第1章 引言

1.1 研究背景

生物信息学（bioinformatics）是生物学与计算机科学以及应用数学等学科相互交叉而形成的一门新兴学科。它通过对生物学实验数据的获取、加工、存储、检索与分析，进而达到揭示数据所蕴含的生物学意义的目的。它涉及生物学、数学、计算机科学和工程学，依赖于计算机科学、工程学和应用数学的基础，依赖于生物实验和衍生数据的大量储存。生物信息学不只是一门为了建立、更新生物数据库及获取生物数据而联合使用多项计算机科学技术的应用性学科，也不仅仅是只限于生物信息学这一概念的理论性学科。事实上，它是一门理论概念与实践应用并重的学科。过去二十年，随着人类基因组工程的实施和深入，生物学数据获得前所未有的增加，数据的内容也从生理生化数据向遗传、结构、功能及其相互关系等数据发展。同一时期，计算机微处理器芯片、半导体存储器和系统软件也在按照指数方式增长。如何有效利用组合学、统计学等数学方法和现代计算机的强大计算能力，从这些生物学数据中提取有用的知识、发现重大的科学规律，成为生物学家、数学家、计算机学家们面临的巨大挑战，从而导致了计算生物学、生物信息学的产生和发展。

计算进化生物学是生物信息学的主要研究方向之一。进化生物学研究物种的起源和演化。引入信息学到进化生物学中，使得研究者能够：

- 通过度量DNA序列的改变研究众多生物体间的进化关系（超越了以前基于身体和生理特征观察的研究方法）
- 通过整个基因组的比对，研究更为复杂的进化论课题，如基因复制，基因水平转移等
- 为种群进化建立复杂的计算模型，以预测种群随时间的演化
- 保存大量物种的遗传信息

未来的研究工作包括重建现已相当复杂的进化树，并进行相关的研究。

1.2 研究意义

进化生物学在生物学中是一个非常重要的研究领域，其中进化树，又称系统发生树，是用来研究一群物种进化历史的标准模型，是进化生物学中不可缺少的重要工具。由于生物进化中可能产生的杂交、基因水平转移、基因重组等网状事件，导致一个物种的基因可能来源于多个祖先。多年来对生物进化历史的研究发现，杂交事件分别在植物、鸟类、鱼类的种群中均有发生。自然的杂交事件在哺乳动物，甚至是灵长类动物中都被发现过。研究显示，25%的植物和10%的动物，尤其是年轻的物种，都与杂交事件相关。

已有许多方法能够从一组物种中构建起他们对应的进化树，但由于进化过程中网络事件的存在，一组相同物种基于不同基因的分析可能产生不同的进化树，反之，通过对比这些进化树的相似性是帮助人们发现这些网络事件的重要手段。许多计算生物学家都对此问题非常感兴趣，他们常用的衡量标准有子树剪切再接距离（subtree prune and regraft distance, rSPR）和杂交数（hybridization number, HN）两种。Baroni等人证明了rSPR距离为进化过程中网状事件的数目给出了一个下限。[?]而对这些衡量标准的计算可以很好地抽象为具体的数学问题，因此借助计算机来研究进化树具有十分重要的意义。

1.3 固定参数算法概述

对于NP完全问题（NP难问题），因为当问题的规模增长时，算法执行所需的时间会随着问题的规模呈指数级增长，所以寻找一种有效的精确算法被广泛认为是不可能的。然而许多问题可以得到一个时间复杂度仅随问题的某个固定参数呈指数增长的算法。当问题在$O(f(k)\cdot n^c)$的时间内可解，$f(k)$是一个与n无关且c为常数时便属于固定参数可解类（fixed parameter tractable, FPT）。如果k远小于问题规模n，那么我们便可以获得该问题一个相对更有效的精确算法。例如，对于一个常见的NP问题——顶点覆盖问题，存在一个复杂度为$O(kn+1.274^k)$的固定参数算法[?]，其中n是顶点总数，k是顶点覆盖的大小。这就意味着顶点覆盖问题可以用该问题的解参数化。本文也是根据类似的思路利用固定参数算法来研究进化树的对比问题。

1.4 论文组织架构

本文采用了如下的结构：

第一章：引言。本章主要介绍本论文课题的研究背景，研究意义和主要的研究方法，最后对论文的章节进行了一个合理的逻辑安排。

第二章：问题建模和描述。本章先介绍本论文研究过程中所使用到的相关概念，包括进化树的定义和相关操作的定义，然后对所研究的问题给出了具体的形式化描述，最后简单介绍该问题的研究现状。

第三章：算法思想与设计。本章先介绍Whidden的固定参数算法，随后给出本文基于此算法的改进思路和具体方法和步骤。

第四章：数据结构设计及算法实现。本章给出改进后的固定参数算法的具体实现方法，包括数据结构设计以及具体的算法实现。

第五章：实验分析。本章通过随机生成的数据集和真实的数据集对算法进行实验分析，验证了其改进效果。

第六章：总结及展望。本章对本文研究的课题进行了归纳和总结，指明了改进和完善的方向。

第2章 问题建模和描述

2.1 基本定义

本节将给出在研究进化树结构的过程中所需要的一些重要概念的定义。

定义 2.1.1 **有根二叉进化树**（简称进化树）是一棵叶节点被集合X中的元素所标记的二叉树，满足除叶节点没有子节点外，其余节点都有且仅有两个子节点，记作T。将T的所有边的集合记为E_T。将X称作该进化树的**标识集**。

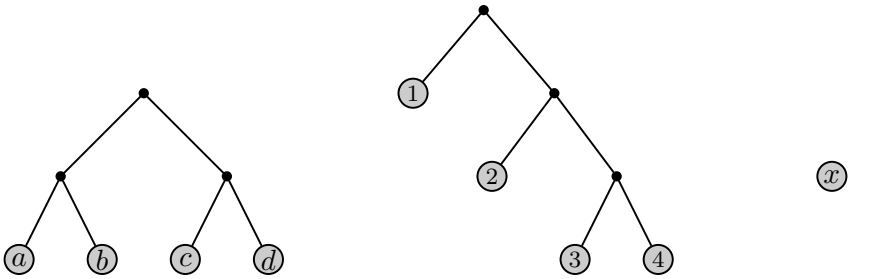


图 2–1 有根二叉进化树示例

定义 2.1.2 如果两棵进化树具有相同的标识集，并且**同构**，则认为这两棵进化树相等，记作$T_1 = T_2$。

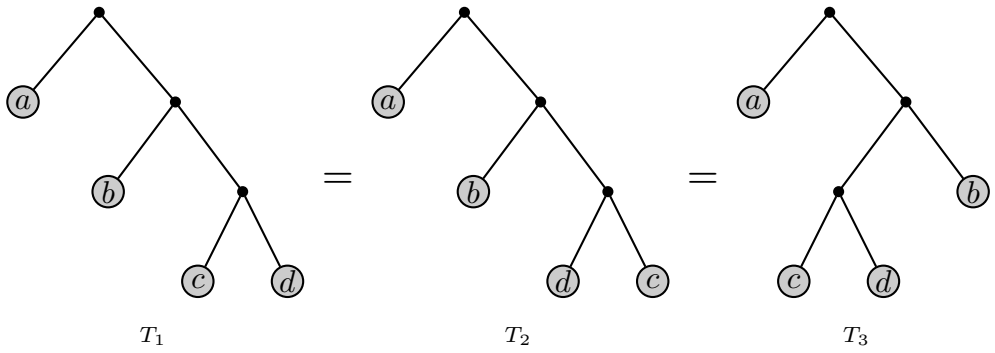


图 2–2 相等的3棵进化树

定义 2.1.3 将一棵二叉树删去所有只有一个子节点的内部节点以及所有未被标记的叶节点，使其变成一棵进化树的操作称为**收缩**。

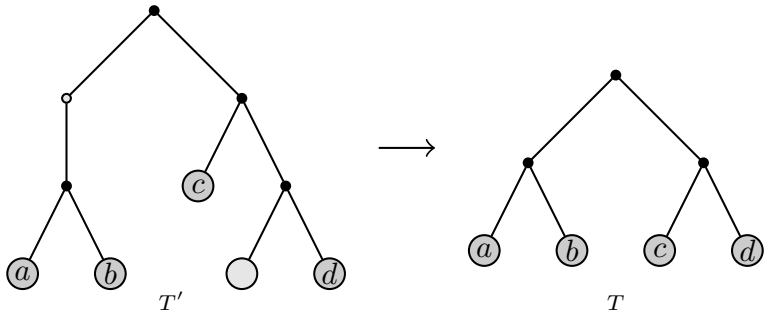


图 2–3 将T'收缩得到T

定义 2.1.4 **子树剪切再接**（rooted subtree prune and regraft，简称rSPR）对于一棵进化树T，剪去任意节点x的父边e_x得到一棵以x为根的子树t_x，将t_x嫁接到余下子树$T-t_x$的一条边上，并对操作后的树进行一次**收缩**，获得一棵新的进化树，该过程称为一次**rSPR操作**。对于两棵进化树T_1,T_2，将其中一棵树转化为另一棵所需的最少rSPR操作次数称为T_1,T_2的**rSPR距离**，记作 $d(T_1,T_2)$。

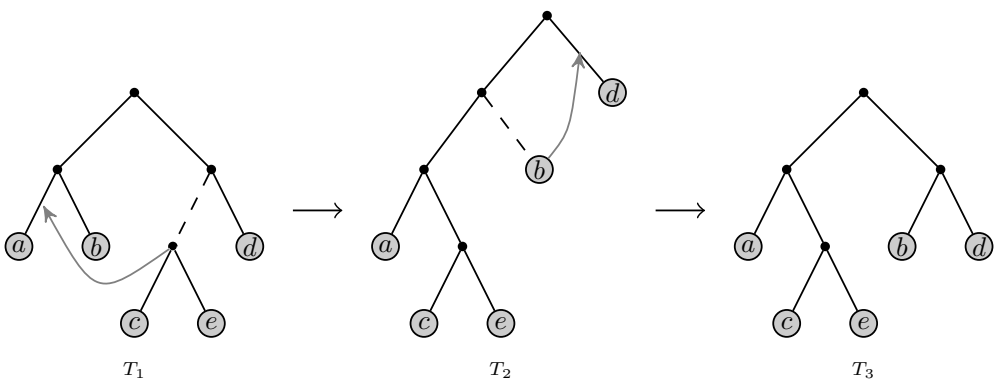


图 2–4 将T_1转化为T_3需要至少2次SPR操作，$d(T_1,T_3)=2$

定义 2.1.5 **进化树森林**（简称森林）是由若干棵进化树组成的集合。记作$F=\{t_1,t_2,...,t_n\}$。将一棵进化树T删去若干条边E，并对每棵独立的子树进行一次收缩操作得到的森林F，记作$F=T-E$。同理，将一个森林F删去若干条边E，并对每棵独立的子树进行一次收缩操作得到的森林F'，记作$F'=F-E$。

定义 2.1.6　最大一致森林（maximum-agreement forest，简称MAF）对于两棵进化树T_1, T_2，若存在$E_1 \subset E_{T_1}$，$E_2 \subset E_{T_2}$，使得$F = T_1 - E_1$并且$F = T_2 - E_2$，我们称F为T_1, T_2的**一致森林**。将T_1, T_2的所有一致森林中，包含最少进化树个数的森林称为**最大一致森林**，其包含的进化树个数记为$m(T_1, T_2)$。

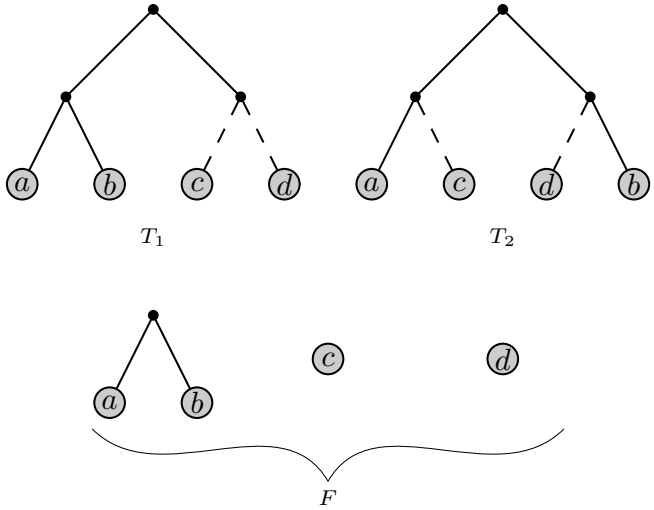


图 2–5 将T_1和T_2删去对应边后获得最大一致森林F，$m(T_1, T_2) = 3$

Bordewich和Semple[?]已经指出两棵进化树的rSPR距离与MAF的大小存在着等价关系，他们证明了如下定理：

定理 2.1.1　对于两棵具有相同标识集X的进化树T_1, T_2，存在

$$d(T_1, T_2) = m(T_1, T_2) - 1$$

2.2 问题描述

根据定理**??**我们可以将求解两棵进化树的rSPR距离转化为求解它们的最大一致森林的最优化问题，$d(T_1, T_2)$也可等价为将T_1, T_2转化为它们的MAF所需要删除的最少边数。而因为该问题是NP完全问题，我们可以将其进一步转化为可以用固定参数算法求解的判定性问题，问题的具体描述如下：

给定两个具有相同标识集X的进化树T_1, T_2，以及一个参数k。
判断 $d(T_1, T_2) \leq k$ 是否成立？

2.3 相关研究

尽管rSPR距离在生物学上具有十分重要的意义，但计算它却被证明是NP难的。因此研究针对该问题的近似算法，参数算法以及一些启发式算法成为了计算rSPR距离的主要手段。在介绍本文提出的参数算法之前，先回顾之前针对此问题各种算法的研究进展。

2.3.1 近似算法

自从Jotun Hein等人在[?]中引入MAF的概念以来，MAF已经成为解决该问题最有效的工具，不少以此为基础的近似算法得以提出。Hein首先在[?]中提出了一个近似比为3的近似算法，但却被Rodrigues在[?]中证明在某些情况下其近似比不可能小于4。同时，Rodrigues也提出了一些修正，宣称修正后的算法的近似比为3。然而，Bonet在[?]中给出的反例证明了[?]和[?]中的算法近似比实际上均为5，并且可以在线性时间复杂度下实现。Bordewich在[?]中提出了一个正确的近似比为3的算法，然而是以时间复杂度增加到$O(n^5)$为代价①。在[?]中提出的第二个近似比为3的算法的时间复杂度为$O(n^2)$。最终，线性时间复杂度下近似比为3的算法由Whidden在[?]中提出。

2.3.2 固定参数算法

在生物进化过程中网状事件发生的次数远远小于物种的数量，因此固定参数算法往往是计算rSPR距离精确值的最好途径。将两棵进化树的rSPR距离作为参数k的参数算法便是一个很有前景的研究方法。以此为基础，Bordewich在[?]中提出了一个时间复杂度为$O(4^k \cdot k^5 + n^3)$参数算法。之后，Hallett，Allen，Bonet等人针对该问题的相关问题也有许多有意义的研究。[?][?][?]Whidden在[?]中提出的算法可以说是该问题在参数算法上的一个突破，该算法的时间复杂度为$O(2.42^k n)$，并且从实验结果可以看出其性能远优于先前的算法，有能力处理具有更多节点数更大rSPR距离的进化树对比问题。随后，Chen和Wang在[?]中对Whidden的算法中最差的情况进行了许多改进，最后获得了一个时间复杂度为$O(2.34^k n)$的参数算法，但其改进方法分类较多，比较复杂，因此算法实现难度很大。本文也是基于相同的思想，但使用相对更简单的方法得到了更好的改进效果。

① 适当地改进数据结构可以将时间复杂度降低到$O(n^4)$

2.3.3 启发式算法

许多关于SPR距离的启发式算法在近年也得以提出，并开发了相关的软件。其中，Hallett和Lagergen开发的程序LatTrans[?]针对一些特殊的rSPR操作进行建模，只考虑进化树只可能在两种情况下相异，在这种特殊的条件下，它的时间复杂度可以减少到$O(2^k n^2)$。Macleod开发的HorizStory[?]可以计算多叉树的SPR距离，但只考虑SPR操作对象是只有一个叶节点的子树。SPRdist[?]和TreeSAT[?]是两个计算rSPR距离精确值的软件，他们分别把计算MAF的问题转化为整数线性规划问题（integer linear programming, ILP）和可满足性问题（satisfiability problem, SAT），然后利用求解对应问题的有效手段来获得rSPR距离的解。但根据实验结果，它们的性能均不能与Whidden所提出的算法相比。[?]

第3章 算法思想与设计

本文的算法与[?]中的相似，都是基于Whidden的参数算法[?]改进而来。因此，在这里有必要先简单介绍Whidden算法的内容，并给出简要的时间复杂度分析。之后，会详细介绍本文的改进思路，并给出详细的方法和步骤。

3.1 Whidden的参数算法[?]

3.1.1 算法概述

Whidden的算法所解决的正是本文在第**??**节中所提出的问题。设$MAF(T_1,F_2,k)$为针对此问题的判定函数，T_1代表第一棵进化树，F_2代表T_2删除某个边集后所得的森林，k是一个非负整数。若F_2能够在删除至多k条边后变成T_1,T_2的最大一致森林，则$MAF(T_1,F_2,k)$返回$true$，否则返回$false$。初始时，$F_2=T_2$。求解rSPR距离，只需要从0开始枚举k的值，直到$MAF(T_1,T_2,k)$返回$true$。因为时间复杂度是关于k的指数，所以相对于计算$MAF(T_1,T_2,k)$，计算rSPR距离只会在时间复杂度的常数上有所增加。算法采用递归的思想，MAF函数的具体步骤如下：

1. 如果$k<0$，返回$false$
2. 如果在T_1中存在一对兄弟叶节点a,b，并且它们在F_2中的对应节点也是兄弟叶节点，那么就合并a,b，然后把它们的在T_1,F_2中的父节点作为对应的叶节点。重复此步骤，直至没有节点可以合并。
3. 如果在F_2中存在只有一个节点的子树x，则将x从T_1,F_2中移除。重复此步骤，直至没有节点可以移除。此时如果产生新的可以合并的兄弟叶节点，则转至**??**，否则继续。
4. 如果F_2为空，返回$true$
5. 任意选择T_1中的一对兄弟叶节点a,b（注意到此时a,b在F_2中一定不是兄弟叶节点），分三种情况讨论（如图**??**）：
   (a) Case 1：若a,b在F_2中属于不同的连通分量。可以证明[?]，a的父边e_a和b的父边e_b两条边中至少有一条需要删除。对于两种情况下修改

后的$F_2' = F_2 - \{e_a\}$和$F_2' = F_2 - \{e_b\}$，分别调用两次$MAF(T_1, F_2', k-1)$。只要有任意一次结果为$true$，则返回$true$，否则返回$false$。

(b) Case 2：若a,b在F_2中连通，并且a,b之间有且只有一个悬挂节点p。可以证明[?]，一定存在最优解$E \subset E_{T_2}$，使得$T_2 - E$是T_1, T_2的MAF，并且$e_p \in E$。因此，只需要直接修改F_2，得到$F_2' = F_2 - \{e_p\}$。若$MAF(T_1, F_2', k-1)$结果为$true$，则返回$true$，否则返回$false$。

(c) Case 3：若a,b在F_2中连通，并且a,b之间有至少两个悬挂节点。设a,b之间所有悬挂节点的集合为$P = \{p_1, p_2, ..., p_m\}, m \geq 2$。可以证明[?]，一定存在最优解$E \subset E_{T_2}$，使得$T_2 - E$是$T_1, T_2$的MAF，并且$e_a \in E$或者$e_b \in E$或者对于所有$1 \leq i \leq m$，有$e_{p_i} \in E$。因此，分别修改$F_2$得到$F_2' = F_2 - \{e_a\}$、$F_2' = F_2 - \{e_b\}$和$F_2' = F_2 - E_P$（$E_P$为P中所有节点对应的父边的集合），然后分别调用两次$MAF(T_1, F_2', k-1)$和一次$MAF(T_1, F_2', k-m)$。只要有任意一次结果为$true$，则返回$true$，否则返回$false$。

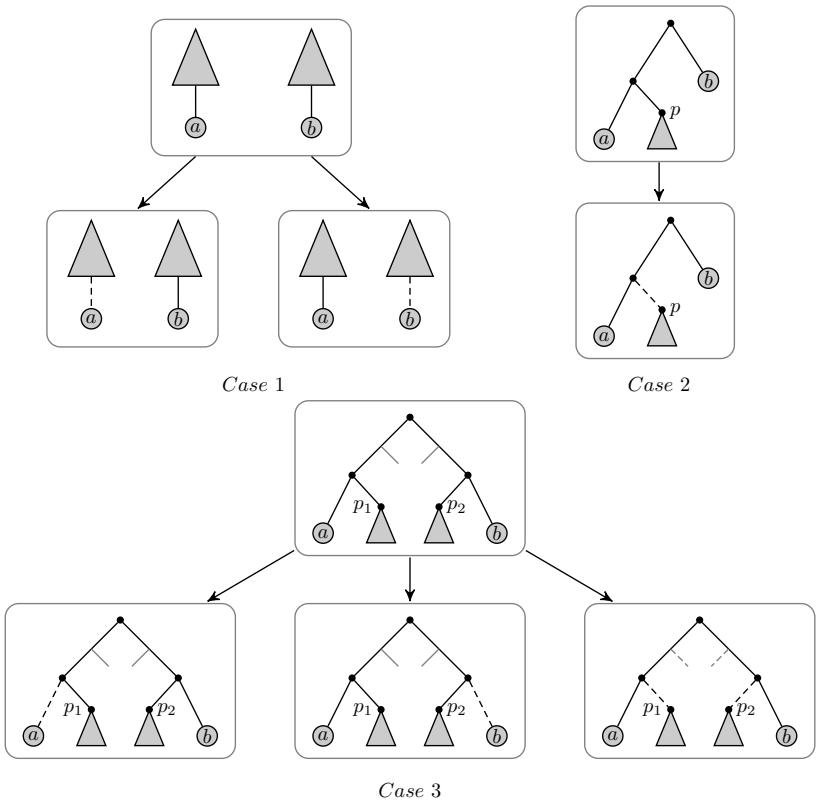


图 3–1 Whidden算法第**??**步的三种情况

3.1.2 时间复杂度分析

不难看出，适当地设计数据结构，Whidden算法中前四步可以在$O(n)$的时间内完成。因此，我们关键需要分析搜索树的大小，而搜索树的大小与参数k有直接关系。设$C(k)$为参数为k时最坏情况下搜索树的大小。根据第**??**步中的三种情况，我们可以得到下面三个不等式：

1. 若a,b在F_2中属于不同的连通分量。

$$C(k) \leq 2C(k-1)$$

2. 若a,b在F_2中连通，并且a,b之间有且只有一个悬挂节点。

$$C(k) \leq C(k-1)$$

3. 若a,b在F_2中连通，并且a,b之间有至少两个悬挂节点。

$$C(k) \leq 2C(k-1)+C(k-m),\ m \geq 2$$

可以看出，$C(k)$的最坏情况出现在第3种情况$m=2$时，此时有

$$C(k) \leq 2C(k-1)+C(k-2)$$

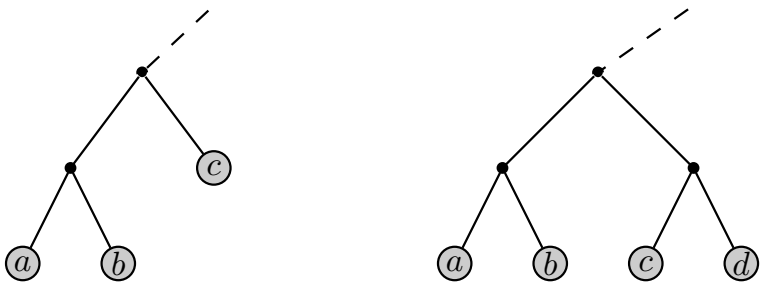
设$C(k)=\alpha^k$，带入可得

$$1 = 2\alpha^{-1}+\alpha^{-2}$$

解得$\alpha = 1+\sqrt{2} \approx 2.42$，因此Whidden参数算法复杂度为$O(2.42^k n)$。Whidden的算法最大贡献在于发现了Case 2中当a,b直接只有1个悬挂节点便可以直接删除其父边的规律。而该算法的限制则在于Case 3，本文正是试图寻找方法改进Case 3的最坏情况，具体内容将在下一节讨论。

3.2 改进思路

显然，当叶节点个数n小于3时，两棵进化树的rSPR距离一定是0，因此在后面的讨论中，我们假设$n \geq 3$。为了改进Whidden的算法，我们试图在Whidden所利用的两个兄弟叶节点的基础之上加入更多的节点，以此获取更多的信息用以改进算法。一种自然的想法是在T_1中寻找形如图**??**中(a)的结构。但(a)所表示的结构并非在所有进化树中都会出现，例如一棵叶节点个数为4的满二叉树。然而可以证明，若(a)中的结构不存在，一定存在形如图**??**中(b)所表示的结构。



(a) (b)

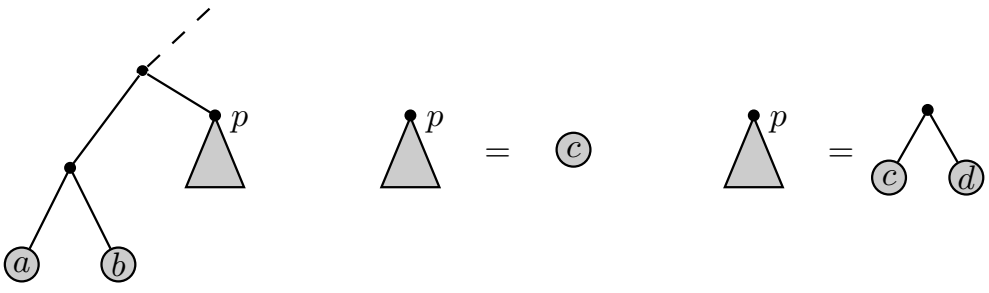
图 3–2 两种结构

引理 3.2.1 对于任意一棵叶节点数大于3的进化树T，存在下面至少一种情况：

- 存在T的三个叶节点a,b,c，满足a,b是兄弟节点，并且a,b的父节点与c是兄弟节点（图**??** (a)）
- 存在T的四个叶节点a,b,c,d，满足a,b是兄弟节点，c,d是兄弟节点，并且a,b的父节点与c,d的父节点是兄弟节点（图**??** (b)）

证明：不妨设a是离T的根节点最远的叶节点，根据进化树的定义，一定存在a的兄弟节点b。（图**??** (a)）设p为a,b的父节点的兄弟节点，因为a离根节点的距离最远，那么以p为根节点的子树T_p只有两种可能。

- p为叶节点，如图**??** (b)，此时出现引理**??**中第一种情况
- p不为叶节点，则p一定有且只有两个叶节点，如图**??** (c)，此时出现引理**??**中第二种情况



(a) (b) (c)

图 3–3 引理**??**证明

于是，基于引理**??**我们可以改进Whidden算法的第**??**步。

3.3 改进算法

本文改进算法的前四步与第**??**节中叙述的完全一致，主要针对第**??**步进行优化。为了方便叙述，我们作如下两个定义：

定义 3.3.1 设Q为节点a到节点b的路径上所有节点的集合，但不包括a,b。有节点x，记其父节点为f_x，若满足$x \notin Q$且$x \notin \{a,b\}$且$f_x \in Q$，则称x为a,b之间的悬挂节点。

定义 3.3.2 记$P(a,b)$为节点a到节点b的路径上所有悬挂节点的集合，E_P表示集合P中所有节点父边的集合。

在详细叙述本文的改进方法之前，我们再明确一下当T_1,F_2经过前四步的处理后所具有的两点性质：

1. T_1中的任意一对兄弟叶节点a,b在F_2中一定不是兄弟节点。换句话说，a,b在F_2中要么不连通，要么连通但$|P(a,b)| \geq 1$。
2. T_1,F_2中的任意一个叶节点x都不可能是某棵树的根节点。这意味着任意叶节点x的父边e_x一定存在。

改进算法的第五步首先是在T_1中寻找图**??** (a)所表示的结构，这可以通过遍历T_1在最多$O(n)$的时间内完成。接下来，我们根据子树T_p的两种情况分类讨论。

3.3.1 当p为叶节点

Case 1: 当p为叶节点时，我们得到T_1中的3个叶节点a,b,c，使其结构如图**??**。此时，根据a,b,c在F_2中的不同形态，我们采取不同的策略。

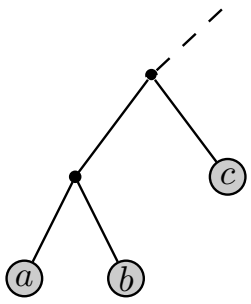


图 3–4 *Case 1* T_1中的a,b,c

Case 1.1: 若a,b在F_2中属于不同的连通分量，此时出现Whidden算法第**??**步中的情况**??**（简称**情况??**），用相同方法处理。

Case 1.2: 若a,b在F_2中属于相同的连通分量，但$|P(a,b)|=1$时，此时出现Whidden算法第**??**步中的情况**??**（简称**情况??**），也用相同方法处理。

Case 1.3: 若a,b在F_2中属于相同的连通分量，且满足$|P(a,b)|\geq 2$，但a,b与c不属于同一连通分量（如图**??**）。

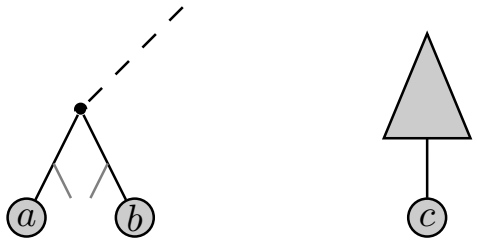


图 3–5 *Case 1.3* F_2中的a,b,c

设x为节点a在最终的MAF中的兄弟节点，根据节点x可能的情况，我们进行如下分类讨论：

1. 如果$x\in\varnothing$，即a最终为孤立节点，那么必须删除e_a。删除后在T_1中b,c为兄弟，而在F_2中b,c不连通，此时出现**情况??**，可以有两种选择，删除e_b或者e_c。因此分别调用两次递归函数，$MAF(T_1,F_2-\{e_a,e_b\},k-2)$和$MAF(T_1,F_2-\{e_a,e_c\},k-2)$。
2. 如果$x=b$，则需删除F_2中a到b之间所有的悬挂节点的父边$E_{P(a,b)}$。因此只需调用一次递归函数，$MAF(T_1,F_2-E_{P(a,b)},k-|P(a,b)|)$。
3. 如果$x\notin\varnothing$且$x\neq b$，由F_2的结构可知必须删除b的父边e_b。删除后在T_1中a,c为兄弟，而在F_2中a,c不连通，此时出现**情况??**，可以有两种选择，删除e_a或者e_c。因此分别调用两次递归函数，$MAF(T_1,F_2-\{e_a,e_b\},k-2)$和$MAF(T_1,F_2-\{e_b,e_c\},k-2)$。$MAF(T_1,F_2-\{e_a,e_b\},k-2)$与1中重复，可省略。

综上，对于*Case 1.3*，我们需要调用四次递归函数：

$$MAF(T_1,F_2,k)=\begin{cases} MAF(T_1,F_2-\{e_a,e_b\},k-2) & \text{or}\\ MAF(T_1,F_2-\{e_a,e_c\},k-2) & \text{or}\\ MAF(T_1,F_2-\{e_b,e_c\},k-2) & \text{or}\\ MAF(T_1,F_2-E_{P(a,b)},k-|P(a,b)|) & \end{cases}$$

Case 1.4: 若a,b在F_2中属于相同的连通分量，满足$|P(a,b)| \geq 2$，且a,b与c属于同一连通分量。则需要进行更细节的讨论。

Case 1.4.1: 在*Case 1.4*的前提下，若在F_2中存在$|P(a,c)| = 0$或者$|P(b,c)| = 0$，即a,c或者b,c是兄弟节点。（如图**??**）

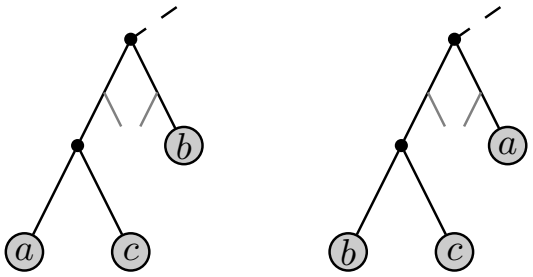


图 3–6 *Case 1.4.1* F_2中的a,b,c

因为在T_1中有$P(a,c) = \{b\}, P(b,c) = \{a\}$，此时出现**情况??**。对应的，应该删除$e_b$或者$e_a$。因此，对于*Case 1.4.1*有：

$$MAF(T_1,F_2,k) = \begin{cases} MAF(T_1,F_2-\{e_b\},k-2) & \text{for} \quad |P(a,c)| = 0 \\ MAF(T_1,F_2-\{e_a\},k-2) & \text{for} \quad |P(b,c)| = 0 \end{cases}$$

Case 1.4.2: 在*Case 1.4*的前提下，若F_2中满足$|P(a,c)| = 1$且$|P(b,c)| = 1$，此时F_2中的a,b,c只可能是图**??**中的两种情况。

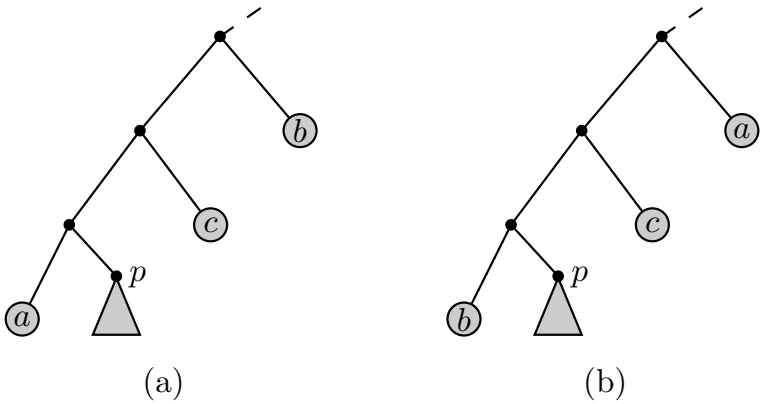


图 3–7 *Case 1.4.2* F_2中的a,b,c

同样，设x为节点a在最终的MAF中的兄弟节点，根据节点x可能的情况，我们首先针对图**??** (a)的情形进行如下分类讨论：

1. 如果$x \in \varnothing$，即a最终为孤立节点，那么必须删除e_a。删除后在T_1中b,c为兄弟，而在F_2中$P(b,c) = \{p\}$，此时出现**情况??**，因此可以直接删除b,c之间的悬挂节点p的父边。需调用一次递归函数，$MAF(T_1, F_2 - \{e_a, e_p\}, k-2)$。
2. 如果$x = b$，则需删除F_2中a到b之间所有的悬挂节点的父边$E_{P(a,b)}$。注意到此时$P(a,b) = \{c,p\}$，需调用一次递归函数，$MAF(T_1, F_2 - \{e_c, e_p\}, k-2)$。
3. 如果$x = c$，则需删除e_b和F_2中a到c之间所有的悬挂节点的父边$E_{P(a,c)}$。注意到此时$P(a,c) = \{p\}$，需调用一次递归函数，$MAF(T_1, F_2 - \{e_b, e_p\}, k-2)$。
4. 如果$x \notin \varnothing$且$x \notin \{b,c\}$，则需要删除e_b, e_c。调用一次递归函数，$MAF(T_1, F_2 - \{e_b, e_c\}, k-2)$。

然而，通过观察，实际上我们可以发现$x = b$和$x = c$两种情况处理后的T_1, F_2实际上是等价的。）对于如图**??**的两片森林，只需要简单地交换节点b,c的标识即可让它们等价，而这并不会影响最终MAF的大小。

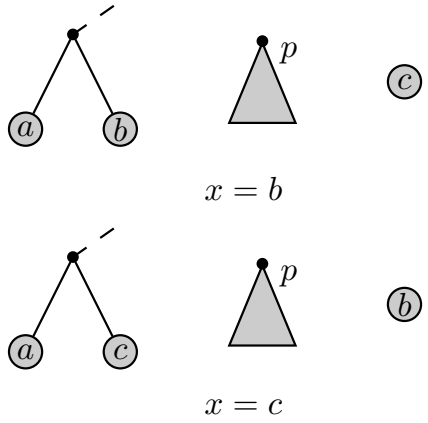


图 3–8 $x = b$和$x = c$两种情况处理后的F_2

因此，我们可以在$x = b$和$x = c$两种选择中省略一种，不妨省略$x = c$。而对于图**??** (b)的情况，我们交换a,b所代表的节点后可以用完全相同的方法处理。

综上，对于*Case 1.4.2*，我们需要调用三次递归函数：

$$MAF(T_1, F_2, k) = \begin{cases} MAF(T_1, F_2 - \{e_a, e_p\}, k-2) & \text{or} \\ MAF(T_1, F_2 - \{e_c, e_p\}, k-2) & \text{or} \\ MAF(T_1, F_2 - \{e_b, e_c\}, k-2) \end{cases}$$

Case 1.4.3: 在*Case 1.4*的前提下，排除*Case 1.4.1*和*Case 1.4.2*，F_2中一定有$|P(a,c)| \geq 1$且$|P(b,c)| \geq 1$且存在$x \in \{a,b\}$使$|P(x,c)| \geq 2$。不妨设$x = a$，综合之前的条件，有

$$\begin{cases} |P(a,b)| \geq 2 \\ |P(a,c)| \geq 2 \\ |P(b,c)| \geq 1 \end{cases}$$

在满足上述条件的情况下，F_2中a,b,c有可能为如图**??**三种形态：

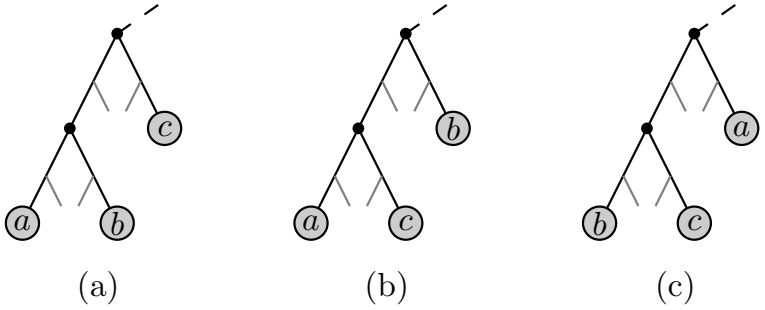


图 3–9 *Case 1.4.3* F_2中的a,b,c

虽然是三种不同形态，实际上我们都可以采取类似*Case 1.4.2*的策略来处理。跟之前一样，设x为节点a在最终的MAF中的兄弟节点，根据节点x可能的情况，进行如下分类：

1. 如果$x \in \varnothing$，即a最终为孤立节点，那么必须删除e_a。需调用一次递归函数，$MAF(T_1, F_2 - \{e_a\}, k-1)$。
2. 如果$x = b$，则需删除F_2中a到b之间所有的悬挂节点的父边$E_{P(a,b)}$。需调用一次递归函数，$MAF(T_1, F_2 - E_{P(a,b)}, k - |P(a,b)|)$。
3. 如果$x = c$，则需删除e_b和F_2中a到c之间所有的悬挂节点的父边$E_{P(a,c)}$，即$E_{P(a,c)\cup\{b\}}$。需调用一次递归函数，$MAF(T_1, F_2 - E_{P(a,c)\cup\{b\}}, k - |P(a,c) \cup \{b\}|)$。
4. 如果$x \notin \varnothing$且$x \notin \{b,c\}$，则需要删除e_b, e_c。调用一次递归函数，$MAF(T_1, F_2 - \{e_b, e_c\}, k-2)$。

综上，对于*Case 1.4.3*，我们需要调用四次递归函数：

$$MAF(T_1,F_2,k)=\begin{cases} MAF(T_1,F_2-\{e_a\},k-2) & \text{or} \\ MAF(T_1,F_2-E_{P(a,b)},k-|P(a,b)|) & \text{or} \\ MAF(T_1,F_2-E_{P(a,c)\cup\{b\}},k-|P(a,c)\cup\{b\}|) & \text{or} \\ MAF(T_1,F_2-\{e_b,e_c\},k-2) \end{cases}$$

为了后文时间复杂度的分析，我们在此先分析一下$|P(a,c)\cup\{b\}|$的大小。

引理 3.3.1 **设a,b,c为同一进化树上的三个节点，且满足$b\in P(a,c)$。则存在如下关系**

$$|P(a,c)|=|P(a,b)|+|P(b,c)|$$

证明：记R_{xy}为节点x,y的最近公共祖先。如果节点y是x的祖先，则记为$x\prec y$。根据R_{ab},R_{ac},R_{bc}的不同情况可能出现如图**??**的两种形态。

首先我们针对$R_{ab}\prec R_{ac}$的情况进行分析。根据该情况下进化树的结构和悬挂节点的定义可得到如下几个等式：

$$\begin{cases} |P(a,b)|=|P(a,R_{ab})|+|P(R_{ab},b)| \\ |P(a,c)|=|P(a,R_{ab})|+|P(R_{ab},c)|+1 \\ |P(b,c)|=|P(b,R_{ab})|+|P(R_{ab},c)|+1 \end{cases}$$

又因为节点b与R_{ab}直接相连，可知$P(b,R_{ab})=P(R_{ab},b)=0$。带入之前的三个等式即可得到：

$$|P(a,c)|=|P(a,b)|+|P(b,c)|$$

同理，对于$R_{bc}\prec R_{ac}$的情况，可以用完全相同的方法证明。

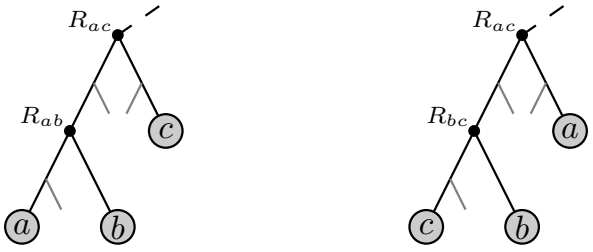


图 3–10 当$b\in P(a,c)$时a,b,c的可能形态

对于$|P(a,c)\cup\{b\}|$的大小，我们考虑b是否属于$P(a,c)$

- 当$b\notin P(a,c)$，则有$|P(a,c)\cup\{b\}|=|P(a,c)|+1$。
  而根据*Case 1.4.3*的条件$|P(a,c)|\geq 2$，因此$|P(a,c)\cup\{b\}|\geq 3$。
- 当$b\in P(a,c)$，则有$|P(a,c)\cup\{b\}|=|P(a,c)|$。
  但由引理**??**，可得$|P(a,c)|=|P(a,b)|+|P(b,c)|$。
  而根据*Case 1.4.3*的条件$|P(a,b)|\geq 2$且$P(b,c)\geq 1$，因此$|P(a,c)\cup\{b\}|\geq 3$。

综上，在*Case 1.4.3*中，$|P(a,c)\cup\{b\}|\geq 3$。

3.3.2 当p不为叶节点

Case 2: 当p不为叶节点时，我们得到T_1中4个叶节点a,b,c,d，使其结构如图**??**。此时，如果继续完整地讨论a,b,c,d在F_2中的不同形态将会十分复杂，而本文通过将4点情况转化为3点情况巧妙地避免了很多复杂的讨论。首先我们根据a,b,c,d在F_2中的连通性进行分类讨论。

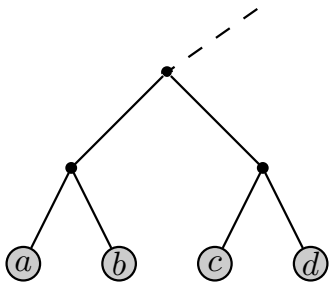


图 3–11 *Case 2* T_1中的a,b,c,d

Case 2.1: 若a,b（或c,d）在F_2中属于不同的连通分量，此时出现Whidden算法第**??**步中的情况**??**，用相同方法处理。

Case 2.2: 若a,b（或c,d）在F_2中属于相同的连通分量，但$|P(a,b)|=1$（或$|P(c,d)|=1$）时，此时出现Whidden算法第**??**步中的情况**??**，也用相同方法处理。

Case 2.3: 若a,b在F_2中连通，c,d也在F_2中连通，且满足$|P(a,b)|\geq 2,|P(c,d)|\geq 2$，但$a,b$与$c,d$不属于同一连通分量（如图**??**）。

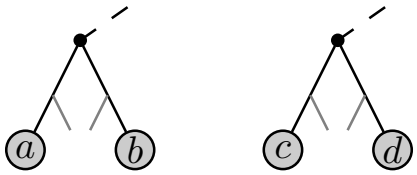


图 3–12 *Case 2.3* F_2中的a,b,c,d

类似*Case 1*中的分析，设x为节点c在最终的MAF中的兄弟节点，根据节点x可能的情况，进行如下分类讨论：

1. 如果$x \neq d$，根据T_1中的形态可知e_c, e_d必然有至少一个会被删除。若删除e_d，那么此时a,b,c满足*Case 1.3*的条件。同理，若删除e_c，则a,b,d也会满足*Case 1.3*的条件。因此，我们只需要调用两次处理*Case 1.3*的函数，$CASE_1_3(T_1, F_2 - \{e_d\}, k-1, a, b, c)$和$CASE_1_3(T_1, F_2 - \{e_c\}, k-1, a, b, d)$。
2. 如果$x = d$，则需删除F_2中c到d之间所有的悬挂节点的父边$E_{P(c,d)}$。然后只需调用一次递归函数，$MAF(T_1, F_2 - E_{P(c,d)}, k-2)$。

综上，对于*Case 2.3*，我们需要调用3次递归函数：

$$MAF(T_1, F_2, k) = \begin{cases} CASE\_1\_3(T_1, F_2 - \{e_d\}, k-1, a, b, c) & \text{or} \\ CASE\_1\_3(T_1, F_2 - \{e_c\}, k-1, a, b, d) & \text{or} \\ MAF(T_1, F_2 - E_{P(a,b)}, k-2) & \text{or} \end{cases}$$

Case 2.4: 若a,b,c,d在F_2中都属于同一连通分量，且满足$|P(a,b)| \geq 2, |P(c,d)| \geq 2$。此时，$a,b,c,d$在$F_2$中会有很多种不同形态，但我们可以通过讨论$c,d$在最终MAF的情况，将问题转化为*Case 1.4.3*的情况。

类似*Case 1*中的分析，设x为节点c在最终的MAF中的兄弟节点，根据节点x可能的情况，进行如下分类讨论：

1. 如果$x \neq d$，根据T_1中的形态可知e_c, e_d必然有至少一个会被删除。若删除e_d，那么此时a,b,c满足*Case 1.4.3*的条件。同理，若删除e_c，则a,b,d也会满足*Case 1.4.3*的条件。因此，我们只需要调用两次处理*Case 1.4.3*的函数，$CASE_1_4_3(T_1, F_2 - \{e_d\}, k-1, a, b, c)$和$CASE_1_4_3(T_1, F_2 - \{e_c\}, k-1, a, b, d)$。
2. 如果$x = d$，则需删除F_2中a到b之间所有的悬挂节点的父边$E_{P(c,d)}$。然后只需调用一次递归函数，$MAF(T_1, F_2 - E_{P(c,d)}, k-2)$。

综上，对于*Case 2.4*，我们需要调用3次递归函数：

$$MAF(T_1,F_2,k)=\begin{cases} CASE\_1\_4\_3(T_1,F_2-\{e_d\},k-1,a,b,c) & \text{or} \\ CASE\_1\_4\_3(T_1,F_2-\{e_c\},k-1,a,b,d) & \text{or} \\ MAF(T_1,F_2-E_{P(c,d)},k-2) & \text{or} \end{cases}$$

3.4 改进算法的时间复杂度

类似Whidden的算法，我们每次递归的函数也可以在$O(n)$的时间内完成。设$C(k)$为参数为k时最坏情况下搜索数的大小。对于*Case 1*中的所有情况以及*Case 2.1, Case 2.2*可以直接列出以下不等式：

$$\begin{cases} C(k)\le 2C(k-1) & \textit{Cases } 1.1,\ 2.1 \\ C(k)\le C(k-1) & \textit{Cases } 1.2,\ 1.4.1,\ 2.2 \\ C(k)\le 3C(k-2)+C(k-m),\ m\ge 2 & \textit{Case } 1.3 \\ C(k)\le 3C(k-2) & \textit{Case } 1.4.2 \\ C(k)\le C(k-1)+C(k-2)+C(k-m_1)+C(k-m_2),\ m_1\ge 2, m_2\ge 3 & \textit{Case } 1.4.3 \end{cases}$$

对于*Case*2.3和*Case*2.4，由于需要转移到*Case*1.3和*Case*1.4.3，我们设Q_1为*Case*1.3所限制的搜索树大小，Q_2为*Case*1.4.3所限制的搜索树大小。根据之前的分析，可以得到不等式：

$$\begin{cases} C(k)\le 2Q_1(k-1)+C(k-2) & \textit{Case } 2.3 \\ C(k)\le 2Q_2(k-1)+C(k-2) & \textit{Case } 2.4 \\ Q_1(k)\le 3C(k-2)+C(k-m),\ m\ge 2 & \textit{Case } 1.3 \\ Q_2(k)\le C(k-1)+C(k-2)+C(k-m_1)+C(k-m_2),\ m_1\ge 2, m_2\ge 3 & \textit{Case } 1.4.3 \end{cases}$$

我们取*Case 1.3*和*Case 1.4.3*的最坏情况带入，即$m=2, m_1=2, m_2=3$时，可以得到

$$\begin{cases} C(k)\le C(k-2)+8C(k-3) & \textit{Case } 2.3 \\ C(k)\le 3C(k-2)+4C(k-3)+2C(k-4) & \textit{Case } 2.4 \end{cases}$$

综合*Case 1*和*Case 2*，最差的情况为*Case 2.4*。设$C(k)=\alpha^k$，带入可得

$$1=3\alpha^{-2}+4\alpha^{-3}+2\alpha^{-4}$$

解得$\alpha\approx 2.27$，因此改进算法的复杂度为$O(2.27^k n)$。

第4章 数据结构设计及算法实现

为了验证上一章所提出的参数算法的性能，我们需要将算法用一种编程语言实现。考虑到编程语言的复杂性和效率，我们选用C++语言，利用其面向对象的程序设计方法来实现该算法。精心设计的数据结构可以大大降低编程的复杂度并且提高程序的运行效率。因此选取合适的数据结构是本章的首要任务。

4.1 数据结构设计

4.1.1 树节点

在设计树节点的数据结构时，需要考虑到能够方便地访问其父节点，子节点。为了简便与扩展性，将子节点指针用vector存储，实际运行中容器的大小不会超过2。

同时，因为两棵进化树的节点需要进行匹配，如果两棵进化树中存在同构的子树，便将两子树的根节点配对，记录对方的节点id为pairId，若未配对则pairId为-1。

其次，定义此类相关的存取方法和一些辅助方法。

TreeNode类定义如下：

```
class TreeNode{
  private:
    TreeNode * parent; // 父节点的指针
    vector<TreeNode *> children; // 子节点的指针
    int id; // 节点id
    string label; // 叶节点标识，非叶节点则为空
    int pairId; // 对应节点id

  public:
    // TreeNode的构造函数与析构函数
    TreeNode();
    TreeNode(int id);
    TreeNode(int id,int label,int pairId);
    TreeNode(TreeNode *p); // 拷贝构造函数
    ~TreeNode();
```

```
16
17      void AddChild(TreeNode *pChild); // 添加子节点
18      void RemoveChild(TreeNode * p); // 删除子节点
19      void SetId(int x);
20      void SetLabel(const int str);
21      void SetPairId(const int id);
22      int GetId();
23      int GetLablel();
24      int GetPairId();
25      int GetChildrenSize();
26      TreeNode * GetChild(int i);
27      TreeNode * GetParent();
28      TreeNode * GetRoot();
29      TreeNode * GetSiblingNode();
30      bool IsLeaf(); // 若是叶节点返回true，否则返回false
31      bool IsRtLeaf(); // 若节点已配对则返回true，否则返回false
32      bool IsRoot(); // 返回该节点所在子树的根节点
33      bool IsSibling(const TreeNode * p); // 判断与另一节点是否是兄弟节点
34      string ToString(); // 若是叶节点则返回标识，否则返回空
35  };
```

4.1.2 进化树

考虑到需要同时表示进化树和进化树森林，我们将PhylogenyTree类定义成包含多个根节点的森林，将所有的根节点用vector保存。

算法实现过程中需要通过节点id或label访问节点的操作，用unordered_map为容器用idMap和labelMap进行映射，unordered_map内部原理是哈希表，单次映射操作复杂度为$O(1)$。

最后定义各种相关的读写方法和辅助方法，其中比较重要的有Contract、DeleteEdge、DeleteEdges和randomSPR等方法，这些都是进化树的几个基本操作，详见注释。

PhylogenyTree类定义如下：

```
1  class PhylogenyTree{
2    private:
3      vector<TreeNode *> roots;
4      unordered_map<int,TreeNode *> idMap; // 节点id映射到节点指针
5      unordered_map<string,TreeNode *> labelMap; /* 节点label映射到节点
6                                                    指针，非叶节点则不存在*/
7      void BuildMaps(); // 遍历森林，生成idMap和labelMap
```

```
8   public:
9     PhylogenyTree();
10    PhylogenyTree(PhylogenyTree * tree); // 拷贝构造函数，复制另一棵树
11    ~PhylogenyTree();
12
13    void BuildByNewick(string newickStr); // 根据Newick格式构造进化树
14    TreeNode * GetRootNode(int k); // 获取第k个根节点
15    TreeNode * GetNodeById(int id);
16    TreeNode * GetNodeByLabel(int label);
17    int GetLabeledNodeNum(); // 获取被标记节点数
18    int GetNodeNum(); // 获取节点总数
19    int GetRootNum(); // 获取根节点数
20    vector<TreeNode *> GetAllNode();
21    vector<TreeNode *> GetAllLabeledNode();
22    vector<TreeNode *> GetAllPairedNode();
23
24    void AddRoot(TreeNode * p); // 添加根节点，相当于添加一棵子树
25    void RemoveRoot(TreeNode * p); // 删除根节点，相当于删除一棵子树
26    void Contract(); // 对所有子树进行收缩操作
27    void DeleteEdge(int nid); // 删除某节点的父边
28    void DeleteEdges(vector<int> &nids); // 删除一些节点的父边
29    void EraseNode(TreeNode * p); // 将节点从idMap和labelMap中移除
30
31    void randomSPR(int k); // 随机进行k次rSPR操作（生成随机数据用）
32    string ToString(); // 将树或森林表示成Newick格式的字符串
33  };
```

4.2 算法实现

从上一章的算法描述部分可以看出，我们的算法实际上是通过限制深度的深度优先搜索算法来寻找解，并且在每次搜索时判断当前状态所属情况，然后进行分类讨论。因此，我们将不同情况分为不同的模块，模块之间可以互相调用，这样即理清了程序的结构也增加了代码重用，使代码结构清晰简洁明了。

同样，为了遵循面相对象的设计方式，我们用FPTsolver类来实现算法的主体。将每种情况用一个方法实现，并定义相关的辅助方法。由于篇幅限制，在这里只给出类FPTsolver的定义和递归主体函数MAF的具体实现方法。MAF方法主要完成了算法的前四步，然后寻找可以处理的一组点进行情况判定，并调用相应的方法进行处理。完整源代码详见FPT_faster。

FPTsolver类定义如下：

```
class FPTSolver{
private:
    PhylogenyTree * ans; // 保存最终的最大一致森林解
    void FindPendantNodes(TreeNode * p, TreeNode * r,
                          vector<int> &nids);
                        // 寻找p到其祖先r路径上的悬挂节点
    bool MergePair(PhylogenyTree *T1, PhylogenyTree *F2,
                   TreeNode * a,TreeNode * b);
                    // 在T1,F2中合并a,b两个叶节点
    bool MergeSiblingNodes(PhylogenyTree * T1, PhylogenyTree * F2);
                  // 合并T1,F2中所有能合并的叶节点，若不存在则返回false，
                     否则返回true
    bool MoveTree(PhylogenyTree * F, PhylogenyTree * T1,
                  PhylogenyTree * F2); /* 将F2中所有合并完的子树移动
                           到F中，若不存在则返回false，否则返回true*/
    Group FindGroup(PhylogenyTree * tree); /*需找一组符合要求的叶节点，
                                          可能返回3点或4点两种情况*/
    bool Case_1(PhylogenyTree * F,PhylogenyTree * T1,
                PhylogenyTree * F2, int k, Group &grp2);
    bool Case_1_1(PhylogenyTree * F,PhylogenyTree * T1,
                  PhylogenyTree * F2, int k,Group & grp2);
    bool Case_1_2(PhylogenyTree * F,PhylogenyTree * T1,
                  PhylogenyTree * F2, int k,int nid);
    bool Case_1_3(PhylogenyTree * F,PhylogenyTree * T1,
                  PhylogenyTree * F2, int k,Group & grp2,
                  vector<int> & pedant);
    bool Case_1_4(PhylogenyTree * F,PhylogenyTree * T1,
                  PhylogenyTree * F2, int k, Group &grp2,
                  vector<int> & pedant);
    bool Case_1_4_1(PhylogenyTree * F,PhylogenyTree * T1,
                    PhylogenyTree * F2, int k,int nid);
    bool Case_1_4_2(PhylogenyTree * F,PhylogenyTree * T1,
                    PhylogenyTree * F2, int k,Group & grp2,
                    vector<int> & pedant);
    bool Case_1_4_3(PhylogenyTree * F,PhylogenyTree * T1,
                    PhylogenyTree * F2, int k,Group & grp2,
                    vector<int> & pedant_ab,vector<int> & pedant_ac);
    bool Case_2_1(PhylogenyTree * F,PhylogenyTree * T1,
                  PhylogenyTree * F2, int k, int nid1, int nid2);
    bool Case_2_2(PhylogenyTree * F,PhylogenyTree * T1,
                  PhylogenyTree * F2, int k, vector<int> & pedant_one);
```

```
    bool Case_2_3(PhylogenyTree * F,PhylogenyTree * T1,
                  PhylogenyTree * F2,int k,Group &grp2,TreeNode * Rab);
    bool Case_2_4(PhylogenyTree * F,PhylogenyTree * T1,
                  PhylogenyTree * F2, int k, Group &grp2);
    bool MAF(PhylogenyTree * F,PhylogenyTree * T1,
             PhylogenyTree * F2, int k); // 递归主函数
public:
    bool MAF_K(PhylogenyTree * T1, PhylogenyTree * T2,int k);
                                // 判断T1,T2的rSPR距离是否小于k
    int MAF_Calc(PhylogenyTree * T1, PhylogenyTree * T2);
                                // 通过枚举计算T1,T2的rSPR距离
    PhylogenyTree * GetMAF(); // 获取最终的最大一致森林解
};
```

MAF函数的实现如下：

```
bool FPTSolver:: MAF(PhylogenyTree * F,PhylogenyTree * T1,
                     PhylogenyTree * F2, int k){
    if (k<0) return false; // 第一步
    while (MergeSiblingNodes(T1, F2) | MoveTree(F, T1, F2));
                            // 第二、三步
    if (F2->GetRootNum() == 0) { // 第四步
        ans = new PhylogenyTree(F); return true;
    }
    if (k==0) return false; // 此时至少需要删掉一条边，k==0返回false
    Group grp1 = FindGroup(T1),grp2; //寻找3点组或4点组，由type字段注明
    grp2.type = grp1.type;
    grp2.a = F2->GetNodeById(grp1.a->GetPairId()); //获取F2中对应节点组
    grp2.b = F2->GetNodeById(grp1.b->GetPairId());
    grp2.c = F2->GetNodeById(grp1.c->GetPairId());
    if (grp1.type == 4) grp2.d = F2->GetNodeById(grp1.d->GetPairId());
    TreeNode * roota = grp2.a->GetRoot();
    TreeNode * rootb = grp2.b->GetRoot();
    bool res = false;
    if (grp1.type == 3){ // 3点情况
        if (roota != rootb) {
            res = Case_1_1(F,T1,F2,k,grp2);
        } else {
            TreeNode * Rab = F2->GetNodeById(lca.ask(grp2.a->GetId(),
                             grp2.b->GetId())); // 获取a,b的最近公共祖先
            vector<int> pedant_ab; // 获取a,b之间的悬挂节点
            FindPendantNodes(grp2.a, Rab, pedant_ab);
            FindPendantNodes(grp2.b, Rab, pedant_ab);
```

```
28              if (pedant_ab.size()==1){
29                  res = Case_1_2(F, T1, F2, k, pedant_ab[0]);
30              } else {
31                  TreeNode * rootc = grp2.c->GetRoot();
32                  if (roota != rootc) {
33                      res = Case_1_3(F, T1, F2, k, grp2, pedant_ab);
34                  } else {
35                      res = Case_1_4(F, T1, F2, k, grp2, pedant_ab);
36                  }
37              }
38          }
39      } else { // 4点情况
40          if (roota != rootb) {
41              res = Case_2_1(F,T1,F2,k,grp2.a->GetId(),grp2.b->GetId());
42              return res;
43          }
44          TreeNode * rootc = grp2.c->GetRoot();
45          TreeNode * rootd = grp2.d->GetRoot();
46          if (rootc != rootd) {
47              res = Case_2_1(F,T1,F2,k,grp2.c->GetId(),grp2.d->GetId());
48              return res;
49          }
50          TreeNode * tmp;
51          vector<int> pedant_one;
52          if ((tmp=IsOneDistance(grp2.a, grp2.b))!= NULL){
53              pedant_one.push_back(tmp->GetId());
54          } // 若a,b之间只有1个悬挂节点
55          if ((tmp=IsOneDistance(grp2.c, grp2.d))!= NULL){
56              pedant_one.push_back(tmp->GetId());
57          } // 若c,d之间只有1个悬挂节点
58          if (pedant_one.size() != 0) {
59              res = Case_2_2(F,T1,F2,k,pedant_one);
60              return res;
61          }
62          if (roota != rootc){
63              TreeNode * Rab = F2->GetNodeById(lca.ask(grp2.a->GetId(),
64                                                       grp2.b->GetId()));
65              res = Case_2_3(F,T1,F2,k,grp2,Rab);
66              return res;
67          }
68          res = Case_2_4(F,T1,F2,k,grp2);
69      }
70      return res;
71  }
```

第5章 实验分析

实验分析同样也是算法研究的一个重要部分，文章我们利用随机生成的数据和真是的生物数据来测试算法的改进效果。为了尽量去除因为算法实现方式不同而造成的效率差异，我们在完全相同的数据结构和程序框架下重新实现了Whidden原本的参数算法。为了方便，我们将此程序成为FPT，而将改进后的程序称作FPT_faster。实验结果显示，FPT_faster在效率较之前有显著的提升。

两个算法均在以下环境中编译运行并进行测试。

表 5–1 实验环境

操作系统	MAC OS X 10.10.2
处理器	2.4GHz Intel Core i5
内存	4GB 1600MHz DDR3
编译器	Apple LLVM version 6.0

5.1 随机数据测试

首先我们按照如下步骤生成随机数据：

1. 设R为一些子树的根节点集合，初始时R为n个叶节点的集合。
2. 在R中随机选择两个节点作为兄弟节点合并，并将它们移除R，将它们的父节点加入R。
3. 重复第二步，直到R中只有1个节点，将该节点为根的树作为T_1。
4. 在T_1上随机进行k此rSPR操作，得到T_2。

按照上述方法，我们可以得到两棵叶节点数为n，rSPR距离不大于k的两棵进化树T_1, T_2。

为了测试随rSPR距离增加算法运行时间的变化，首先生成了叶节点数均为100，rSPR距离由0递增至40的100对进化树。FPT和FPT_faster的运行结果如图**??**所示。因为算法时间复杂度为指数级增长，我们对纵坐标时间做了对数处理。由图可以看出，FPT在rSPR距离增大到30以上时已经很难在可接受的时间范围内找到最优解。而FPT_faster则可以在更短的时间内找到更复杂数据的解。实际上，对于rSPR距离不大于30的情况，FPT_faster基本都能在1分钟内给出最优解。

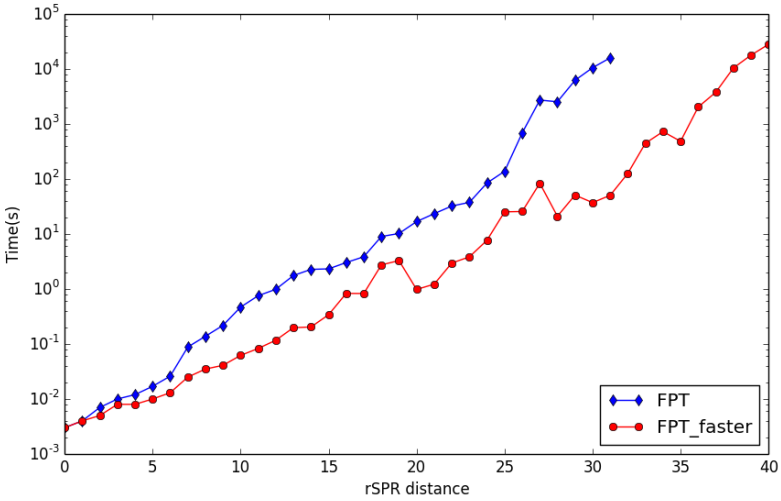


图 5–1 叶节点数均为100，rSPR距离从0到40

两种算法实际上一个共同的特点是根据进化树的不同形态划分为不同情况，进而采取相应地策略。而不同的策略，意味着不同的复杂度，因此算法实际运行的时间与在搜索过程中每种情况所遇到的次数有很大关系。对于每种策略，我们用第三章的分析方法解出一个特征根α，α的值越小意味着该情况复杂度越小。同时，我们对FPT和FPT_faster分别统计了所有随机数据计算过程中每种情况的占比。（如表5-2，表5-3）由表可以看出，在FPT中三种情况的分布相对均匀，各占三分之一。这也意味着有三分之一的情况是该算法的最坏情况。而在FPT_faster中，不仅最坏情况的复杂度得以降低，而且最坏情况的占比也大大减小，有超过90%的情况的α值小于等于2。因此，算法运行的时间理所当然地大大减少了。

表 5–2 FPT各情况复杂度及分布

情况	α	占比
Case 1	2	36.07%
Case 2	1	31.45%
Case 3	2.42	32.48%

表 5–3 FPT_faster各情况复杂度及分布

情况	α	占比	情况	α	占比
Case 1.1	2	59.26%	Case 1.4.3	2.15	8.52%
Case 1.2	1	12.08%	Case 2.1	2	3.82%
Case 1.3	2	13.51%	Case 2.2	1	2.03%
Case 1.4.1	1	0.38%	Case 2.3	2.17	0.11%
Case 1.4.2	1.73	0.00%	Case 2.4	2.27	0.30%

5.2 生物数据测试

第6章 总结及展望