第六讲、弦图的判定

给定一个弦图，LexBFS 与 MCS 算法可以找到一个完美消除序列。本讲中，我们将了解这两种算法实现的细节，并给出简要的正确性证明。同时我们将看到 LexBFS 在 O（n+m）的时间内得到完美消除序列。

1．介绍

每个弦图都有完美消除序列。我们将用 LexBFS 算法在 O（n+m）的找到这个序列。回忆以下 LexBFS 算法：每次选择标号最大的顶点并更新它的相邻点。直接做到这一点需要多余线性的时间，我们使用了特殊的数据结构使得寻找、更新顶点的时间降为 O（1）。

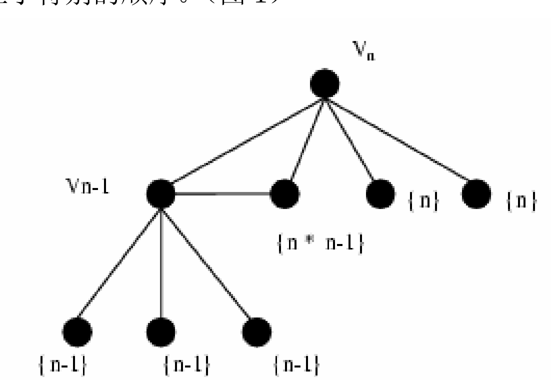
判定一个图是否是弦图，我们需要测试 LexBFS 算法返回的序列是否是完美消除序列。朴素的算法是测试每个顶点的前驱集合是否构成一个团，这需要 $O(\sum_{v\in V}(deg(v)^2) = O(mn)$的时间。但事实上，我们可以知道使用恰当的测试顺序，一个完美消除序列的每条边至多被检查两次。因此我们得到了一个时空复杂度都为 O（m+n）的算法。

2．**LexBFS** 算法的复杂度

回忆以下 LexBFS 算法的流程：

For all vertices v, set $L(v) = \emptyset$;
For $i = n \ldots 1$
    among all vertices $\neq v_{i+1}, \ldots, v_n$
    pick up v_i with the lexicographically largest label $\mathrm{L}(v_i)$;
    for each unnumbered vertex w that is adjacent to v
        Set $L(w) = L(w) \circ i$

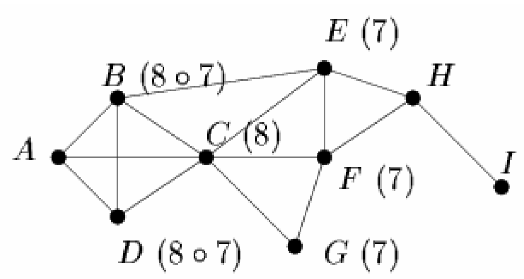
我们可以指出该算法是一个广度优先搜索。观察一下算法生成的搜索树，顶点 Vn 的相邻点在 Vn－1 的相邻点之前被访问。LexBFS 与标准 BFS 的唯一区别是 LexBFS 给 Vn 的相邻点加上了特别的顺序。（图 1）

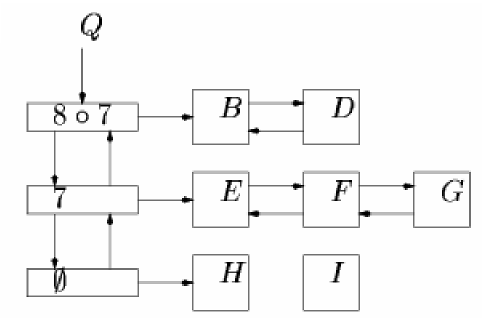


3．数据结构

为了高效地执行 LexBFS 算法，我们使用一个链表形的数据结构，其中表 Q 中的每个节

点是指向另一个链表的指针，称为一个“桶”。表 Sl（l 表示一个标号）包含了所有的满足 L（V）=l 顶点 V，L 是顶点 V 的标号。Q 中不能包含空链表。Q 中的各桶按字典序排列，最大的在最前面。我们看一个例子：





假设我们首先选择顶点 A 并标号它的所有相邻点；接着，我们选择顶点 C 并更新它的相邻点，我们得到的表 Q 将如图 3 所示。

实际中，每个顶点所在的桶 S（L（Vi））也被记录下来。另外，每个桶在 Q 中的位置也被记录下来。注意：为了插入与删除的方便，这些链表都是双链表。

对于这种数据类型的世纪操作，初始时所有顶点都标号 $\emptyset$，因此 Q 只有一个桶 $S_\emptyset$ 包含了所有顶点。显然这个初始化能在 O（n）时间内解决。

为了获得下一个顶点并更新数据，我们进行一下操作：

1. 令 Vi 是第一个桶中的第一个元素（显然 Vi 是目前标号最大的一个顶点）。
2. 将 Vi 从桶 S（L（Vi））中删去。
3. 如果 S（L（Vi））已空，将它从 Q 中删去。
4. 对于每个 Vi 的相邻点 W：
5. 如果 W 仍在 Q 中（W 尚未选择，必须更新它的标号和在 Q 中的位置）
6. 找到 S（L（W））以及它在 Q 中的位置。
7. 寻找 Q 中 S（L（W））上一个桶。
8. 如果这样的桶不存在，或它不是 $S_{L(w)\circ i}$
9. 在 Q 中的当前位置建立一个桶 $S_{L(w)\circ i}$。
10. 将 W 从 S（L（W））中取出并加入 $S_{L(w)\circ i}$ 中。
11. 如果 S（L（W））已空，将它删除。
12. 将 L（W）更新为 $L(w)\circ i$。

现在分析该算法的复杂度。获得 Vi 以需要 O（1）时间。从桶中删除一个元素也只要 O（1）时间。

更新 Q 时，我们必须更新 Vi 所有未选择的相邻点。对每个顶点需要 O（1）时间（所有需要的值都被储存下来了）。因此用时为 O（顶点 Vi 的度）。总的时空复杂度都为 O（n+m）。

空间也是线性也许不很明显。注意每个标号的长度可能到达Ω（n），例如对于完全图。但注意我们只在选择某个顶点 W 的相邻点时才增长它的标号。因此每个顶点的标号长度与它的度成正比，所以总空间复杂度为 O（n＋m）。

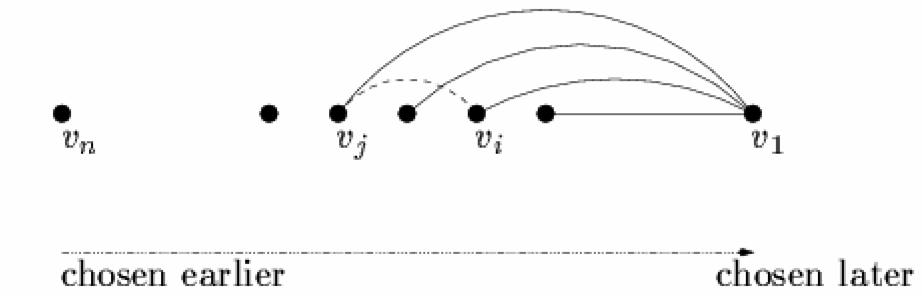
我们下面证明这个算法的确找到一个完美消除序列（如果有的话）。

定理 1：令 G＝（V，E）是一个图，{V1..Vn}是 LexBFS 算法得出的序列（Vn 是首先被选

择的）。如果 G 是弦图，则{Vn..V1}是一个完美消除序列。

证明：我们只需要证明 V1 是单纯点，接下来使用归纳法就可以解决。我们只简略地说明这个证明，详细的证明见[Gol80]。

假设 V1 不是单纯点，那么存在 V1 的两个相邻点 Vi 和 Vj，它们之间没有边。不妨令 j>i，即 Vj 在完美消除序列（假设是）中在 Vi 之前。（图 4）



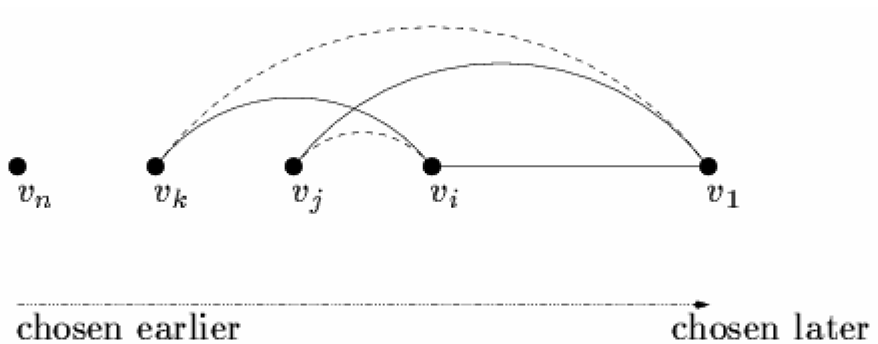
注意 Vi 和 Vj 在 LexBFS 算法中都在 V1 之前被选择。当我们选择 Vj 时，我们将 j 加到它所有的未选择的相邻点的标号中。那么 j 也被加入到 L（V1）中，但由于 Vi、Vj 不相邻，我们没有将 j 加入到 L（Vi）中。于是我们知道：

$$\begin{aligned} L(v_1) &= \cdots j \cdots \\ L(v_i) &= \cdots \not{j} \cdots \end{aligned}$$

注意到 Vi 在算法中先于 V1 被选择，那么 Vi 的标号必须不比 V1 小。但由于 L（V1）包括 j 而 L（Vi）不包括，那么只可能在标号更早的地方，L（Vi）大于 L（V1）。也就是说，j 之前必定存在一个数 k，它在 L（Vi）中，但不在 L（V1）中：

$$\begin{aligned} L(v_1) &= \cdots \not{k} \cdots j \cdots \\ L(v_i) &= \cdots k \cdots \not{j} \cdots \end{aligned}$$

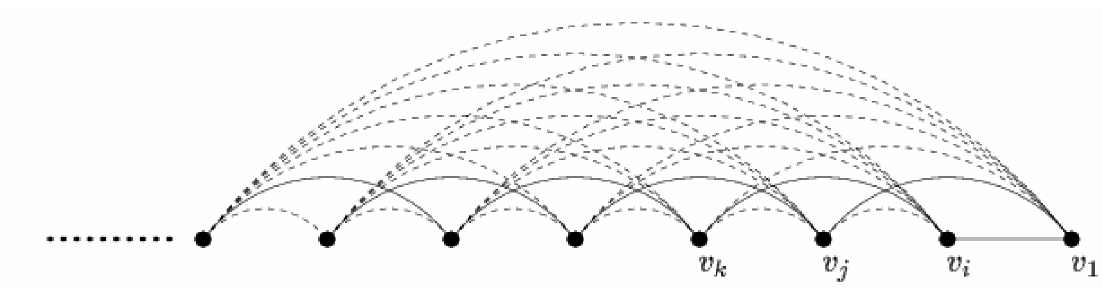
那么，一定有顶点 Vk，k>j，Vk 于 Vi 相邻但不与 V1 相邻，见图 5：



注意边（Vj，Vk）不会存在，因为如果有这条边，G 将有一个无弦的 4 阶环 V1，Vi，Vk，Vj，这与 G 是弦图矛盾。

我们小结一下我们的证明。Vi 的标号包含 k 但 Vj 的标号不包含。那么我们为什么在 LexBFS 算法中在 Vi 之前选择 Vj 呢？则一定存在另一个 Vk 之前的顶点与 Vj 相邻但不与 Vi 相邻。【通过一系列论证（这些就是这里简略的部分），我们可以证明那个点与 V1、Vk 都不相邻】（后者比较容易解决，只要用弦图的性质）。

于是我们又一次重复论点：为什么 Vk 在 Vj 之前被选择？于是又有之前的另一个顶点……这样的证明不断进行，知道推出矛盾（图 6）：每次都将导出一个新的顶点，但 G 的顶点是有限的，矛盾。因此 V1 是单纯点。根据上一讲的归纳证明，定理 1 得证。



∎

4．检验完美消除序列

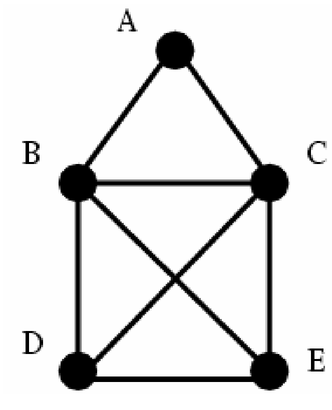
定理 1 证明了如果 G 是弦图，LexBFS 能够找出完美消除序列。为了判定弦图我们只需要验证 LexBFS 返回的序列是否是完美消除序列。我们的算法用时为 O（n+m）。

假设顶点 V 有一些前驱（否则 V 不需检验），令 U 是这些前驱中的最后一个。如果这是一个完美消除序列，那么 V 的前驱集构成一个团，所以 U 必须与 V 的所有其他前驱相邻，我们检验这一点。另一方面，如果检验成立，则 V 的其他前驱都是 U 的前驱，我们只要对 U 进行测试就可以了。

该算法的流程如下：

1. for $j = n$ down to 1 do
2.   if v_j has predecessors
3.   Let u be the last predecessor of v_j.
4.   Add $Pred(v_i) - \{u\}$ to $Test(u)$.
            ($Test(u)$ denotes the multi-set of vertices for which we want to test whether they are neighbours of u.)
5.   (Now test $Test(v_j)$.)
6.   Mark all vertices in Pred(v_j) as touched
7.   for every vertex w in $Test(v_j)$,
8.   if w is not touched, return FALSE.
9.   Mark all vertices in Pred(v_j) is untouched
10. return TRUE

我们用图 7 的图了解这个算法的运行过程：



假设我们将检验序列{A，D，B，C，E}，这不是完美消除序列。算法运行如下：

- 第一轮：Pred（E）＝{B，C，D}，U＝C，Test（C）＝{B，D}。由于 Test（E）＝

$\emptyset$，未发现错误，继续。

- 第二轮：Pred（C）＝{A，B，D}，U＝B，Test（B）＝{A，D}。由于 Test（C）＝{B，D}，C 与它们都相邻，未发现错误，继续。
- 第三轮：Pred（B）＝{A，D}，U＝D，Test（D）＝{A}。由于 Test（B）＝{A，D}，B 与它们都相邻，未发现错误，继续。
- 第四轮：Pred（D）＝$\emptyset$，Test 集合不变。但由于 Test（D）＝{A}，但 D 不与 A 相邻，因此返回 FALSE。

译者注： 1：这个算法也可以从前向后进行，可能更便于理解。

2：代码第 4 行有个错误：Pred（Vj）→Pred（Vi）。

定理 2：以上算法当且仅当 V1..Vn 是完美消除序列时返回 TRUE。

证明：如果算法返回 FALSE。则一定存在顶点 W∈Test（Vj）－Pred（Vj）。由于 W 在 Test（Vj）中，则 W 与 Vj 都是某个顶点 Vi 的前驱。由于 W∉Pred（Vi），那么 Vi 的前驱集合不是一个团，所以 V1..Vn 不是完美消除序列。

另外，假设 V1..Vn 不是完美消除序列，但算法却返回 TRUE。令 I 为满足以下条件的最小值：存在 Vj、Vk（j<k）是 Vi 的两个前驱，它们不相邻。令 U 未 Vi 最后的前驱。检验 Vi 时，U 的前驱是一个团，因此 Vi 与 Vj 中至少又一个就是 U（否则算法在检验 U 时会返回 FALSE）。令 U＝Vj，我们处理 Vi 时会将 Vk 加入到 Test（U）中，于是检验 U 时，我们会发现 Vk∈Test（U），但 Vk∉Pred（U），算法将返回 FALSE，矛盾。定理 2 得证。█

我们可以发现，整个算法的时空复杂度为 $|V|+\sum_{v\in V}|Adj(v)|+\sum_{v\in V}|Test(v)|$。对于每个顶点 V，算法只会将 Pred（V）加入到某一个 Test（U）中去即 $\sum_{v\in V}|Test(v)|=\sum_{v\in V}|Deg(v)|$，因此算法的复杂度为 O（n+m）。

将这个算法与 LexBFS 算法结合在一起，我们就得到了本章的最终结论：

定理 3：弦图的判定可以在线性时间内解决。