基于求解状态的递归程序设计公式化方法

张德同, 周明全

(西北大学 计算机科学系, 陕西 西安 710069)

摘要: 讨论了递归程序设计的公式化方法, 指出现有方法的不足, 并提出了一种新的基于求解状态的递归程序设计公式化方法, 在一定程度上达到了递归程序设计公式化、简单化的目的。最后通过两个具体例子, 说明了如何使用这种新的公式化方法进行递归程序设计。

关 键 词: 递归; 程序设计; 公式化方法; 求解状态

中图分类号: TP18 **文献标识码**: A **文章编号**: 1000-274X(2002)05-0507-04

根据解的特点, 用递归法处理的问题可以分成 3 类: ① 第一类具有确定的惟一解, 这一类问题的递归程序设计相对容易; ② 第二类问题具有多个解, 且各个解之间没有优劣之分, 通常要求列出所有可能的解; ③ 第三类问题也具有多个解, 但各个解之间有优劣之分, 通常要求给出最优解。后两类问题均可以用回溯法递归求解[1,2], 但在具体应用时, 不同问题的处理方法之间有较大变化, 不易把握。

文献[3]对递归程序设计的公式化方法进行了有益的探讨。递归程序设计的公式化方法, 就是首先把问题表示成一个递归定义的数学函数, 然后设计相应的递归程序计算递归函数值, 在求得函数值的同时, 设法得到原问题的解, 但原文中所提出的方法存在以下不足: 首先, 没有区分具有确定惟一解的问题和具有多个解的问题, 文中讨论的汉诺塔问题属于第一类问题, 其公式化方法与通常方法相比, 没有特别的优越性。其次, 原文仅针对具体问题给出了具体的公式化方法, 没有针对问题类型特点提出一般化的公式化方法。另外, 文章认为组合问题与八皇后问题类型不同, 并在文献[4]中以组合问题(五皇后问题)为例提出一种称为伴随序列法的方法。我们认为, 组合问题(五皇后问题)与八皇后问题同属上述第二类问题, 可以用同一种公式化方法求解。

本文针对上述第二类问题, 提出了一种基于求解状态的递归程序设计公式化方法, 给出了一个统一的公式化抽象模型, 在一定程度上达到递归程序设计公式化、简单化的目的。

1 基于求解状态的递归公式化方法

基于求解状态的递归程序设计公式化方法涉及以下几个方面。

1.1 求解状态的表示

求解状态由问题参量和部分解两方面组成, 问题参量描述了当前要解决的问题, 部分解记录当前已经求得的不完全解。

1.2 状态转换方法

一个状态转换方法能改变当前求解状态的某些状态分量, 得到一个后继状态。状态转换方法的作用体现在两方面: 一是将当前问题转化成一个子问题, 另一方面修改部分解。

1.3 状态恢复方法

状态恢复方法是特殊的状态转换方法, 能够将当前求解状态恢复到其父状态。显然, 状态恢复方法的作用也体现在两方面: 一是将当前问题恢复为其父问题, 另一方面恢复原来的部分解。状态恢复方法与相应的状态转换方法一一对应。

1.4 递归函数的参数

递归函数的参数只有一个, 统一取当前求解状态。

1.5 递归函数的返回值

统一取当前求解状态(当前问题)的解数目, 即

收稿日期: 2001-12-20
基金项目: 国家自然科学基金资助项目(60072044)
作者简介: 张德同(1963-), 男, 山东济南人, 讲师, 在职博士生, 主要研究领域为智能信息处理、模式识别。

从当前状态出发所能得到的解的个数。

1.6 递归函数的抽象定义

solution (S_k) = 0 如果 S_k 为非法状态;

solution (S_k) = 1 如果 S_k 为完全解状态;

solution (S_k) = $\sum_{j=1}^{n}$ solution ($S_{k+1}^{(j)}$) 如果 S_k 为其他状态。

其中 $(S_{k+1}^{(j)})$ 表示对 S_k 使用第 j 个状态转换方法所得到的后继状态。

1.7 递归函数的实现

可以用下面统一的程序框架,描述以上抽象递归函数的实现过程。

```
typedef struct{
问题参量;
部分解参量;
} State; /* State 是表示求解状态的类型 */
void transform (int j, State * s){
if (j= = 1){利用第 1 个状态转换方法 T1 改变当前求解状态 s 的状态分量 Δs1; return; }
if (j= = 2){利用第 2 个状态转换方法 T2 改变当前求解状态 s 的状态分量 Δs2; return; }
……
}
void restore(int j, State * s){
if (j= = 1){利用第 1 个状态恢复方法 R1 恢复当前求解状态 s 的状态分量 Δs1; return; }
if (j= = 2){利用第 2 个状态恢复方法 R2 恢复当前求解状态 s 的状态分量 Δs2; return; }
……
}
long solution(State * s)
{int j; long num;
if (illegal(s)) /* s 为非法状态 */
return (0);
else if (complete(s)) /* s 为完全解状态 */
{use(s); return (1); }
else /* s 为其他状态 */
{num = 0;
for(j= 1; j< = 状态转换方法数目; j+ + )
{transform (j, s);
num = num + solution(s);
restore(j, s);
}
return (num); }}
main()
{State s0; long num; s0= 初始状态;
num = solution(&s0);
printf("\n%ld solutions in all !", num); }
```

对于同类型的具体问题,只需给出求解状态 State 的定义和状态转换方法 transform (j, s) 等的定义,就可以使用上述统一的程序框架实现递归处理。

2 应用示例

下面通过具体例子,说明如何使用上述统一的程序框架实现递归处理。为了便于与原文方法进行比较,仍然讨论八皇后问题[3]和五皇后问题[4]。

2.1 八皇后问题

八皇后问题要求在 8×8 的国际象棋棋盘上摆放 8 个皇后,使她们不致互相攻击。算法从第一行开始每行放一个皇后,其中每一行都从第一列开始尝试,如果与前面已放好的皇后冲突就换下一列,如果每一列都不行则回溯到前一行,直到 8 个皇后全部放好。

八皇后问题的求解状态定义如下:

```
typedef struct {
int result[9]; /* result[i]存放第 i 行上皇后所在的列号。0 号单元不用。 */
int len; /* len 记录已放好皇后的行数(部分解长度)。 */
} State;
```

其中 len 有双重作用,一方面作为部分解参量,记录部分解长度;另一方面作为问题参量,表示当前问题为:从第 len+ 1 行开始,逐行按要求摆放皇后。

八皇后问题的状态转换方法定义如下:

```
void transform (int j, State * s){
(* s). len+ + ; /* 指向下一行 */
(* s). result[(* s). len]= j; /* 在当前行的第 j 列摆放皇后, 1≤j≤8 */
}
```

显然,状态转换方法改变了当前求解状态,包括问题参量和部分解参量。

八皇后问题的状态恢复方法定义如下:

```
void restore(int j, State * s){
(* s). len- - ;
}
```

由于 len 具有双重作用,所以通过恢复 len 的

值, 能够同时恢复问题参量和部分解参量。由于各恢复方法完全相同, 对该问题而言, 参数 j 已没有实质作用。

限于篇幅, 下面略去通用的 solution( ) 函数:

```
int illegal(State * s)
{int i, k;
if((* s). len= = 0) return (0); /* 初始状态合法 * /
k= (* s). len;
for(i= 1; i< = k- 1; i+ + )/* 检查第 k 个皇后是否与前 k- 1 个皇后冲突 * /
if ((* s). result[i]= = (* s). result[k])
return (1);
else if(i+ (* s). result[i]= = k+ (* s). result[k])
return (1);
else if(i- (* s). result[i]= = k- (* s). result[k])
return (1);
return (0); /* 与前面皇后不冲突 * /
}
int complete(State * s)
{ if((* s). len= = 8) /* 已放好 8 个皇后 * /
return (1);
else return (0);
}
void use (State * s)
/* 使用当前解, 在此为统计和打印 * /
{int i; static long count= 0;
printf("\n%ld", + + count);
for(i= 1; i< = (* s). len; i+ + )
printf(" %d", (* s). result[i]);
}
main()
{ State s0; long num;
s0. len= 0; /* 初始问题为: 从第一行开始逐行摆放皇后。当前部分解的长度为 0 * /
num= solution(&s0);
printf("\n%ld solutions in all!", num);
}
```

程序运行后共找到 92 个解, 其中前 4 个解为: 15 863 724, 16 837 425, 17 468 253, 17 582 463; 后 4 个解为: 75 316 824, 82 417 536, 82 531 746, 83 162 574。

2.2　五皇后问题

五皇后问题要求在 8×8 的国际象棋棋盘上摆放 5 个皇后, 使她们能控制整个棋盘。算法基本思想与文献[4]相同: 求出棋盘上 64 个位置中取 5 个位置的所有组合, 并依次检查每个组合是否满足五皇后问题的要求。

五皇后问题的求解状态定义如下:

```
typedef struct {
int m; /* m 和下面分量 n 是问题参量, 表示从 m 个数中取 n 个数的组合问题 * /
int n;
int result[6]; /* result[i]存放一个组合中的第 i 个数。0 号单元不用 * /
int len; /* len 记录当前组合的长度(部分解长度) * /
} State;
```

五皇后问题的状态转换方法定义如下:

```
void transform(int j, State * s){
if(j= = 1)/* 状态转换方法 T1 * /
{(* s). len+ + ;
(* s). result[(* s). len]= (* s). m; /* 将 m 加入当前组合 * /
(* s). m- - ; (* s). n- - ; /* 在取了 m 的基础上, 进一步要解决的后继问题是: 在 m- 1 个数中再取 n- 1 个数 * /
return; }
if(j= = 2)/* 状态转换方法 T2 * /
{(* s). m- - ; /* 如果不取 m, 进一步要解决的后继问题是: 在 m- 1 个数中仍取 n 个数。当前部分解维持不变。 * /
return; }}
```

五皇后问题的状态恢复方法定义如下:

```
void restore(int j, State * s)
{if(j= = 1)
{(* s). len- - ; (* s). m+ + ;
(* s). n+ + ; return; }
if(j= = 2)
{ (* s). m+ + ; return; }}
```

限于篇幅, 下面仍略去通用的 solution() 函数:

```
int illegal(State * s){
if((* s). m= = 0&&(* s). n> 0)/* 非法 * /
return (1);
else return (0);
}
```

```
int complete(State * s)
{if((* s). n= = 0) return (1); /* 找到一个组合 */
else return (0);
}
void use(State * s)
{test((* s). result); /* 检查当前找到的位置组合是否满足五皇后问题的要求 */
}
main()
{State s0; long num;
s0. m= 64; s0. n= 5; /* 初始问题 */
s0. len= 0; /* 当前部分解的长度为 0 */
num= solution(&s0);
printf("\n %ld combinations!", num);
printf(" %ld solutions !", count); }
```

程序运行后共找到 7 624 512 个棋盘位置组合, 其中满足五皇后问题要求的组合有 4 860 个。下面列出五皇后问题的前 5 个解, 每个解用两种形式表示: 棋盘位置序号的组合; 棋盘位置行列号的组合。

1) 64 63 51 32 20; (8, 8) (8, 7) (7, 3) (4, 8) (3, 4)
2) 64 63 35 29 20; (8, 8) (8, 7) (5, 3) (4, 5) (3, 4)
3) 64 63 35 26 20; (8, 8) (8, 7) (5, 3) (4, 2) (3, 4)
4) 64 62 50 31 12; (8, 8) (8, 6) (7, 2) (4, 7) (2, 4)
5) 64 62 42 32 12; (8, 8) (8, 6) (6, 2) (4, 8) (2, 4)

3 结束语

递归法是一种重要的程序设计方法, 用递归法处理的问题, 可以根据解的特点分成 3 类。本文针对第二类问题, 提出了一种基于求解状态的递归程序设计公式化方法, 在一定程度上达到了递归程序设计公式化、简单化的目的。另外, 将 illegal(s), complete(s) 等辅助子函数改为宏定义或内联函数, 可以进一步提高程序的运行速度。

参考文献:

[1] ROBERT S. Algorithms[M]. Boston: Addson-Wesley Publishing Company, 1983.
[2] 王晓东. 计算机算法设计与分析[M]. 北京: 电子工业出版社, 2001.
[3] 朱玉龙, 任文岚. 递归程序设计的公式化方法[J]. 小型微型计算机系统, 2001, 22(11): 1 389-1 390.
[4] 朱玉龙, 任文岚. 五皇后问题与伴随序列法[J]. 小型微型计算机系统, 2000, 21(11): 1 221-1 222.

(编 辑 曹大刚)

The formula method of recursive programming based on solving state

ZHANG De-tong, ZHOU Ming-quan
(Department of Computer Science, Northwest University, Xi'an 710069, China)

Abstract: Formula method of recursive programming is discussed. The imperfectness of the present method is pointed out, and a new formula method of recursive programming based on solving state is proposed. This leads to the formularization and simplification of recursive programming to some extent. Finally two examples are presented to explain how to use the new formula method to design recursive program.

Key words: recursion; programming; formula method; solving state