;
  Writeln(Best);
End.
```

第四课 启发式搜索

课题：启发式搜索
目标：
知识目标：启发式搜索的原理与实现
能力目标：启发式搜索的应用
重点：启发式搜索的应用
难点：启发式函数的选择

板书示意：
1. **启发式搜索的过程与框架**
2. **用启发式搜索的性质**
3. **用启发式搜索实现“八数码问题”**
4. **搜索算法的比较**

授课过程：

启发式搜索是利用问题拥有的启发信息来引导搜索，达到减少搜索范围，降低问题复杂度的目的。这种利用启发信息的搜索过程称为启发式搜索方法。在研究启发式搜索方法时，先说明一下启发信息的应用，启发能力度量及如何获得启发信息这几个问题。

一．评价函数

为了提高搜索效率，引入启发信息来进行搜索，在启发式搜索过程中，要对 OPEN 表进行排序，这就需要有一种方法来计算待扩展节点有希望通向目标节点的程度，我们总希望能找到最有希望通向目标节点的待扩展节点优先扩展。一种常用的方法就是定义评价函数 f(Evaluation funtion)对各个节点进行计算，其目的就是估算出“有希望”的节点来。如何定义一个评价函数呢？通常可以参考如下原则：

一个节点处在最佳路径上的概率；

求出任意一个节点与目标节点集之间的距离度量或差异度量；

根据格局（博弈问题）或状态特点来打分。

通常，我们可以将评价函数 f 拆分成两个部分；即对节点 n，有 f(n)=g(n)+h(n)。其中函数 g 是节点 n 对历史（初始节点 s）的评价值，函数 h 是节点 n 对未来（目标节点 t）的评价值。严格地说：

f(n)规定为对从初始节点 s，通过约束节点 n，到达目标节点 t 上，最小耗散路径(最佳路径)的耗散值 $f^*(n)$的估计值。其中，

g(n)规定为对从 s 到 n 最小耗散路径(最佳路径)的耗散值 $g^*(n)$的估计值。

h(n)规定为对从 n 到 t 最小耗散路径(最佳路径)的耗散值 $h^*(n)$的估计值。

因此，$f^*(n)=g^*(n)+h^*(n)$就表示从初试节点出发到达目标节点的最小耗散路径(最佳路径)的耗散值。

显然，g(n)的值很容易从到目前为止的搜索树上计算出来。也就是根据搜索历史对 $g^*(n)$做出估计，显然，$g(n)\geq g^*(n)$。h(n)则依赖启发信息，通常称为**启发函数**。

回顾图搜索过程框架，对 DATA 中的节点要按某种规则进行排序，如果我们按照 f(n)递增的顺序来排列 Open 表的节点，因而优先扩展 f(n)值较小的节点，体现了好的优先搜索思想。

如在 8 数码问题中，可以用不在目标位置的数字个数作为启发函数来作为度量：

g(n)=搜索的深度。

h(n) =不在目标位置的数字个数。

利用这个评价函数 f=g+h，得出的搜索树如图 10，其中每个节点所标的数值为该节点按评价函数计算的耗散值。与图 3 比较，它少扩展了 21 个节点。

二、最佳图搜索算法（A*）

比较一下宽度优先搜索、深度优先搜索和启发式搜索，可以看出，他们的唯一区别在于选取节点扩展的顺序有所区别。换句话说，就是在节点插入到 OPEN 表中的方式不一样，决定了选取节点的次序不同。

宽度优先是将当前扩展的节点插入到OPEN表的尾部,OPEN表组织为一个队列的形式。这样，先产生的节点先扩展，也就是深度浅的节点优先扩展。

宽度优先是将当前扩展的节点插入到OPEN表的最前端，OPEN表组织为一个堆栈的形式。这样，后产生的节点先扩展，也就是深度大的节点优先扩展。

启发式搜索是按照启发函数的值将节点插入到OPEN表中，启发函数值较小的节点优先扩展。启发式搜索吸取了宽度优先和深度优先的优点，它是按照启发信息，“智能”地选取比较好的节点优先扩展。

如果在搜索过程中，始终满足 $h(n) \leqslant h^*(n)$，则启发式搜索算法为最佳图搜索算法（A*）。A*算法找到的解，为问题的最优解。因为每次扩展都是选取耗散值最小的节点优先扩展，而 $h(n) \leqslant h^*(n)$可以确保最优解一定在扩展的搜索树以内。

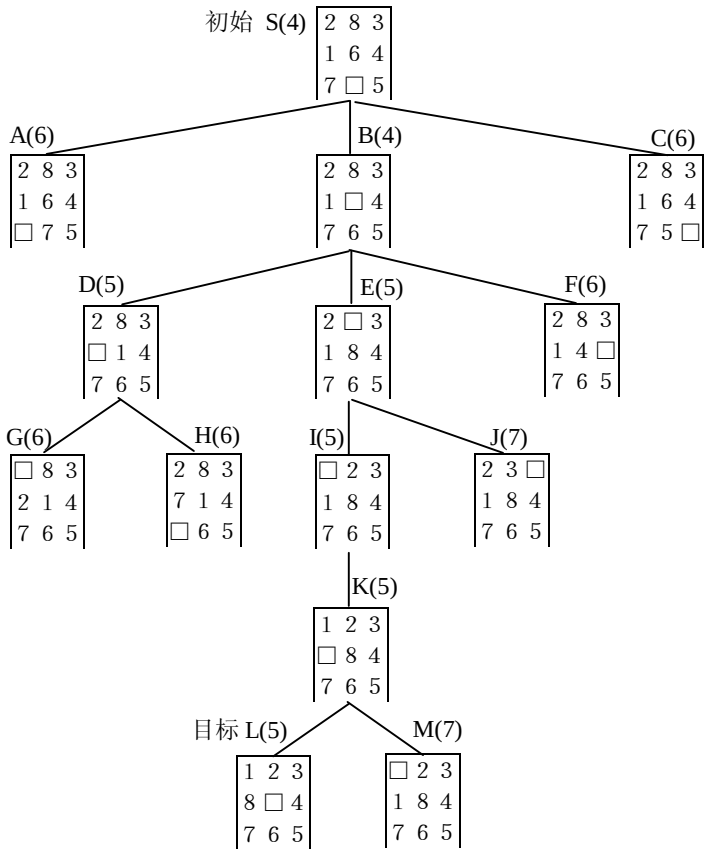


图 10 八数码问题启发搜索过程

启发式搜索的算法框架：

```
PROCEDURE A*-SEARCH;(算法 5)
1.    Open:=(s),f(s)=g(s)+h(s);        {把 s 加入 Open,并计算 s 耗散值的估计值}
2.    Closed:=nil;                     {置 CLOSED 表为空}
3.    Repeat
4.     IF Open=nil then Exit(Fail);    {如果 Open 为空，输出无解并退出}
5.     n:=FIRST(OPEN);Remove(n,Open);{取出 Open 表首节点,从 Open 删除 n}
6.     Add(n,Closed);                  {把 n 加入到 Closed 表}
7.     If n=Goal then Exit(Success); {如果 n 为目标，就退出并输出解}
```

8. m_i:=Expand(n);计算 f(n,m_i)=g(n,m_i)+ h(n,m_i)
{扩展 n，得到后继 m_i,计算从 s 出发经过节点 n 达到目标节点耗散值的估计值}
9. 标记和修改 m_i 的指针 { m_i=m_j∪m_j∪m_j}
.Add(m_j,Open)，标记 m_j 到 n 的指针;
.if f(n,m_k)< f(m_k)then f(m_k):= f(n,m_k)，标记 m_k 到 n 的指针;
. if f(n,m_l)< f(m_l)then f(m_l):= f(n,m_l)，标记 m_l 到 n 的指针，Add(m_l,Open);
10. 按递增值，重排 OPEN（相同最小值可根据搜索树中的最深节点来解决）；
11. Until false;

三．A*算法满足的一些性质

性质 1：若有两个 A*算法 A1 和 A2,R 若 A2 比 A1 有较多的启发信息（即对所有非目标接点均有 h2(n)> h1(n)）,则在具有一条从 s 到 t 的隐含图上，搜索结束时，由 A2 所扩展的每一个节点，也必定由 A1 所扩展，即 A1 扩展的节点数至少和 A2 一样多。

根据这个性质可知，使用启发函数 h(n)的 A*算法，比不使用 h(n)（h(n)=0）的算法，求得最佳路径时扩展的节点要少，使用 h(n)启发信息强的 A*算法比使用 h(n)启发信息弱的 A*算法扩展的节点数要少。因此，我们在考虑问题时，在满足 h(n)≤h^*(n)的条件下，要尽量扩大 h 函数的值。

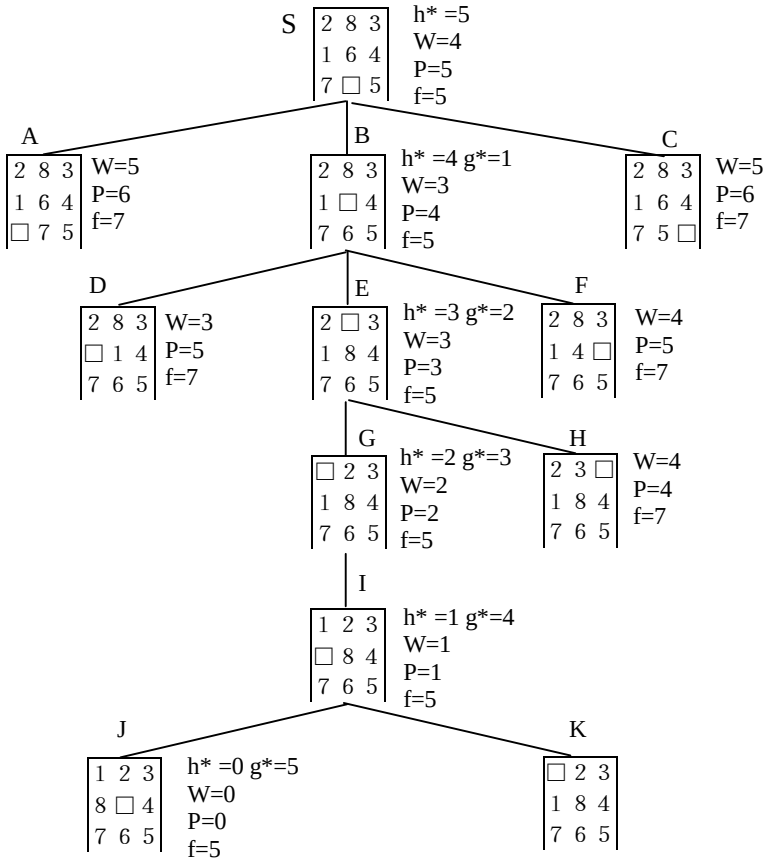


图 11 八数码问题启发搜索过程

对于八数码问题，如果我们采用每个不在位数码到目标位置的距离之和（记为 P(n)）作为启发函数，则显然要比用不在位数码的个数（记为 W(n)）作为启发函数要强一些。下面是用 h(n)=P(n)作为启发函数的搜索树，可以看出，只扩展了 12 个节点。

性质 2：若存在初始节点 s 到目标节点 t 的路径，则 A*必能找到最佳路径。如果 A*选作扩展的任一节点 n，有 f(n)≤f^*(s)。

性质 3：若 h(n)满足单调限制条件，也就是说启发函数 h 满足单调性，则 A*扩展了节点 n 之后就已经找到了到达节点 n 的最佳路径。即若 A*选 n 来扩展，在单调限制条件下有 g(n)=g^*(n)，其由 A*所扩展的节点序列，其 f 值也是非递减的，即 f(n_i)≤f(n_j)。

由性质 3 可知，设 A*扩展了节点 n’，则 $f(n') \leqslant f(n) \leqslant f^*(s)$;再设搜索进程的某一时刻扩展的节点中，扩展前 f 值最大的为节点 m，$f(m) \leqslant f^*(s)$。这时对在 Open 中的节点，若 $f(n_i)<f(m)$，则 n_i 必定被扩展几次，因为 $f(n_i)< f^*(s)$必成立。因此 n_i 留在表中，目标节点 t 就不能从 Open 表中取出来，即使在扩展 m 之后，扩展全部满足 $f(n_i)<f(m)$的节点也有用。如果这样的节点有多个，扩展的顺序不以 f 排列，而以 g 的大小排列，则在扩展期间，同一个节点就不会被扩展多次了。利用这点可以修正 A*算法，提高其搜索效率。

下面采用 A*算法解决八数码问题：

算法描述如下：

```
1．List[1]=source; Open:=1;                {加入初始节点,List 为扩展节点的表}
2．Repeat
3．    Index:=First(List);                 {取出一个未扩展的节点}
4．    If Index=0 then Exit(Fail);
5．    N:=List[List];
6．    If N=Goal then  输出;
7．    For x:=1 to 4 do [
8．       New:=Move(N,4);                  {对 N 节点使用第 x 条规则，得到 New}
9．       New.Father:=Index;
10．          New.dep:=N.dep+1;
11．          New.H:=H(New.State);
12．          New.F:=New.Dep+New.H;
13．          Add(New,List);               {加入 New}
14．       ]
15．    Until (Index=0) or  找到解;
```

```
program No8_A; {八数码的 A*算法}
Const
  Dir : array[1..4,1..2]of integer
      = ((1,0),(-1,0),(0,1),(0,-1));
  N=10000;

Type
  T8No = array[1..3,1..3]of integer;

  TList = record
    Father : integer;                    {节点的父指针}
    dep : byte;                          {扩展的深度}
    f : byte;                            {启发函数值}
    expanded : boolean;                  {该节点是否已经扩展的标志}
    X0,Y0 : byte;
    State : T8No;
  end;

Var
  Source,Target : T8No;
  List : array[0..10000] of TList;
  Open,Best : integer;  {无 Closed}
  Answer : integer;
  Found : Boolean;

procedure GetOutInfo;   {见程序 1}
```

```
procedure Move(N:tList;d:integer;var ok:boolean;var New:tList); {见程序 1}
procedure GetInfo;                      {见程序 1}
procedure Initialize;                   {见程序 1}
Function Same(A,B : T8No):Boolean;      {见程序 1}
procedure Add(New : tList);             {插入新节点}
  var i : integer;
  begin
    for i:=1 to Open do
        if Same(List[i].State,New.State) then   {如果出现重复节点}
      if List[i].f<New.f then exit    {选择两者中一个较优的保存}
      else if List[i].f>New.f then begin
        List[i]:=New;
        exit;
      end;
    Inc(Open);      {如果没有和以前的扩展的节点重复，就加入 Open}
    List[Open]:=New;
  end;

function H(S : T8NO):integer;
  var x,y,i : integer;
begin
    i := 0;
    for x:=1 to 3 do
      for y:=1 to 3 do if S[x,y]<>Target[x,y] then inc(i);
    H:=i;
  end;

  function First:integer;   {从 List 中选择一个没有扩展的 F 值最小的节点}
  var min,i,j : integer;
  begin
    min:=maxint;j:=0;
    for i:=1 to Open do if not List[i].expanded and (List[i].F<min) then begin
      min:=List[i].F;j:=i;
    end;
    First:=j;
  end;

  procedure Expand(Index : integer;var N : tList);  {扩展 N }
  var i : integer;
      New : tList;
      OK : boolean;
  Begin
    N.expanded:=true;
    If Same(N.State , Target) then begin
     Found := true;
     Best :=N.Dep;
     Answer:=Index;
     Exit;
    end;
    For i := 1 to 4 do begin
      Move(N,i,OK,New);
```

```
      if not ok then continue;
      New.Father := Index;
      New.Dep :=N.dep + 1;
      New.f:=New.Dep+H(New.State);
      New.expanded:=false;
      Add(New);
    end;
  end;

  procedure Main;
  var
 Index : integer;
  begin
    Repeat
      Index:=First;{取出 F 值最小的节点}
      if Index<>0 then Expand(Index,List[Index]){扩展首节点}
                  else break;
    Until Found;
    If Found then GetOutInfo {有解}
             else Writeln('No answer');{无解}
  end;

Begin
  Assign(Input,'input.txt');ReSet(Input);
  Assign(Output,'Output.txt');ReWrite(Output);
  GetInfo;
  Initialize;
  Main;
  Close(Input);Close(Output);
End.
```

五、各种搜索算法的关系

综合深度优先算法、宽度优先算法和启发式搜索，可以看出它们有联系非常紧密的联系。唯一区别在于选取节点扩展的先后顺序不同而已。其中宽度优先搜索是启发式搜索 H=0 的极端情形，而 A*算法是符合 H≤H*时的 A 算法。

各种搜索算法的复杂度见下表：

	空间复杂度	时间复杂度	实现复杂度	思维复杂度
深度优先搜索	*	***	*	*
宽度优先搜索	***	***	*	*
A*搜索	**	**	***	***
A 搜索	**	*	**	**

第五课 搜索的优化

课题：搜索的优化
目标：
　　知识目标：优化的方法，剪枝和改变搜索的对象
　　能力目标：利用各种优化方法提高搜索的效率
重点：剪枝策略，最优化剪枝和可行性剪枝
难点：剪枝条件的选取

板书示意：
1. 剪枝（可行性剪枝和最优化剪枝）
2. 生日蛋糕问题的优化
3. 剪枝的原则（邮票问题的应用）
4. 可行性剪枝（分数分解问题）
5. 选取不同的搜索对象（汽车问题）
6. 应用实例（列车调度、埃及分数）

授课过程：

搜索算法其实质相当于穷举，其时间复杂度是相当高的，因此，如果不对 搜索加以优化，搜索的效率将非常底，效果自然也不理想，本节将着重讨论一下搜索的优化问题。尽可能的使搜索达到：**准确性、高效性、可推广**。

下面就竞赛中的几个问题着重介绍一下优化的方法：

1. 利用剪枝条件，减少搜索对象。
2. 选择适合的搜索对象，提高搜索效率。

一、剪枝

我们知道，搜索的进程可以看作是从树根出发，遍历一棵搜索树的过程。所谓剪枝，就是通过某种判断条件，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”。

图 12 是一个求最短路径扩展的搜索树。描述了剪枝的过程。

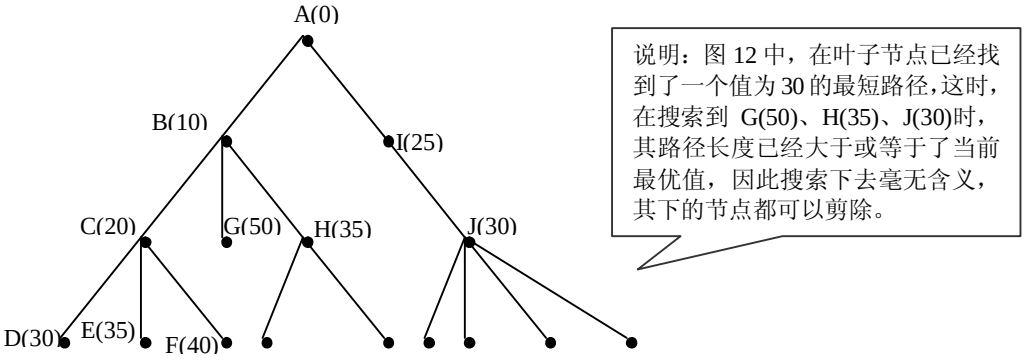


图 12 搜索树的剪枝图

例 4 生日蛋糕(Cake)的优化。

试题描述见例 3。

算法如下：

首先确定最底层的半径和高(R_1,H_1)，然后根据题目给出的要求从下往上递归搜索余下的

M-1 层。

算法 1:

Recursive - Search (i,R_i,H_i,S_i,V_i)

1. If i=M then 更新最优值
2. else
3. for R_{i+1} ß R_i-1 downto 1
4. for H_{i+1} ß H_i-1 downto 1 [
5. S_{i+1} ß S_i+2*R_{i+1}*H_{i+1}
6. V_{i+1} ß V_i-R_{i+1}*R_{i+1}*H_{i+1}
7. Recuresive - Search (i+1,R_{i+1},H_{i+1},S_{i+1},V_{i+1})
8.]

Problem-Cake

1. for R_1 ß n downto m do
2. for H_1 ß sqrt(n-R_1*R_1) downto m do [
3. S_1=2*R_1*H_1+R_1*R_1
4. V_1=n-R_1*R_1*H_1
5. Recursive - Search (1,R_1,H_1,S_1,V_1)
6.]

对上述算法进行分析，有很大剪枝的余地。

为什么会出现这种情况呢？

R_i,H_i 分别等于此层可能的最大半径与最大高，由于 R_i,H_i 都是递减的，所以 R_i,H_i 分别为上一层的实际半径与高减 1。假设当前层为 i,则以上还有 M-i+1 层。FS_i 表示估计的余下的最小测面积,V_i 表示 当前余下体积。S_i 表示原有的面积。

如果当前面积加上预测可能的最大面积小于已经得到的最优面积，就可以剪掉当前枝。即:如果 $FS_i+S_i<=BestS$，就放弃继续搜索。

$FS_i = 2*\Pi*\sum_{k=i}^{M} R_k * H_k \qquad V_i = \Pi*\sum_{k=i}^{M} R_k * R_k * H_k$

这个函数的最值比较难求，而且求和的计算量很大，那么，让我们看一下下面的结果：

我们可以把 FS_i 的面积看成一个圆锥的面积{R_i 依次减小是圆锥的性质}。设 $S2_i$ 为体积的圆柱的测面积。

由同体积的圆柱的测面积一定比圆锥的小可以知道，满足 H<H*的最优化剪枝的条件(启发函数的必要条件)，而且 $S2-FS_i$ 的值比较小，计算量也十分小，所以我们就可以以圆柱的测面积近似的作为最小面积。

$FS_i \approx V_i/(R_{i-1}*R_{i-1})*R_{i-1} = V_i/R_{i-1}$

加入这条剪枝后，算法的效率可以提高许多。

算法 2

Recursive - Search (i,R_i,H_i,S_i,V_i)

1. *If $V_i/R_{i-1}+S_i$>=最优值 then exit;*
2. If i=M then 更新最优值
3. else
4. for R_{i+1} ß R_i-1 downto 1
5. for H_{i+1} ß H_i-1 downto 1 [
6. S_{i+1} ß S_i+2*R_{i+1}*H_{i+1}
7. V_{i+1} ß V_i-R_{i+1}*R_{i+1}*H_{i+1}
8. Recuresive - Search (i+1,R_{i+1},H_{i+1},S_{i+1},V_{i+1})
9.]

Problem-Cake

1. for R_1 ß n downto m do
2. for H_1 ß sqrt(n-R_1*R_1) downto m do [
3. S_1=2*R_1*H_1+R_1*R_1
4. V_1=n-R_1*R_1*H_1

5. Recursive - Search (1,R_1,H_1,S_1,V_1)
6.]

是不是这个问题再找不到其它剪枝条件了呢？仔细分析，我们可以发现此算法并不是最优的。我们发现某些情况下，由当前的 R_i,H_i,V_i 找不到任何一种满足要求的蛋糕。于是我们可以由此得到另外两条剪枝条件。

剪枝 1 如果 $V_i<\sum_{K=1}^{i} K*K*K$，就没有必要搜索下去了。

因为余下的 I 层的体积 Total 最小为

最上层半径和高都为 1，$R_1=H_1=1$

第二层半径和高都为 2，$R_2=H_2=2$

……

第 i 层半径和高都为 i， $Ri=H_i=i$

Total= $\sum_{K=1}^{i} K*K*K$

剪枝 2 如果 Vi> $\sum_{k=1}^{i} Max\{S_k\}=\sum_{k=1}^{i}(R_{i-1}-k)*(R_{i-1}-k)*(H_{i-1}-k)$,那么就没有必要搜索下去了。

由于 R,H 递减，$R_{i-k}=R_{i-1}$-1-k；$H_{i-k}=H_{i-1}$-1-k；所以

$\sum_{k=1}^{i} Max\{S_k\}=\sum_{k=1}^{i}(R_{i-1}-k)*(R_{i-1}-k)*(H_{i-1}-k)$，即余下 i 层可能的最大体积。

可以发现，这个 $\sum_{k=1}^{i} Max\{S_k\}=\sum_{k=1}^{i}(R_{i-1}-k)*(R_{i-1}-k)*(H_{i-1}-k)$ 中有三个变量 i,R_{i-1},H_{i-1}

如果我们在搜索的过程中计算的话，显然要重复计算多次，白白浪费时间，但如果首先算出来，然后保存的话，须 20*100*10000=20M 的空间。时间复杂度也为 2*10^7，这样势必会影响程序的效率。但我们可以先计算其中经常引用到的一部分，而其余引用次数比较少的临时计算。于是我们可以得到算法 3:

Recursive - Search (i,R_i,H_i,S_i,V_i)

1. *If $V_i/R_{i-1}+S_i$>=最优值 then exit* *{剪枝 1}*
2. *If $Min_i>V_i$ then exit;* *{剪枝 2 }*
3. *If Max[i,R_i,H_i]<V_i then exit;* *{剪枝 3}*
4. If i=M then 更新最优值
5. else
6. for R_{i+1} ß R_i-1 downto 1
7. for H_{i+1} ß H_i-1 downto 1 [
8. S_{i+1} ß S_i+2*R_{i+1}*H_{i+1}
9. V_{i+1} ß V_i-R_{i+1}*R_{i+1}*H_{i+1}
10. Recuresive - Search (i+1,R_{i+1},H_{i+1},S_{i+1},V_{i+1})
11.]

Problem-Cake

1. for i ß 1 to sqrt(n) do $Min_i=\sum_{K=1}^{i} K*K*K$;
2. 计算 Max[i,R,H]= $\sum_{k=1}^{i}(R-k)*(R-k)*(H-k)$
3. for R_1 ß n downto m do
4. for H_1 ß sqrt(n-R_1*R_1) downto m do [

5. $S_1=2*R_1*H_1+R_1*R_1$
6. $V_1=n-R_1*R_1*H_1$
7. Recursive - Search $(1,R_1,H_1,S_1,V_1)$
8.]

显然，应用剪枝优化的核心问题是设计剪枝条件，即确定哪些枝条应当舍弃，哪些枝条应当保留的方法。好的剪枝，往往能够使程序的运行效率大大提高；否则，也可能适得其反。下面，我们来分析一下设计剪枝条件需要遵循的一些基本原则。

二．剪枝的原则

原则一：**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，是因为它能够“剪去”搜索树中的一些“无用枝条”。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，再好的剪枝都将失去意义。所以，在进行剪枝时，必须保证不剪掉包含解或最优解的枝条，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。

原则二：**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，由于是利用有解的“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四个字：**正确、高效**。

当然，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：

最优性剪枝和可行性剪枝。

三．最优化剪枝

如果可以根据当前枝条的状态，预计出从这个枝条往下搜索一定无法得到比当前解更优的解，我们可以＂剪断＂这条枝，这种剪枝就叫做最优化剪枝.

例 4 中 CAKE 问题的第一条剪枝就是最优化剪枝。对于最优化剪枝，Best 值是我们剪枝的标准，所以，Best 的值越优，我们就有可能越早的剪掉非最优的枝。如果一开始就是最优值的话，那么搜索就可在根部剪掉所有的枝了。

在利用最优化剪枝方面，有很多技巧：

1. 用贪心法获得一个局部最优解
2. 在搜索中使用一些贪心策略，使解尽快地朝最优解靠拢
3. H 函数尽可能的接近 H*。
4. H 值的计算量与误差反比定理：“H 值的计算量*误差=定值”。如果用过于复杂的 H 函数，反而浪费了搜索的时间，但又不能忽略误差，所以我们要处理好误差与计算时间的关系，要做到细中有粗，粗中有细。

不同的题目一般有不同的剪枝，这有赖于做题者对题目的认真的分析，没有一般的方法与公式。

例 5 邮票面值设计(1999 分区联赛第四题)

试题描述：

给定一个信封，最多只允许粘贴 N 张邮票，计算在给定 K（N+k<=40) 种邮票的情况下（假定所有的邮票数量都足够），如何设计邮票的面值，能得到最大 max ，使得 1－max 之间的每一个邮资值都能得到。

例如，N=3，K＝2，如果面值分别为 1 分、4 分，则在 l 分-6 分之间的每一个邮资值都能得到（当然还有 8 分、9 分和 12 分）：如果面值分别为 1 分、3 分，则在 1 分-7 分之间的每一个邮资值都能得到。可以验证当 N=3，K＝2 时，7 分就是可以得到连续的邮资最大值，所以 MAX=7，面值分别为 l 分、3 分。

样例：

INPUT

N=3 k=2

OUTPUT

1 3

MAX=7

分析：

邮票面值设计这道题中 MAX 和 N、K 难以用数学方法确定他们之间的关系，只能通过搜索来求得。所以本题的重点就放在搜索优化上，来提高搜索的效率。

搜索的过程实质上是枚举 K 种邮票的面值，然后计算这 K 种邮票对应的 MAX 值。于是可以得到如下算法：

算法 1

Procedure Search(m) 搜索第 m 种邮票的面值

1. If m = K+1 Then [
2. If 当前方案更优 Then 保存当前方案;
3. Exit;
4.]
5. For I ß A_{m-1}+1 to N＊A_{m-1}+1 do [{N*A_{m-1}+1 肯定不能有前 K 中邮票构成}
6. A_m := I;
7. Search(m+1);
8.]
9. End;

搜索的过程可以分成两步：生成 K 种邮票的面值和计算这 K 种邮票对应的 MAX 值。我们发现上面对于 MAX 的估计是很不精确的，于是有下面的方法，给定 K 种邮票的面值 A_i(i=1,2,…,K)，计算对应的 MAX 值，这个过程可以采用动态规划来解决。下面我们针对这点进行讨论。

方法是检查是否 1～MAX 之间的每种邮资值可以通过不超过 N 张邮票贴出，而邮资值为 MAX+1 时则不可以。在判断邮资值 P 是否可用不超过 N 张邮票贴出时，如果我们贴一张面值为 M 的邮票，则问题转化为判断邮资值 P-M 是否可用不超过 N-1 张邮票贴出。令 S(x) 表示贴出邮资值 x 最少使用的邮票数目，则可以建立如下递推方程：

S(0)=0，

S(x)=min{S(x-A_i)+1|i=1,2,…,K 且 x>=A_i}。

若 S(x)<=N，x=1,2,…,P，当 S(P)>N 时，则可确定 MAX=P。

下面我们设计搜索思想：显然一定有面值为 1 的邮票。可以按递增的次序来搜索各种邮票的面值，关键在于确定每一层搜索的上界。设当前搜索到第 m 层，用不超过 N-1 张前 m-1 种邮票可以得到的 MAX 值记为 MAX(m-1)，则第 m 种邮票面值的上界是 MAX(m-1)+1，否则邮资值 MAX(m-1)+1 就无法用这 m 种邮票贴出来了。

我们将搜索的过程用伪码描述如下：

算法 2

Procedure Search(m) 搜索第 m 种邮票的面值

1. If m = K+1 Then [

2. If 当前方案更优 Then 保存当前方案;
3. Exit;
4.]
5. For I ß A_{m-1}+1 to Max(M-1) do [
6. A_m := I;
7. Search(m+1);
8.]
9. End;

```
{$A+,B-,D+,E+,F-,G-,I+,L+,N-,O-,P-,Q-,R-,S+,T-,V+,X+}
{$M 16384,0,655360}

Program Stamp4;
  Const
    Maxn=500;
    Maxs=-150;
  Type
    Tp=Array[Maxs..Maxn]of Byte;
  Var
    P:Tp;                                          {p[i]表示拼成 i 所需邮票数}
    Max:Array[0..40]of Integer;                    {见伪代码}
    List,Save:Array[0..40]of Integer;              {综合数据库，解}
    Best,Now,N,K:Integer;                          {最优值，当前值，N,K 见题}

    Procedure Getinfo;
      Begin
        Write('N=?');readln(N);
        Write('K=?');readln(K);
      End;

    Procedure Make(Step:Byte);                     {通过动态规划求 Max}
      Var
        i,j:Integer;
      Begin
        i:=0;
        Fillchar(P,Sizeof(P),$FF);
        P[0]:=0;
        Repeat
          Inc(i);
          For j:=1 to Step do
            if (P[i-List[j]]+1<P[i]) then P[i]:=P[i-List[j]]+1;{动态规划}
        Until P[i]>N;
        Max[Step]:=i-1;
      End;

    Procedure Search(Step,Pre:Integer);                {递归主过程}
      Var
        i:Integer;
      Begin
        If Step>K then                                 {step 上界}
          Begin
            If Max[K]>Best then                        {更新最优值}
```

```
            Begin
              Save:=List;
              Best:=Max[K];
            End;
          Exit;
        End;
      For i:=Pre+1 to Max[Step-1]+1 do              {枚举当前邮票}
        Begin
          List[Step]:=i;
          Make(Step);
          Search(Step+1,i);
        End;
    End;

  Procedure Main;                                   {主过程}
    Var i:Integer;
    Begin
      Best:=N;                                      {初始化}
      List[1]:=1;
      Max[1]:=N;P[0]:=0;
      For i:=1 to N do
        Begin
          Save[i]:=1;
          P[i]:=i;
        End;
      Search(2,1);                                   {递归搜索}
    End;


  Procedure GetoutInfo;                             {输出}
    Var
      i:Integer;
    Begin
      For i:=1 to K do
        Write(Save[i],' ');
      Writeln;
      Writeln('Max=',Best);
    End;

Begin
  Getinfo;
  Main;
  GetoutInfo;
End.
```

四. 可行性剪枝

如果可以根据当前枝条的状态，预计出从这个枝条往下搜索一定无法得到可行解，我们可以＂剪断＂这条枝，这种剪枝就叫做可行性剪枝.

在 CAKE 问题中的后两个剪枝就是可行性剪枝。

可行性剪枝条件一般有以下两种情况：

1. 根据函数 F2，估计未来原料消耗的最大值，如果比当前 Source 小的话，显然可以剪掉此枝。

2. 根据函数 F2，估计未来原料消耗的最小值，如果比当前 Source 大的话，显然可以剪掉此枝。

这两条剪枝是算法效率的关键，切不可小视。

例 6 分数分解(湖南 1999 选拔赛第一试第一题)

近来 IOI 专家们正在进行一项有关整数方程的研究，研究涉及到整数方程解集的统计问题，问题是这样的：

对任意的正整数 N，我们有整数方程：$1/X_1+1/X_2+\cdots+1/X_N=1$,该整数方程的一个解集$\{x_1,x_2,\cdots\cdots,x_n\}$是使整数方程成立的一组正整数，例如{n,n,n,…，n}就是一个解集，在统计解集时，IOI 专家把数据值相同但顺序不一样的解集认为是同一个解集，例如：当 n=3 时，我们把{2,3,6}和{3，6，2}义为是同一个解集。

现在的任务是：对于一个给定的 m ,在最多只允许 1 个 x_i 大于 m 时，求出整数方程不同解集的个数。

输入：输入文件共一行，有 2 个空格分开的正整数，它们分别是 n,m(n<=20,m<=100)。

输出：输出文件为一行，是不同解集的总个数。

分析：

本题找不到什么很好的解决方法，只能用搜索。因此，剪枝优化便成为了是否能够在时限内出解的关键。为了保证搜索到的解不会出现重复，不妨首先限定 $X_1<=X_2<X_3<\cdots\cdots<X_n$，从 X_1 搜起。像这种用搜索统计所有的可行状态总数的问题，可行性剪枝就更为重要了。

我们先来考虑一下问题的上下界。假设当前要搜 X_i ，而和已经是 t_1/t_2 ，因此，若以后的(n-i+1) 个数都取最大值，即 $1/X_{I-1}$ ，也不可能达到 1，则必然是无解的；同理，如果后(n-i+1)个数中，最后一个分数无穷小，而后(n-1)个分数都取最小值，即 1/m，它们的和也会大于 1，则必然也是无解的。利用上面的这两个剪枝条件，搜索的效率便得到了很大的提高。在解决此问题时，特别要注意的是过程中 t_1/t_2 要为既约分数，需避免计算时超界。

算法设计如下：

```
procedure search(i,t1,t2);
1 reduce(t1,t2);                                    {化简分数}
2 if (n-i+1)*1/X[i-1]+t1/t2<1 then exit;//剪枝 1
3 if (n-i+1)*1/m+t1/t2>1 then exit; //剪枝 1
4 for X[i] ß X[i-1] to m do
5     search(i+1,t1+t2*X[i],t2*X[i]) //递归搜索下一层
```

```
{$E+,N+}

program fraction;
const
   maxn = 20;
   limit = 1e10;
var
   tot : longint;
   n,m : integer;                                   {读入数据}
   x : array[0..maxn]of integer;                    {当前解}

procedure reduce(var t1,t2 : comp);                 {欧几里得辗转相除法}
   var c,tmp,a,b : comp;
   begin
      a:=t1;b:=t2;
      repeat
         tmp:=t2;
```

```
        c:=int(t1/t2);
        t2:=t1-t2*c;
        t1:=tmp;
      until t2=0;
      c:=t1;
      t1:=a/c;t2:=b/c;
    end;

  procedure search(step : integer;previous : integer;t1,t2 : comp); {递归搜索}
    var i : integer;
    begin
      reduce(t1,t2);
      if (n-step+1)/previous+t1/t2<1 then exit;     {剪枝 1}
      if (n-step+1)/m+t1/t2>1 then exit;            {剪枝 2}
      if n=step then                                {step 上界}
        if t1+1=t2 then inc(tot)
        else exit;
      if t2>Limit then exit;                        {上限}
      for i:=previous to m do
        search(step+1,i,t1*i+t2,t2*i);
    end;

  var i : integer;
  begin
    assign(input,'input.txt');reset(input);
    assign(output,'output.txt');rewrite(output);
    read(n,m);                                      {读入数据}
    tot:=0;
    for i:=2 to m do
      search(2,i,1,i);
    writeln(tot);                                   {输出数据}
    close(input);close(output);
  end.
```

五、选择合适的搜索对象，提高搜索效率

一个问题抽象后，能选取的对象可能有很多，从不同的角度分析问题，可能会建立不同的搜索对象，适当的选择搜索的对象，有利于建立好的搜索模型，提高搜索效率。下面看一个例题。

例 7 汽车问题(IOI 第二试第二题)

有一个人在某个公共汽车站上，从 12:00 到 12:59 观察公共汽车到达本站的情况，该站被多条公共汽车线路所公用，他依次记下公共汽车到达本站的时刻。

- 在 12:00－12:59 期间，同一条线路上的公共汽车以相同的时间间隔到站。
- 时间单位用“分”表示，从 0 到 59 。
- 每条公共汽车线路至少有两辆车到达本站。
- 公共汽车线路数 K 一定≤17，汽车数目 N 一定小于 300。
- 来自不同线路的公共汽车可能在同一时刻到达本站。
- 不同公共汽车线路的车首次到站时间和到站的时间间隔都有可能相同。

请为公共汽车线路编一个调度表，目标是：公共汽车线路数目最少的情况下，使公共汽车到达本站的时刻满足输入数据的要求。

例如：

汽车编号								
到达时间	0	3	5	13	14	14	21	25

那就可能存在这样一个解，由以下 3 条汽车线路组成：

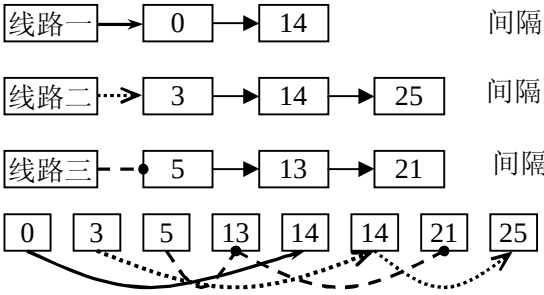


图 13 示例图

分析：

经仔细分析，本题宜采用深度搜索解题。

对问题，首先提取出三个关键要素：时间、车、路线。

☆决定车辆的属性是“时间”，

☆决定路线的属性是“首发车时间”和“间隔时间”，这等效于“第一辆车”和“第二辆车”。

面对这三个关键要素，下面就要从中确定搜索对象和搜索策略。可以看出，题目要求的是车和线路的关系，而时间在其中起的是描述作用和条件制约作用，因此，本题的搜索对象应该选择车或线路这两个关键要素。

分析搜索对象及策略

1．用车作搜索对象

由此对象而产生的搜索策略是：按到达时间顺序，依次对于那些没有确定属于哪条线路的车进行枚举，该车属于某新线路的第一辆车或属于某已有线路的第二辆车，若为后一种选择，则可确定路线上其它所有的车辆。

还是看给出的示例来构造搜索树，大致如下：

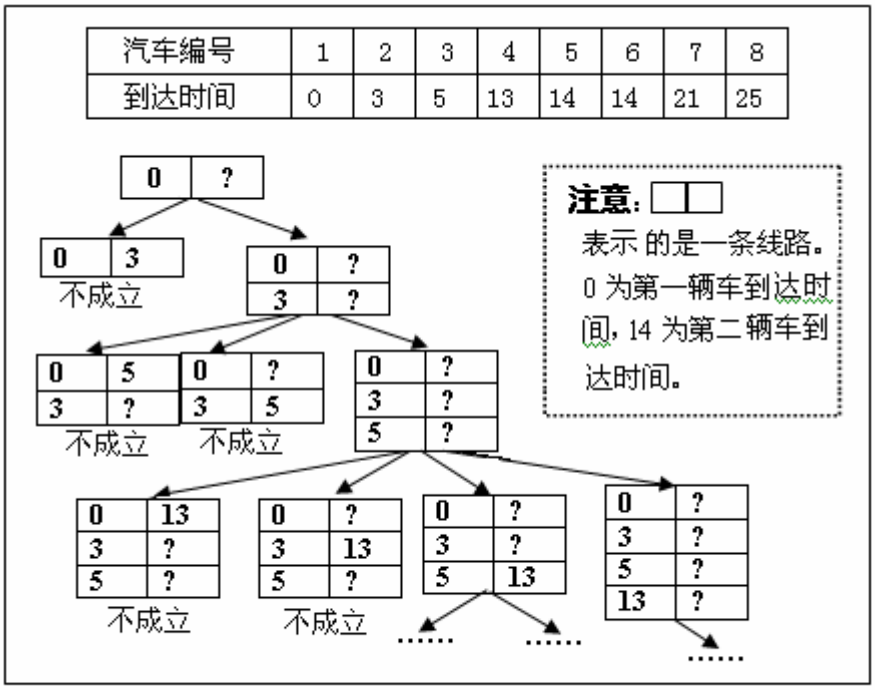


图14 选择车为搜索对象的搜索树

观察该搜索树发现：随着搜索树层数的递增，每层节点所扩展出的树叉数目逐渐增大。从直观上说，该搜索树从根开始，“分叉”越来越多，“枝叶”越来越茂盛。这就是采用车为搜索对象的搜索树的特点。

2．选用线路作搜索对象

由此对象而产生的搜索策略是：枚举每条路线包含哪些车，确定该路线。

实际上，对每条路线，我们只要枚举其特征：第一辆车和第二辆车。再根据“有序化”思想，固定所有路线是按照第一辆车的到达时间为关键字排序的。

大致搜索方案为：

搜索每层都要确定一条路线：将未确定归属路线的到达时间最小的车固定为新路线的第一辆车，其后枚举这条路线的第二辆车，从而确定该路线。

同样，根据先前给的那个简单例子我们也构造出了方法二的搜索树。

观察这个搜索树，和方法一的搜索树对比，我们发现，两者特性截然相反，该搜索树从根开始，“分叉”越来越少。

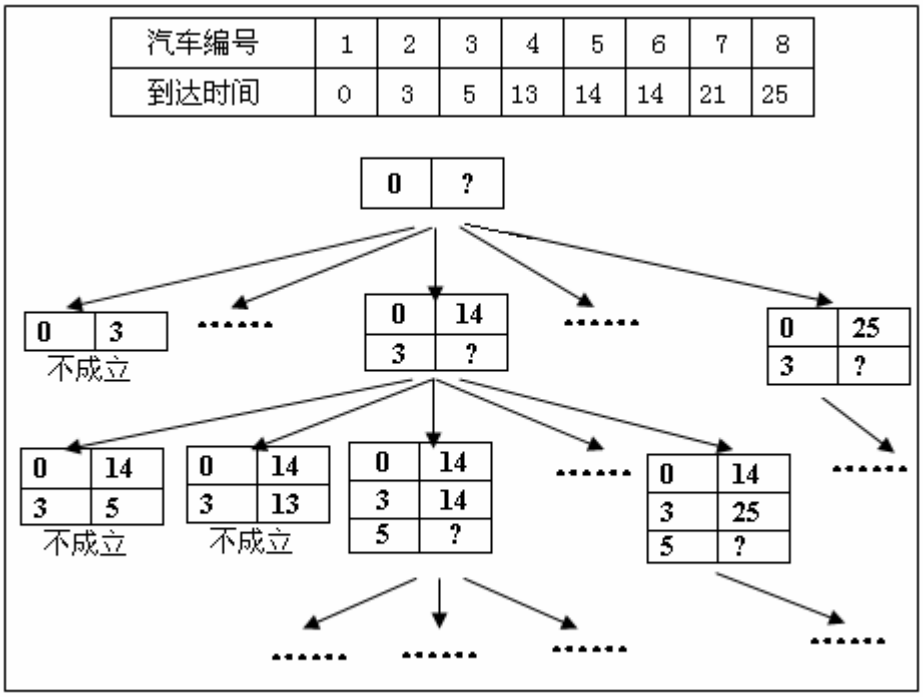


图 15 选择线路为搜索对象的搜索树

两种搜索对象及策略是完全不同的，搜索树特性又截然相反。从宏观上，搜索树上节点多少，两者相差无几。如何抉择呢？这时就要从微观上比较：

★谁易于优化剪枝

★谁的操作量小

A）关于谁易于优化剪枝

本题的主要剪枝有三种，逐一分析。

◇ 可行性剪枝

当路线的特征确定了，就可以判断该路线是否成立。

方法一，搜索中，每层枚举当前车是哪条路线的第二辆车，都要用到该判断；方法二，每层是确定一条路线，也用到该判断。关键是，根据两者搜索树，用方法一剪枝，将剪去的是一大片“茂盛”的树枝，显然，相比之下，方法二剪枝的效果就差了许多。

故在此剪枝应用上，方法二比方法一逊色许多。

◇ 最优化剪枝

这就要看谁能很快找到可行解了。显然，由于方法一可行性剪枝的优点，每次剪枝都能

删去很多的不可行的节点，找到解的速度就不比方法二慢了。

此剪枝，方法二不比方法一要好。

◇ 排除重复剪枝

注意到题目中时间这个关键要素的范围为 0~59，而车辆数目可达 300，说明，在同一时间到达的车辆数目很多！前面那个简单例子中，就出现了两个 14，而到达时间为 14 的两辆车各属于那条路线是等效的，这就有重复。

方法一对于同时到达的且未确定归属的车，若编号小的车为某路线的第一辆车，则编号大的车为也必为一条路线的第一辆车。

方法二，对到达时间相同的且未确定归属的车，固定只选编号最小的车为第二辆车。

两者剪去的都是重复的枝，所以，效果是一样的，故此剪枝上，双方平分秋色。

结论：由于方法一搜索树的良好特性，使得方法一在剪枝优化方面前景更广阔。

B）谁的操作量小

操作量是“主递归程序操作量”的简称，由主递归程序的枚举循环和剪枝函数决定的。

◇主递归程序的枚举循环

方法一，每层利用循环来枚举一辆车是属于新路线的第一辆车还是已知路线的第二辆车，而且搜索树上这个循环枚举量是由未确定“第二辆车”的线路数目决定的，最大为 17。

方法二，每层枚举哪辆车是第二辆车。由于用了排除重复剪枝，这个循环量最大为 60。

直观上看两者的最大界限就已知道，方法二不比方法一好。

◇剪枝函数消耗时间

由于本题特殊性，主要剪枝函数基本上差异不大，消耗时间也差不多。

结论：操作量大小方面，方法二不比方法一好。

通过以上微观理论分析，对两种方法进行了比较，得出最终结论是：方法一比方法二好！选定了搜索对象和策略，刚才又分析了如何剪枝，编写程序自然就得心应手了。

就本题而言，从宏观上看，很难知道两种方法效率的差异，为什么方法一更好呢？关键原因在于：它适合程序上的剪枝优化，且操作量小。

那为什么选择这两个方面为标准呢？

我们知道：深度搜索消耗时间 ≈ 每个节点操作系数 × 节点个数

从上面一个公式，我们很显然地能从微观上看出，要减少消耗时间，

一是减少节点个数——这就是我们所说地剪枝优化；

二是减少每个节点的操作系数——即刚才分析的程序操作量。

为了提高搜索效率，根据这个公式，我们往往在以上两方面反复进行改进。殊不知，从宏观前提上，如何才能使我们“可以、充分、有效”剪枝，同时，如何才能使我们“可以、充分、有效”降低程序操作量也都是很重要的！

于是，搜索对象和策略为剪枝，降低操作系数创造前提条件的好坏就成了我们的标准！

但是，在以这两方面为标准比较的时候，我们要注意到：这两个标准紧密关联，要剪枝多，就要有好的复杂的剪枝函数，但这就增大了每个节点操作系数。如图所示：

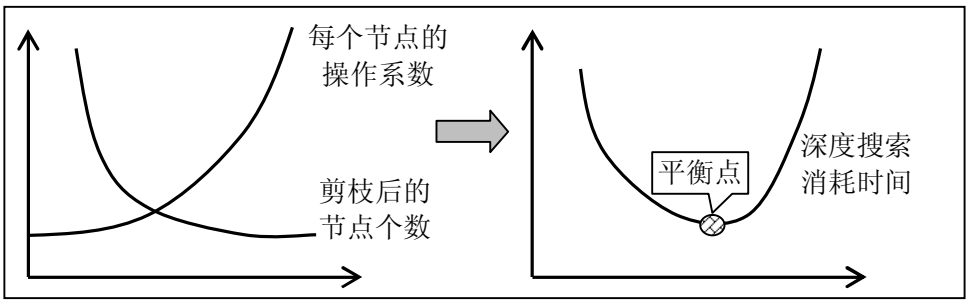


图 16 操作量和剪枝之间的关系图

两者在目的上是统一的，效果上却是对立的。在以这两者为标准的时候，要把握好“如何协调，找准两者平衡点”。

下面我们给出选用线路作搜索对象的程序，选用车作搜索对象的程序由读者完成。

```
PROGRAM Buses;
CONSt
  Inp = 'input.txt';
  Outp = 'output.txt';
VAR
  State : array[0..59] of integer;
  {时刻表.state[i]--i 时刻经由本站的汽车数(0≤i≤59)}
  Arrive,Timelength : array[1..300] of integer;
  {Arrive[i],Timelength[i]--第 i 条汽车线路的开始时间和时间间隔}
  Open,Closed : array[0..59] of integer;
  {open[i]--i 时刻出发的未确定时间间隔的线路数}
  {closed[i]--i 时刻出发的已确定时间间隔的线路数}
  List : array[0..59,1..300] of integer;
  {list[i,j]--i 时刻出发的第 j 辆车所属的线路}
  best,lev : integer;
  {best--目前求出的所有方案中最少的线路数;lev--当前线路数}

PROCEDURE Init;
var n,i,k : integer;
    fn : string;                                {文件名串}
begin
  Assign(Input,inp);reset(Input);
  readln(n);                                    {读入汽车数}
  fillchar(State,sizeof(State),0);
  {各时刻经由本站的汽车数初始化为零}
  for i:=1 to n do begin
  {以递增顺序输入汽车到达本站的时刻,并统计各时刻经由本站的汽车数}
    read(k);
    inc(State[k]);
  end;
  close(input);
  best:=maxint;                                  {最少线路数初始化}
  lev:=0;
  {各时刻出发的已确定时间间隔的线路数和未确定时间间隔的线路数初始化}
  for i:=0 to 59 do Closed[i]:=1;
  fillchar(Open,sizeof(Open),0);
  fillchar(List,sizeof(List),0);
  {各时刻出发的汽车线路号初始化}
  {各条线路的时间间隔和开始时间初始化}
  fillchar(Timelength,sizeof(Timelength),0);
  fillchar(Arrive,sizeof(Arrive),0);
end; {init}

FUNCTION Right:boolean;
{若当前 lev 条线路都确定时间间隔,则返回 true,否则返回 false}
var
  i : integer;
begin
  i:=0;
  repeat
    inc(i);
```

```
    until (i>lev) or (Timelength[i]=0);
    Right:=i>lev;
  end; {right}

  PROCEDURE Search(t:integer);    {从 t 时刻出发,递归搜索最佳方案}
  var
    i,j,k : integer;
    function can:boolean;
    {若时刻 t 后每隔 J 时间都有汽车到站,(构成一条出 t 时刻开始,间隔时间为 j 的汽
车线路),则返回 TRUE,否则返回 FALSE}
    begin
      k:=t;
      repeat
        inc(k,j);
      until (k>=60) or (State[k]=0);
      can:=k>=60;
    end; {can}
  begin
    while (t<60) and (State[t]=0) do inc(t);
                                {求 T 时刻后第一辆经由本站的时刻}
    if t=60 {T 时刻后无车辆到达本站}
      then begin
        if (lev<best) and right
                                {若线路数目前最少且每条线路确定了时间间隔}
        then begin
          best:=lev;            {则记下和输出当前方案}
          for i:=1 to lev do
            writeln(i:3,') ',Arrive[i]:4,Timelength[i]:4);
        end; {then}
      end {then}
    else begin
      for i:=0 to t-1 do
      {在 0..T-1 间,搜索所有 T 时刻经由本站的线路的开始时间 i}
      if Open[i]>=Closed[i]      {i 为未匹配时刻}
        then begin
          j:=t-i;
          if can then begin
            {T 时刻后每隔 T-i 都有汽车到站}
            k:=t;
            {从 T 时刻开始撤去以 T-i 为间隔的线路}
            repeat
              dec(State[k]); inc(k,j);
            until k>59;
            Timelength[List[i,Closed[i]]]:=j;
            {确定由 i 时刻出发的第 close[i]线路的时间间隔为 t-i}
            inc(Closed[i]);   {i 时刻出发的、已确定时间间隔的线路数增 1}
            Search(t);        {递归搜索 t 时刻后经由本站的线路}
            dec(Closed[i]);   {恢复递归调用前的参数}
            Timelength[List[i,Closed[i]]]:=0;
            k:=t;
            while k<=59 do begin
```

```
            inc(State[k]);
            inc(k,j);
          end; {while}
        end; {for}
      end; {then}
    if (lev+1)〈best
    {在确定了 0..t-1 期间出发的线路后,再增一条线路,线路数仍小于目前最佳方案}
      then begin
        inc(lev);          {则新增一条由 T 时刻出发的暂未确定时间间隔的线路}
        Arrive[lev]:=t;
        inc(Open[t]);
        List[t,Open[t]]:=lev;
        dec(State[t]);     {在时刻表中撤去 T 时刻}
        Search(t);         {递归搜索 T 时刻后经由本站的线路}
        inc(State[t]);     {恢复递归调用前的参数}
        dec(Open[t]);
        dec(lev);
      end; {then}
  end; {else}
end; {search}

BEGIN
  Assign(Output,' output.txt' );ReWrite(Output);
  Init;                    {输入汽车数和汽车到站的时刻表}
  Search(0);               {从 0 时刻出发,递归搜索最佳方案}
  Close(Output);
END. {main}
```

六、搜索优化的综合运用

例 8 列车调度(集训队选手自拟题目)

2020 年，很多工作已经由机器人来担当了，比如 X 火车站的列车调度员就是一个机器人。X 车站很小，其结构如下图所示：

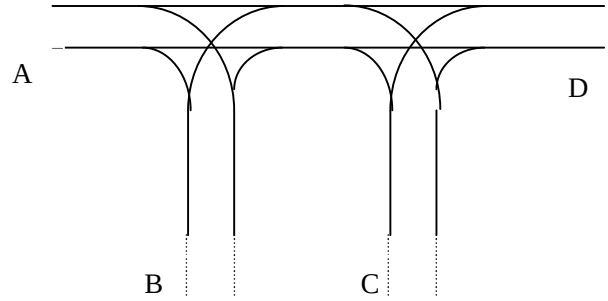


图 17 列车轨道

其中，A 是车站的入口轨道，D 是出口，B 和 C 是两个长度足够的中转轨道(最底端没有出口)。所有的轨道都是单线，即任一时刻，至多只有一辆列车通过，所以，A 端列车只能顺次进站，中转轨道 B 和 C 中，也只有最顶端的列车(如果有的话)可以移动。

调度员只可以执行下述 6 种调度操作(当然，他一次只能移动一辆可以移动的列车)：

让 A 端的进站列车进入 B，记作 A->B；

让 A 端的进站列车进入 C，记作 A->C；

让 A 端的进站列车直接移动到 D 出站，记作 A->D；

让中转轨道 B 中的顶端列车移动到 C 的顶端，记作 B->C；

让中转轨道 B 中的顶端列车移动到 D 出站，记作 B->D;

让中转轨道 C 中的顶端列车移动到 D 出站，记作 C->D。

作为列车调度员，[What]3 的工作是这样的：每天早晨，有 N 辆列车从 A 端进站(不妨按进站顺序编号为 1..N)，然后，他将要按照指令，通过调度操作，使列车按照某个给定的顺序出站。

然而，最近[What]3 向 CRR(Commission of Robot Rights)提出了申诉，因为他怀疑有些命令是否真的能够实现。所以，CRR 委派你编写一个程序，根据给定的出站序列，判断是否存在可行的调度方案，更进一步的，如果这样的调度方案存在，请找出其中操作步数最少的一种（如有多种这样的方案，只需找到其中的一种即可）。

输入：

输入数据存放于文件 input.txt 中；

输入文件的第一行是列车数 N(N<=50)，第二行是列车的出站序列(先出站者在前)，各编号之间用空格隔开。

输出：

输出数据到文件 output.txt 中；

如果无解，则输出“No Answer!”，否则第一行输出最优的操作次数 t，后继的 t 行是一个最优的操作序列，每个操作占一行，输出格式遵循前面所述的调度操作的记法。

分析:

本题的状态很容易确定——所有列车在各轨道上的排列就构成了问题的状态，我们可以用四个序列组成的四元组来表示状态：

状态 $S=\langle A,B,C,D\rangle$

$$\begin{cases} A=\{a_i\} & (1\le i\le t_A) \\ B=\{b_i\} & (1\le i\le t_B) \\ C=\{c_i\} & (1\le i\le t_C) \\ D=\{d_i\} & (1\le i\le t_D) \end{cases}$$

其中，$A \text{Y} B \text{Y} C \text{Y} D=\{1..n\}, t_A+t_B+t_C+t_D=n$ ，都采用自底向上的顺序，即可移动的元素都在序列的尾部。

则初始状态为 $\langle \{a_i=n+1-i\}, f, f, f\rangle$，如果给定的出站序列为 $\{T_i\}$（最先出站的列车为 T_1），则末状态为 $\langle f, f, f, \{T_i\}\rangle$。

由于题目规定列车的数目 n 至多可达 50，所以即便是最经济的状态保存方法，也至少需要 50 个字节来保存一个状态。显然，如果使用宽度系列的搜索算法，则状态的保存无疑会带来很大的空间消耗，因此，对空间要求比较低的回溯法搜索，就成了一个很不错的选择。

于是我们可以得到如下伪代码：

PROCEDURE search(step,A,B,C,D);step 表示当前的移动次数，A,B,C,D 分别表示各车道的车辆序列

1 if step>=best then exit;

2 if A=B=C= f then 更新最优值;

3 if A<> f then[Move(A,B);search(step+1,A’,B’,C’,D’);]//A’,B’,C’,D’表示变化后的序列

4 if A<> f then [Move(A,C); search(step+1,A’,B’,C’,D’);]

5 if (A<> f)and(A[top]=D[top]) then [Move(A,D);search(step+1,A’,B’,C’,D’);]

6 if B<> f then [Move(B,C);search(step+1,A’,B’,C’,D’);]

7 if (B<> f)and(B[top]=D[top]) then [Move(B,D);search(step+1,A’,B’,C’,D’);]

8 if (C<> f)and(C[top]=D[top]) then [Move(C,D);search(step+1,A’,B’,C’,D’);]

PROGRAM train;

1 初始化

2 search(1,Source, f , f ,Target); {Source 表示初始序列，Target 表示目标序列}

3 输出结果

确定算法后，我们再来考虑有没有什么好的最优化剪枝。应用最优性剪枝的关键在于设计一个准确的估价函数（一个状态的估价函数值不大于该状态到达解的路径长度的下限）。

1.最优化剪枝

对于本题，一种最简单的估价函数可以是$t_A + t_B + t_C$（假设所有的未出站列车都可以一步出站）。然而，这个估价函数的准确性实在太差（与被评估状态到达解的路径长度的下限相差甚远），于程序效率提高的作用十分有限。

那么，如果要提高估价函数的准确性，一种自然的思路就是判断哪些列车不可能一步出站。一辆列车何时不能直接出站呢？显然，是在该列车所在的轨道中，存在需要更早出站，而位置又更接近底端的列车的时候，即若对轨道 *R*(*R*=*A* 或 *B*)，$\exists j_{1\le j<i}\left(p(r_j)<p(r_i)\right)$，则列车 r_i 不能直接出站。

所以，只要统计出当前状态下，不能直接出站、需要给别的列车“让路”的列车的数目，再加上$t_A + t_B + t_C$，就可以得到一个较为准确的估价函数（假设这些不能直接出站的列车需要再移动两步方可出站）。

2.可行性剪枝

对于本题可行性剪枝很多，首先我们看一个简单的栈.

图 20，假设目标序列为{1,2}，而栈中我们得到的序列是{1,2}，因此无论怎么调度，我们也无法得到目标序列.

仔细想一想，这个栈也就相当于题目中的 C 栈．于是，我们可以的到如下的剪枝：

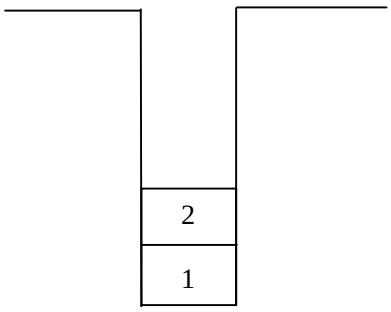


图 18 无法调度示例 1

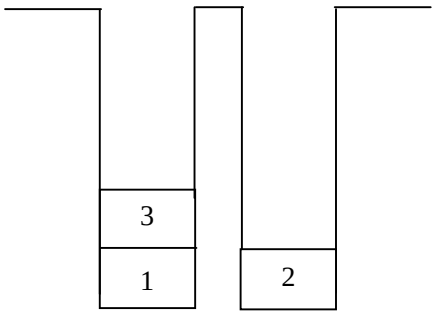


图 19 无法调度示例 2

令 pos_target[i]表示 i 号车在目标中的位置.

C[i]表示当前 C 栈中第 i 辆车的标号

```
For i:=tc downto 1 do
   For j:=i-1 downto 1 do
If pos_target[C[i]]<pos_target[C[j]] then fail;
```

相应的我们可以得到一个比较复杂的可行性剪枝.

如图 21，设目标序列是{1,2,3}，我们可以发现无论如何移动，现在的状态无法到达目标状态．于是我们可以得到一个关于两个栈的剪枝.

```
For x:=tb downto 2 do
   For y:=x-1 downto 1 do
      If (pos_target[B[y]]<pos_target[C[top]]) and (pos_target[B[x]]>pos_target[C[top]])
      then fail
```

3.简化规则

本题除了有如上的剪枝外，还可以在搜索的规则上优化.

图中有 6 条规则：

A à D
B à D
C à D
A à B
B à C
A à C

但只要我们稍加分析,便可得：

1.A à B B à D ó A à D
2.A à C C à D ó A à D
3.B à C C à D ó B à D

这样一来，由六条规则转化成了 3 条规则，搜索树的度一下由 6 降到了 3，重复的节点便少了。同时也减少了 6^MaxDep-3^MaxDep 次判重操作。因为是深度优先搜索算法，因此空间复杂度并未降低。从测试情况中可以看出，指数级的算法中，如果能够降低它的底数，那么算法的空间和时间（特别是时间）的复杂度会有很大的降低。

下面让我们分析一下，这一条看似简单的优化的效率.

设作用于节点的规则数量为 N，扩展的状态空间为 S，节点的最大深度为 MaxDep,那么：{不考虑判重}

$S=N^{Dep}$

如图 22，显然当 N 比较大时 $N^{Dep}>>(N-1)^{Dep}$，所以我们应尽可能的使 N 减小。

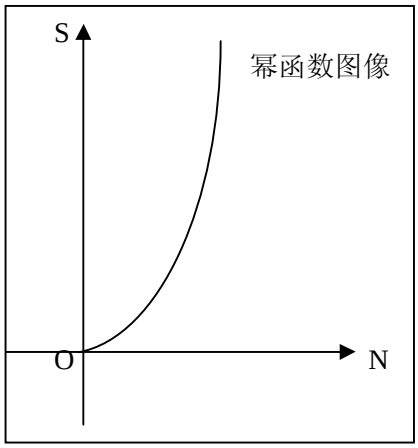


图 20 指数函数的曲线图

```
{$R-,S-,Q-}

program trains;
const
   maxN = 60;
type
   tList = array[0..maxN]of byte;
var
   List,Save : array[0..maxN*3+1]of byte;
   {List,Save 分别保留当前移动方案和最优值移动方案}
   {其中 List[i]的值分别表示：1:A-B;2:A-C;3:A-D;4:B-C;5:B-D;6:C-D}

   Block,                    {Block[i]标志 i 是否需要出站让路}
   A,B,C,                    {A,B,C 分别表示入口，栈 1 ，栈 2 的情况}
   order : tList;            {Order 表示每一辆车的出站顺序}
   nBlock,                   {nBlock 为 Block[i]之和}
   iD,                       { iD 表示当前出栈序列的长度}
   best,N,iA,iB,iC : integer;  {iA,iB,iC 分别表示 A,B,C 的长度}

   procedure GetInfo;{读入}
   var i,j : integer;
   begin
      read(n);
      for i:=1 to n do begin
         read(j);
         order[j]:=i;
```

```
    end;
end;

function cut2:boolean;   {可行性剪枝 2 }
var min,i : integer;
begin
    min:=N;
    for i:=1 to iB-1 do if B[i]<min then min:=B[i];
    for i:=1 to iC do if (B[iB]>C[i])and(C[i]>min) then begin
        cut2:=true;
        exit;
    end;
    cut2:=false;
end;


procedure search(step : integer); {主搜索过程}
var    moveD,more : boolean;
       i,k,oldstep,oldblock : byte;
begin
    if step-1+iA+iB+iC+nBlock>=Best then exit; {最优化剪枝}

    if iD=n+1 then begin
        best:=step-1;
        Save:=List;
        exit;
    end;
    oldstep:=step;

    {处理所有的 A->D,B->D,C->D}
    moveD:=false;more:=true;
    while more do begin
        more:=false;
        if (iA>0)and(A[iA]=iD) then begin
            more:=true;
            List[step]:=3;
            inc(step);Dec(iA);Inc(iD);
        end;
        if (iB>0)and(B[iB]=iD) then begin
            more:=true;
            List[step]:=5;
            inc(step);Dec(iB);Inc(iD);
        end;
        if (iC>0)and(C[iC]=iD) then begin
            more:=true;
            List[step]:=6;
            inc(step);Dec(iC);Inc(iD);
        end;
        if more then moveD:=true;
    end;

    {如果有移动到 D 的}
    if moveD then begin
        search(step);
        {恢复}
        for k:=step-1 downto oldstep do
            case List[k] of
```

```
            3:begin inc(iA);Dec(iD);A[iA]:=iD;end;
            5:begin inc(iB);Dec(iD);B[iB]:=iD;end;
            6:begin inc(iC);Dec(iD);C[iC]:=iD;end;
            end;
        step:=oldstep;
    end;
    {A->B 的情况}
    if iA>0 then begin
        inc(iB);B[iB]:=A[iA];Dec(iA);
        List[step]:=1;{记路径}

        oldblock:=Block[B[iB]]; {计算 Block 数组，更新 Block}
        Block[B[iB]]:=0;
        for i:=iB-1 downto 1 do
            if B[i]<B[iB] then begin Block[B[iB]]:=1;break;end;
        inc(nBlock,Block[B[iB]]-oldblock);
        if not Cut2 then search(step+1); {可行性剪枝 2 }
        dec(nBlock,Block[B[iB]]-oldblock); {恢复}
        Block[B[iB]]:=oldblock;
        inc(iA);A[iA]:=B[iB];dec(iB);
    end;
    {A-->C,或者 B-->C}
    if iB>0 then
        if List[step-1]=1 then begin {A-->C}
            dec(step);
            List[step]:=2; {记路径}
            inc(iC);C[iC]:=B[iB];Dec(iB);
            oldBlock:=Block[C[iC]];
            Block[C[iC]]:=0;
            for i:=iC-1 downto 1 do
                if C[i]<C[iC] then begin Block[C[iC]]:=1;break end;
            inc(nBlock,Block[C[iC]]-oldblock);
            if (block[C[iC]]=0) and not cut2 then Search(step+1); {可行性剪枝 1,2}
            dec(nBlock,Block[C[iC]]-oldblock); {恢复}
            Block[C[iC]]:=oldBlock;
            Inc(iB);B[iB]:=C[iC];dec(iC);
            List[step]:=1;
            inc(step);
        end
        else begin                                          {B-->C}
            List[step]:=4; {记路径}
            inc(iC);C[iC]:=B[iB];Dec(iB);
            oldBlock:=Block[C[iC]];
            Block[C[iC]]:=0;
            for i:=iC-1 downto 1 do
                if C[i]<C[iC] then begin Block[C[iC]]:=1;break end;
            inc(nBlock,Block[C[iC]]-oldblock);
            if (block[C[iC]]=0) and not cut2 then Search(step+1); {可行性剪枝 1,2}
            dec(nBlock,Block[C[iC]]-oldblock);{恢复}
            Block[C[iC]]:=oldBlock;
            Inc(iB);B[iB]:=C[iC];dec(iC);
        end;
end;

var i,j : integer;
begin
```

```
    assign(input,'input.txt');reset(input);
    assign(output,'output.txt');rewrite(output);
    GetInfo;
    {初始化}
    for i:=1 to N do A[i]:=Order[N-i+1];
    iA:=N;
    iB:=0;iC:=0;iD:=1;
    for i:=iA downto 1 do
      for j:=i-1 downto 1 do
        if A[j]<A[i] then begin
          inc(nBlock);
          Block[A[i]]:=1;
          break;
        end;
    best:=3*N+1;{可能的最大移动次数+1}
    search(1);
    {输出}
    if best=3*N+1 then writeln('No Answer!')
    else begin
      writeln(best);
      for i:=1 to best do
        case Save[i] of
          1:writeln('A->B');
          2:writeln('A->C');
          3:writeln('A->D');
          4:writeln('B->C');
          5:writeln('B->D');
          6:writeln('C->D');
        end;
    end;
    close(output);
    close(input);
  end.
```

例 9 埃及分数

[题目描述]

在古埃及，人们使用单位分数的和(形如 1/a 的，a 是自然数)表示一切有理数。如：2/3=1/2+1/6,但不允许 2/3=1/3+1/3,因为加数中有相同的。

对于一个分数 a/b,表示方法有很多种,但是哪种最好呢？首先,加数少的比加数多的好,其次，加数个数相同的，最小的分数越大越好。

如：

19/45=1/3 + 1/12 + 1/180

19/45=1/3 + 1/15 + 1/45

19/45=1/3 + 1/18 + 1/30,

19/45=1/4 + 1/6 + 1/180

19/45=1/5 + 1/6 + 1/18.

最好的是最后一种，因为 1/18 比 1/180,1/45,1/30,1/180 都大。

给出 a,b(0<a<b<1000),编程计算最好的表达方式。

输入：a b

输出：若干个数，自小到大排列，依次是单位分数的分母。

例如：

输入：19 45

输出：5 6 18

分析:

这个题目虽然看起来很简单，但是我们很难想到用一个数学模型的方法或理论方法解决，因此搜索势在必行。最基本的搜索方式有深度优先和宽度优先。这道题目用深度优先很难出解，因为我们无法预知解的深度，因此考虑宽度优先搜索。如果用宽度优先搜索，先来看看算法的实现：

1.节点类型

是一个 K 元组（a1,a2,...ak），代表当前解中的分母 a1,a2..ak.

2.节点扩展方法

按照 a1<a2<a3..<ak 的顺序扩展,扩展第 k 层节点的时候,最简单的办法就是从 a[k-1]+1 开始枚举 a[k],一直到预先确定的最大值。

但是这个最大值怎么确定呢？

直观的讲,a[k]总不能太大,因为如果 a[k]太大,1/a[k]就很小,1/a[k+1] .. 1/a[k+2].. 就更小，那么，尽管加了很多项，还可能不够 a/b.

例如：

例如已经扩展到

19/45=1/5+????

如果第二项是 1/100，那么由于 19/45-1/5=2/9=0.22222...

那么接下来至少要 0.2222/(1/100)=22 项加起来才足够 2/9，所以继续扩展下去至少还要扩展 22 项,加起来才差不多够 a/b。

3.确定搜索算法

我们可以发现这道题目其实和上文提到的“分数分解问题”形式上差不多，但原题的分数 k 是固定的一个数，所以比较容易处理，而“埃及分数问题”的 k 是可变的，所以才会出现上文提到的问题，既然它的 k 可变，我们何必不固定 k 然后再仿照分数分解用深度优先搜索解决这道问题呢？

事实证明这种方法是可行的。我们可以令 k=1,2,3,4,5,6…,由于本题需要 k 最小，所以我们只要找到一组解，就可以停止对于 k 的枚举而得到解. 这种深度可变的深度优先搜索叫做**深度可变的搜索**. 它的实现方式很简单：

设 depth 为当前最大深度；maxdepth：问题理论最大深度；

```
for depth:=1 to maxdepth do begin
  best:=maxint;
DFS;
end;
```

对于深度无法预知，而且深度可变的范围不大的题目，采取深度可变搜索是一种比较好的选择. 它主要有以下优点：

1.空间开销小

因为每个深度下实际上是一个深度优先搜索，不过深度有限制，而它吸收了 DFS 的空间消耗小的优点。

2 时间效率不与简单深度优先搜索相当

虽然重复搜索，但是前一次搜索跟后一次相比搜索量是微不足到的。算法是指数级的，depth+1 和 depth 的搜索量差别自然很大，所以 depth 的搜索量相对来说是可以忽略的.

3 可利于深度剪枝

对于上文中提到的例子,如当前深度是 3,那么 22 肯定就不要了。而如果最优解是 4，肯定在 depth=4 的时候就能得到最优解，从而可立刻终止搜索。也就是说，22 每次都没用，这和我们主观的剪枝动机是一致的。

此题的剪枝和分数分解的剪枝基本相同,于是我们可以如下伪代码：

```
PROCEDURE Search(k,a,b); {决定第 k 个分母 d[k]}
1.  Reduce(a,b);                                    {化简 A,B}
2.  If k=depth+1 then exit
```

```
3.    else if (b mod a=0) and (b div a>d[k-1]) then [        {a 整除 b 的情况}
4.      d[k]:=b div a;
5.      if not found or (d[k]<answer[k]) then  更新解;
6.    ]
7.    else [
8.     确定 d[k]的上下界 s,t;
9.     for i:=s to t do [
10.      d[k]:=I;
11.      search(k+1,(i*a-b) div m,(b*i) div m);
12.    ]
13.  ]
PROGRAM Fraction;
1.   初始化
2.   for depth:=1 to maxdepth do [
3.     search(1,a,b);
4.     if 找到解 then 输出
5.   ]


{$A+,B-,D+,E+,F-,G-,I+,L+,N+,O-,P-,Q-,R-,S-,T-,V+,X+}
{$M 16384,0,655360}
program fraction;
const
   maxdepth=10;   {理论最大深度}
var
   a,b,i,depth:longint;
   found:boolean;                                    {解存在标志}
   answer,d:array[0..maxdepth] of longint;           {存储所有分母}

function gcd(a,b:longint):longint;                   {求公约数}
var t:longint;
begin
   t:=a mod b;
   while t<>0 do
   begin
      a:=b;
      b:=t;
      t:=a mod b;
   end;
   gcd:=b;
end;

procedure search(k,a,b:longint); {决定第 k 个分母 d[k] }
var
   i,m,s,t:longint;
begin
   if k=depth+1 then exit
   else if (b mod a=0) and (b div a>d[k-1]) then      {a 整除 b 的情况}
   begin
      d[k]:=b div a;
      if not found or (d[k]<answer[k]) then          {比当前解优}
         answer:=d;
      found:=true;
      exit;
```

```
    end;
    {d[k]的下界为 s,上界为 t}
    s:=b div a;                                          {a 不整除 b 的情况}
    if s<d[k-1]+1 then s:=d[k-1]+1;
    t:=(depth-k+1)*b div a;
    if t>maxlongint div b then t:=maxlongint div b;      {防止溢出}
    if found and (t>=answer[depth]) then t:=answer[depth]-1;

    for i:=s to t do
    begin
      d[k]:=i;                                           {枚举 d[k]}
      m:=gcd(i*a-b,b*i);
      search(k+1,(i*a-b) div m,(b*i) div m);             {继续搜索}
    end;
end;

begin
    assign(input,'input.txt');reset(input);
    assign(output,'output.txt');rewrite(output);
    found:=false;
    d[0]:=1;
    readln(a,b);                                         {读入}
    for depth:=1 to maxdepth do                          {枚举最大深度}
    begin
      search(1,a,b);
      if found then                                      {找到第一组解就退出}
      begin
        for i:=1 to depth do                             {输出}
          write(answer[i],' ');
        writeln;
        break;
      end;
    end;
     close(input);close(output);
end.
```