计算机算法解决俄罗斯方块中的数学与计算问题

王 宇 战学刚 高国伟
(鞍山科技大学计算机科学与工程学院 辽宁 鞍山 114044)

摘 要 针对存在于俄罗斯方块中的两个数学问题,提出原创的解决方案。根据该解决方案,给出两种算法,通过实验比较算法的执行效率,并根据俄罗斯方块的特性提出一种独特的算法改进思想,采用实验证明其可以在解决某些问题时提高算法的效率。该算法可推广至同类问题的解决。

关键词 俄罗斯方块 四向循环链表

SOLVING MATHEMATICAL AND COMPUTATIONAL PROBLEMS IN TETRIS WITH COMPUTER ALGORITHMS

Wang Yu Zhan Xuegang Gao Guowei
(*Institute of Computer Engineering & Science, Anshan University of Science and Technology, Anshan Liaoning 114044, China*)

Abstract This paper will originally presents a method of solving two mathematical problems in Tetris which were presented by a mathematician in 1999. According to the method this paper gives two algorithms. Compare the efficiency of two algorithms with experiments and also using some unique methods which is based on the characteristic of Tetris to improve the computational efficiency of the better algorithm, and proves such methods can enhance the efficiency when applies it to certain problems. The paper also proves the algorithm can not only solve these questions but also solves problems of such kind.

Keywords Tetris Four direction recursive links

1 引 言

本文首先要解决 1999年提出的两个数学问题:

(1) 俄罗斯方块中的 5个方块(每个的面积都为 4个单位格),各一片放入面积为 4×5的矩形区域内(每一片都可以顺时针逆时针旋转,正反面翻转与翻转后旋转后摆放在该矩形区域中,即可以任意摆放),有没有正好将其完全覆盖的方案?[1]

(2) 在面积为 6×6的正方形区域内,从 5个方块中任选 1个方块填充该区域(任意摆放),对于每一个方块来说有没有解?[1]

以上两个问题目前的讨论还仅局限于"有没有解"[1],而通过本文的算法,不但知道有没有解,有多少个解,每个解的具体内容,而且还可以将其可解决的问题范围扩充到:

对于任何面积为 $m\times n$ 的矩形区域,5个俄罗斯方块每片都可以任意次摆放到该区域里(每片可以摆任意次,也可以不摆),共有多少种摆放方法,并计算每种摆放方法的图形。而上面提到的两个问题仅是这个问题的特例而已。当然,($m\times n$ /4)的结果必须是整数,因为每个俄罗斯方块的面积都是 4,所以覆盖区域面积必须要能被 4整除。需要注意的是本文所谓的"所有解决方案个数"是没有考虑到对称性的,即图形上对称的两个解或者是通过翻转,反转以后为相同的两个解仍算做是两个解。

此外只要覆盖的区域是由单位面积为 1×1的小正方形组成的话,即使是不规则的区域,方块不是俄罗斯方块中的那 5个,不管是任何的大小与形状,不管有多少块,都可以采用该算法解决。

图 1 俄罗斯方块

下面以 6×6的矩形摆放区域,每块俄罗斯方块都可以摆放任意次(任意摆放,可以不摆)的情况为例,介绍算法的思想。

2 算法思想

对 6×6的区域中的每一个面积为 1×1的单位小正方形,从左到右、从上到下的依次标上数字以标志每个小方格,如图 2所示。

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

图 2 数字的标法

假设将方形的俄罗斯方块放到该区域的左上角,则其覆盖的区域的标号为:1、2、7、8。

用一个长为 36的 1、0字符串,从前到后,在字符串第 n 位的字符对应正方形内 n 的编号的位置,其中 1代表该位

收稿日期:2004-12-08。王宇,硕士生,主研领域:自然语言理解,信息检索。

置被覆盖, 0代表该位置为空,用字符串表示如下:

11000011000000000000000000000000000000　共 36位

而将方形放到该区域最中间的位置,表示这个覆盖情况的 0, 1串为:

00000000000000011000011000000000000000　共 36位

由上可知, 5个俄罗斯方块中任何一片的所有的覆盖情况都可以用这样的 0, 1串表示。现在将俄罗斯方块的每一块的所有覆盖该区域的情况都表示成这种长度为 36的 0, 1串,每块都要经过翻转,反转,在该区域内移动并摆放,以将其所有的摆放可能全部表示出来,并将所有的串作为一个 0, 1矩阵的行全部加入到这个矩阵里,对于该问题这个矩阵共有 221行, 36列。后文将这个矩阵称为 A。则由于如果存在完全覆盖的可行解的话,每个单位小正方形就必须被某块俄罗斯方块覆盖过,并且只被覆盖过一次,则问题的算法思想可以抽象成:

在该矩阵 A内任意取出 n行作为一个新的矩阵 B的行,如果 B当中的每一列都有且仅有一个 1,那么这组行便是该问题的一个解[2]。

这个结论很好理解,一旦其中的某一行,即某个方块的摆放方法将该区域的某个位置覆盖的话,那其它方块就不能再次覆盖该位置,又因为要求是完全覆盖,所以是每列有且仅有一个 1。

而对于该问题,这个 n的值就为 9,因为面积为 36的区域需要 9块面积为 4的俄罗斯方块才能完全覆盖。选择其中的一行就相当于在该区域内摆了一个俄罗斯方块,所以只能摆 9个才可能有解。

这样就可以得出下面的算法 1思想:

算法 1　如果矩阵 A为空并且已经选过了 9行,则找到一组可行解,成功的返回;

否则选取 1个列, c,如果在 c列中不存在 1的话,不成功的结束该过程;

选择 c列中每一个 $A[r, c]=1$的行, r;

将 r值作为解决方案的一个步骤记录下来;

对于 r行中每一个 $A[r, j]=1$的 j值,从矩阵 A中将第 j列删除;

对于 j列中每一个 $A[i, j]=1$的 i值,从矩阵 A中将第 i行删除;

将已经缩小的矩阵重复进行以上运算;

算法中,将 r值作为解决方案的一个步骤记录下来就是尝试进行一种摆放并记录这种摆放的操作,在这之后的那些删除操作是要排除选择第 r行以后就不能再选的行,之后就是递归的进行下一步摆放的尝试。

3　两个问题的具体解决方案

有了这个思想,就可以知道其它的任何 $m\times n$的覆盖区域问题或者是其它问题只是初始矩阵 A的内容不同而已,都可以采用这种算法解决。像上文提到的问题(2),只要让 A中只有一种方块的所有摆放情况,即如果只用长形俄罗斯方块摆放的话,那么矩阵 A中就只能有所有的标志长形俄罗斯方块摆放的 0, 1串。只要分别对由每一块方块的所有摆放的 5个矩阵 A都分别进行这样的运算,就可知仅摆放那一种俄罗斯方块的所有解的情况;而问题(1),则只要在每个覆盖 0, 1串后加入 5位 0, 1串用来标志这 5个方块(是哪个方块哪位就为 1),假设长形的俄罗斯方块为方块 1,那么标志长形的 5位 0, 1串就为“10000”,对于第 2块方形的俄罗斯方块,标志就为“01000”。将这 5位数字与标志方块的每一种摆放的情况,长为 36位的 0, 1串连在一起,形成长为 36+5=41位的串,这个长为 41的串就标志着某个俄罗斯方块的某种摆法。直接用该算法,就可以让每个方块“出现且仅出现一次”。原因是新加入的 5个位是标志着哪一个方块,如果让其每一个位在使用该算法选择时也“出现且仅出现一次”的话,也就使得每一个方块也“出现且仅出现一次”了。

4　四向循环链表提高算法的效率

为了提高算法的效率,该算法并没有直接采用 0, 1矩阵而是采用了“4向循环链表”的方式,后文将这种算法思想称为“算法 2”,加入了一个回溯过程和一个“S字段”以提高算法效率。

首先清楚这个知识点:

假设元素 x属于一个双向链表,链表每一个元素由三个部分组成:①指向该元素前一个元素的指针;②指向该元素后一个元素的指针;③该元素本身的值。如果用 $L[x]$与 $R[x]$分别表示该元素的“前驱”与“后继”的话,那么下面的操作(注:符号“←”表示将该符号右边的值赋给其左边):

$$L[R[x]]\leftarrow L[x];\ R[L[x]]\leftarrow R[x] \quad (1)$$

会把 x从链表中移走,然后操作:

$$L[R[x]]\leftarrow x;\ R[L[x]]\leftarrow x \quad (2)$$

会把已经被移走的 x放回原来的位置。

步骤(2)这个方法的最大好处就是:只需要 x一个数据,就可以将该数据放回链表的原来位置[3]。

在这个算法当中,将矩阵 A用“4向循环链表”来表示。将 0忽略掉,为每一个 1创建 1个节点 x它包括 5个部分: $L[x]$(指向该节点的左面节点,如果该节点在最左面就循环指到最右面), $R[x]$(指向该节点的右面节点,如果该节点在最右面就循环指到最左面), $U[x]$(指向该节点的上面节点,如果该节点在最上面就循环指到最下面), $D[x]$(指向该节点的下面节点,如果该节点在最下面就循环指到最上面), $C[x]$(指向该节点所在列的“表头”,下面将做说明)。而每一列比原来又多出一个特殊的“表头”y,这个表头在每列的最上端, y除了有正常节点的 $L[y]$, $R[y]$, $U[y]$, $D[y]$, $C[y]$外,还比正常的结点多出两个部分: $S[y]$(该列里的节点个数,也就是 1的个数,主要用于减少初始分支,提高运算效率,这很好理解,因为要是一个列里面的节点数少的话,在这个列所确定的所有备选行的数量就少,自然减少了遍历的行数), $N[y]$(该列的名字)。除此之外,矩阵 A中还包含唯一一个叫“根”的节点 h,它位于所有表头节点的最左端,主要用于判断是否找到合适的解,只用到 $L[h]$与 $R[h]$,用于跟其它表头左右循环相连。

下面举一个例子,假设矩阵 A为:

$$\begin{matrix}0&0&1&0&1&1&0\\1&0&0&1&0&0&1\\0&1&1&0&0&1&0\\1&0&0&1&0&0&0\\0&1&0&0&0&0&1\\0&0&0&1&1&0&1\end{matrix}$$

那么,用算法 2的结构表示,则为图 3所示。

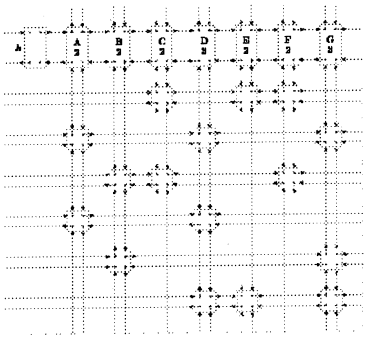

图 3 结构图示

记算法的名称为 Search(k)，算法的基本思想如下：初始时 $k=0$，算法里还有一个指针数组 O，用于记录解决方案所选择的行。该算法里有一个特殊操作“覆盖”，关于如何覆盖将在后面说明。

算法 **2** Search(k)

如果 $R[h]=h$，找到一个解决方案，输出并且返回；

否则选取一列 c（可以是第 1列，也可以按 $S[c]$选）；

覆盖 c列（什么是覆盖操作在下面说明）；

对于每一个 $r\leftarrow D[c]$，$D[D[c]]$，……while r不等于 c（即是由上到下全访问一次）；

赋值：$O[k]\leftarrow r$；（将该行记录）

对于每一个 $j\leftarrow R[r]$，$R[R[r]]$，……while j不等于 r（即是由左到右全访问一次）；

覆盖第 j列（下文说明）；

Search($k+1$)（进入新一层的查找）；

赋值：$r\leftarrow O[k]$并且 $c\leftarrow C[r]$（从该行开始，以下的操作都是为了回溯，它们的操作顺序与前面覆盖的顺序正好相反）；

对于每一个 $j\leftarrow L[r]$，$L[L[r]]$，……while j不等于 r

解除对第 j列的覆盖（下文说明）；

解除对 c列的覆盖并且返回。

所谓覆盖操作，就是将该列的表头从“水平”方向移走，这一步仅断开其 L，R就可以起到避免访问该列的效果了，这样做不但运算量少，而且后面移动行的时候还要用到该列里面的元素。然后将该列的所有节点所对应的行上的节点从“竖直”的方向上移走，这样就会消除与该列有冲突的行，而且还不干涉后面使用这些仅被在“竖直”方向移除的点。本质与算法 1是一致的。具体方法如下：

赋值：$L[R[c]]\leftarrow L[c]$并且 $R[L[c]]\leftarrow R[c]$；

对于每一个 $j\leftarrow D[c]$，$D[D[c]]$，……while i不等于 c

对于每一个 $j\leftarrow R[i]$，$R[R[i]]$，……while j不等于 i

赋值：$U[D[j]]\leftarrow U[j]$，$D[U[j]]\leftarrow D[j]$；并且 $S[C[j]]\leftarrow S[C[j]]-1$。

而解除覆盖的顺序正好与之相反：

对于每一个 $i\leftarrow U[c]$，$U[U[c]]$，……while i不等于 c

对于每一个 $j\leftarrow L[i]$，$L[L[i]]$，……while j不等于 i

赋值：$S[C[j]]\leftarrow S[C[j]]+1$；并且 $U[D[j]]\leftarrow j$，$D[U[j]]\leftarrow j$

赋值：$L[R[c]]\leftarrow c$并且 $R[L[c]]\leftarrow c$。

所谓顺序相反，是指如果覆盖的顺序是由左至右的话，那么解除覆盖的顺序就是由右到左；如果覆盖的顺序是由上至下的话，那么解除覆盖的顺序就是由下到上，这样才能按照原来覆盖的顺序返回到原来的链表状态。

该矩阵的 4向循环链表的表示形式如图 3所示。如果现在按照算法 2的头一次覆盖，要求覆盖第 A列的话，那么图形表示如图 4所示，被曲线围住的节点是将要被移除的节点，被曲线包围的箭头表示保留下来的指针箭头。

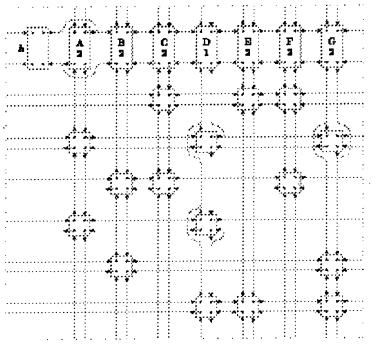

图 4 覆盖的图示

图 4不但演示了覆盖的操作，同时对这个问题而言，也是第一步覆盖以后的结果。表头标号为 A的那一列元素进行了覆盖，也就是将 A列中的两个节点所在的行上的所有节点覆盖，而且将 A列的表头覆盖。只要覆盖 A的表头就可以让程序再也找不到该列元素。

选择列 c的方法如下，先让变量 $s\leftarrow$无穷大：

对于每一个 $j\leftarrow R[h]$，$R[R[h]]$，……while j不等于 h；

如果 $S[j]<s$赋值：$c\leftarrow j$并且 $s\leftarrow S[j]$；

结果的输出方式如下：因为已经得到 O这个数组，那么输出第 k个所选行只要输出 $N[C[O[k]]]$，$N[C[R[O[k]]]]$，$N[C[R[R[O[k]]]]]$……便可。即只要循环输出从 $k=0$到最后的情况，所选的某一行中的每一个元素就被输出了。

该问题共有 1049组解。而在选择这 9个方块的每一块时，通过算法 2的实验，在每一个层次（就是在算法 2中的 Search(k)中的每一个 k值，即表示算法运行的层次，也表示现在在选择第几块）的运行分支情况由表 1所示。

表 1 解决 6×6问题算法的分支情况

层 次	分 支 数
1	7
2	29
3	82
4	241
5	651
6	1510
7	2974
8	4070
9	1409

通过实验，得出采用算法 1与算法 2对于该问题的效率由表 2、表 3所示，其中增删次数是指增加与删除存储单元的次数。

表 **2** 算法 **1**解决 **6×6**问题的效率表

层次	赋值次数	比较次数	增删次数
0	553206	74874	2754000478
1	10484482	1367434	6543785266
2	210118476	5212347	14567398548
3	1051402108	12316595	31038756738

4	2252296396	30667432	102846576836
5	4506541250	62986546	295732993832
6	6151408088	143456756	971846843216
7	3121435672	118305604	184856474372
8	815140204	30752846	7446785336
9	56646	2818	56646
总计	18119436528	405143252	1617633671268

表 3　算法 2解决 6×6问题的效率表

层次	赋值次数	比较次数	增删次数
0	23835	4141	256
2	87431	15391	416
2	212742	39209	1026
3	662584	119152	2588
4	1644409	302855	7200
5	3431566	653531	18726
6	5588301	1137710	42242
7	4491151	1161142	79542
8	657975	300894	100498
9	0	1409	33816
总计	16799994	3735434	286310

由此可见,两者的效率差距是巨大的, 2 4G的 CPU, 512M的内存的电脑上,算法 2可在一瞬间运算完毕,而算法 1则需要在 6个小时左右。其主要原因是算法 2就不必像算法 1那样保存每一步计算以后的矩阵 *A* 自然极大节省了存储空间,同时减少了比较与赋值的次数。

5　对于算法 2的继续改进

这是针对俄罗斯方块的特性进行的改进:每个俄罗斯方块都是由在竖直或水平方向的连续 4个小的 1× 1的单位小正方形组成,即其中的每个单位小正方形都在竖直或水平方向上至少与另一个单位小正方形相连,且这 5个俄罗斯方块是 4个单位小正方形在竖直或水平方向相连的所有情况。根据这个特性,在算法 1与算法 2当中选择一个方块的摆放(在算法 1, 2中都是选定,记录 *r*行的那句话)之前加入一个判断条件:

如果在覆盖区域中加入第 *r*行的这种覆盖以后,未被覆盖的区域如果存在竖直,水平方向相连的所有单位小方块的数量小于 4,那么就不选择 *r*行,改试选 *r*行的下一行。

举例来说,如果放入 *r*以后,未被覆盖的区域中有这样的区域(见图 5)。

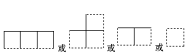

图 5　未覆盖区域基本形状

或者是通过旋转或反转本质是这几种图形,而周围又都是被覆盖的区域,即是说如果相连续的最大未覆盖区域是这样的话,那么就不要选择 *r*行。

这种方法的目的与 *S*字段一样,就是要减少不必要的分支。表面看来似乎需要对每个未被覆盖的单位小正方形(简称未覆盖正方形)所相邻的最大未覆盖区域进行判断,但实际上并不需要如此,通常情况仅需要判断 1个未覆盖正方形所处的未覆盖区域就可以确定 *r*行是否可放了。首先遍历该整个摆放区域找一个未覆盖正方形,查找所有与其在竖直或水平方向相连的未覆盖正方形,并对已经找到的未覆盖的正方形进行同样的递归查找其竖直或水平方向上未覆盖小正方形(注意是所有相连的未覆盖小正方形,即是其所处的最大未覆盖区域),被判断相连续的未覆盖正方形就没有必要再进行遍历来判断其所处的最大未覆盖区域是否合理了,原因是当判断完所有的连续情况时,如果这 1块未覆盖正方形所连接未覆盖的区域不合格,那么就没有必要去判断其他未覆盖正方形而直接退出了, *r*行不能选;如果是合格的,那么对于处在该未覆盖区域内的其它未覆盖小正方形所处的未覆盖区域也是这一个区域,所以没有必要再去重复判断,只要去判断其它不在这个区域内的未覆盖小正方形就可以了。如果所有的未覆盖区域都被判断过且合格的话,那么 *r*行就是可选择的。

而对于该问题,判断一个未覆盖小正方形的未覆盖区域(即为所有相关的未覆盖小正方形)的过程很简单,设这个小正方形的标号为 *n*,函数名为 Judge(*n*):

```
如果标号为 n的小正方形被覆盖
    返回;
否则
    该覆盖区域的面积总数加 1;
    记录该小正方形标号(以便以后不再重复判断其覆盖区域);
    如果 n除以 6余数不为 0(该小正方形不贴在 6×6区域的右边)
        Judge(n+1);          (向右遍历)
    如果 n除以 6余数不为 1(该小正方形不贴在 6×6区域的左边)
        Judge(n-1);          (向左遍历)
    如果 n大于 6(该小正方形不贴在 6×6区域的上边)
        Judge(n-6);          (向上遍历)
    如果 n小于 31(该小正方形不贴在 6×6区域的右边)
        Judge(n+6);          (向下遍历)
```

表 4是改进算法效率:

表 4　采用改进算法解决 6×6问题的效率表

层次	赋值次数	比较次数	增删次数
0	23056	2942	31
1	84388	10888	197
2	204305	26719	778
3	637941	82671	2209
4	1577958	204480	6435
5	3277509	425467	17134
6	5284928	688600	39214
7	4075998	546554	75446
8	514257	88135	99089
9	0	1409	33816
总计	15680340	2077865	274349

这些数据表明,对于小规模的运算,该方法是可以提高效率的。

而对于较大规模的运算,比如说将覆盖区域扩大到 22× 12(这是正规的俄罗斯方块游戏的用于摆放方块的面积区域),本文也做了实验,结果两者相差不大,但是采用这种方法的效率还是在以上三项上略微稍强。经过改良,这种算法在本文所进行的实验上证明其可以对中小规模运算提高效率。

(下转第 158页)

目标对象的周长表示为:

$$P=N_e+\sqrt{2}N_O$$

其中 N_e 表示链码中偶数码的数目, N_O 表示链码中奇数码的数目。

2.2 面积计算

与周长一样, 目标对象的面积只与目标对象的边界有关, 而与目标内部灰度级的变化无关。按像素中心定义的多边形的面积等于所有像素点的个数减去边界像素点数目的一半加 1 即:

$$A=N_O-\left[\left(\frac{N_b}{2}\right)+1\right]$$

式中, N_O 和 N_b 分别是目标对象的像素(包括边界内所有像素数目和边界上像素数目)数目。这种以像素点计数法表示面积的修正方法是基于这样的原理: 在通常情况下, 一个边界像素的一半在目标内而另一半在目标外; 而且, 绕一个封闭曲线一周, 由于目标对象总的来说是凸的, 相当于一半像素的附加面积是落在目标外的。因而可以通过减去周长的一半来近似地修正这种由像素点记数导出的面积。

2.3 高度和宽度计算

目标的高度和宽度指它在图像的垂直和水平方向上的跨度, 搜索目标对象的边缘链码, 可以得到该目标的最大和最小的行、列号, 其差值就是高度或宽度。

2.4 圆形度计算

有一组形状特征被称为圆形度指标, 因为它们在对圆形形状计算时取最小值。它们的幅度值反映了被测量边界的复杂程度。

最常用的圆形度指标是: $C=P^2/A$ 也就是周长的平方与面积的比。这个特征对圆形形状取最小值 4π。越复杂的形状取值越大。圆形度指标 C 与边界复杂性概念有着粗略的联系。

2.5 矩形度计算

反映一个目标对象矩形度的一个参数是矩形拟合因子:

$$R=A_O/A_R$$

其中, A_O 是该目标对象的面积, A_R 是其最小外界矩形(MER)的面积。R 反映了一个目标对象对其 MER 的充满程度。对于矩形目标对象 R 取得最大值 1.0, 对于圆形目标对象 R 取值为 $\pi/4$, 对于纤细、弯曲的目标对象取值变小。矩形拟合因子的值限定在 0与 1之间。

另一个与形状有关的特征是长宽比:

$$A=W/L$$

它等于 MER 的宽与长的比值。这个特征可以把较纤细的目标对象与方形或圆形目标对象区分开。

根据上述的理论, 在边缘跟踪记录集料颗粒数据结果的基础上, 可以得到颗粒参数的计算值见表 2。直观上可以根据这些参数分类, 但实际上, 集料颗粒的粒径是由方孔筛的孔径来度量的。由于集料颗粒分布的随机性, 而且所采集的图像仅是二维的, 不可能准确获得集料颗粒的特征值。但是, 由于分析结果在可以接受的误差范围内, 因此采用该方法来近似地描述特征参数集料颗粒的信息是有意义的。

表 2 颗粒参数

颗粒号	周长	面积	高度	宽度	圆形度	矩形度
1	48.97	110	7	20	21.80	0.786
2	103.11	356	17	41	29.86	0.511
3	88.43	353	19	27	22.15	0.688
4	81.70	305.5	30	17	21.85	0.599
5	220.65	2434	62	63	20.00	0.623
……	……	……	……	……	……	……
16	179.54	1360	28	72	23.70	0.675
17	144.57	1006	32	49	20.77	0.642
18	70.63	152	10	30	32.82	0.507
19	71.56	182.5	8	31	28.06	0.736

此外还需注意数码相机所拍摄到的物体图像要根据一定的比例对应于实际的物体。实际参数计算中还应考虑实际尺寸和所得图像的尺寸进行换算的问题。

3 结 语

本文对沥青混合料数字图像中较大集料颗粒的边缘轮廓跟踪算法进行了有益探索, 同时对跟踪结果进行了分析, 给出了相关特征参数的具体求解方法, 并用 Delphi 语言进行了编程比较、调试与实现, 为自动检测沥青混合料各种成份体积组成提供了可靠依据。

参 考 文 献

[1] 阮秋琪编著, 数字图像处理学, 电子工业出版社, 2001.第 1版.

[2] Caie Z G. Restoration of binary images using contour direction chain codes description. Computer Vision Graphics and Image Processing, 1988, 41: 101~103.

[3] 曹智威等编著, Delphi 6 实用编程技术, 中国水利水电出版社, 2002.第 1版.

[4] Chang L W., Leu K. L., A fast algorithm for the restoration of images based on chain codes description and its application. Computer Vision Graphics and Image Processing, 1990, 50: 296~307.

[5] 沈金安主编, 沥青及沥青混合料路用性能, 人民交通出版社, 2001.第 1版.

[6] 陆宗骐、童韬, "链码和在边界形状分析中的应用",《中国图像图形学报》, 2002.12

[7] 任明武、杨静宇、孙涵, "一种新的基于链码描述的轮廓填充方法",《中国图像图形学报》, 2001.4.

[8] J. D. Frost, J. R W right: Digital Image Processing Techniques and Applications in Civil Engineering. American Society of Civil Engineers, 1993(1).

(上接第 153页)

参 考 文 献

[1] 刘江枫、刘达儒, 俄罗斯方块的数学[J],《数学传播》第 23卷, 第 1期, 1999.

[2] Solomon W. Golomb. Polyominoes[M], second edition (Princeton: New Jersey: Princeton University Press), 1994.

[3] Robert W. Floyd. Nondeterministic algorithms[J], Journal of the ACM, 14: 636~644. 1967.

[4] Hirosi Hitotumatu and Kohei Noshita. A technique for implementing backtrack algorithms and its application[M], Information Processing Letters, 8: 174~175. 1979.