平面嵌入

四川绵阳南山中学 古楠

[引子]

什么是平面嵌入呢？大家还记得冬令营 2005 的蜂窝玉米吗？它涉及到了图的一个性质，那就是所有的边都不能相交。图这个性质叫做图的平面性。当我们需要知道一个图是否具有平面性和这张图实现平面性后的结构时，平面嵌入算法就是一个好的工具。

[摘要]

本文主要由两大部分构成。

第一部分主要介绍了平面嵌入的算法，其中包括了具体的操作，复杂度的分析，正确性的证明和一些相关题目。第二部分主要是附录，包含参考文献和伪码。

[关键字]

平面，嵌入，内部活跃，相关，外部活跃，块，根边，回落边

[目录]

[算法]

1.平面嵌入的相关定义

如果对平面嵌入还有些陌生，希望下面的定义对你有所帮助。

（1）平面作图：一张图能够转化会为一张所有边都不相交的图（在节点上相交不算），转化过程就叫做平面作图。（原来相连的节点在转化后依然相连，图 1 展示了平面作图）

（2）平面图：一张图能够进行平面作图它就是平面图，否则为非平面图。

（3）平面嵌入：和平面作图是等价的，不过在储存方式上是这样的，对于每个节点都顺时针储存和它连接节点。我们将记录用的表叫邻接表。

注意：本篇论文所有的平面都是指这里的相关定义，和几何平面是不同的。

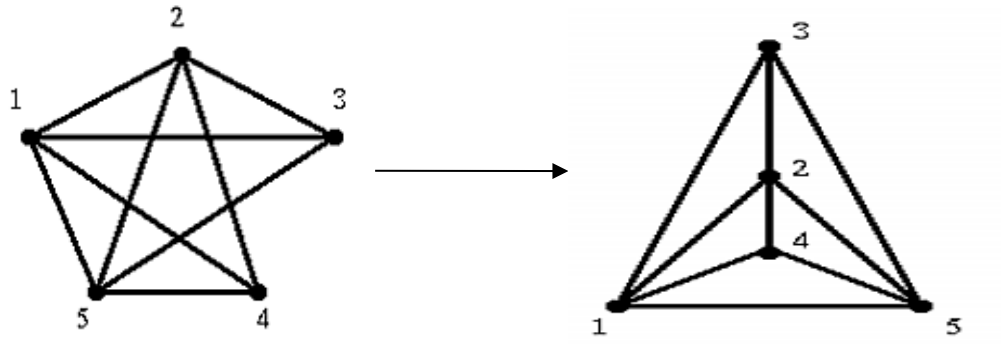


图 1

2．算法的目的

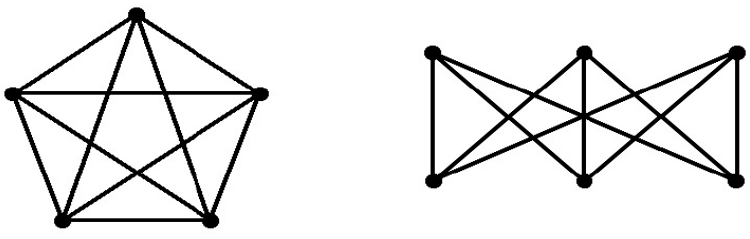
在明白了平面嵌入的一些基本定义以后，对于给定的图 G，它有 n 个节点，m 条边。（以后我们将始终用 n 表示图 G 的节点个数，m 表示图 G 的节点边数）算法的目的就是用 O(n)的时间判断一个图是否为平面图，如果是的话要用 O(n)的时间实现平面嵌入。

3．一些相关的知识

为了实现我们的目的，我们必须先知道一些必要的知识。这些知识将包含深度优先搜索，双连通分量，关节点，计数排序（一个复杂度和关键码范围有关的算法），还要知道一些必要的定理，如一个平面图的边将不能超过 3n-5 条边，否则它将是一个非平面图。Kuratowiski 曾经证明了两个图将阻碍平面嵌入的进行，那就是图 2 所描述的图，我们称作 k5 图和 k33 图。一张图中如果包含了 k5 图和 k33 图的同胚图（如果一个图可以通过延展，弯曲，合并连接节点的方式转化为另一张图，这两张图就是同胚的），那么这张图一定是非平面图，反之也成立。

我们的算法有一个基本操作，就是加边操作。我们创建一个图 GP，我们要将原图中的边按照一定的顺序加入图 GP 中。我们把这个操作称作边的嵌入。

为了叙述的通顺性，有关复杂度的分析和正确性的证明我都会放到论文的末尾。



K5 图　　K33图

图 2

4．算法总览

为了后面的叙述更加清晰，这里将叙述算法的总体过程。

我们只讨论对双连通分量图的平面嵌入，对于有关节点的图，我们可以将图从关节点处断开分别进行平面嵌入后再合并。

首先对图进行一次深度优先遍历，得到一棵深度优先搜索树和图的逆向深度优先搜索序（由于深度优先搜索序简称 DFI 序，所以后面写作逆向 DFI 序）。我们将每个节点和它儿子相连的边叫做树边，将和它后裔相连的边叫做回落边（注意：我们指的后裔是不包括儿子的）。

然后我们按照逆向的 DFI 序依次处理每个节点。所谓处理节点就是：首先将它的树边嵌入图 GP 中，然后将它的的回落边都嵌入到图 GP 中，我们并不处理它和它祖先相连的边（祖先是包括父亲的）。如果嵌入过程失败，那么可以得到图 G 是非平面图的结论，否则就完成构造。

5．边的嵌入

我们算法最重要和操作就是对边的嵌入，我将针对图 3 进行详细的说明，在图 3(a)中，我们要嵌入边(v,w)。在开始的时候该连通分量包含了一个关节点 r，我们将 r 去掉后该连通分量就变成了两个部分，就象图 3(b)。然后分别给两个分量配上一个 r, 这样 r 就被分成了两个，象图 3(c)。然后将边（v,w）嵌入，然后将两个连通分量合并在一起，这时候，r 已经不再是关节点了，象图 3(d)。这样我们就完成了一个最基本的嵌入操作。

为什么要这么麻烦呢？大家注意到图 3(d)了吗？它的下半部分左右颠倒了，这也是嵌入时候的一个操作，叫做反转操作，这个操作是为了保证图 GP 的一个重要性质，这些都将在下文中提到。

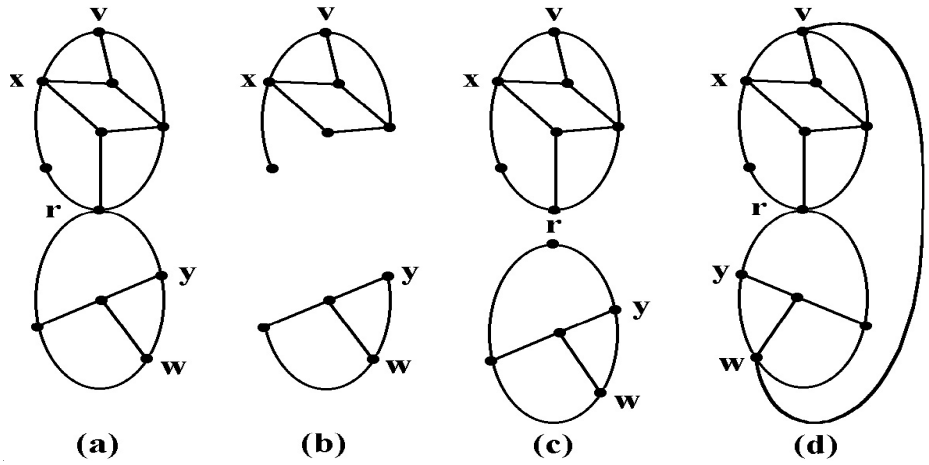


图 3

在我们的嵌入过程中，只要图 GP 中含有关节点，我们都将从这个关节点把它所在的连通分量断为两个部分。在嵌入回落边的时候又会将他们组合在一起。这样，图 GP 就是由很多的双连通分量组成的，请大家注意，在代码的实现中我们必须实际的这样处理，并不是为了理解。

6.外部活跃，相关与内部活跃

反转操作所要保护的性质就是：在嵌入边的过程中所有的外部活跃节点都必须留在外部面上。什么是外部面呢？外部面的名称很形象，也就是图中接触到最外层空间的点和边构

成的面，并且外部面都会是完整的环，如果一个双连通分量只有一条边，那么我们就要进行相应的转化，像图 4 这样。

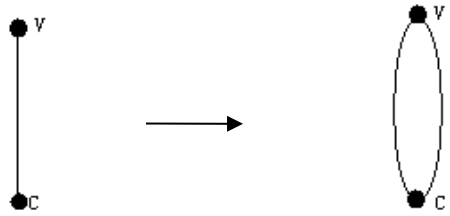


图 4

外部活跃节点的定义就要复杂些，为了清楚的叙述，我们要加入些新的定义。

我们将双连通分量中DFI序最小的节点称作该双连通分量的根节点。由于我们在只有嵌入树边的时候才会产生分离操作，所以我们会发现当一个关节点被分离成多个后，和儿子分到一个分量中的节点一定会成为该分量的根节点 。我们把这些节点称作复制点，把唯一的不是根的点称作原节点。假设r是一个双连通分量B的根节点，那么r 在该双连通分量中一定只有一个儿子，若r有两个儿子c1,c2，由于是深度优先遍历，在访问到c1后，继续访问的过程中将通过双连通分量B的某一条路径不经过r 到达c2,那么c2将不会成为r的儿子，也就与假设矛盾。我们设定r 的这个唯一儿子为c，那么我们将r 到c 的边称做根边（由于我们只有在加入树边的时候才会导致双连通分量的拆分，所以根边一定是树边），将该根节点称作 r^c节点（儿子还没有确定的我们写作r’），将该双连通分量称做r^c块。这样就可以和原r 节点区分开并且双连通分量也可以明确表示（后来的叙述都将遵从这里的设定）。

子块的定义在这里就产生了，我们将r^c块称做原节点r 的子块。

知道了这些后，只要满足下面两个条件的任何一个，它就是外部活跃节点：

(1) 当我们在处理节点 v 的时候，该节点处理过，并且有直接连接 v 的祖先的边。

(2) 当我们在处理节点 v 的时候，该节点处理过，并且该节点的子块中存在外部活跃节点。

就象图 5 中，方形的点都是外部活跃节点。

那么我们怎么得到外部活跃节点的信息呢？我们把每个节点本身能够直接到达的最早祖先的深度记做该节点的 ecpoint，把能够直接或者间接到达的最早祖先的深度叫做该接点的 lowpoint(所谓间接就是通过它的子孙到达)。同时，我们给每个节点配备一个

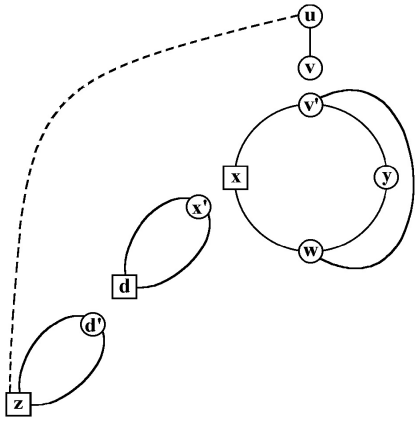


图 5

图 5：*z* 有直接连接 *v* 的祖先 *u* 的边所以，*v* 是外部活跃节点,，*z* 在 *d* 的子块中，所以 *d* 也是外部活跃节点，*d* 在 *x* 的子块中，所以 *x* 也是外部活跃节点。

SDlist ，它是一个双向链表，其中记录了它的每个儿子的 lowpoint。并且，SDlist 中的 lowpoint 值是按照从小到大排序的，为了满足这个条件，我们需要一次排序。因为我们的 lowpoint 值和深度有关，所以深度大小不会超过 n。在求出所有节点的 lowpoint 后，就可以用计数排序将它们按照从小到大排序，再依次加入它们父亲的 SDlist 中。那么如何快速的得到 lowpoint 呢？在深度优先搜索的时候，可以顺便得到它的 ecpoint，在深搜返回的时候再将它的 ecpoint 和它的儿子中最小的 lowpoint 值做比较中，返回较小的给它父亲。这样就可以用 O(n)的时间得到所有的 lowpoint 和 SDlist。当需要知道一个节点的外部活跃信息时，只需要看它的 ecpoint 和 SDlist 的第一个值是否小于当前处理节点的深度。在以后嵌入回落边的时候，儿子和父亲会合并到一个双连通分量中，那么将该儿子在父亲的 SDlist 中的值删掉，这样，我们就可以用常数的时间判断一个节点是否是外部活跃节点。

和外部活跃节点类似的定义还有相关节点，只要满足下面两个条件它就是相关节点：

（1）在处理节点 v 的时候，该节点处理过，并且和 v 有直接连接的边。

（2）在处理节点 v 的时候，该节点处理过，并且该节点的子块中存在相关节点。

注意：一个节点可以既是外部活跃节点，又是相关节点。

和外部活跃节点对立的是内部活跃节点，内部活跃节点就是节点是相关节点，但是不是外部活跃节点。非活跃节点就是既不是外部活跃节点，又不是相关节点。

和这些定义对应的还有：

外部活跃块：包含外部活跃节点的双连通分量。

相关块：包含相关节点的双连通分量

内部活跃块：包含相关节点，但是不包含外部活跃节点的双连通分量。

非活跃块：不包含外部活跃节点，也不包含相关节点的双连通分量。

我们将通过一个叫 walkup 的函数得到这些信息，后面将具体提到。

7．反转操作

反转操作的目的已经在上文中提到过，那就是为了保证在边的嵌入过程中所有的外部活跃节点都留在外部面上。在图 3 中，由于 y 是外部活跃节点，所以在嵌入边前要将下面的双连通分量进行反转，这就是反转操作的形象化体现。那么如何快速的在代码中实现反转操作呢?

最简单的方法就是将该块所包含的所有节点的邻接表都颠倒一次。但是这么做太慢了。记得上文所说的根边吗？每个双连通分量都是只有一条根边的，并且根边一定是树边。我们先将所有树边都初始化一个值，为+1，当一个双连通分量需要进行反转操作，那么我们把它的根边上的值变做-1。判断一个当前节点是否被反转只需要知道从根节点只经过树边到达它的路径上有多少个-1，如果是奇数个，那么该节点就需要反转，否则就不需要反转。在处理完所有的节点后，我们只需要进行一次深度优先搜索，便能够知道所有节点的反转信息了。

由于我们是按照逆向 DFI 序处理每个节点，并且所有的外部活跃节点都留在外部面上，这样，我们每次加入回落边的操作都将在外部面上进行。所以我们在处理节点的时候只需要遍历双连通分量的外部面。这个原因导致我们需要对双连通分量的外部面进行单独的记录。因为外部面都是完整的环，所以对每个外部面上的节点，都记录下与它相临的两个外部节点。那么反转操作对外部面的影响又该怎么处理呢？其实对外部面的遍历我们是不需要一直知道到底是逆时针还是顺时针的，我们只需要知道根节点第一次走的是顺时针还是逆时针

然后知道当前节点是由哪一个相邻节点到达的，那么就走向另一个节点。所以当一个双连通分量需要反转的的话，我们只需要交换它的相邻信息。图 6 对这一操作进行了形象的说明。

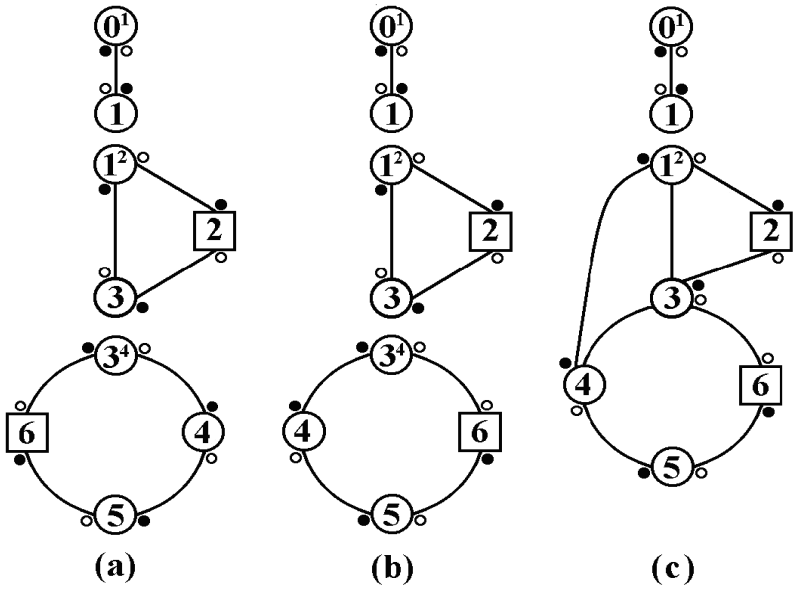


图 6

图 *6*：在图中，我们用黑色和白色的小点来表示方向以便说明。其中白色指顺时针方向，黑色指逆时针方向。在图 *6(a)*中，我们要嵌入边（1^2，*4*），同时要合并块和 1^2块和 3^4块，其中，方形的节点都是外部活跃节点。在图 *6(b)*中，为了能够将 *6* 号节点留在外部面上，我们将下面的块进行了反转。在图 *6(c)*中，我们只需将 *3* 的原来指向 *6*，反转后指向 *4* 的黑色圆指向 *2* 节点，就完成了外部面的反转操作，在 *3* 节点不是根节点后它的顺逆时针方向也不再重要。

8．一个全面的分析

在知道了这么多的信息后，下面将对平面嵌入进行一个相对全面的分析。

将图 GP 初始化为一张空图。（接下来的操作是要将图边嵌入到图 GP 中）

首先将按照逆向的 DFI 序依次处理每个节点。在处理节点 v 的时候，需要知道它有哪些回落边，当有回落边(v, w)的时候，从 w 沿着外部面向上走到 v,得到相关的一些信息，我们把这个函数叫做 walkup, 然后从 v 向下遍历进行边的嵌入，我们把这个过程叫做 walkdown。如果嵌入不成功，将证明该图是非平面子图，可以找出这个图中阻碍平面嵌入的子图，否则将完成平面嵌入。

那么如何在代码中实现边的嵌入呢? 这很简单，假若我们需要嵌入边（v, w），因为我们是顺时针记录节点的相邻信息，所以当我们是顺时针遍历的时候就将 w 节点加入 v 节点的邻接表头，反之加入邻接表尾。一开始只有树边的情况我们统一加入邻接表头。

我们来看看伪代码：

（1） 对图进行一次深度优先遍历，并且计算出每个节点的 lowpoint。
（2） 将图 GP 初始化，包括为每个节点配备上 SDlist，并且将 SDlist 中的 lowpoint 排序。
（3） For 节点 v from n-1 down to 0（按照逆向的 DFI 序处理每个节点）。
（4） For 每个儿子 v 的儿子 c
（5） 将根边(V^c, c) 嵌入图 GP 中
（6） For 每个回落边(v, w)
（7） **Walkup**(GP, v, w) //这个函数用来得到一些信息。

（8） For v 每个 v 的儿子 c
（9） **Walkdown**(GP, V^c) //这个函数会对回落边进行嵌入。
（10） For 每个回落边（v, w）
（11） If (V^c, w)不在图 GP 中 //也就是嵌入失败
（12） Return (非平面，GP)
（13） RecoverPlanarEmbedding(GP) //得到平面嵌入
（14） Return (平面，GP)

大家一定注意到了用黑体注明的两个函数，它们都将在接下来有全面的叙述。

8.1 walkup 函数

Walkup 函数是用来在处理节点 v 的时候得到相关信息的。

当我们在处理节点 v 的时候，我们将要嵌入回落边(v, w)，那么我们将要从 w 节点沿着外部面向上遍历到它的祖先 v。为了记录相关信息，我们给每个节点配备一个表，叫做 proots，记录它有哪些子块是相关块。我们在向上遍历的过程中，当从根节点r^c上升到它的原节点 r 的时候，我们就将r^c加入 r 的 proots 中, 这时候 r 也成为了相关节点。

w 在沿着外部面向上遍历的时候可以选择顺时针方向和逆时针方向，并且这两条路径很可能长度差距很大，如果任意选择一个时间消耗会很大 。那么如果对两条路径同时进行遍历，最多经过较短路径的两倍长度就可以到达块根节点。（这个操作其实保证了算法的复杂度，在后面的叙述中可以体会到）

还有一个问题是：两条回落边(v, w1), (v, w2)导致的遍历中，很可能公共部分是很大的，我们并不想重复的遍历，所以我们给每个节点配备一个 visited 值，在处理节点 v 的时候，我们就将所有访问到的节点的 visited 值置为 v（这样在处理其他节点的时候所有节点的 visited 值都自然清空）。当我们在做向上遍历的时候，访问到一个节点如果 visited 值为 v，那么我们就可以停止对该回落边导致的遍历了。这个操作借助图 7 给大家做详细的说明。

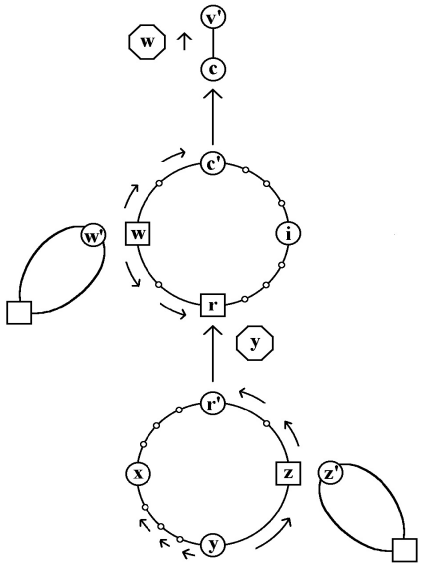


图 7

图 *7*：图 *7* 中的小点我们借以表示很多的节点，方形的点表示外部活跃节点。图 *6* 中首先我们要处理回落边 *(v, w)*，所以我们从 *w* 开始沿着两条路同时遍历。遍历到 *r* 和 *c'*，发现 *c'*为根节点，所以停止对该双连通分量的遍历，上升到 *c*，然后遍历到 *v'*。接着我们要处理回落边（*y, v'*），我们从 *y* 沿两条路向上遍历，通过外部活跃节点 *z* 先到达 *r'*。然后上升到 *r*，发现 *r* 已经访问过，所以终止遍历。

INT： Walkup 函数的伪码参考附录第 2 节。

8.2 walkdown 函数

Walkdown 是利用相关信息对回落边进行嵌入的函数。大概处理是这样的：类似 walkup 它也是在外部面上进行操作。但和 walkup 不同的是，walkdown 的遍历中是不能经过外部活跃节点的（有一个特殊情况例外，后面将提到）。在处理节点 v 的时候，需要从它的每个子块向下进行两次遍历，既顺时针方向遍历和逆时针方向遍历，来嵌入所有的回落边，并且处理嵌入后的影响。在遇到 proots 不为空的节点，需要下降到它子块中进行遍历的时候。不过将要重新选择方向。

当我们在向下遍历的过程中访问到一个节点，它是外部活跃节点但不是相关节点的时候我们就会终止遍历，所以我们把该节点叫做终止节点。由于终止节点的存在，如果过早的终止，将导致需要嵌入的边没有被嵌入。所以我们的程序必须遵从一个原则：尽量晚的终止遍历。为了这个原则我们遍历的顺序将受到两个法则的约束，这将在下文中提到。

当我们遍历到了节点 w，我们会遇到这些情况，他们要按顺序判断并且处理，图 8 会对这些情况有详细的说明：

（1） 当节点 w 是相关节点的时候，我们首先嵌入边(v, w)。

（2） 当节点 w 不是外部活跃节点的时候，我们走向下一个节点。

（3） 当节点是外部活跃节点但不是相关节点的时候，我们终止这次遍历。

（4） 当节点是一个相关节点并且是关节点的时候，我们向下遍历到它的子块。（这就是唯一可以经过外部活跃节点的情况）

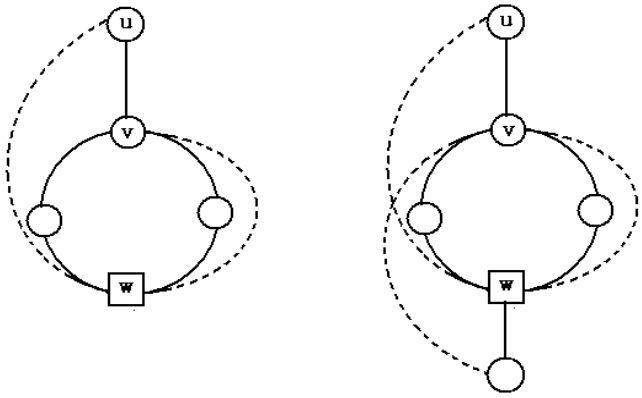


图 8

图 *8*：图 *8* 所示的是在处理节点 *v* 的情况下，在左边的图中，*w* 既是一个相关节又是一个外部活跃节点，首先我们将回落边嵌入图 *GP* 中。现在它就是一个外部活跃节点但是不是相关节点了，所以就成为了一个终止节点，于是终止遍历。在右边的图中，*w* 依然既是外部活跃节点又是相关节点，但我们在嵌入回落边后，*w* 依然是相关节点。所以它不是一个终止节点，我们下降到它的子块中继续遍历。

如果我们在需要下降到子块的时候发现该节点有很多相关子块怎么办呢？我们采取的方式是先下降到内部活跃子块中，由于内部活跃子块中不存在终止节点，所以定会返回到该节点，这时候我们就可以遍历它的其余子快。这就是 walkdown 执行时必须要遵从的，我们把它叫做法则 1。

法则 1：当下降时遇到多个子块需要遍历，我们总是先下降到内部活跃块中遍历，在返回后再下降到其他的子块中进行遍历。

怎样保证法则 1 呢，我们可以注意到内部活跃子块的 lowpoint 都是等于 v 的，而外部活跃子块的 lowpoint 都是小于 v 的，所以我们只需要将 lowpoint 等于 v 的子块加入原节点 proots 的表头，然后顺次处理就可以了。

那么下降到一个相关外部活跃子块中时，方向又应该怎样选择呢？为了我们的原则，方向选择必须遵从这样的法则：

法则 2：首先选择能够直接到达内部活跃节点的方向（直接就是不经过其他相关或者活跃节点），然后选择直接到达相关的外部活跃节点的方向。当遍历到终止节点或者根节点的时候，终止遍历。

当嵌入边的时候，该边就会覆盖当前所经过的外部面，成为新的外部面的一部分。

有了两个法则，我们的遍历就有了约束。那么我们在嵌入完回落边后怎样完成双连通分量的合并呢？在下降到子块的时候，为了后面的合并必须要记录这样几个信息，我们用栈将他们记录下来：非复制点，复制点，下降的方向，返回的方向。合并时这些信息也会合并。之所以要记录下下降和返回的方向是为了处理反转操作的影响，当进入和返回的方向有矛盾的时候，也就是要进行反转操作后再合并。对于反转操作，这里不在累赘的叙述。

为了保证复杂度，我们还有一个提高效率的操作。大家观察图 9，其中 s 是一个外部活跃节点，p 是一个内部活跃节点。由于我们的操作都在外部面上进行，所以外部面的遍历将直接关系程序的效率。在图中，我们会先访问到节点 p,然后会访问到终止节点 s,其中经过的 p 到 s 的这段路其实没有什么用，但是它仍然留在外部面上，下一次遍历还会遍历到它。为了避免重复的遍历，我们在遭遇 s 节点终止的时候连接一条边(v，s),由于这个边其实并不存在，我们把它叫虚边。这样我们在遍历外部面的时候就不会遍历到 p 到 s 这一段了。由于每个双连通分量最多遭遇两次终止节点，就最多只有两条虚边。总共也就不会超过 2n 条虚边，最后我们只需要将这 2n 条虚边删除就行了。

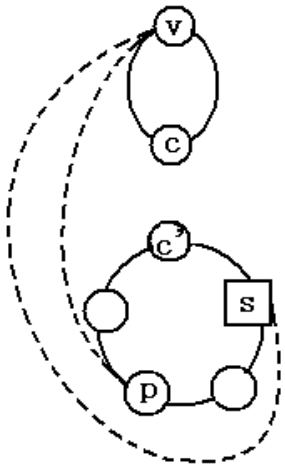


图 9

伪码请参考附录底 3 节。

9．复杂度分析

我们的算法叙述完了，那么我们实现了我们开始所说的复杂度了吗？这里我们将对它进行一个完整的分析。

深度优先遍历和 lowpoint 的计算是很显然的线形时间处理，为了知道外部活跃信息，我们为每个节点建立的 SDlist 总体复杂度是 O(n)，对 SDlist 中元素的删除操作每次是常数的时间，总体复杂度是 O(n)（参考第五节中对 SDlist 的描述）。反转操作一定只发生在合并双连通分量的时候，所以最多只会有 n 次反转，并且每次反转也是常数时间的操作。嵌入每

条边也是常数时间，总体复杂度也是 O(n)。

在 walkup 和 walkdown 中，我们遍历的的过外部面会因为边的嵌入被更新，从而不在被遍历到。虽然在遍历内部活跃子块的时候不会有嵌入虚边，但是内部活跃子快一定不会被遍历第二次，因为它没有和更早祖先相连的边。所以所有的边都只被遍历了常数次，由于边不会超过 3n-5 条，加上虚边也是线性的，所以总体复杂度依然是 O(n)。这些 O(n)的操作都是并行的，所以我们的总体复杂度就是 O(n)!

我们实现了 O(n)的时间复杂度，并且编程复杂度较简单！

10. 正确性的证明

如果算法根本就是错的，那么怎样的简单都是没用的。所以这一节也很重要。

除了一种特殊情况，我们在 walkdown 中是不会经过外部活跃节点的，所以在嵌入边的过程中不会覆盖外部活跃节点，如果所有得外部活跃节点都留在了外部面上。那么也就不会出现交叉的边。

那么那种特殊情况会不会覆盖外部活跃节点呢？其实不会的，在情况 2 的证明中说明了这个问题。

还记得文章开始我们提到的 k5 图和 k33 图吗。对于是非平面图，我们只需要证明该图中存在 k5 或者 k33 的同胚图就可以了。我们将依次讨论失败的情况，逐个找出他们构成的 k33 或者 k5 图。

情况 1：当节点有三个或者以上的外部活跃相关子块的时。

前面提到对于一个节点，由于我们只进行顺时针和逆时针两次遍历，当遍历一个外部活跃子块的时候一定会终止，那么最多遍历两个外部活跃子块我们的遍历就结束了。所以一旦节点包含了 3 个外部活跃子块，嵌入就失败了。不过在这样的图中，我们是可以找到一张 k33 图的。如图 10，图中我们正处理节点 v, u 是节点 v 的祖先节点。a , b , c 是 r 的儿子，也代表了它拥有的三个外部活跃相关子块。其中（u, v , r）,(a, b, c)分别是上三节点和下三节点，他们组成了一张 k33 的同胚图。

这样我们就证明了一个节点最多只能拥有 2 个外部活跃相关子块。

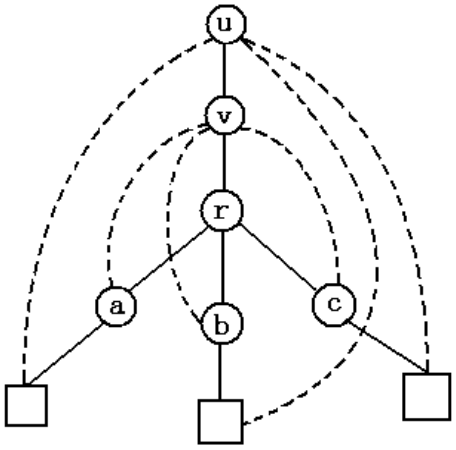


图 10

情况 2：当经过节点是活跃节点，下降到了它的相关子块只遍历时。

为什么我们经过了外部活跃节点，就不怕嵌入边的时候不能再将它留在外部面上了吗？当我们经过了外部活跃节点，对它的子块及其后裔块都遍历后，所有的嵌入边都在图的一边。由于拥有子快的节点一定是关节点，所以嵌入边后它依然会留在在外部面上。然后它们

会合并，节点就不再拥有该子块了，第二次遍历也就不再会产生这种情况。我们看图 11。

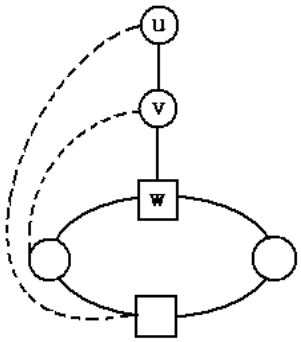


图 11

图 *11*：图中方形的节点是外部活跃节点，我们可以发现，在遍历了 *w* 的子块后嵌入边并不影响 *w* 留在外部面上。

大家会发现对外部活跃子块我们都只能遍历一次就会终止，那么如果需要两次遍历就一定是非平面图吗?

子情况 2.1：当 w 的子块或者后裔块中存在一个终止节点将两个或两个以上的相关节点分开了。如图 12。

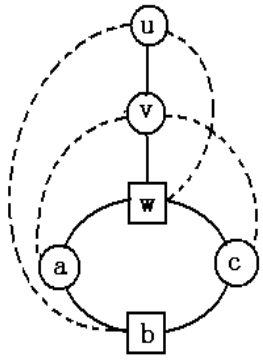


图 12

图 *12*：在图 *12* 中，*a,c* 被 *b* 分开了，所以要嵌入 *a,c* 的回落边需要对 *w* 的子快进行两次遍历。

在图 12 中，{(u, a, c), (v, w, b)}是一个 k33 同胚图。所以该图是一个非平面图。

子情况 2.2：节点 w 的子块或者后裔块中存在两个外部活跃相关节点。如图 13。

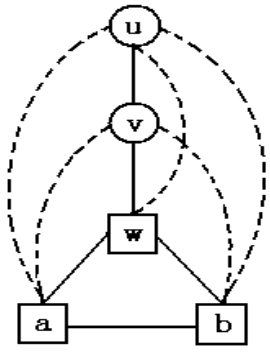


图 13

图 *13*：*a* ,*b* 是 *w* 子块中的两个外部活跃相关节点。

在图 13 中该图是一个 k5 同胚图，所以该图也是一个非平面图。

子情况 2.3：节点 w 有两个外部活跃相关块。

如果他们都只需要一次遍历，那么节点 w 依然会留外部面上，如果有三个这样的子块就回到了情况 1，如果一个块需要两次遍历就回到了上两种情况。

情况 3：一个相关节点由于终止节点的存在不能被遍历到。如图 14。

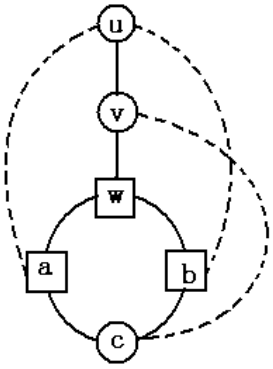


图 14

图 *14*：图中 *c* 节点是一个相关节点，但是他被 *a,b* 两个节点阻隔了而无法遍历到。

图 14 的这种情况也会导致嵌入失败，它其实也是一个 k33 同胚图，由{(u, r, c),(v ,a ,b)}构成。

这里我们的讨论了所有导致嵌入失败的情况，发现他们都是非平面图，所以我们的算法是正确的！

11．相关题目

1．冬令营 2004：蜂窝玉米

2．mipt： 090 planrity graph

3．对于有兴趣的同学测试自己的程序，可以参考附带题目：电路板问题。其中也附带了测试数据和源程序。

12．总结

算法或许在竞赛中用途并不广泛，但是对于学习图论知识将有一个更加深刻的认识。对更困难问题的挑战也能提高自己的勇气，毕竟没有什么困难是不可逾越的，也只有不断的在知识的海洋中遨游，才能完善和充实自己。

如果对算法还有问题，希望附录中的伪码对你有所帮助。

[附录]

资料参考：

John.M.Boyer： Simplified *O*(*n*) Planarity by Edge Addition

伪码说明：

“ <- ”为赋值号。设有节点为k，那么kin 表示它是由什么方向到达它的，kout表示它走向了什么方向。在外部面上方向将用1和0表示。当一个节点有复制点的的时候，由于该复制点所在块只有一个儿子c，我们将该复制点储存为 c+n

函数说明:

GetSuccessorOnExternalFace(v, vin):到达下一节点的函数。

GetActivesuccessorOnExternalFace(v, vin):到达下一个活跃节点的函数。

MergeBiconnectedComponent(stack):将栈中所保存的块的信息合并的函数，包括反转操作也在其中完成。

EmbedBackEdge(v, vin, w, win):嵌入回落边的函数。

Walkup(w,v):从w向上遍历到v的函数。

Walkdown(v):从v向下遍历的函数。

1．MergeBiconnectedComponent伪码

当前需要合并栈S中的信息。

（1）从栈S中取出4个元素，分别为r ,rin, w, wout

（2）if (rin和wout相等)

（3） 交换w的两个方向

（4） 将r,w这条根边置为-1

（5） wout<- wout xor 1

（6）将w从r的proots中移除。

（7）将 w – n 从它父亲的 SDlist中移除

（8）x<-1 xor wout

（9）将v的vin方向指向x

（10）将x的指向w的方向指向v

1．walkup伪码

当前处理节点为V ,要从节点w 向上遍历到它。

（1） 由于V的回落边指向w 将w的回落旗标标记为V

（2） (x,xin)<-(w, 1)

（3） (y,yin)<-(w, 0)

（4） while (x 不等于 v)

（5） if (x或y的visited值等于v) then break;

（6） 将x和y的值置为 v;

（7） if (x 是根节点) z1 <- x

（8） else if (y是根节点) z1 <- y

（9） else z1 <- nil

（10） if (z1不等于nil)

（11） c <- z1-n

（12） if (c的lowpoint 值比v的深度小)

（13） 将z1加入z的proots表尾

（14） else

（15） 将z1加入z的proots表头

（16） else (x,xin)<-GetSuccessorOnExternalFace(x,xin)

（17） (y,yin)<-GetSuccessorOnExternalFace(y,yin)

2．walkdown 伪码

当前处理节点v，根节点用v0表示。需要从v0向下遍历

```
（1）将栈S清空
（2）for vout in{0,1}
（3）  (w,win)<-GetSuccessorOnExternalFace(v)
（4）  while(w不等于v0)
（5）   if (如果w的回落旗标等于v)
（6）     while (栈S不为空)
（7）     MergeBiconnectedComponent(S)
（8）     EmbedBackEdge(v0,vout,w,win)
（9）     将w的回落旗标清空

（10）    if (w的proots不为空)
（11）     将(w,win)压入栈S中
（12）     w0 <- w的proots中的第一个。
（13）     (x,xin)<-GetActiveSuccessorOnExternalFace(w0,0)
（14）     (y,yin)<-GetActiveSuccessorOnExternalFace(w0,1)

（15）     if(x是内部活跃节点) then (w,win)<-(x,xin)
（16）     else if (y是内部活跃节点) then (w,win)<-(y,yin)
（17）     else if (x是相关节点) then (w,win)<-(x,xin)
（18）     else (w,win)<-(y,yin)

（19）     if (w和x相等) wout<-0
（20）     else wout<-1
（21）     将（w0,wout）压入栈S中

（22）   else if (w是内部活跃)
（23）     (w,win)<- GetSuccessorOnExternlFace(w,win)

（24）   else 意味着w是一个终止节点
（25）     if (c的lowpoint是比v的深度小的，并且栈S为空)
（26）       嵌入虚边(v0,w)
（27）     break while

（28） if(栈S不为空) then break for
```