实现 pushpush in 3D

——<计算几何>试验报告

● 问题分析

"pushpush in 3D",翻译成"三维推箱子问题".

先谈一下问题本身.这个问题的提出是为了解决具有一定能力的机器人在一个特定格局的路径里清除有一定特性的障碍之后能否从给定起点到达给定终点的问题.

在这个问题中，涉及到三个对象，机器人，障碍物，和路径，下面对这三个对象分别加以解释:

机器人.有移动能力,并能推动一个障碍物(箱子),但不能推动两个以上的箱子.并且只能在路径上移动,同时也不具备拉箱子的能力.

障碍物，即箱子.被推后能顺路径无阻力的滑至遇到另一个障碍物(箱子)或"墙壁"为止.(该问题中,平面上除了路径之外全是墙壁,而机器人,障碍物均只能在路径上静止或移动)

路径.路径并非自然的路径,而是与某一和取形式的逻辑表达式相对应.路径实际上分三种:变量,子句,联接.若某变量在该和取式的某子句项中出现,则从该变量到那个子句有一个联接路径,并在靠近子句的地方有一个障碍.

三者的关系以及任务在这里的具体体现:

机器人可以在路径上自由移动,只要不碰到障碍物和到达路径的尽头.而障碍物的初始分布很有规律,每个变量头部又一个障碍物,以确定机器人走变量的哪一个分支;每个变量的尾部有两个障碍物,确保机器人可以在原分支中行走,可以走出变量,但又阻止机器人从尾部进入另一个分支.在连接变量和子句的路径上,靠近子句有一个障碍物,若被推动后会滑到子句上,确保机器人在后面的行走中可以走过那个子句.机器人的任务就是从起点穿过串联的变量,在走过变量时有可能接近某个子句去开启那个子句,然后穿过串联的被开启过的子句,最后到达终点.这中间若是有一个子句没被开启,则任务不能完成.

● 具体实现

由于在 2D 中已实现了 CrossOver 的构造,所以在 2D 中实现也是可能的,但工作量太大.所以便选择在 3D 中实现.

实现分两大部分:

(一) **求解.**

求解即在给定和取形式的逻辑表达式后求出相应的解,若无解便警告,有解便列出以供选择一组解.

在这里对程序所给的求解方法给出比较详细的解释.

先对求解的问题给一个描述,本部分解决的问题如下:

给定一个逻辑表达式的标准的合取范式,求出它所有可能的解.对输入作了如下规定:用~表示非操作,&表示与操作,|表示或操作,变量以字符 x 后加标号(1--9)表示;每个子句用()括起来,子句内只能有变量和合法的或非形式,每个变量在一个子句内只能出现一次;子句之间(即括号之间)只能用&来连接;一个合取范式由若干子句组成.那么一个标准的合取范式，以三个变量为例，可以有(x1|~x3)&(~x2|x3),(x2|x3)&(~x2|x1)这样的形式.

对这样的一种表达式,只要输入变量在 x1 到 x9 之间,并满足上面的输入要求(对子句的多少没有限制),便可以进行求解,并给出所有的解集.

以一个具体的例子讲述求解过程.

(x1|~x3)&(~x2|x3)

要求的这个表达式的真解,我们先说一下总体的思路,在说明在程序中怎么实现.

对这类形式的表达式,由于各个子句之间是与操作,那么我们只要求出各个子句的真解的集合,最后进行交运算,那么我们就可以有交集得出最后的真解.然而在计算机中这种思路实现起来有些麻烦.我们稍微作一下转化,我们先给出所有可能出现的解的集合,对上例而言,有三个变量,(x1,x2,x3),实际上有(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1)这八种可能的解.用一个长度为 8 的整形数组,表示这八个解.并对数组全部置成 1.

我们先求出令第一个子句成假的解的集合,并对其中的每一个解对应的数组中的值改为 0,那么数组中值仍为 1 的项对应的解便是使第一个子句为真的解.然后,我们接着求第二个子句的假解,并把相应的数组中的某些项的值置成 0,这个时候,数组中值为 1 的相对应的解实际上是令第一个子句和第二子句都为真的解.

这样,我们的这个例子实际上已经得到了解决.而且,这个解决的方法对这种类型的所有的逻辑表达式都通用.这样,我们就可以得到这一类型逻辑表达式的通用解法.步骤如下:

<1> 先对整个逻辑表达式进行扫描,找出变量出现的个数,设为 VariableNumber,并申请一个大小为2的VariableNumber的幂的整形数组(用位来描述会更省空间),每一项对应一个一个解,方便起见,就用下标来表示该组解.如三个变量,那么 ResultSet[6]的下标是 6,是 110,即表示(x1,x2,x3)为(1,1,0)的这组解.那么 ResultSet[6]的值就表示(1,1,0)是否为真解,如果 ResultSet[6]为 1,那么就表示(1,1,0)是表达式的一组解,为 0 就表示不是表达式的一组解.先对数组项全部置 1,默认全部为解集.

<2>完成扫描和申请空间后,就对逻辑表达式的第一个子句进行分析,即对第一个表达式求出假解集并在 ResultSet 数组中进行相应的标识,即在相应项标 0.完成第一个子句分析后,接着进行第二个(如果有的话),第三个...,直到结束.

<3>对逻辑表达式的分析完成后,最后对 ResultSet 数组进行扫描,只要某一项的值为 1,那么该项下标对应的解就是一组真解,把该解加入真解列表,全部扫描完成后,真解列表中的集合就是真解集合.至此,求解过程完毕.

由上面<1>,<2>,<3>三步,就完成了合取范式的逻辑表达式的求解.

在实际实现中注意以下几点:

★语法分析.在对表达式求解时,实际上由一个语法分析过程,能够识别大部分的输入错误,但有些问题没有细化,例如不识别空格,当表达式中出现空格时,将报错;在同一个子句中若多次出现同一个变量,不报错,但程序运行结果将有问题,但只要正常输入,程序将正常工作.

★对一个子句的求解问题.一个子句中不允许两次以上出现同一个变量,但允许不足额出现所有变量,例如(x1|~x4),x2 和 x3 就都没有出现,这是允许的,这在程序里进行了解决.用了一个递归函数实现.每个子句对应一个"原解",用 OneResult 记录,例如(x1|~x4),那么它的"原解"里记录的是(0,2,2,1),意思是假解为 x1=0,x4=1,x2,x3 为 0,1 两种可能的情况.那么用递归函数进行遍历,便可以遍历 (0,0,0,1),(0,0,1,1),(0,1,0,1),(0,1,1,1)这四个假解,并给下标为

0001(1),0011(3),0101(5),0111(6)的 ResultSet 数组项置 0.(这里假设只有 x1--x4 四个变量)

★在求解过程中,建立了一个链表,这个链表中的记录记录了某个变量的一些情况。b 表示变量 x 在该子句中的取值为 b,x 表示这个记录是关于第 x 个变量的,而 c 则表示 x 出现在第 c 个子句中。并按照 x,c 排序,即先按 x 从小到大从前往后排,x 相同时,按 c 从小到大从前往后排.这个链标其实是为程序的画图部分提供必要的数据.

(二) **画图**.

在实现中,以一个方块代表机器人,实心圆代表箱子(障碍物),直线代表路径.在求解完成后作如下工作:若无解,仅画出该逻辑表达式对应的初始图形,无动画出现.若有解,在用户选择一组解后,便按照该组解,由机器人完成相应的移动,走变量,推箱子,走子句,最后到达终点.

在这里对程序所给的画图方法给出比较详细的解释.

这部分程序的输入部分有: 一个由逻辑表达式建立的链表, 链表按变量和子句的顺序依次从小到达排好了顺序,详细叙述见对求解部分的描述;还有一个由操作者在有解的情况下选定的一组解.

本部分完成的工作就是根据这个链表画出逻辑表达式对应的总体图,根据给定的一组解产生动画描述机器人从给定起点走到给定终点的经过.分两步进行:

(1)根据逻辑表达式确定的链表画总体图.绘制总体图分三个步骤来完成:

1>绘制"变量图".为每个变量绘制一个"变量图",根据子句个数确定每个"变量图"的长度.每个"变量图"包括变量头,变量体,变量尾三部分.完成后只是一个空架子,并不包括从中引出的路径.

2>绘制"子句图".为每个子句绘制一个"子句图",根据变量个数确定每个"子句图"的长度.每个"子句图"包括子句头,子句体,子句尾三部分.完成后只是一个空架子,并不包括向它引入的路径.

3>绘制"主框架".对每个变量,在它的"变量图"中引出路径画至含有它的所有子句对应的"子句图";并根据变量在某一子句中的取值情况判断是否绘制 xor 门(用一个小长方形代替),根据路径交叉情况在交叉点上绘制 crossover 门(用一个十字的图形表示);再画出连接变量图的尾部和子句头部的连线.

(2)根据选定的一组解来产生动画.为产生动画,先做准备工作.即提前预备好所有可能移动的对象在动画过程中可能出现的位置.在这里,有可能移动的对象有两种:机器人,道路上的障碍物,机器人只有一个,用一个小方块来表示.而障碍物则有很多,用实心圆来表示,它们分布在以下地方:每一个"变量图"的变量头有一个,变量尾有两个,每一个"子句图"的子句头有一个,在由"变量图"指向"子句图"的每一条路径的靠近"子句图"的部位有一个.

具体实现采用以下方法:

用一个链表 0,其中一个节点对应一个移动对象及其当前的位置.然后对给定的解所对应要走过的路径进行提前计算并储存.由于对象移动实际上是分段进行的,某一段是机器人在移动,另一段可能又是障碍物在滑动,所以一段一段的记录总的路径是合理的,这样对于每段可以方便地描述在它上面移动的对象及其位置.所以就用另一个链表 P,每个节点对应路径的某一段中,并记录在该段由哪一个对象移动及其要移动的起始位置和终止位置(即这一段的起始位置和终止位置).那么当所有要走的各段路径走完后,总的路径也就走完.

看看移动开始后程序运行的具体情况:Ontimer()(windows 时钟事件)每触发

一次,就去对链表 P 进行查询,看当前在哪一段上移动,并由哪一个对象移动.如果一段没有完成,就由该对象继续在该段中移动.如果一段移动完成,就去读下一段的信息,找出要移动的对象,由它走完这一段.为了方便对链表 P 进行操作,对走过的部分进行了删除,这样当查询到链表 P 为空时,路径就走完了.

值得注意的是,Ontimer()每触发一次,实际上是对链表 P 进行查询,找到要移动的对象及要在哪段上移动,计算出在该段上移动一步后该对象的位置.然后在链表 O 中找到相应的对象,把前面刚计算过的位置赋给这个对象,然后对 O 中的所有对象重画一遍,于是便有了移动的效果.移动的对象只在某段上移动一步,而每段都可能要走很多步才完成.走完一步,在该段中删去这一步(具体实现见程序);当走完一段后,就删去这一段(实际上P就删去了一个节点),所以当P为空时,动画操作完成.

在这儿补充几点:

★在第一阶段画总体图时,实际上已经建立了移动对象的链表 O,并给定了初始位置.所以在画总体图时,遍历了链表 O 并进行了绘制.这就是无解或没有选取一组解时,仍能看到各个对象的原因.

★当无解或没有选取一组解时,链表 P 实际上没有建立,在 Ontimer()触发时,并不能对 P 进行查询,所以并没有出现动画,这也是合理的.

● 总结

在求解部分,由于在实现时设计了一般情况,所以只要机器条件允许,用户有足够的耐心,并对求解过程的语法分析稍作修改(程序中只允许出现 x1--x9),便可目睹在给出任意合取表达式,任意多的变量,并有解的情况下的"push"过程.但实际上变量以增多,空间消耗以幂的形式增大.

在动画部分,最终采取了用 windows 时钟做动画触发器.这样有闪烁的问题.做动画时尝试过另外两种办法.一种是用位图法,即对背景和移动对象都采用位图形式画出,但是需要的内存很大,而且在拖动时进行背景位图和移动对象的位图的显示部分的计算过于复杂.另一种方法是用 DirectX 技术,但是其与硬件的结合太过紧密,经常白屏死机,也就没敢进行下去.所以最终还是采用目前的画图方法.

吕浩进　　王晓娟
006109　　008045

2001,01,10