基因表达式编程

——用一种人工智能方法进行数学建模

前 言

1999 年九月到十月之间，在我产生关于基因表达式编程（gene expression programming, GEP）的基本思想的时候，我几乎没有意识到它的独特性。当时我正在读 Mitchell 的《An introduction to Genetic Algorithms》(Mitchell 1996)，并小心翼翼地逐个解答每一章后面提供的计算机练习。所以，我实现了我的第一个遗传算法的程序而且实现了一个我认为是遗传程序设计（genetic programming, GP）的系统。正如 GP 系统一样，这个新的系统能够通过进化得到大小和形状不同的计算机程序，但是，令人惊讶的是，它的效率比老的遗传程序设计系统高出 100－60，000 倍。那么，这其中到底发生了什么？是什么导致了如此让人惊奇的性能差异？对于一个进化生物学家来说，这个问题的答案很简单：这个新的系统——基因表达式编程——仅仅是跨越了表现型的极限。这意味着由 GEP 进化的复杂的计算机程序（表现型）被完全编码在一个固定长度的简单字符串里面（染色体或者基因型）。这种基因型与表现型的分离就像打开一个盛满好事情和各种可能性的潘多拉魔盒。这些好事情中最重要的可能就是所使用的遗传算子的种类或数量实际上不存在任何限制。另一个很重要的方面在于产生更高级别的复杂度实际上成为一个微不足道的任务。的确，要由一个单基因系统产生一个多基因系统或者由一个单元胞产生一个多元胞本身就是一个很小的任务。而且，每个新的系统都会产生新的可能性，这些新的可能性将极大地扩展该技术的应用领域。

在这本也是第一本关于基因表达式编程的书里面，我详尽地描述了基本基因表达式算法的及其诸多变型算法，给出其所有的实现细节以使任何具有基本编程技巧（或者愿意学习这些技巧）的人能够自己实现该算法。第一章简要介绍在生物学中基因表达式中的几个主要概念，以便说明它们与广义上的人工进化系统和狭义上的 GEP 系统中的主要概念之间的关系。第二章介绍基因表达式编程的几个主要概念并详细介绍它们的结构和功能组织。第三章详细描述基本基因表达式算法和基本遗传算子。另外，通过彻底分析一个非常简单的问题，给出发现过程中所产生的所有的个体程序，并以此来阐明其中的适应性和进化机理。第四章描述一些基本基因表达式算法的应用，其中包括大量未曾发表的材料，即参数优化、Kolmogorov-Gabor 多项式进化、时间系列预测、分类器系统、连接函数的进化、多元胞、自动定义函数、用户定义函数等。第五章所有演示的关于如何用基因表达式编程方法激励完整的神经网络的材料也是全新的。这一章采用这些 GEP-网解决了两个基准问题，并对它们的效率提出了有效的衡量标准。第六章演示如何用基因表达式编程方法来解决组合优化问题。这一章介绍多基因族和许多组合优化专用算子，并通过两个调度问题对其效率进行评估。最后一章讨论一些重要、同时也是很有争议的进化学问题，这些问题的讨论可能会让进化计算机学家和进化生物学家感到耳目一新。

致 谢

一种新范例的发明经常会招致强烈的抵制，特别是当它似乎会危及早已确立其稳固地位的技术和事业的时候更是如此。当我的工作在本该成为讨论和交流思想的论坛的科技期刊和会议上发表的时候，我简直如同作噩梦一般，我的工作和我自己都被彻底排斥，遭到蔑视。尽管最开始有反对意见，但是我身处的幸福环境和身边这些足智多谋的人，使我最终能够让我的工作为人所知、为人所用。我深深地感谢 José Simas，一个熟练的图形及网页设计和软件开发者，因为他从一开始就对我和 GEP 深信不疑，并不遗余力地在网络上推广 GEP。我们一起创办了 Gepsoft 公司，一起开发基于基因表达式编程的软件，这些软件今天已经在世界各地帮助无数的科学家和工程师解决他们所遇到的问题。同时，Gepsoft 公司使我有可能专注于这本书的写作和其它一些新算法的开发。的确，我在 Gepsoft 公司的工作和我的写作有很多相互受益的地方。

我也十分感谢 Pedro Carneiro，一个极具天赋、富有渴望的音乐家，感谢他阅读并编辑原稿的前三章。José Simas 也阅读了原稿的许多草稿，他在整个过程中自始至终陪伴我并贡献了许多有价值的讨论和建议。实际上他是我的第一个读者，我在写作的时候也一直考虑到他的意见。

最后，我也想要感谢 José Gabriel，一个天赋极高的印刷者，一个能工巧匠，感谢他在这本书中自始至终的参与和在处理本书的印刷过程中所倾注的特别心思。

献给 José Simas
因为所有的梦想

同时

献给我的祖父，Domingos de Carvalho
因为他的远见

简 介 1

本章的目的是希望把焦点投向基因表达式编程（gene expression programming，GEP）与它的前身，即遗传算法（genetic algorithms, GAs）和遗传程序设计（genetic programming, GP），之间的根本区别。所有这三种算法都属于一个更广的遗传算法类（Genetic Algorithms，这里采用大写字母是为了将更广义的遗传算法类与经典遗传算法区别开来），因为它们都采用个体的种群，根据适应度对个体进行选择，通过一个或者多个遗传算子来引入遗传变化。这三种算法的根本区别在于其个体的本性不同：GAs 中的个体是固定长度的线性符号串（染色体）；GP 中的个体是不同尺寸和形状的非线性实体（分列树），GEP 中的个体也是不同尺寸和形状的非线性实体（表达式树），但是这些复杂的实体被编码成固定长度的简单字符串（染色体）。

如果我们对于地球上生命的历史有所了解（见参考文献，如 Dawkins 1995 或 Maynard Smith 和 Szathmáry 1995），那么我们会发现 GAs 和 GP 之间的区别只是表面的：这两种系统都只采用一种实体，该实体既起基因组的作用，又起主体（表现型）的作用。人们经常抱怨这些系统有这样或那样的限制：如果该系统的遗传操作容易进行，那么它就会失去功能的复杂性（GAs 中的情况），然而如果该系统表现出某种程度的功能复杂性，那么该系统就极难由修饰获得繁殖（GP 中的情况）。

R.Dawkins（1995）在他的书《*River Out of Eden*》中列举了任何生命爆发的一些极限，其中第一个极限就是构成一个自复制系统的“复制体极限”，在该系统中存在遗传变化。同样重要的是复制体是因为其自身的性质才得以存活。第二个极限就是“表现型极限”，其中复制体是因为其对某些其它物质的偶然作用才得以存活。这里的“某些其它物质”是被称为表现型或主体的实体，即那些直接面对环境同时完成所有工作的实体。一个简单的复制体/表现型系统的实例就是地球生命的 DNA/蛋白质系统。人们相信，如果生命要演变到能够超越最基本的发展阶段，就必须超越这个表现型极限（Dawkins 1995 或者 Maynard Smith 和 Szathmáry 1995）。

与此相似，GAs 和 GP 的实体（简单复制体）都是由于它们自身的性质才得以存活。所以我们不难理解为什么近年来在科学领域中有人努力试图突破进化计算中的表现型极限。这些努力中最杰出的成果就是发展的遗传程序设计(developmental genetic programming)或者 DGP（Banzhaf 1994），其中的二进制串用来对数学表达式进行编码。这些表达式被解码成一个长 5-位的二进制编码，称为遗传代码。与它所类比的自然遗传代码恰好相反，这种“遗传代码”在运用于二进制串的时候，经常产生非法的表达式（在自然界中不存在这种结构上不正确的蛋白质）。所以，巨大的计算资源被浪费在编辑这些非法结构中，这就在很大程度上限制了系统的效率。因此，DGP 相对于 GP 的性能优势十分微小也就不足为怪了（Banzhaf 1994，Keller 和 Banzhaf 1996）。

基因表达式编程是一个成熟的复制体/表现型系统的实例，这其中染色体/表达式树构成了一个功能完善、不可分割的整体（Ferreira 2001）。的确，在 GEP 中不存在这种表达式树或程序非法的情况。显然，GEP 染色体和表达式树之间的相互作用要求一个明确的翻译系统来将染色体语言转化成表达式树的语言。更进一步，我们将会看到 GEP 染色体的组织结构允许对基因组进行不受限制的修饰，从而为进化的出现提供了一个完美的条件。的确，为了向 GEP 种群中引入遗传修饰而开发的各式各样的遗传算子集所产生的表达式树始终是合法的，这使 GEP 作为一个简单的人工生命系统通过超越复制体极限而完整地建立起来。

为了帮助不太熟悉生物学的读者理解 GEP 与其它遗传算法之间的根本区别以及为什么

GEP 能够成为进化计算中的重要一跃，了解一点关于生物学基因表达式中的主要分子的结构和功能以及它们一起作用的机理是将会是十分有用的。接下来的部分简单介绍进行信息新陈代谢的主要分子的组织结构以及蛋白质变异如何与进化相关的问题。如果读者想进一步了解这些问题，可以参考任何一本生物化学的教科书，而我个人则推荐最近出版的 Mathews 等（2000）著的《生物化学》一书，因为该书叙述清晰，文字优雅。

1.1.生物学基因表达式实体

在细胞中，遗传信息的表达是一个十分复杂的过程，涉及到数百个分子。为了我们的需要，我们只需要了解几类主要分子，即 DNA，RNA 和蛋白质，的结构与功能的基本知识。

DNA 是遗传信息的载体，蛋白质读出这些信息并把它们表达出来。RNA 是 DNA 的工作副本，虽然它的存在在细胞的环境中也有意义，但是它的等价体在像 GEP 这样的计算机系统中却没有什么用处。尽管如此，为了理解 GEP 和其它遗传算法之间的根本区别，知道这些分子的结构和性质还是很重要的。

1.1.1.DNA

DNA 的分子是由四种核苷酸（分别用 A，T，C，G 来表示）构成的长线性串。每个 DNA 分子实际上是一条双螺旋链，其中每条链与另一条链形成互补关系，因此，增加一条单链不会增加任何信息。在这个双链的结构中，A 与 T 配对，C 与 G 配对（如图 1.1）。DNA 的这种双链的，互为补充的本性对于细胞中遗传信息的复制来说至关重要，但是这种本性对于像 GEP 或 GAs 这样的计算机系统来说却并不重要。的确，GEP 和 GAs 的染色体都是单链的而且它们的复制操作是由简单的程序指令完成的。

DNA 中存储的信息由四种核苷酸的序列构成，这些核苷酸被称为 DNA 的基本结构。DNA 的二级结构由其中所能形成的双螺旋结构构成，DNA 缺少一种由唯一的、三维的分子排列构成的三级结构。DNA 分子相互堆叠，形成一个随机的螺旋线。由于像催化活性这样的复杂功能与三级结构密切相关，所以 DNA 分子对于许多在细胞中完成的工作并没有什么作用。

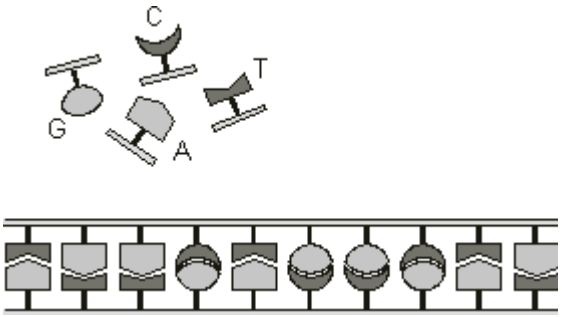


图 1.1.DNA 分子双链中的基对。注意较大的 G 和 A 对，分别与较小的 C 和 T 对，将两条链安排在完全相同的间隔距离上。还要注意，一条链中所包含的信息与其它链中包含的信息在本质上是相同的。事实上，DNA 链是互补的。

然而，简单的 DNA 分子是存储信息的绝佳选择。在双螺旋结构中，核苷酸两两互补并固定在双螺旋的内部。这使得 DNA 在化学上具有惰性和稳定性，从而使其具有保存信息的理想特质。事实上，在细胞中，DNA 进一步被受保护的真核细胞的细胞核或原核细胞中的类核环境所保护。但是对我们来说最重要的是，DNA 不具有催化活性和结构多样性：首先，

那些潜在的功能组（A，T，C，G 这些基础）被锁在双螺旋的内部，第二，分子缺少三级结构，另一个催化活性和结构多样性的前提。

简单的说，DNA可以看作一个由四种不同字符构成的长链（A，T，C，G），其中字符的序列构成了遗传信息。遗传信息或者说地球上所有生物体的蓝图都被写在由这四个字符构成的DNA语言里面。例如，一个普通的哺乳动物的基因组含有大约 5×10^9个DNA的基对（或者字符，如果只考虑单链）和大约 300，000 个蛋白质和RNA基因的编码，而这些RNA的基因是基因组表达的中间产物。下面我们将看到遗传信息如何被表达成蛋白质。

1.1.2.RNA

如我早先提到的一样，在整个信息的新陈代谢全景图中，RNA 可以被视为某个特定的 DNA 的工作副本。当一个蛋白质基因被表达的时候，基因的某个副本以信使 RNA（mRNA）的身份被产生出来并用来指导蛋白质的合成。所以，从信息的角度来说，这个 RNA 的副本所包含的信息与原 DNA 所包含的信息完全相同。

信使RNA 并不是细胞中唯一一种起作用的 RNA 分子。尽管在信息解码中起到中心作用，mRNA 在结构上显得十分单调。下面讨论的结构多样性是其它的 RNA 类别的典型情况，也就是转移 RNA（tRNA）和核醣体 RNA（rRNA）。

和 DNA 一样，RNA 的分子也是由四种核苷酸（核糖核酸，在这里是 A，U，C 和 G）构成的长链。与 DNA 不同，RNA 分子是单链的，而且其中有些分子可以在一个唯一的三维结构中堆叠起来。RNA 分子能够堆叠的一个原因就是由于短的序列存在，这些短的序列与同一分子中的其它序列形成互补。显然，如果这些互补的序列能够相遇，就会生成短的双螺旋。这些分子间的双螺旋就是某些 RNA 分子的唯一的三维结构所必需的。

因此，像蛋白质一样，许多 RNA 分子可以有一个唯一的三维结构（三级结构），并因此表现出某种程度的结构多样性和功能多样性。RNA 双螺旋结构中的互补规则与 DNA 十分类似，A 与 U 配对，C 和 G 配对。在含有三级结构的 RNA 分子中，有些核苷酸参与螺旋构成而且因此不具有化学作用，但是其它一些功能组则不受限制地可以参与到各种相互作用中甚至参与生物催化过程。的确，加入这种唯一的三维结构使得 RNA 分子能够发挥生物催化剂的作用（核糖酶）。

因此，虽然 RNA 需要的化学元素减少了，但是 RNA 分子还能够同时起到基因型和表现型的作用，即扮演一个简单复制体的角色。但我们必须注意这其中基因型和表现型是如何被绑在一起的：任何复制体上的修饰都将在它的性能上直接反映出来。这些系统中没有细微变化或者中性变化存在的余地。而这些细微变化或者中性变化是一个高效的进化过程的根本所在（Kimura 1983；同时可参考本书 7.4 节）。

这些简单复制体系统的另一个重要限制可以借助 GP 这个人工系统很好地诠释，因为 GP 本身也是一个简单复制体系统。在 GP 中，发生在分列树上的大部分修饰都会导致非法的结构。这种系统的问题在于一些非常高效的搜索算子，比如点变异算子，在其中不能使用。作为其代替办法，采用一种效率低下的子树交换来产生合法的分列树。然而，无论在使用遗传算子的时候多么细心，嫁接和剪枝所能起到的作用都显然十分有限，而且在这样的系统中，永远不可能彻底遍历搜索空间。

1.1.3 蛋白质

蛋白质是由 20 种不同的氨基酸构成的线性长串，这些氨基酸由DNA中的存储遗传信息的直接表达式构成。这意味着由四个字母构成的DNA语言被翻译成更复杂的由 20 个字母构

成的蛋白质语言。显然，必定有某些编码（遗传代码）将这种由 4 个核苷酸构成的语言翻译成由 20 个氨基酸构成的语言。为了区分这 20 个氨基酸，至少需要 20 个DNA“ 词汇 ”。通过对每个氨基酸使用核苷酸的三联体（密码子），3 个字母的可能的“ 词汇 ”有 4^3＝64 个。使用这些“ 词汇 ”来对 20 个氨基酸进行编码已经足够了，事实上，大多数的氨基酸都有多个密码子，因为 64 个密码子中的 3 个用来对“ 停止合成 ”指令进行编码。当然也存在一种用来对“ 开始合成 ”指令进行编码的密码子，但是这种密码子同时也对蛋氨酸进行编码，其中蛋氨酸是蛋白质中发现的 20 种氨基酸之一。遗传代码实际上是通用的，也就是说地球上的所有生物体除了极少的特例以外，均采用相同的密码子来将它们的基因语言翻译成蛋白质语言（遗传代码的结构见 1.2.4 节，如图 1.6）。

因此，蛋白质的信息就是这样对每个三联体逐个解码并表达成氨基酸的线性序列。虽然蛋白质的氨基酸序列反映了相应的 DNA 分子的序列，但是蛋白质具有其唯一的三维结构并表现出唯一的性质。因为蛋白质所包含的化学元素更加丰富，氨基酸的线性串通过特殊的方式相互堆叠从而让每个蛋白质具有其自身的三维结构。这种唯一的三维结构或者蛋白质三级组织结构和氨基酸的无尽的化学结构一起，使得蛋白质能够起到各式各样的作用，其中包括生物催化剂和酶的作用。事实上，蛋白质才是细胞中真正起作用的成分。

我们还要注意，与 RNA 一样，蛋白质也可以同时起到基因型和表现型的作用。但是，我们要注意，尽管这样的系统存在的功能多样性更加丰富，它们搜到一些限制：任何复制体上的修饰都将立即在其性能上直接反映出来。

从理论上说，DNA、RNA 和蛋白质之间的差别对于理解 GAs，GP 和 GEP 之间的根本区别来说是最有用的。GAs 和 GP 都是单复制体系统，只采用一种实体：这种实体在 GAs 中是 0 和 1 构成的线性字符串，在 GP 中是由多种元素构成的复杂分叉结构。很多人相信，在生命历史的早期曾经存在一个与我们今天的简单“ 蛋白质世界 ”同时代的所谓简单“ RNA 世界 ”。RNA 和蛋白质由于某种原因开始一起发生作用，同时 DNA 也加入其中。现在地球生命复杂的 DNA/蛋白质系统就是这个进化过程的产物。

令人惊奇的是，在地球生命产生 40 亿年以后的今天，计算机科学家们也迈出了与自然界十分相似的步伐：他们首先发明了一个简单的复制体系统，进而又发明了一个更加成熟的复制体/表现型系统。Holland（Holland 1975）在 60 年代发明的遗传算法就是对简单 RNA 复制体的类比，它具有线性染色体和有限的功能，而 Koza（1992）推广的算法可以看作是功能更加丰富的简单蛋白质复制体的类比。但是奇怪的是，在经历了很多对 DNA/蛋白质系统的模拟后，人们试图创造一个基因组/表现型的有意识的努力仍然与所期待的“ 向前一跃 ”相去甚远（Banzhaf 1994，Ryan 等 1998）。

另一方面，1999 年我本人发明了 GEP 这个成熟的基因组/表现型系统（Ferreira 2001），当时我全然没有注意到其他研究者为了创造出一个基因型/表现型系统所付出的艰苦努力。事实上，我最初是从 Mitchell 的书（Mitchell 1996）中读到 GP 的，GP 给我留下了很深的印象，我试图自己编出一个 GP 的系统。我假设我使用的仅仅是我所了解的生物化学和进化方面的知识，从来没有想到要去做一个没有自治染色体的系统。很显然，关于信息新陈代谢的那些复杂的事情被舍弃了，因为这些与一个计算机系统无关，而且这些规则不受化学的约束。所以，双链染色体，与 RNA 类似的中间产物，以及具有复杂的翻译机制的复杂的遗传代码没有能够进入到 GEP 系统中。更进一步，我也知道要让一个基因型/表现型的机器能够平稳地运行，必须对遗传算子进行限制，使其总是产生合法的结构。以上这些方面的努力的最终的结果就是第一个功能完全的基因型/表现型系统，该系统能够在任何编程语言下实现，因为该算法与具体的语言的工作方式没有任何关系。

1.2.生物学基因表达式

在存储信息的 DNA 分子中，最让我们感兴趣的是对蛋白质进行编码的基因。在真核细胞中，平均只有 2%是对蛋白质进行编码的基因，其余的都是重复序列和基因内区（中断基因编码序列的非编码序列）。而原核细胞和病毒则更加紧凑，它们是用来对蛋白质进行编码的基因中不可缺少的部分。为了满足我们的需要，我们只需要了解最简单的基因组，因为这些基因组已经使生物机器变得极其复杂。

1.2.1.基因组复制

当 James D.Watson 和 Francis Crick 在 1953 年共同提出双螺旋结构的时候，DNA 的复制机制开始逐渐广为人知，要将这一机制中必不可少的参与者分离出来进行研究，并理解这个机制的细节问题只不过是一个时间问题。这些互补的，双螺旋结构的 DNA 分子将自己打开，每个链成为用来与其互补的链合成的模板。例如，当一个细胞内的病毒复制的时候，它将自己的基因组复制数百次，从而在几分钟以内形成数百个病毒粒子。

如同在生物学中的基本地位一样（不仅是在 DNA 复制中，而且在 DNA 的修复中），DNA 分子双链结构的本性对于像 GEP 和 GAs 这样的计算机系统里面的基因组复制来说，并没有什么用处，因为这些系统中的单链染色体是通过简单的计算机指令复制的。更重要的是，某些修饰操作能够产生基因多样性。下一节我们将简要分析遗传变化的基本机理。

1.2.2.基因组重构

在细胞中，信息重构的过程和很多不同的功能相关，比如对环境压力的反映，DNA 的修复，基因表达式的控制及基因多样性的产生。为了满足我们的需要，我们只需要了解他们在进化中的作用，特别是如何产生遗传修饰，以及遗传多样性如何反映自身在某个物种的基因池内众多蛋白质变体中的作用。

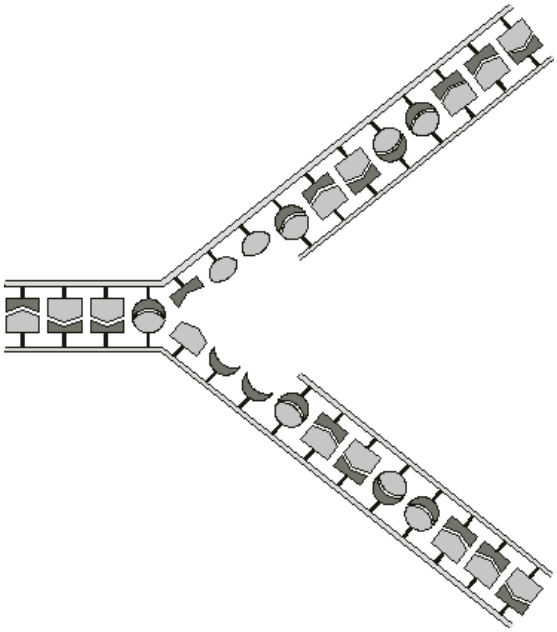


图 1.2.DNA 分子的复制。每条链作为一个新的互补链的模板。当拷贝完成的时候，将会产生两个子 DNA 分子，每个分子在序列上与分子母体完全相同。

在自然界中的所有生物，包括活着的和已经灭绝的生物，其多样性让人难以置信。这很大程

度上是重构过程的结果，这种重构过程发生在生物体的基因组上，从而产生各式各样的蛋白质功能。同样，在 GEP 中，个体的种群（计算机程序）通过发展新的能力不断进化而越来越适应环境，因为经过一定代数后，遗传修饰得以积累。但是，由于某些简单的，特别是那些防止大规模灭绝的机制存在，使得在计算机系统中有可能极大地加快进化的速度，使得系统进化的速度比新的病毒变种的产生速度还快。

1.2.2.1.变异

当某个特定的基因组进行自我复制，并把其遗传信息传给下一代的时候，子代分子的序列有时会与母代分子的序列在一点或者多点上有所不同。虽然这种复制机制近乎完美，但是有些时候新合成的链上会引入一个不匹配的核苷酸。尽管细胞中存在某些机制能对大部分不匹配的情况进行纠正，但是其中还是有些不匹配的情况没有被修复而被直接传给了下一代（图 1.3）。

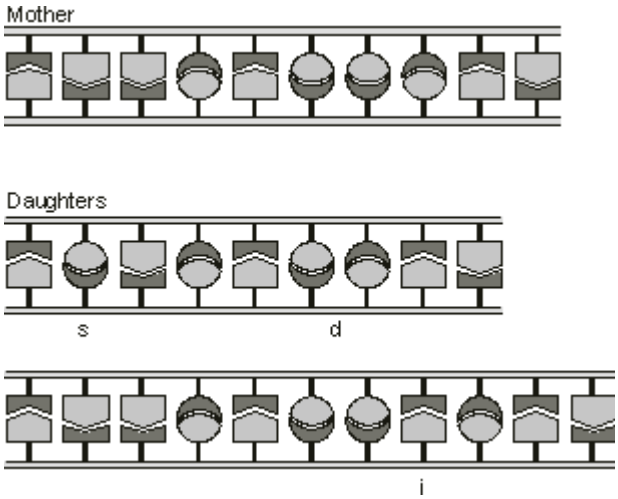


图 1.3.基因的 DNA 序列中的变异。此处演示一个基替换（s），一个小的删除（d）和一个小的插入（i）。

在自然界中，变异出现的比例是受严格控制的，各种不同的生物体有不同的变异比例，例如病毒和细菌的变异比例就比真核细胞的变异比例要高。而在这其中，病毒的变异比例当然最高，因为一个病毒粒子可以在每个受感染的细胞里面留下上百个甚至上千个后代，测试某一代中的几个新的基因组。

进一步分析蛋白质上的点变异对蛋白质本身的影响也是很重要的工作。在一个基因内部，某个核苷酸被另一个核苷酸所取代可能产生几种不同的影响：(1）新的密码子可能对一个新的氨基酸进行编码（这称作错义变异）；(2）新的密码子可能对一个“ 终止 ”密码子进行编码，截断该蛋白质，抑或一个“ 终止 ”密码子变异成一个氨基酸密码子，将链延长（无义变异）；(3）更常见的情况是，基因内的点变异对蛋白质序列完全没有影响，因为新的密码子可能编码成了一个相同的氨基酸（中性变异）；(4）还有一种情况是，在真核细胞中，大部分基因序列都被非编码区（基因内区）所干扰，如果变异发生在基因内区里面的话，那么这种变异对蛋白质序列根本没有任何影响（这也是中性变异的一个实例)。

在另一种蛋白质变异中，或大或小的基因片断可能被插入到基因的编码区域中，或者从基因的编码区域中被删除。插入或者删除较大的基因片段几乎总会导致生成的蛋白质存在缺陷。插入或者删除较小的基因片段所产生的影响取决于这些修饰操作是否会导致基因的阅读框发生移位（如果导致基因的阅读框发生移位，则称之为移码变异)。如果插入或者删除的基因片段恰好是三的倍数，那么一个或者多个密码子就会被相应地移除或者插入，导致蛋白质中的一个或多个氨基酸被删除。非移码删除/插入操作的后果与错义变异导致的结果相似。

这些结构和功能上的变异对蛋白质造成的影响可能各不相同。点变异造成的影响可能是中性的，可能对氨基酸毫无改变，也可能在该蛋白质所在的位置上用一个功效相同的蛋白质去取代它。密码子的插入/删除也可能不产生什么明显的后果，只对蛋白质的功能有很小的改变。在某些偶然的情况下，这些变异会提高蛋白质的效率，为该生物体赋予某些选择上的优势。另一方面，无义变异和移码变异几乎每次都会导致致命的影响，特别是当新产生的蛋白质对该生物体的存活来说至关重要的时候，尤其如此。这种变异有时也能导致一些新的、革命性的特性出现，虽然这种情况十分少见。

我们将会看到，在 GEP 中，大部分的变异，包括点变异和小的插入操作，都会对表达式树的结构和功能产生深刻的影响，这一点与自然界中的无意义变异和框架转移变异十分相似。尽管这种变异对 GEP 的进化能力和其它几种新特性的引入来说极其重要。但是，一些不那么剧烈的变异操作在 GEP 中也会出现。事实上，某些变异操作会非常平缓地改变表达式树，它们会或多或少地提高表达式树的效率。进一步说，在 GEP 中，有些变异操作也会产生十分明显的中性效果。例如，无论表达式树的结构如何，发生在基因的非编码区域内的变异操作均对表达式树的结构没有影响。其它的一些中性变异操作也很容易发现，因为它们会产生一些结构不同的表达式树。在这种情况下，新的表达式树与父代表达式树（在数学上）等价。我们还将看到许多变异操作，包括从最保守的到最彻底的，对进化得到好的计算机程序来说都是很重要的。

1.2.2.2.重组

蛋白质通过世世代代地不断积累各种各样的变异达到逐渐进化的目的。但是变异并不是遗传多样性的唯一来源。在本节的剩余部分中将会提到一些其它的遗传算子。其中的一种算子就是重组。在自然界中存在各种各样的重组过程，它们牵涉到不同的过程同时发挥不同的功能。然而，在所有的重组操作中，两个供体分子中的某些遗传元素的片断相互交换，这样，每个供体中的遗传信息得以在后代中表现出来（如图 1.4）。例如，在有性繁殖时，两对同源染色体交换其 DNA 片断。

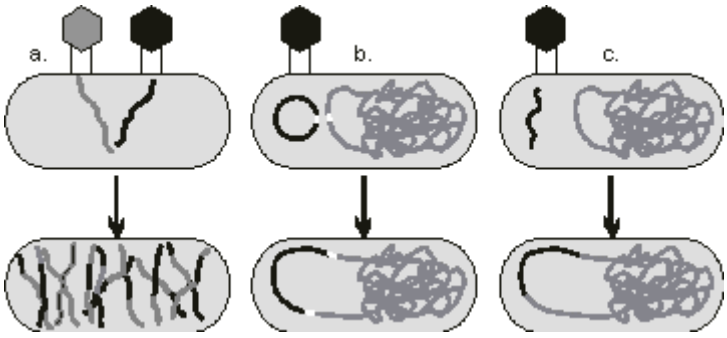


图 1.4.三种重组过程。a)同源重组；b）特定位置的重组；和 c)非同源重组。注意特定位置的重组和非同源重组比同源重组具有更强的变换能力。

尽管在 GEP 中并不要求同源序列，同源染色体配对时同源重组操作的简单图示还是能够帮助我们了解 GEP 如何通过重组过程来产生种群中的遗传多样性。这个简单的图示无疑是十分方便的，因为在 GEP 中发生重组的染色体具有相似的结构，而且重组过程中会产生两个子代染色体。因此，在重组时，两个染色体（并不需要同源）配对并交换它们的部分材料，生成两个新的子代染色体。需要注意的是，由于结构同源性的原因，染色体中某个特定基因中特定位置上的片断将绝对不会和另一个基因中不同位置上的片断相互交换，或者说一个基因尾部的片断绝对不会和一个基因头部的片断相互交换。如此说来，GEP 的重组操作虽然表面上看起来和同源重组非常相似，但是在生物学上没有对应的概念。

1.2.2.3.转座

转座遗传元素由可以在基因组中移动的基因构成。生物学中有三类结构和转座机制不同的转座元素。它们同时存在于真核细胞和原核细胞中，并对它们转座的目标染色体造成不同的影响。比如，它们可以通过干扰基因的编码序列降低基因的活性；它们也可以通过提供一个启动区和核糖核酸聚合酶相结合的脱氧糖核酸分子或者转录触媒来激活一个相邻的基因；或者它们也可以通过产生同源序列对一个染色体进行重构，而这些同源序列可以在以后的同源重组中使用。这些不必考虑边界的，甚至包括物种边界，可以在染色体之间移动的“ 跳跃基因 ”的存在，极大地改变了我们对进化所抱有的观点，即只是中性变异平缓地推动进化过程的观点。转座操作的影响非常剧烈，所以很少显式采用。与此类似，非同源重组和无义变异以及移码变异的影响也十分剧烈。尽管如此，转座（以及非同源重组）在生物界不仅十分频繁而且分布还很广，并且在生物中留下了许多假象。虽然如此，在大部分情况下转座操作以及一些非同源重组都是有害的。只有在很少的情况下会形成一些某些“ 有希望的怪胎 ”，产生一个全新的蛋白质。实际上在自然界，结构上和功能上都非常不同的蛋白质都有相同的结构主体（域）可能就是因为这些遗传变化的存在。

各种类型不同的转位子的结构细节与其多样的机制对于像 GEP 这样的计算机系统来说并不重要。GEP 中采用的转座元素是自然界中发现的转座元素的简化体。首先，在 GEP 中，所选定的转座元素只在同一个染色体内转座。其次，GEP 的转座元素可以是整个基因，也可以是基因片段。第三，任何基因或者片断都可以成为一个转座元素，而不必满足是特定的可区分序列的要求。第四，转座元素被完全复制到目标位置。最后，在基因转座中，供体序列在原位置上被删除，然而在转座片断中供体序列保持不变，通常会产生驻留在同一个染色体中的两个同源序列。我们会看到，在 GEP 中，采用转座会产生简单、重复的序列。

1.2.2.4.基因复制

基因复制在蛋白质的进化中起着重要的作用。虽然基因复制的机理还不为人知，但是一个基因偶尔会在复制过程中被复制两次。如果需要生成大量的蛋白质的话，这样的转换并没有什么潜在的危害，甚至有可能有一定的优势。另一方面，如果时间足够的话，这两个基因有可能开始独立进化。其中一个可能继续表达原来那个蛋白质，而另一个则有可能进化到一个完全不同的蛋白质中。

在 GEP 中，基因偶尔也会被复制。虽然对于基因复制没有特别的算子，但是一个基因可以在基因转座和基因重组的联合作用下进行复制。有趣的是，染色体中有重复基因的情况经常出现在 GEP 种群的那些最佳个体中。

1.2.3.转录

将遗传信息表达到蛋白质中的过程并非直接源于 DNA。将 DNA 语言转换成蛋白质语言，需要一个中间分子。该分子是一类特别的 RNA，称为信使 RNA，其合成过程称为转录。

在转录的过程中，基因序列被复制到一个 mRNA 中，该过程采用一个 DNA 分子的一条链作为模板（如图 1.5）。DNA/RNA 双链的互补规则和 DNA 中的互补规则十分相似，rA 与 dT 配对（r 代表 RNA，d 代表 DNA），rU 与 dA 配对，rG 与 dC 配对，rC 与 dG 配对。根据互补规则，产生一个与基因完全相同的 rRNA 形式的拷贝。该信使 RNA 含有蛋白质合成所需的所有信号，即“ 开始 ”信号和“ 终止 ”信号。这个信使 RNA 被放到细胞中合适的位置，成为蛋白质合成的模板。比如，在真核细胞中，在细胞核中合成的 mRNA 就必须进入翻译机制所在的细胞质中去。

我们前面已经提到，在细胞环境下对 mRNA 的需求是有意义的，但是对于像 GEP 这样

的简单计算机系统来说却没有什么意义，因为在计算机系统中，小而简单的基因组在当前情况下不具有基因表达式规则的成熟机制。也许不久的将来我们会看到像 GEP 这样的计算机系统具有更加复杂的基因组并能进行复杂的细胞体分化。也许到那时就需要像 mRNA 这样的中间体来保证基因表达式的模式各不相同。

图 1.5.DNA 与 mRNA 的关系。注意 mRNA 与它转录的来源 DNA 链是互补的。还要注意开始信号前面以及终止信号后面的序列都不会被翻译成氨基酸。为了描述开始信号和终止信号，这里所给出的序列比实际序列要短很多。

1.2.4.翻译和翻译后修饰

细胞内的蛋白质合成过程涉及到非常复杂的机制，关系到数百个分子。翻译过程中的主要问题包括：(1) 作为蛋白质合成模板的 mRNA 分子；(2) 核糖体，实际解码过程就发生在其中；(3) 一类特殊的 RNA（转移 RNA），该 RNA 将恰当的氨基酸运送到复杂的 mRNA 或者核糖体中。

1.2.4.1.翻译

DNA 和 RNA 合成的逻辑关系与蛋白质的合成相比来说相当简单，这种关系以核苷酸间的互补规则为基础。从化学角度来说，不存在简单的办法能够直接使三联体密码子与恰当的氨基酸配对。事实上，许多翻译方式经过进化得到了许多成熟的机制，这些机制正好能够解决这个问题。氨基酸必须能够正确地附着到那些特殊分子上，这些特殊分子进一步与 mRNA 中正确的密码子结合起来的。这一类特殊的分子也是 RNA 分子（转移 RNA），但是它们与 mRNA 在结构上和功能上均不相同。我们已经看到，tRNA 有三级结构，因而具有各式各样的功能。虽然识别正确的氨基酸的任务并不由 tRNA 本身完成，但是它们的三维结构对于能够识别它们的特定的酶来说则是至关重要的。每一种酶同时识别一个特定的氨基酸和相应的 tRNA，并进一步将这个氨基酸附着在这个 tRNA 上。更进一步，每个 tRNA 还含有一个与相应的密码子互补的核苷酸序列（反密码子），正是这样，正确的氨基酸载体与 mRNA 的配对才得以完成。

用来将 mRNA 的三联体密码子翻译成氨基酸的规则集就是遗传代码。图 1.6 所示的是 64 个密码子及其所编码的氨基酸或指令。氨基酸用三个字母的缩写来表示，并用一个字母的缩写来描述蛋白质中的主要结构。（本章的所有图都将使用这种一个字母的缩写来表示蛋白质链。）

mRNA 的消息被读出来，每次只读出一个密码子，每个密码子通过反密码子与相应的 tRNA 正确配对，这些被转运的氨基酸一个接一个地连接起来，形成一个长的线性蛋白质链，这个链的序列正好反映基因的序列（如图 1.7）。值得注意的是，核糖体——蛋白质合成过程中的关键粒子——是由很多蛋白质和其它种类的 RNA 分子，即核糖体分子构成的巨大的大分子结构。与 tRNA 类似，rRNA 也有唯一的三维结构。所以 rRNA 在核糖体这个大的机器中无数的化学反应中起到真正的酶的作用。

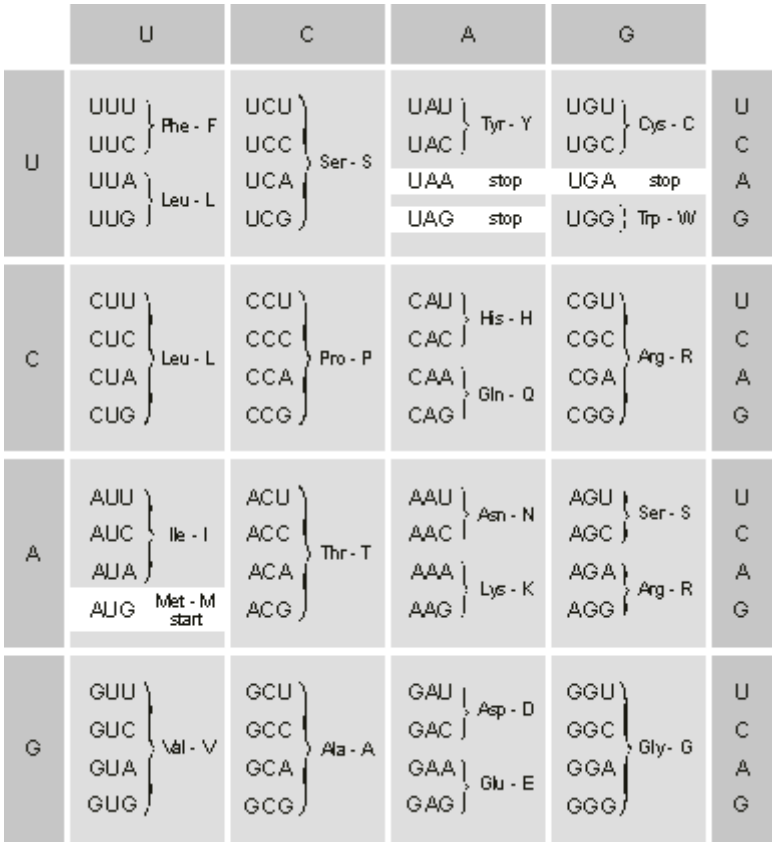


图 1.6.以 mRNA 形式表达的遗传代码。64 个密码子中有 3 个是终止信号。开始密码子也对 metionine 进行编码。注意该编码是冗余的，因为有许多密码子对相同的氨基酸进行编码。每个密码子所对应的氨基酸由三个字符和蛋白质中通常用于描述氨基酸序列的一个字符的缩写构成。

图 1.7.一个简化的翻译过程，仅显示必须的成分：mRNA 模板，开始和终止密码子，一个变化的 tRNA 和紧急蛋白质链。为了描述开始信号和终止信号，这里所给出的序列比实际序列要短很多。

然而对我们来说幸运的是，翻译过程中的错综复杂的化学过程对于像 GEP 这样的计算机系统来说意义有限。在这样的一个计算机系统中，翻译的规则（从更广义的角度来说）只需要简单地定义，简单地应用：我们不需要处理太多的化学问题。是的，我们既不需要处理复杂的中间转录过程，也不需要处理复杂的翻译机制和复杂的遗传代码。GEP 的遗传代码仅仅是一个基因组符号与其所代表的函数以及变量（在进化计算的术语中也叫做终端节点或者叶子）之间的一一对应关系。

1.2.4.2.翻译后修饰

当翻译过程到达一个停止信号的时候，一个非功能蛋白质链就被释放。这些蛋白质链一旦被释放，马上要接受一系列的修饰。其中第一个就是所有蛋白质中十分普遍的所谓翻译后修饰操作，这种修饰同时也构成了其唯一的三维结构中的蛋白质链的堆叠。某些蛋白质还要进一步接受其它一些修饰，比如对某些氨基酸进行的化学修饰将使蛋白质语言丰富许多，影响某些特定氨基酸之间的共价键的形成，以及能够通过某些片断的移除操作使链长缩短。最后，某些堆叠的蛋白质链（子单元）必须和其它一些子单元合并形成一个多子单元蛋白质。

这些多子单元蛋白质包括细胞中许多最重要的酶和转移蛋白。我们称这些蛋白质含有一种四元结构，也就是最高级别的蛋白质组织结构。

这里需要着重理解的是，每个层次的蛋白质都是建立在较低层次的蛋白质的基础上，一切均由蛋白质的基本结构控制，归根结底是被基因所控制。另一个重要的事情是，当一个蛋白质自己堆叠的时候，从很长的线性链上分离出来的氨基酸有可能被一起带到最终的三维结构中去，相反的事情也可能发生，蛋白质链中临近的氨基酸也可能面对完全不同的另一半，参与到蛋白质的完全不同的结构和功能当中去。最后一个与蛋白质结构和功能相关的重要事实是某个特定的蛋白质的功能由其唯一的三维结构决定，即最终由基因序列决定。但是要注意，一个蛋白质与它的 DNA 序列之间的差距有多么大！这就是简单转化的威力和突创论的美之所在。

GEP 的表达式树也是简单转化的产物，其结果也同样势不可挡。我们会看到，在 GEP 中，表达式树也通过某种特殊的方式堆叠，将基因中相距遥远的元素拉近，也将相距很近的元素分开。我们还将看到，有些表达式树也有四元结构，它们由许多较小的子单元（子表达式树）构成，这些较小的子单元通过各种翻译后的相互作用连接起来。

1.3. 适应性和进化

我们在生物界看到的许多多样性都是蛋白质变异积累（从广义上讲）的结果。如果我们拿任何蛋白质，例如血色素来作例子，并分析某个种群中个体的序列，我们会发现，其中存在各种各样的蛋白质变体，它们之间的差别仅仅在于一个或几个氨基酸。这些蛋白质变体中的大部分都具有相同的功效，但是它们之间也表现出一些功能上的细微差别。在某些环境下，这些变体具有更好的适应性，有一些可能还在个体的表达方面具有一定的优势。如果我们进一步研究并分析不同物种的血色素分子，我们会发现其血色素之间存在很大的不同。虽然这些不同的血色素功能完全相同，但是这些血色素似乎能够很好地适应某些特定物种的自然环境。的确，蛋白质中出现在分子级别的修饰操作能够使生物体的种群产生新的能力，从而更好地适应新环境，并最终成为新的物种。

种群为了能够在很长时间的进化过程中逐渐适应，生物个体必须通过选择进行复制。从进化学的角度来说，某个特定生物体的存活只有在它能够留下后代的情况下才会显得重要。只有该个体的后代才能够表现出某些新的特性并更好地适应自然环境。一个生物越能适应环境，其被选择并留下后代的可能性越大。我们在自然界中的生物体里发现的变化或者遗传多样性实际上是生物体用于选择的原始材料，这些生物体通过利用与其它物种相比的各种优势来保证其存在。越成功的个体留下的后代越多，这使得这些适应性较好的个体在种群中出现的频率增加（判断生物的适应性好坏的标准在于它们是否能存活下来），并随着时间的变化改变其自身的特性。但是在自然界，适应的进程从未停止过，这是因为生物体不断改变选择所出现的同一个环境，而且产生的个体不一定都能存活。

虽然“ 适应度 ” 一词在进化计算中被广泛采用，但是其意义与今天它在进化论中的意义并不相同。进化计算中适应度的含义和达尔文时代的含义一样：生物体被选择所偏爱的一种性质。实际上，遗传算法中的所有个体都是根据其适应度被选中的。然而在进化论中，适应度是结合了生存和繁殖的成功两方面因素的一个衡量尺度。

这说明适应性计算机系统与自然系统之间存在一个重要的不同之处。在自然界，生物的选择基于很多的因素，一个新的特性为什么被选中以及如何被选中的原因并不十分明确。然而个体的适应度只能通过它留下的后代的个数多少来衡量。但是在计算机系统中，某个环境下的个体的适应度很容易进行评价，而这种评价标准可以用来严格地决定选择过程。然而有些科学家喜欢引入随机因素来模拟自然界中的选择过程，这类选择最简单的一个实现方法

就是赌盘轮采样策略（Goldberg 1989）。每个个体用圆形赌盘上的一块来代表其适应度的比例。赌盘旋转的时候，较大的块被选中的概率较大。因为这完全是一种非确定性的现象，所以有时候有些不太可能的事情会发生而很可能发生的事情却没有发生。但是我偏向于使用这种选择办法，因为它对自然界的模拟更加真实，而且对于所有的种群来说都很适用（不同的选择策略将在第 7 章讨论）。的确，这种选择方法以及上一代最优个体的克隆操作（简单精英策略）在整个适应度图中具有非常高的搜索效率。

最后，另外一个自然系统和计算机系统的重要不同之处在于，在计算机系统中有可能严格地衡量适应度，因为我们知道我们所面临的对象是什么以及我们想要的是什么，因此只有那些或多或少地适合完成这种预定工作的个体才会被选中。因此，仔细研究手头的工作，选择在什么样的情况下产生个体，选择个体就显得尤为重要，因为我们一旦开始，就要得到我们想要得到的东西。

1.4. 遗传算法

J.Holland 在 20 世纪 60 年代发明的遗传算法将生物进化的理论运用到了计算机系统中（Holland 1975）。如所有进化的计算机系统一样，GAs 也是生物进化的一种简化。在其中，问题的解被编码成线性字符串（通常是 0 和 1），一个由备选解构成的种群通过不断进化来发现当前问题的一个最优解。种群，也就是解，因为解个体在修饰之后不断繁殖所以能够进化。如我们所看到的，这是进化发生的前提条件。在最初的 GA 中，修饰是通过变异，杂交和倒置引入的。另外，为了使进化得以发生，个体必须经过选择的筛选。算法根据个体的适应度进行选择，个体的适应度按照一定的方式严格计算，个体的适应度决定了它们繁殖的比例。个体的适应度越高，它们留下较多后代的可能性越大。

遗传算法只使用一种实体：染色体。因此，遗传算法的染色体就是简单的复制体。这些染色体由固定长度的线性符号串构成，代表当前问题的一个可能解。对每个问题来说，为了评价每个染色体的适应度，我们必须生成并严格地定义一个表示方法。实际上，在实现一个遗传算法的过程中，最困难的方面就是产生这样一个表示方法，但是这个表示方法一旦确定，那么这个问题就可以用一个符号染色体来编码。这些染色体（备选解）构成的初始群体是随机产生的，这个种群经过若干代的进化直到发现一个问题的最优解。

像所有的简单复制体一样，GAs 的染色体同时起基因型和表现型的作用：它们既是选择的对象也是遗传信息的保护者，这些遗传信息经过修饰后复制并传输到下一代中。基因组中发生的任何变化都将最终影响到其适应度和选择。为了更清楚地说明 GAs 的这一重要特征，我们将 GAs 中的这种情况与自然界中的这种现状相比，在自然界中个体选择只取决于它们本身的特性，我们会发现：只有个体本身和它所具有的能力对选择过程来说是重要的；与基因组的状态无关。

GAs 染色体所具有的功能多样性存在极大的限制。这种限制主要是由染色体所担当的两项功能（基因型和表现型）和它们的组织结构，特别是染色体的简单语言和固定长度造成的。这和简单的 RNA 复制体非常类似，在这个复制体中，线性的 RNA 基因组也能够表现出有限的结构多样性和功能多样性。在这两种情况中，复制体的整个结构决定其功能，并相应地决定个体的适应度。比如，在这样的一个系统中不太可能仅仅使用复制体的某个特定区间作为一个问题的解：总是整个复制体，而且不多不少恰好是整个复制体，才是问题的解。

1.5.遗传程序设计

遗传程序设计，最初是 Cramer 在 1985 年发明的，后来 Koza 进一步发展了它。GP 通

过产生大小和形状不同的非线性的实体来解决那些解的长度固定的问题。用来产生这些实体的字符集也更多样化，这同时也导致其表达方式成为一个更丰富、功能更强的系统。然而 GP 的个体缺乏一个简单的、自治的基因组，同时起基因型和表现型的作用。因此，这些个体也是简单的复制体。

这些 GP 的非线性实体（分列树）与蛋白质分子在使用的字符集的丰富程度和它们复杂的、唯一的层次结构的表达方式方面非常相似。因此，这些分列树也由很多的功能。但是这些实体难于复制，因为它们十分巨大，需要大量的空间。但是最重要的是，因为其中的基因修饰操作是直接在其分列树上完成的，因此其基因修饰操作受到很大的限制。比如，在大部分情况下，简单高效的点变异操作就不能使用，因为它经常会产生结构上不可能的情况。作为比较，值得强调一下的是，在自然界中任何蛋白质基因总是产生合法的蛋白质结构。

虽然没有染色体，但是 GP 仍然是一种遗传算法，因为它采用个体种群，根据个体的适应度进行选择，采用遗传算子引入遗传变化。因此，GP 与 GAs 之间的根本区别在于个体的本性不同，以及它们通过修饰操作进行复制并允许进化的方法不同。

GP 的个体通常在 LISP 语言中被表示成分列树的形式（如图 1.8），虽然这种形式咋看起来很有优势，但是这样会对该技术造成很大限制（就像不可能仅仅通过嫁接和修剪就让柑桔树长出芒果一样）。其遗传算子在应用的时候必须非常小心才能形成合法的结构。比如，在杂交这个 GP 中最常用也经常是唯一被使用的遗传算子中，被选中的子树在两个父代树之间互换产生新的后代（如图 1.9），这一实现过程后的基本思想就是要通过互换较小的、数学上较为简明的程序块来进化得到更加复杂的、由较小的基因块构成的层次结构的解。GP 的杂交和树的剪枝及嫁接非常相似，而且一样存在很大的功能限制。使用这样的杂交操作是因为在 LISP 中这种操作很容易实现而且所产生的分列树总是合法的 LISP 程序。

GP 中的变异算子也和生物学上的点变异操作不一样，这样做是为了保证产生的 LISP 程序在句法上的正确性。变异操作在分列树上选择一个节点，并用一个随机产生的子树来替换该节点上的子树（图 1.10）。这样一来，这种变异并没有对整个树的结构作很大的改变，

a. `(-(*(-a(sqrt(*ba)))(/(/(-aa)(+bb))(sqrt(+ab))))(+ba))`

b.

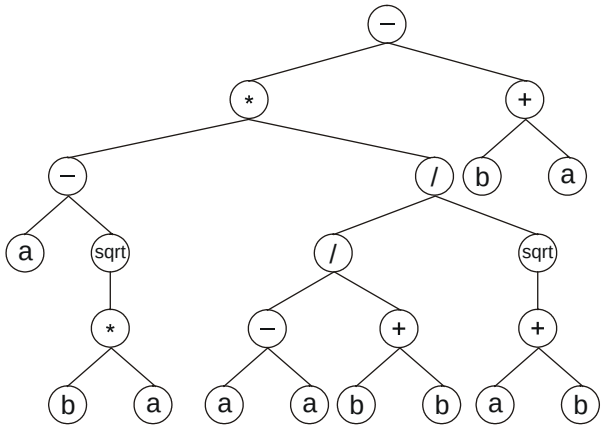


图 1.8.LISP 中的一个计算机程序(a)及其树型表达(b)。注意在 LISP 中操作符在参数之前，如*(a+b)*写成*(+ab)*。

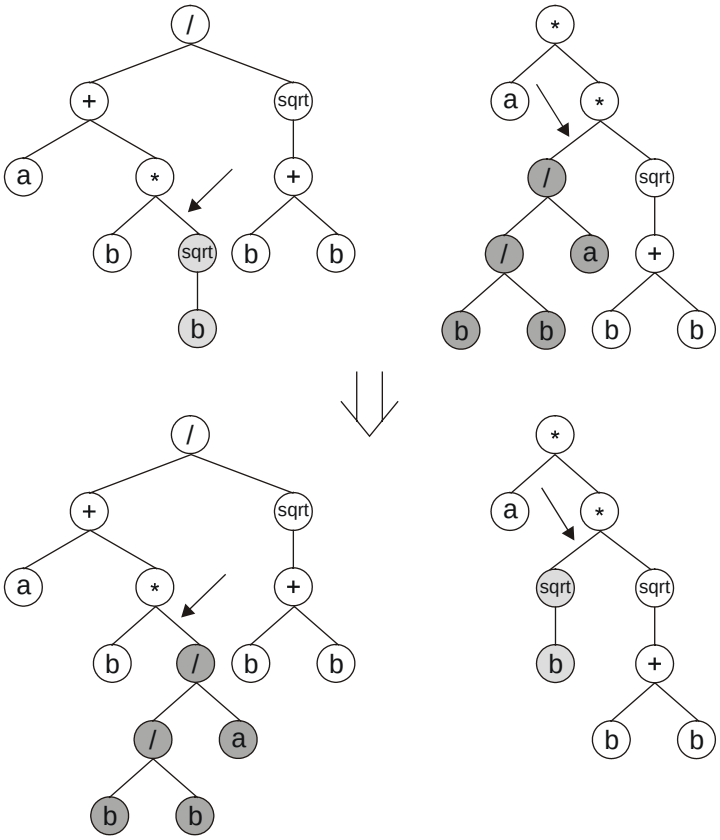


图 1.9.遗传程序设计中的树杂交。箭头表示杂交点。

特别是当比较可能被选为变异目标的点处在树中较深层次的节点的时候，更是如此。

置换是 GP 中使用的第三种算子，也是这三种算子中最保守的一种，在置换的过程中，某个随机选定的函数的参数随机置换，在这种情况下树的整体结构保持不变。

虽然 J.Koza 将这三种算子描述成基本 GP 算子，但是实际上针对树的杂交算子是绝大部分 GP 应用中所使用的唯一遗传算子。由此，实际上 GP 种群的基因池中并没有引入任何新的元素。所以必须使用数量巨大的分列树以使所有必需的基因块都已经存在于初始种群中，只有这样才能保证仅仅通过将最初产生的那些基因块来回移动去发现一个好的解。

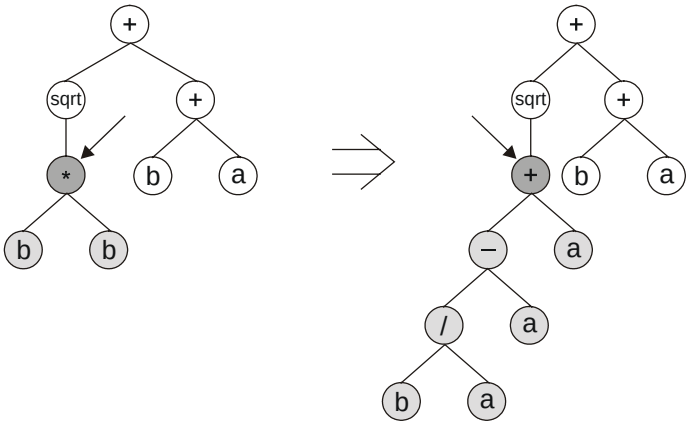


图 1.10.遗传程序设计中的树变异。箭头所指为变异点。注意变异算子在变异点产生一个随机生成的分支。

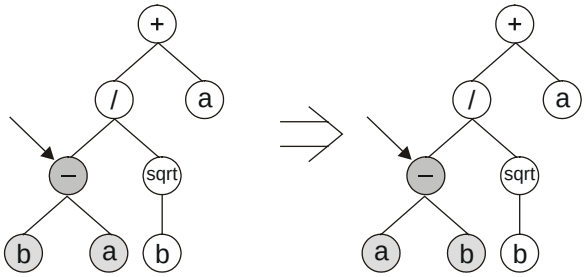


图 1.11.遗传程序设计中的置换。箭头所指为置换的位置。注意置换函数的参数在子树中变换了位置。置换后树的形状保持不变。

总之，在 GP 中，这些算子更像一个有意识的数学家，不太像自然界中的盲目选择。我们将会看到，在适应性系统中，盲目选择比 GP 这样高度限制性的系统更加有效。比如，在 GP 中实现其它一些算子，如某些与点变异一样高效的等价算子（Ferreira 2002a）的时候，就不是十分有效，因为大部分变异操作会产生句法上错误的结构（如图 1.12）。显然，其它一些算子，例如变换和倒置算子也会产生同样的困难，而且 GP 中的很大部分的搜索空间都没有被搜索到。

最后，由于分列树的双重作用（基因型和表现型），GP 不能采用简单的、基本的表达方式：在所有情况下，整个分列树就是解。

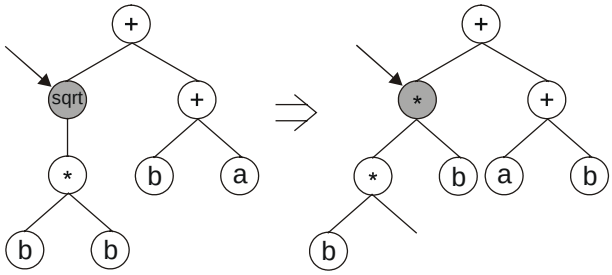


图 1.12. 遗传程序设计中的点变异的一种假设情形的图示。注意子树是一个非法结构。

1.6.基因表达式编程

基因表达式编程是我本人 1999 年发明的，它也是 GAs 和 GP 的一种必然的发展，GEP 结合了 GAs 的简单线性染色体的思想和 GP 中使用的大小和形状不同的分叉结构的思想。

GEP 采用与 GP 相同的表达方法，但是 GEP 中的进化实体（表达式树）是线性基因组的表达式。因此，采用 GEP，我们跨越了第二个进化极限，即表现型极限，从而产生了进化计算中的一系列新的可能。

GEP 的关键在于其中发明的能够表达任何树结构的染色体。为此我发明了一种新的语言——*Karva* 语言——来读取和表达编码在 GEP 染色体中的信息。

进一步说，染色体的结构设计允许多基因的产生，每个基因对一个较小的程序或者子表达式树进行编码。值得强调的是，GEP 是唯一的多基因遗传算法。的确，用多个基因构成的复杂个体的产生大大简化了构建一个功能强大的基因型/表现型系统的过程。实际上，自从这些系统产生以来，它们好像把自己放在了更高级别的复杂度的快车道上，并且有数不清的新思想等待人们去发现。在这本书里面，我们将遇到这种更复杂的实体，例如，不同细胞表达不同基因构成的多细胞个体或者适应性基因型/表现型人工神经网络等。

这些新奇事物的根本在于 GEP 基因革命性的结构。这种基因简单但富有弹性的结构不仅可以对任何想象得出的程序进行编码，也可以允许它们高效地进化。由于它们的这种组织

结构，可以很容易地实现一组高效的遗传算子并在搜索空间进行有效的搜索。像在自然界中一样，GEP 的遗传算子总是产生合法的结构，因而很适合产生遗传多样性。

下一章我们将学习 GEP 染色体的结构和功能组织；如何将染色体的语言翻译成表达式树的语言，染色体如何起基因型的作用，表达式树如何起表现型的作用；程序个体如何产生，成熟，繁殖，并产生具有新的特性，能够进化的后代。

基因表达式编程的实体 2

GEP 的基因表达式和它所类比的细胞基因表达式比起来要相对简单得多。在 GEP 中有两种东西起主要作用：染色体和表达式树，后者由编码在前者中的遗传信息的表达式构成。信息解码的过程（从染色体到表达式树）称为翻译。这种翻译明显地暗示了一种编码方法和一套编码规则。它的遗传代码十分简单：就是染色体符号与其所表示的函数或者终点之间的一一对应关系。编码规则也很简单：它们决定了表达式树中函数和终点的空间组织结构和多基因系统中子表达式树相互作用的类型。

因此，在 GEP 中有两种语言：基因语言和表达式树语言，我们将看到这两种语言中的任何一种的序列或者结构都足以推断出另外一种语言。在自然界中，虽然根据给定的基因序列可以推断出蛋白质的序列，而且反之亦然，但是对于决定蛋白质堆叠的规则我们还是知之甚少。而且在蛋白质堆叠以前蛋白质基因的表达式是不完整的。也就是说，只有当蛋白质正确地堆叠到它们本来的三维结构中去的时候，氨基酸链才会变成蛋白质。关于蛋白质堆叠我们唯一确切知道的是氨基酸的序列决定了堆叠过程。然而，协调这一堆叠过程的规则仍然不为人知。但是在 GEP 中，由于决定表达式树结构和它们之间相互作用的规则非常简单，使得我们有可能立即根据给定基因的序列推断出表现型（与堆叠后的蛋白质分子对应的最终结构），反之亦然。这种明确的双语系统称为 *Karva* 语言。本章将介绍该语言的细节。

2.1.基因组

在 GEP 中，基因组或者染色体由一个或多个基因组成的固定长度的线性符号串构成。虽然它们的长度固定以外，但是我们还将看到，GEP 染色体可以对长度和形状不同的表达式树进编码。

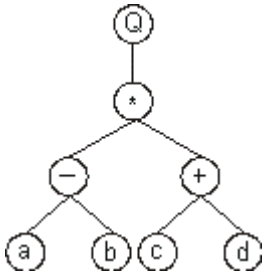
2.1.1. 开放阅读框和基因

GEP 的结构组织从开放阅读框（Open Reading Frames，ORFs）的角度来说更容易理解。在生物学上，一个 ORF 序列，或者基因编码的序列，开始于一个“起始”密码子，接着是一些氨基酸密码子，最后止于一个“终止”密码子。然而一个基因不仅仅包含各自的 ORF，还包含起始密码子之前的序列和终止密码子之后的序列。虽然在 GEP 中起始点通常都是某个基因的第一个基因位，然而终点并不一定与基因的最后一个基因位重合。对于 GEP 来说，在终点以后有未编码区域是十分常见的。（这里我们暂时不考虑这些未编码区域，因为它们并不影响表达式的产生。）

比如，考虑如下代数表达式：

$$\sqrt{(a-b)\times(c-d)} \tag{2.1}$$

它可以表示成如下所示的图或者表达式树：



这里“ Q ”表示开方函数。

这种图实际上代表了 GEP 个体的表现型，根据这些表现型很容易推断出如下基因型：

```
01234567
Q*-+abcd
```
(2.2)

该基因型是直接将表达式树从左到右，自上而下读出的结果（就像我们读普通文本一样）。表达式(2.2)就是一个 ORF，从“ Q ”（第 0 位）开始，止于“ d ”（第 7 位）。我将这些 ORF 称为 K-表达式（源自 *Karva* 语言）。

注意，这种符号表示法不同于在各种有数组或栈的 GP 实现中所使用的后缀表达式或前缀表达式（Keith 和 Martin 1994）。图 2.1 对 *Karva* 表示法和前缀表达式和后缀表达式进行了比较。

K-expression:

```
01234567
Q*-+abcd
```

Postfix:	**Prefix:**
01234567	01234567
ab-cd+*Q	Q*+dc-ba

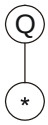
图 2.2.Karva 表示法与后缀及前缀表示法的比较。在各种情况下，表达式(2.1)都得以表示。

考虑另外一个 ORF，如下的 K-表达式：

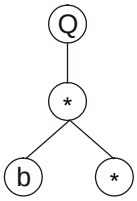
```
01234567890
Q*b**+baQba
```
(2.3)

该表达式作为一个表达式树也十分简单而且直接。对于其完整表达式，必须遵循其函数和终点的空间分布的控制规则。首先，基因的起点对应表达式树的根（虽然这种根在树的顶端），使这个节点构成表达式树的第一行。第二，根据每个元素的参数个数（不同函数的参数个数可能不等，但是终点的操作数为 0），在下一行放置与该元素的参数个数相当的节点。第三，将这些节点从左到右逐个按照基因中元素的顺序进行填充。反复执行该过程，直到某一行仅含有终点。所以，对于上面的 K-表达式，第 0 位上的符号就是表达式树的根：

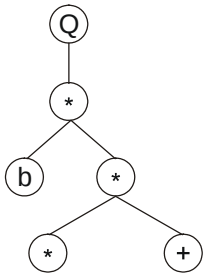
开方函数只有一个参数，所以下一行只有一个节点，用第 1 位的符号来填充：



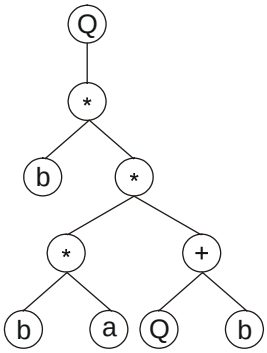
乘法函数有两个参数，因此下一行放置两个节点，这两个节点用第 2 位和第 3 位的符号来填充，得到：



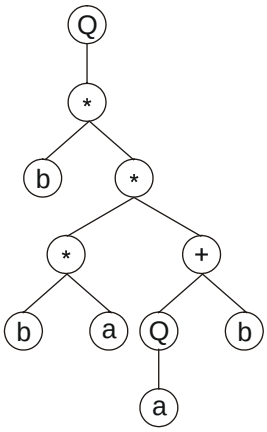
这一行有一个叶节点和一个表示含有两个参数的“ 芽节点 ”。所以下一行需要两个节点。这里用第 4 位和第 5 位的元素来填充，得到：



现在最后一行有两个函数，每个函数都有两个参数，（复印的书上面这个地方不清晰） 这里分别用第 6，7，8，9 位的元素来填充，得到：



在新一行中，虽然有 4 个节点，但是只有一个节点是芽节点。这里，所要求的节点放置在函数的下面，并用 ORF 中的下一个（第 10 位）元素来填充：



这一步完成后，这个 K-表达式的表达就算完成，因为最后一行只含有终点构成的节点。我们将要看到，由于 GEP 基因的结构组织，导致所有的表达式树的最后一行都仅仅含有终点。因此也可以说通过 GEP 进化得到的所有程序在语法上都是正确的。

如果仅仅看这些 GEP 中的 ORF，我们除了看到其简洁和优雅之外，很难甚至根本就不可能发现这种表达方式有什么优点。然而，当我们在基因中分析这些 ORF 时，这种表达方式的优点就会显而易见。如我前面提到的那样，GEP 的染色体具有固定的长度，而且这些染色体由一个或多个相同长度的基因构成。而且每个基因的长度也是固定的。因此，在 GEP 中，基因的长度是固定不变的，变化的是 ORF 的长度。当然，ORF 的长度可以和基因的长度相同，也可以比基因的长度短。在前面一种情况下，终点就与基因的尾部重合，而在后一种情况下，终点在基因的尾部之前。

那么，GEP 基因中非编码区域有什么作用呢？实际上，它们是 GEP 中不可缺少的部分，也是其进化能力的精华所在，因为它们允许遗传算子对基因组进行不受限制的修饰，产生句法上始终正确的程序。因此，在 GEP 中，基因型/表现型系统的根本特性——句法封闭性——

是与生俱来的，允许对基因型进行完全不受限制的操作，并由此促成高效的进化。的确，这是 GEP 与以前的 GP 实现之间极其重要的区别，即是否使用线性基因组（请参考 Banzhaf 等 1998 中线性基因型 GP 的评论文章）。

为了理解 GEP 的基因如何永恒不变地编码出语法正确的程序，以及它们如何允许对几乎任何遗传算子无限制地应用，让我们现在来分析一下 GEP 基因的结构组织。

2.1.2.基因的结构和功能组织

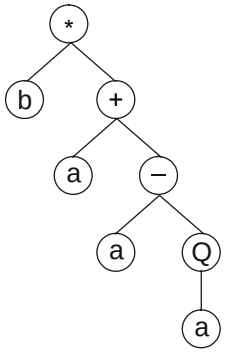
GEP 的基因由一头和一尾两部分构成。头部由包含既代表函数又代表终点的符号构成，而尾部仅仅含有终点。对每个问题而言，头的长度 h 是选定的，而尾的长度 t 是 h 和 n 的函数，其中 n 是所需变量数最多的函数的参数个数（也称为最大操作数），t 的大小由下面的方程得到：

$$t = h(n-1)+1 \tag{2.4}$$

考虑如下这个由函数集为 $F=\{Q,*,/,-,+\}$ 和终点集为 $T=\{a,b\}$ 构成的基因。在这里，$n=2$；如果我们选择 $h=15$，那么 $t=15\times(2-1)+1=16$；所以基因 g 的长度为 $15+16=31$。下面给出一个这样的基因（尾部用粗体标识）：

```
012345678901234567890123456789O
*b+a-aQab+//+b+babbabbbababbaaa
```
(2.5)

它编码成如下表达式树：

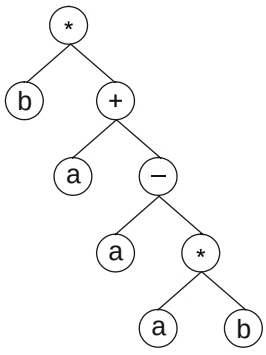


需要注意的是，ORF 终止于第 7 位，但是基因终止于第 30 位。

假设现在第 6 位出现一个变异，将“ Q ”变成“ * ”，那么得到如下的基因：

```
012345678901234567890123456789O
*b+a-a*ab+//+b+babbabbbababbaaa
```
(2.6)

和如下的表达式：

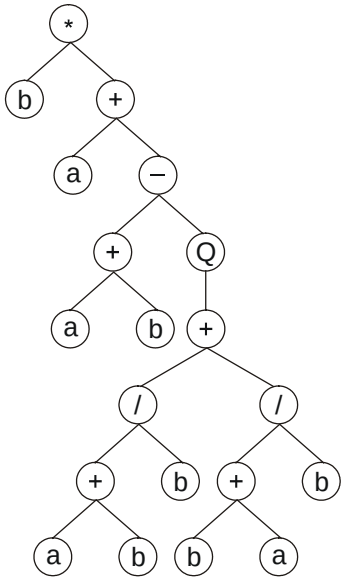


在此，ORF 的终止点向右移动了 1 位（到第 8 位）。

考虑上面的染色体（2.5）出现另一个变异的情况，用“ ＋ ”来替换第 5 位上的“ a ”。得到如下的染色体：

```
                    01234567890123456789012345678 90
                    *b+a-+Qab+//+b+babbabbbababbaaa
```
(2.7)

它给出的表达式树是：

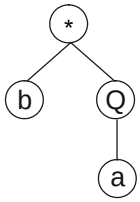


在此，ORF 的终止点向右移动了 12 位（到第 19 位）。

显然，相反的情况即 ORF 被缩短的情况也可能出现。比如，考虑上面的基因（2.5），假设在第 2 位发生变异，由“ ＋ ”变为“ Q ”，得到：

```
                    01234567890123456789012345678 90
                    *bQa-aQab+//+b+babbabbbababbaaa
```
(2.8)

其表达式产生如下的表达式树：



这里，ORF 止于第 3 位，使原表达式树减少 4 个节点。

虽然基因的长度已定，但是它们还是具有转换成不同大小和形状的表达式树的潜力，最简单的情况是整个表达式树仅仅由一个节点构成(当基因的第一个元素是终点时)，最复杂的情况是表达式树由基因中的所有节点构成（当头中所有的元素都是函数，而且函数都具有最大参数个数 n）。

从以上的例子很容易看出，不论对基因组进行多么深刻的修饰，产生的表达式树始终是正确的。显然，基因的组织结构必须保留，即始终保持头部和尾部的边界而且禁止表示函数的符号出现在尾部里面。在下一章（3.3 节）我们将继续讨论这些问题，主要包括详细分析不同遗传算子的机制和效果。

2.1.3.多基因染色体

在自然界中，通常染色体所编码的基因不止一个，因为复杂的个体需要复杂的基因组。的确，更复杂的个体只有通过产生多基因的基因组才可能进化。毫不奇怪，GEP 也开发了多基因系统的优势。

GEP 染色体通常由等长的多个基因构成。对于每个问题，基因的数目以及头的长度都

是根据情况来选定。每一个基因可以对一个子表达式树进行编码而且子表达式树相互作用构成一个更复杂的实体。这些相互作用的细节将在 2.2 节进一步讨论。我们现在仅仅讨论如何由基因构成子表达式树的问题。

例如，考虑如下长度为 39，由 3 个基因长度均为 g=13 的基因构成的染色体（尾部由黑体标识）：

```
012345678901201234567890120123456789012
*Qb+*/bbbabab-a+QbQbbababa/ba-/*bbaaaaa
```
(2.9)

该染色体含有 3 个ORF，每个ORF对一个子表达式树进行编码（如图 2.2）。第 0 位标示每个基因的开始；每个ORF的结束对于对应的子表达式树的结构来说是显然的。如图 2.2 所示，第一个ORF止于第 9 位（sub-ET_1），第二个ORF止于第 6 位（sub-ET_2），最后一个ORF止于第 2 位（sub-ET_3）。

简言之，GEP 染色体含有多个 ORF，每个 ORF 对一个结构和功能唯一的子表达式树进行编码。我们会看到，根据手头的任务，这些子表达式树可以根据它们各自的适应度进行选择（比如在多个输出的问题中），或者它们可以形成一个更加复杂的，多级表达式树，而这些多级表达式树作为一个整体根据其适应度进行选择。这种表达方式和选择的细节将在这本书中经常讨论。但是，请记住，每个子表达式书既是一个独立的实体，也是一个复杂的层次结构系统的一部分，而且，像所有的复杂系统一样，整体大于部分之和。

a.
```
012345678901201234567890120123456789012
*Qb+*/bbbabab-a+QbQbbababa/ba-/*bbaaaaa
```

b.

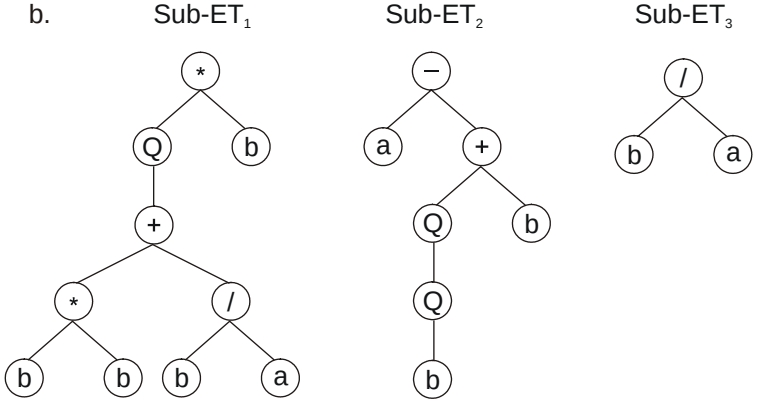


图 2.2.GEP 基因的表达式作为子表达式树。**a)**一个 3-基因染色体，其中尾部用黑体标识。0 号位表示基因的开始。**b)**每个基因所编码的子表达式树。

2.1.4.染色体的结构多样性和功能多样性

到现在为止，我们所讨论的染色体，也就是 GEP 中最常见的染色体，对于解决很多类的问题来讲都是十分理想的。但是，还存在其它一些为解决不同类型的问题而特别设计的，具有不同结构和功能组织的染色体。这一节我将给出一些这样的染色体结构以显示如何将 GEP 染色体简单地修饰一下就能通过进化解决各种不同的问题。

因此，最简单的染色体仅仅对一个基因进行编码，由一个终点构成，比如：

a (2.10)

的确，当 $h=0$ 和 $n=0$ 的时候就会得到这样的基因。尾部的长度很容易由方程（2.4）计算得到，从而得到 $t=0$，因此，$g=1$。虽然单基因染色体没有什么用处，但是这种仅由一个元素构成的简单基因甚至 ORF 在结合成更复杂的染色体的时候却是十分重要的。

实际上，单个元素的表达式树不仅能作为单个元素的基因的表达式还可以在头部长度大于 0 的更大的基因的表达式中出现。比如，在如下的 3-基因染色体中，2 号基因就对一个单元素的 ORF 进行编码（尾部由黑体标识）：

```
012345678901201234567890120123456789012
Q+aaa/babbabaa+*b+-bababbb-***aabaaaaba
```
(2.11)

该染色体中的 3 个子表达式树如图 2.3 所示。

a.
```
012345678901201234567890120123456789012
Q+aaa/babbabaa+*b+-bababbb-***aabaaaaba
```

b.

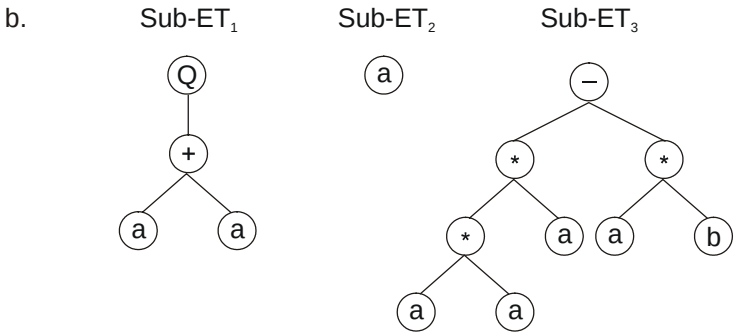


图 2.3.对单元素子表达式树进行编码的 3-基因染色体(sub-ET_2)。**a)**3-基因的染色体，其中尾部用黑体标识。**b)**每个基因所编码的子表达式树。

尽管如此，单个元素的基因在多基因系统中也是非常有用的，因为它们可以构成多基因族（以下简称 MGF）。这些 MGF 由对一类特殊的变量进行编码的相关基因簇构成。因此，在 MGF 中，每个基因对某个特定的终点或任务进行编码。我们将看到，这种由 MGF 构成的染色体在进化得到调度问题的解时很有用（见第 6 章）。例如，旅行商问题中不同的城市可以编码在一个多基因族里面，每个对一个城市进行编码。考虑以下由 9 个成员构成的 MGF 所表示的染色体：

```
012345678
GCDAHEIFB
```
(2.12)

其中每个元素代表一个城市。在此，表达式由子表达式树的空间组织，即它相互作用的路径顺序构成（如图 2.4）。我们在第 6 章还将看到这种染色体的表达式还可以用来表示旅行商问题的一条巡回路径。

a.
```
012345678
GCDAHEIFB
```

b.

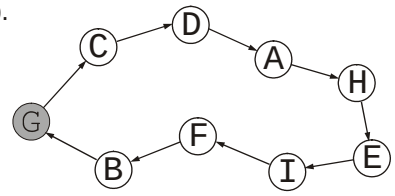


图 2.4.单元素基因表达式作为子表达式树的空间组织结构。如果每个符号代表某个商人游历的城市，染色体**(a)**的完整表达式将与**(b)**中 9 个城市的路径对应。起始和终止城市用灰色标识。

然而，对于更加复杂的调度问题，可以使用一个以上的 MGF，对于这些问题，染色体含有两个或者更多的 MGF。例如，下面这个染色体就是为了解决一个简单的安排问题而涉及的（多基因族用不同的阴影表示）：

```
012345012345
453621DABFEC
```
(2.13)

在第 6 章我们还将看到这些染色体是如何表达的，以及它们所能表达的相互作用的类型。虽然本书中涉及的多基因族的例子都是非常简单的，但是这些例子说明 MGF 在解决更大范围的调度问题时是非常有用的。

单元素基因在解决其它一些问题时也是十分重要的。在这些问题中，这些简单基因的产物在完成翻译后的修饰之后由某个被选中的函数相互连接起来。例如，复杂的布尔函数可以通过这种方式来建模。下面这个例子就是一个由 9 个单元素基因构成的染色体，其中子表达式树由布尔函数 $IF(x, y, z)$ 三个三个地连接起来（如果 $x = 1$ 则返回 y ，否则返回 z ）：

```
012345678
bbcaedcff
```
(2.14)

符号 $\{a, b, c, d, e, f\}$ 分别表示布尔函数的 6 个参数。其表达式见 2.5。

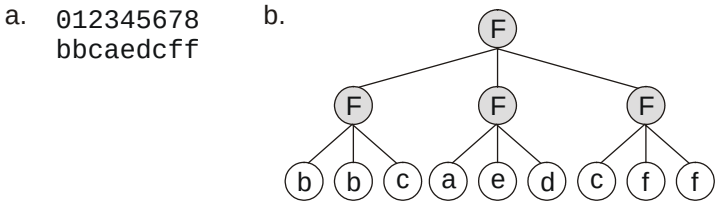


图 2.5.单元素基因作为布尔函数簇的表达式。**a)**一个由 9 个单元素基因构成的染色体。**b)**单元素子表达式树翻译后由 3-参数函数“ F ”连接。

以上这些例子中的染色体比 2.1.3 节描述的基本 GEP 染色体要简单。然而，在 GEP 中，还有一些染色体的组织结构比基本多基因染色体还要复杂。这些复杂的染色体由一些具有函数功能的簇构成，这些簇由传统的头/尾域加上一个或者多个附加域构成。这些附加域一般都用来对单元素的子表达式树进行编码。编码在这些不同域中的所有子表达式树相互作用，形成一个更复杂的实体。这样的染色体是为了解决采用数值常量的符号回归问题和复杂的神经网络的问题而设计的，在进化神经网络的时候，这种附加域用来对神经网络的权值和阈值进行编码（见第 5 章）。例如，以下染色体长为 34，由两个基因构成，这两个基因的头部长度均为 2，尾部长度均为 7，长度为 8 的附加域对神经网络的权值进行编码（不同的基因用不同的阴影标识）。

```
01234567890123456012345678901234 56
Tbababaab29418263DQbabaaba55157646
```
(2.15)

符号“ D ”、“ T ”、“ Q ”分别表示有两个参数的函数，有三个参数的函数和有四个函数的参数。每个基因中，对权值进行编码的域从第 9 位到第 16 位，数值符号表示相应的头/尾域中的神经网络的权值。

根据当前所要解决的问题的不同，还可以设计出其它的染色体来对更多的域进行编码。例如，如果我们还想让神经网络的阈值进化，还可以生成第二个域（不同的域用不同的阴影来标识）：

```
0123456790123456 78
QDabaabbb08100883 24
```
(2.16)

在这个单基因染色体中，第一个附加域从第 9 位到第 16 位，这个附加域用来对权值进行编码，第二个附加域从第 17 位到第 18 位，这个附加域用来对头/尾域中的神经网络的阈值进行编码。

总而言之，本节中所展示个各式各样的染色体说明 GEP 的基因型/表现型表示方法所具有的弹性，而且这些染色体色展示可以激发我们去设计各种不同的染色体结构来解决各种不同的问题。

2.2.表达式树和表现型

在自然界中，表现型有多种级别的复杂度，而其中最复杂的就是有机体本身。但是，tRNA，蛋白质，核糖体，细胞等等，也是表达式的产物，并且它们最终都被编码到基因组里中。

与自然界中的情况相反，在 GEP 中，遗传信息的表示十分简单。尽管我们在 2.1 节已经看到，GEP 的染色体可以有很多种不同的组织结构，而且编码在这些结构中的个体也显然具有不同程度的复杂度。最简单的个体编码在单个基因中，而这里，“有机体”是单个基因，即只含有一个子结构的表达式树的产物。在其它一些情况下，“有机体”是多级表达式树，这些各不相同的子表达式树通过一种特殊的连接函数或者相互作用方式相互连接。还有一些情况下，“有机体”是多级表达式树，其中各不相同的子表达式树的组合通过一种特殊的连接函数相互连接，用来表达一个特定细胞的发展的程序。在另外一些情形下，“有机体”凸现于不同的子表达式树的空间组织中。还有一些情况下，“有机体”从传统的子表达式树和编码在特殊域中的更小的子表达式树的相互作用间相互凸现出来，从而构成一个复杂网络之间的相互作用。然而，在所有的情形下，整个“有机体”都被编码成一个线性基因组。

本节我们将更加详细地讨论遗传信息表达的不同方面，考虑不同层次的表现型复杂度。

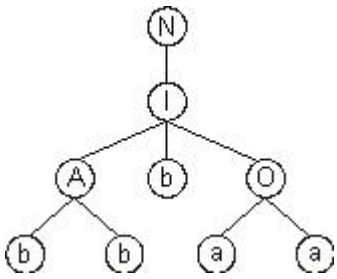
2.2.1.信息解码：翻译

不论是最简单的个体还是最复杂的个体，GEP 中遗传信息的表达都从翻译开始。2.1.1 节已经对翻译进行了简要介绍，而且我们还学习了如何将 K-表达式解码。同时我们也认识到每个染色体的微小变化都需要不同级别和不同方式的表达。但是，在所有的情况下，遗传信息表达的第一步都必须含有子表达式树的建立。

考虑如下的单基因染色体（尾部用黑体标识）：

```
0123456789012345
NIAbObbaaaabaabb                    (2.17)
```

符号 $\{A,O,N,I\}$ 分别表示布尔函数的 AND，OR，NOT 和 IF，其中前两个需要 2 个参数，NOT 只有 1 个参数，最后一个函数有 3 个参数。在此，翻译的产物是下面这个含有 9 个元素的表达式树：



对于这个简单的个体，单个的表达式树就是“有机体”，所以，遗传信息的表达止于其翻译过程。请注意这个特定的程序就是 NOR 函数的一个解。

当基因组含有的基因个数多于一个的时候，每个基因都被独立地翻译成一个子表达式树。例如，图 2.6 所示的三个子表达式树就可以由以下这个由 3 个基因构成的染色体编码（尾部用黑体标识）：

```
012345678901234567890123456789
AOaabaaaabNabaaaaaabINNbababaa      (2.18)
```

a. 012345678901234567890123456789
AOa**abaaaab**Nab**aaaaaab**INN**bababaa**

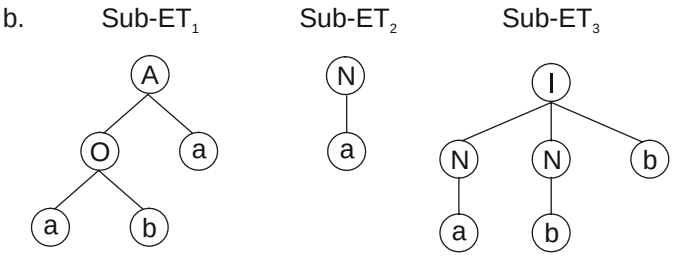


图 2.6.作为子表达式树的 GEP 基因的翻译。a)由 3 个单元素基因构成的染色体，尾部用黑体标识。b)每个基因所编码的子表达式树。注意子表达式树之间应该相互作用以使染色体得以完整表达。的确，编码在染色体中的程序只有当其相互作用得以确定的时候才有意义。例如，如果连接由 OR，AND 或 IF 完成，应该得到 3 个不同的程序。

这个个体的完整表达显然要求子表达式树之间存在某种相互作用。例如，如果我们用布尔函数 IF 来连接这些子表达式树，那么上面的染色体（2.18）就可以称为 NOR 函数的一个复杂的解。下一节将详细分析这些问题。

对于由单元素基因构成的染色体，如下面这个染色体（2.19）:

aab (2.19)

每个基因都被翻译成一个单元素的子表达式树（如图 2.7）。这样的染色体的完整表达也要求这些简单的子表达式树之间存在某种相互作用。例如，如果我们用 IF 来连接这些编码在染色体中的子表达式树，该个体就可以成为布尔函数 OR 的解。

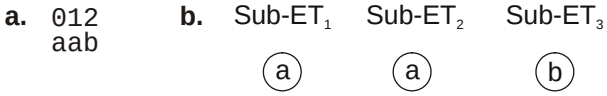


图 2.7.作为子表达式树的单元素基因的翻译。a)由 3 个单元素基因构成的 3-基因染色体。b)每个基因所编码的子表达式树。注意子表达式树之间应该相互作用以使染色体得以完整表达。的确，编码在染色体中的程序只有当其相互作用得以确定的时候才有意义。例如，如果连接由 OR，AND 或 IF 完成，应该得到 3 个不同的程序。

对于由多基因族构成的染色体，翻译的过程也很简单。例如，考虑下面这个由两个 MGF 构成的染色体（MGF 用不同的阴影标识）：

012345012345

453621DABFEC (2.20)

这里，每个单元素基因对一个单元素子表达式树进行编码（如图 2.8）。“ 有机体 ” 只有在子表达式树相互作用的时候才凸现出来。在第 6 章中当我们讨论到置换问题时，我们将回过头来讨论其中各种各样的相互作用。

a. **012345**012345
453621DABFEC

b.

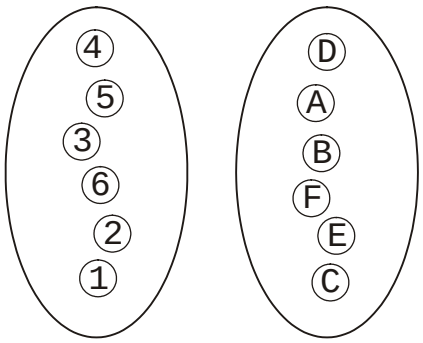


图 2.8.由多基因族构成的染色体的翻译。**a)**含有 2 个多基因族的染色体。**b)** 两个多基因族中每个成员所编码的子表达式树。每一族的成员放在一起。注意染色体的完整表达需要子表达式树间的相互作用。还要注意可能存在许多相互作用的方式。

2.2.2.翻译后的相互作用和连接函数

我们已经看到翻译可以导致不同复杂度的子表达式树的形成，但是遗传信息的完整表达需要这些子表达式树之间的相互作用。也只有这样，个体才能被完全表达。解决该问题的一个简单的办法就是将这些子表达式树用一个特殊的函数连接起来。这个过程与多级蛋白质中的不同蛋白质相互聚合的过程很类似。

当这些子表达式树是代数表达式或者布尔表达式的时候，任何参数多于一个的数学函数或者布尔函数都可以用来连接这些最后的多级结构的子表达式树。对于代数表达式树最常用的连接函数是加法和乘法函数，对布尔子表达式树最常用的是 OR，AND 或 IF 函数。

考虑如下对 3 个代数子表达式树编码的染色体（尾部用黑体标识）：

```
012345678901201234567890120123456789012
QaQ+-Qbbaaaba+Q+ab+abababa*-**b+aabbaba          (2.21)
```

用加法函数连接的子表达式树见图 2.9。注意，这个编码在染色体 2.21 中的多级子表达式树也可以被线性化编码成如下的 K-表达式树。

```
01234567890123456789012
++*Q+-*aQ+*b+aab+abbaab                          (2.22)
```

然而，对复杂问题的解采用多基因染色体来进化更恰当一些，因为它们允许构成更复杂的、层次结构的模块，在这些复杂的、层次结构的模块中每个基因对一个小的基因块进行编码（参见第 7 章的讨论）。这些小的基因块相互分离，但是可以在某种程度上独立进化。

考虑另一个染色体，这一次染色体对 3 个布尔子表达式树进行编码（尾部用黑体标识）：

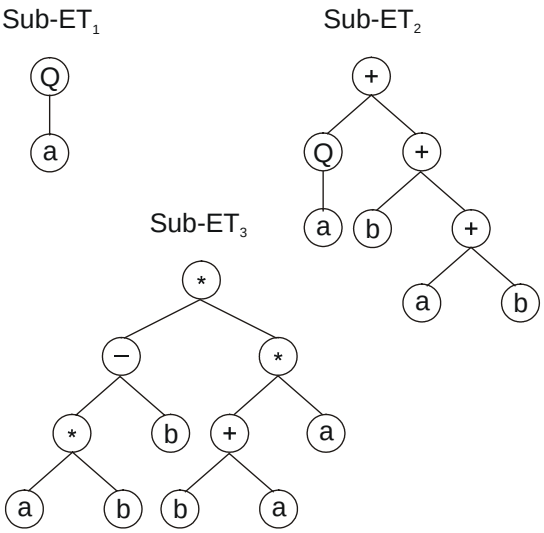
```
012345678901234501234567890123450123456789012345
IOaIAcbaaacaacacAOcaIccabcbccbacIONAAbbbbacbcbbc  (2.23)
```

用 IF 函数连接的子表达式树如图 2.10 所示。同样，染色体 2.23 所编码的多级子表达式树也可以被线性化地编码成如下的 K-表达式树。

```
012345678901234567890123456789O
IIAIOaIOcONAAcbaaaIAbbbbacccaac                  (2.24)
```

a. 012345678901201234567890120123456789012
QaQ+-Q**bbaaaba**+Q+ab+**abababa***-**b+**aabbaba**

b.



c.

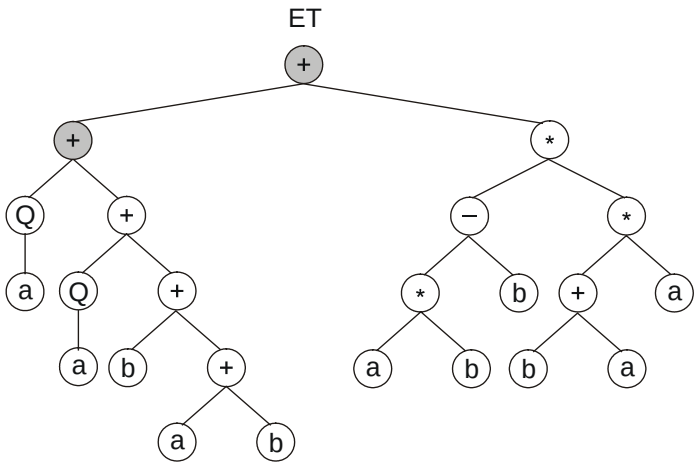
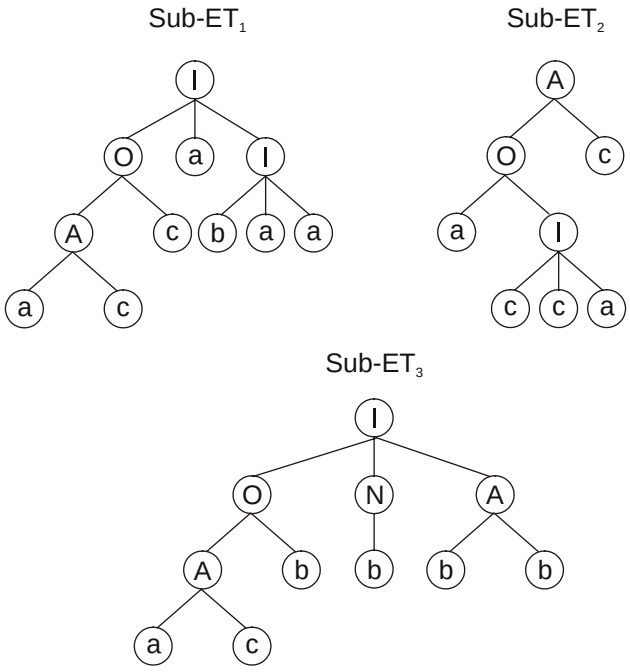


图 2.9.对作为多级结构的子表达式树的代数表达式进行编码的多基因染色体的表达。**a)** 一个 3-基因染色体，其中尾部用黑体标识。**b)**每个基因所编码的子表达式树。**c)**用加号连接得到的结果。连接函数用灰色标识。

a. 012345678901234501234567890123450123456789012345
IOaIA**cbaaacaacac**AOcaI**ccabcbccbac**IONAA**bbbbacbcbbc**

b.



c.

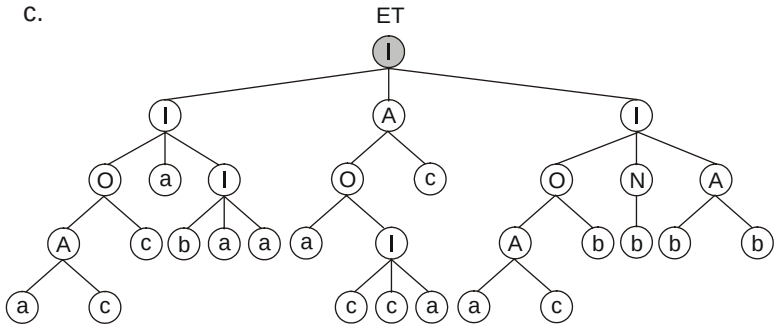


图 2.10.对作为多级结构的子表达式树的布尔表达式进行编码的多基因染色体的表达。**a)** 一个 3-基因染色体，其中尾部用黑体标识。**b)**每个基因所编码的子表达式树。**c)**用加号连接得到的结果。连接函数用灰色标识。

a. 012345678
2bba42a31

b. 2 b b a 4 2 a 3 1

c.

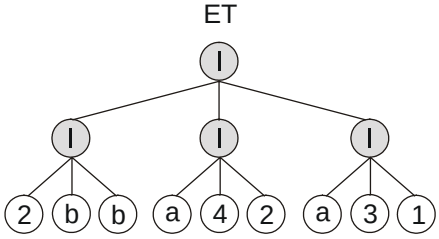


图 2.11.对由作为表达式树的单元素基因构成的多基因染色体的表达。**a)** 一个 9-基因染色体。**b)**每个基因所编码的子表达式树。**c)**由 IF 连接的完整表达的个体。连接函数用灰色标识。

图 2.11 给出了另外一个翻译后修饰的例子，其中的染色体是最简单的类型（单元素子表达式树）。在此，IF 函数将这 9 个子表达式树 3 个 3 个地连接起来，形成 3 个新的簇，这三个簇也由 IF 函数连接起来，最后形成一个多级结构的子表达式树。这样的染色体结构可以用来找到复杂的布尔函数问题的解。同样，图 2.11 中的染色体所编码的多级子表达式树也可以被线性化地编码成如下的 K-表达式树。

```
0123456789012
IIII2bba42a31                                    (2.25)
```

总之，要想完整地表达一个染色体，还必须提供子表达式树之间的相互作用的信息。所以，对于每个问题，连接函数的类型或子表达式树之间的相互作用也根据情况来选定。我们可以在开始的时候为代数表达式数选择加法作为连接函数，为布尔表达式数选择 OR 作为连接函数，但是在某些情况下，使用别的连接函数也许更加恰当（例如乘法或者 IF）。我们的目的是要找到一个好的解，所以可以采用不同的连接函数来搜索不同的适应度图的凹陷之处，从而增加找到最高峰的机会。显然，可以对基本基因表达式算法稍作修饰让连接函数得以进化。解决这个问题的一个极好而且非常有趣的办法就是通过产生一种 homeotic 基因，这种基因用来对发展的程序进行编码（见下一节）。

在另一种翻译后的相互作用下，子表达式树表示一个点的坐标（例如，旅行商问题中的一个城市），这些子表达式树用它们之间的距离连接。还有一些情况下，不同的多基因族的产物被安排在空间中，并相互连接，形成一个复杂网络间的相互作用。

最后，某些染色体的完整表达需要一些小的计划顺序执行。其中第一个子表达式树完成一部分工作，第二个子表达式树继续完成一部分工作，依此类推。在这种情况下，最终的计划由顺序执行的子计划构成。

以上只是子表达式树之间的相互作用的为数不多的几个实例，但是我们可以很容易实现其它的方法来解决不同的问题。

2.2.3.细胞和连接函数的进化

从进化学的角度来说，用一个特定的连接函数来连接子表达式树是简单、高效的。的确，从单基因系统到多基因系统，效率得到很大的提高。尽管这种人工的复杂度提高了（这里“人工”一词的意思是说这种复杂度的增加不是由系统自身的进化引起的），但是多基因系统的进化仍然非常高效，所以可以用它们来进化得到不同问题的解。原则上说来，不可能对系统从外部强加一个更高级别的复杂度，但是这并不保证系统的性能会提高。复杂的进化系统不是这样产生的：高级别的复杂系统建立在较低级别的复杂性之上，而且复杂度的进化或多或少具有连续性。虽然 GEP 中的子表达式树间的相互作用确实可以用更高级别的复杂度来编程。为了达到这个目的，我们创造了一类特别的基因——homeotic 基因——来控制个体的发展。这些基因的表达会导致不同的程序或者细胞。因此，homeotic 基因决定哪些基因在哪个细胞中表达以及它们之间如何相互作用。

那么 homeotic 基因的结构是什么样的呢？它们具有的结构和传统基因一样，而且是通过同样的过程形成的。只是在这里，头部含有连接函数和一类特殊的终点——基因终点——来表示传统的基因。尾部显然仅含有基因终点。

例如，考虑如下的染色体：

```
012345601234560123456012345678 90
*-*aaaa*a/aaaa+/+aaaa+*2+Q322122                 (2.26)
```

传统的基因像往常对 3 个不同的子表达式树进行编码。homeotic 基因控制不同子表达式树之间的相互作用。注意，homeotic 基因有其特殊的长度和特殊的函数集。对于这个特定的

例子，homeotic 基因的头部的长度 h_H 等于 5，而其它基因的头部长度等于 3；homeotic 基因的函数集 F_H 为 $F_H = \{+,*,Q\}$，其它基因的函数集为 $F = \{+,-,*,/\}$。如图 2.12 所示，这种表

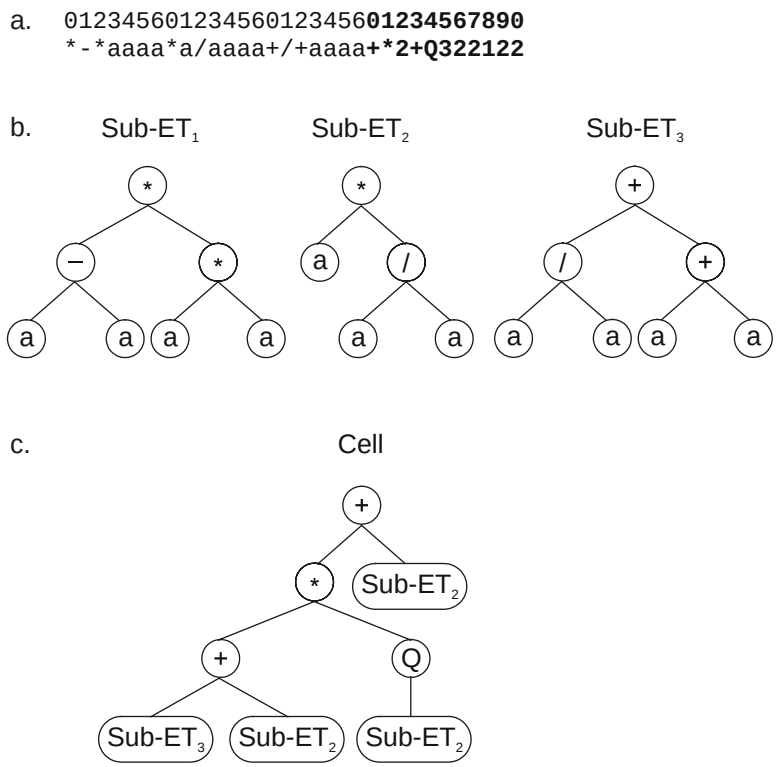


图 2.12.对包含单个 homeotic 基因的染色体的表达。**a)** 由 3 个传统基因和一个 homeotic 基因（用黑体标识）构成的染色体。**b)**每个基因所编码的子表达式树。**c)**最终的程序或细胞。注意该细胞系统允许代码重用，因为每个传统基因对一个自动定义函数进行编码。

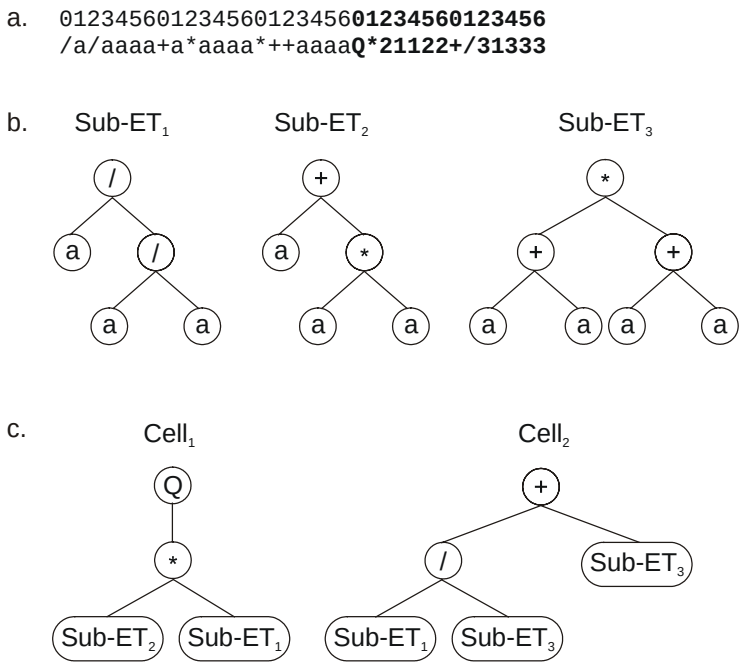


图 2.13.对包含多个 homeotic 基因的染色体的表达。**a)** 由 3 个传统基因和 2 个 homeotic 基因（用黑体标识）构成的染色体。**b)**每个基因所编码的子表达式树。**c)**在 2 个不同细胞中表达的不同程序。注意不同的细胞是如何将不同基因组合结合起来的。

达方式不仅允许连接函数的进化，还允许代码重用。也许读者已经注意到，GEP 中产生的这种极其优雅而且简单的形式就是大家所熟知的自动定义函数（ADF）。的确，这种细胞表示方法中的任何传统基因不仅可以根据需要使用任意次，而且在不同基因之间建立了联系。

使用一个以上的 homeotic 基因显然会导致一个多细胞系统，其中，每个 homeotic 基因将不同组的基因放到一起。例如，考虑如下的染色体：

```
012345601234560123456012345601234560
/a/aaaa+a*aaaa*++aaaaQ*21122+/31333
```
(2.27)

该染色体对 3 个传统基因和 2 个 homeotic 基因（用粗体标识）编码。其表达式导致两个不同的细胞或程序，每个细胞或程序分别用不同的方法来表达不同的基因。

2.2.4.其它级别的复杂度

如我们以上看到的例子，GEP 虽然是一个简单的系统，但是它却已经表现出复杂的发展过程。GEP 进化得到的最复杂的个体，除了头部和尾部以外，还包括用来对单元素子表达式树进行编码的附加域。这些单元素子表达式树与编码在头/尾域中的主要子表达式树相互作用，形成一个更加复杂的具有复杂相互作用网络的实体。

在符号回归问题种，我发展了一种这样的结构来操纵随机数值常量（Ferreira 2000）。例如，如下的染色体

```
012345678901234567890123
+*?+?*+a??aaa??09081345
```
(2.28)

含有一个附加域 Dc，这个附加域用来对随机数值常量进行编码。该染色体的翻译过程如图 2.14 所示。在 4.2 节我们将会学习这些子表达式树如何相互作用才能使个体得到完整的发展。

a.
```
012345678901234567890123
+*?+?*+a??aaa??09081345
```

b.

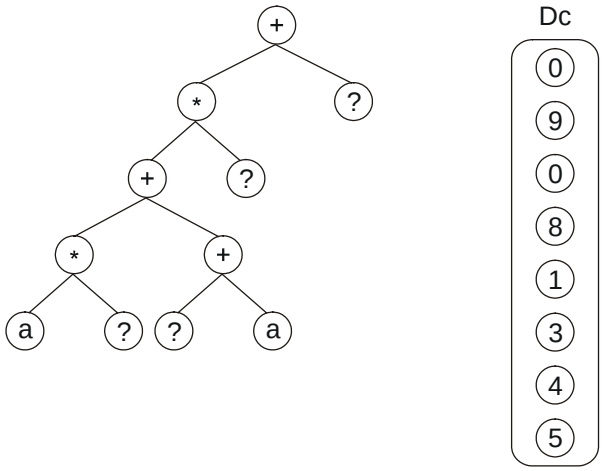


图 2.14.对含有多个域的染色体的表达。**a)** 由一个传统的头/尾域和一个对随机数值常量进行编码的附加域（Dc）构成的多域染色体。**b)**每个基因所编码的子表达式树。Dc 中编码的单元素子表达式树单独放置。“？”代表 Dc 中数字所编码的随机数值常量。

在设计线性基因组来对神经网络进行编码的时候还将用到多个域（见第 5 章）。这些神经网络是 GEP 进化得到的最复杂的个体中的一种。这里，网络结构编码在传统的头/尾域中，然而权值和阈值编码在两个附加域，即 Dw 和 Dt 中，这两个域分别由单元素基因构成。下

面这个染色体就含有两个附加域，这两个附加域分别对权值和阈值进行编码（不同的域用不同的阴影标识）：

```
012345678901234567890123456789012345
DUbUTdaedbfaabad42998409791482467584
```

(2.29)

其翻译过程如图 2.15 所示。第 5 章我们将学习其完整发展过程的规则以及由这种复杂的个体构成的种群如何进化，最后找到这些以 GEP 编码形式出现的神经网络问题的解。

a.

```
012345678901234567890123456789012345
DUbUTdaedbfaabad42998409791482467584
```

b.

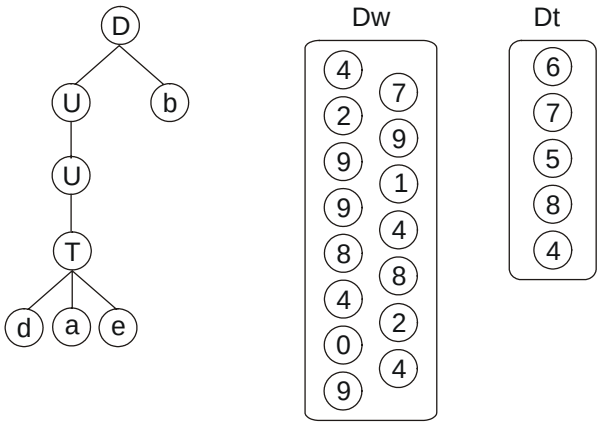


图 2.15.对含有多个域的染色体的表达。**a)** 由一个对神经网络结构进行编码的传统的头/尾域和 2 个附加域构成的多域染色体，其中这 2 个附加域中一个对权值(Dw)进行编码，另一个对阈值(Dt)进行编码。**b)**每个基因所编码的子表达式树。Dw 和 Dt 中所编码的单元素子表达式树单独放置。“ U ”, “ D ”和“ T ”分别表示具有 1 个，2 个和 3 个连接分量的函数。这些子表达式树之间如何相互作用详见第 5 章。

2.2.5. *Karva* 语言：GEP 的语言

我们已经看到，每个基因对一个特定的子表达式树进行编码，而且每个子表达式树对应一个特定的 K-表达式或开放阅读框。由于这种对应十分简单、优雅，所以 K-表达式实际上是非常紧凑、容易理解的计算机程序。我们已经看到多级子表达式树如何和轻易地转化成线性 K-表达式，这种转换对任何代数的，布尔的，或者非传统的表达式来说都是十分容易的。的确，GEP 的语言——*Karva* 语言——是一种功能十分强大的表达方式，它可以以简单、非常紧凑的符号串形式进化相对比较复杂的程序。事实上，已经有商业软件可以自动将 K-表达式和 GEP 染色体转换成传统的 C＋＋或 Visual Basic 函数，例如 *Gepsoft* 公司的 *Automatic Problem Solver*。

Karva 表达方式的另外一个好处在于它可以采用任何编程语言来进化复杂的程序。的确，最初的 GEP 程序是用 C＋＋编写的，但是也可以在任何编程语言下实现。作为比较，值得强调的是，早期的 GP 的实现很大程度上依赖 LISP，因为采用该语言很容易实现复制过程中的子树交换。下一章将详细分析基因表达式算法的实现细节，我们将从产生初始种群开始，以选择和复制操作作为结束。

基本基因表达式算法 3

基本基因表达式算法的基本步骤示意图见图 3.1。该过程从随机产生一定数量的染色体个体（初始种群）开始。然后对这些染色体进行表达，依靠一个适应度样本集（也称为选择环境，即问题的输入）计算出每个个体的适应度。然后个体按照其适应度被选中，进行有修饰的复制。留下具有新特性的后代。接下来，这些新的个体也要经历相同的发展过程：基因组的表达，面临选择环境，选择和有修饰的复制。该过程重复若干代，直到发现一个优良解。

本章中我们将详尽分析该算法的基本步骤，从产生初始种群中染色体个体的随机产生开始，以复制操作作为结束。我们的目的不仅是研究算法的逻辑结构，还包括计算机程序构成的种群为什么能够以及如何进化，并找到特定问题的优良解。我们已经看到，不论是有机体还是计算机程序所构成的种群，它们之所以能够进化，是因为个体通过有修饰的复制产生了遗传多样性，而遗传多样性是进化的原始元素。这种遗传多样性是选择的基础，因而它在进化过程中扮演重要的角色。因此，我们将彻底地分析遗传多样性的代理——遗传算子的机制及其影响。我们将用这些遗传算子来解决同一个简单的问题，这样我们就可以对整个种群进行分析，也可以完全理解每个遗传算子的效力。本章中用来解释遗传算子的内部运行机制的问题十分简单也很容易理解，这个问题由三个参数的布尔多数函数构成。

进一步来说，本章最后详尽地分析了一个符号回归问题，以此演示了用 GEP 解决一个特定的问题所需要的所有步骤。

3.1.个体种群

像所有的遗传算法一样，GEP 采用个体种群，而且最开始必须产生初始种群。后续的种群都是这些初始种群或者奠基种群经过遗传修饰得到的后代。我们已经看到，在基因型/表现型系统中，我们只需要产生个体染色体，然后由发展过程来控制后续工作。因此，在 GEP 中，只需要随机产生初始种群的简单个体染色体结构。而且这是一个非常小的任务。对每个问题而言，我们必须选择产生染色体的符号，也就是说，我们必须选择适合解决当前问题的函数集和终点集。我们还要选定每个基因的长度，每个染色体中的基因个数，以及这些染色体的表达式之间如何相互作用。最后，我们必须提供一个选择环境（适应度函数集）来计算个体的适应度。然后，个体按照其适应度被选中，进行有修饰的复制，在下一代中产生新的成员。这个种群经历同样的发展过程，产生另一代新的群体。该过程重复若干代，直到发现一个优良解。

3.1.1.初始种群的产生

初始种群的染色体由解决特定问题的分别代表函数和终点的符号随机产生。其初始个体是当前问题的第一个备选解集。因为它们是随机生成的，所以这些奠基种群几乎不可能是非常优良的解。但是它们含有几乎所有需要的部分，因为进化过程会完成其它的工作，而且很快非常优良的解就会出现。我们下面将用一个具体的例子来说明。

例如，假设我们想知道如何用 AND，OR 和 NOT 来表示 Majority(a,b,c)函数。这种情况下，函数集的选择并不复杂，函数集的构成为 $F=\{A,O,N\}$，分别代表布尔函数 AND，OR 和 NOT。终点集的选择也不复杂，终点集的构成为 $T=\{a,b,c\}$，分别代表多数函数的三个参数。因此，对于该问题而言，基因的头部将从六个不同符号中随机生成 (A,O,N,a,b,c)，

而基因的尾部从一个较小的字符集随机中生成，该符号集包含三个符号 (a,b,c)。

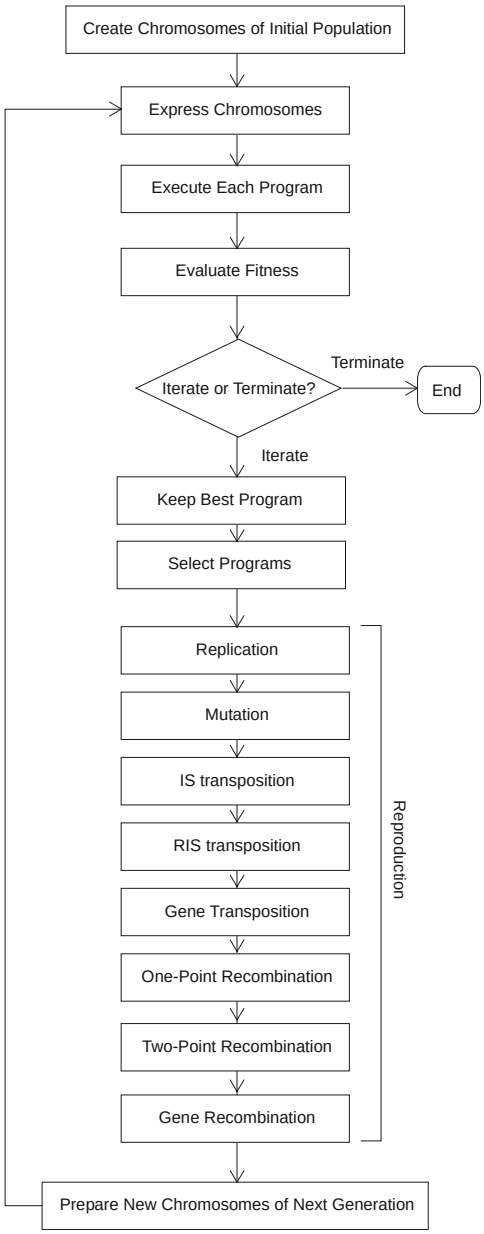


图 3.1. 基因表达式算法流程图。

多数函数的真值表如表 3.1 所示。对于该问题而言，所有的过渡状态集用来构成选择环境，以计算每个程序的适应度。选择环境也可以较正式地称之为适应度样本集。该问题的适应度函数也不难猜测，它对应每个特定个体所正确计算的适应度样本的个数。

表 3.1 多数函数

a	b	c	d
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

因此，一旦得到来自 F 和 T 的符号，就可以随机产生初始种群的染色体。显然，基因的头部可以使用 F 和 T 中的元素，而基因的尾部则只能使用终点符号。图 3.2 给出了一个小的初始种群（也称为第 0 代），该种群由用来解决多数问题的 10 个随机产生的个体构成。染色体含有两个基因，每个基因的长度为 7（$h=3$，$t=4$）染色体对子表达式树进行编码，这些子表达式树翻译后由布尔函数 OR 连接。

值得注意的是，GEP 的初始群体的产生非常容易。染色体的简单结构只需要使用函数集和终点集的符号随机产生就可以，不需要其它的任何东西来监控结构的完整性：我们确信所有的程序都是合法的，而且绝无例外。作为比较，GP 随机产生的树结构时必须非常小心，以保证产生的程序在句法上是正确的。毫不奇怪，GP 的初始种群的产生是一个复杂而且耗时的过程。

```
Generation N: 0
01234560123456
NacababAANaccb-[0]
AOAaccaNcOabcc-[1]
ObaaabaNONaacb-[2]
OObacbcANNcccc-[3]
NNNabcbOAaacbc-[4]
NbbbcccAaOabbc-[5]
AbNbbaaAaAbcac-[6]
OcOccaaOOAaabb-[7]
OOcacabNNNbbcb-[8]
ObbbabbAbAccbc-[9]
```

图 3.2.一个用来求解 Majority 函数问题的较小初始种群的染色体。这些随机生成的染色体由两个基因构成，对由 OR 连接的子表达式树进行编码。

用随机生成染色体个体的办法产生初始种群之后，还要对染色体进行表达，计算其适应度。图 3.3 给出了多数函数任务的每个个体的适应度（为了简化，只给出了个体的染色体）。然后个体根据其适应度被选中并进行有修饰的复制。例如，图 3.2 种所示的初始种群中的 10 个个体全部是可存活的，所以这些个体可以被选中进行复制。这些可存活的个体，在它们中间产生与当前种群个数相同的新个体。这样，经过一次运行之后，种群的大小将保持不变。

```
Generation N: 0
01234560123456
NacababAANaccb-[0] = 2
AOAaccaNcOabcc-[1] = 4
ObNaabaAONaacb-[2] = 3
OObacbcANNcccc-[3] = 4
NNNabcbOAaacbc-[4] = 4
NbbbcccAaOabbc-[5] = 4
AbNbbaaAaAbcac-[6] = 5
OcOccaaOOAaabb-[7] = 5
OOcacabNNNbbcb-[8] = 5
ObbbabbNbAccbc-[9] = 4
```

图 3.3.初始种群个体的染色体及其相应的适应度（等号之后的值）。适应度对应于每个染色体正确求解适应度样本的个数。

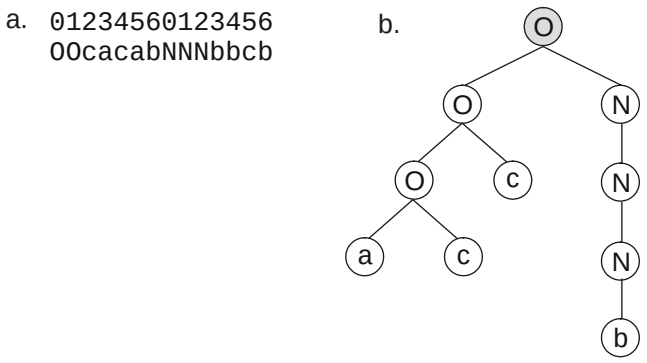


图 3.4.第 0 代中某个最佳个体的表达。**a)**由 2 个基因构成的染色体。**b)**编码在染色体中的程序。连接函数用灰色标识。该布尔函数正确求解 8 个适应度样本中的 5 个。

被选中个体的后代由新一代中的成员构成。第 0 代中最优的表达式之一（8 号染色体）如图 3.4 所示。该个体能够正确求解 5 个适应度样本，因而其适应度等于 5。

也许读者已经注意到，初始种群中随机产生的所有基因的第 0 位上都是函数。虽然我选择让初始种群中随机产生的所有基因的第 0 位上都是函数，但是这对于多基因的个体来说并不十分重要。初始种群的特点只是 GEP 自身进化过程中的一点残余，因为第一次实现 GEP 的时候，染色体只含有一个基因。在这些系统之中，对单元素表达式树进行编码的基因没有什么用处，因而被排斥在种群之外。当多基因系统开发出来以后，我没有改变初始种群的这一特点，因为在后续代中，终点可以被插入到基因的起始位置上。而且如果这种方法可以对某个任务而言有优势的话，这样的基因当然会得到进化。我们在本章的后面将看到，点变异操作将会允许单元素表达式树进行编码的基因得到进化。

随机产生初始种群的一个问题是，有时候，特别是当个体的数量或者适应度样本的个数较少的时候，染色体所编码的个体都是可存活的个体，导致运行异常中断（这种情况对于布尔型的问题很可能出现）。回避这种问题的一个简单的办法就是引入一个启动控制。该控制默认设置为一个可存活的染色体，本书中的所有问题都将采用这种方法。我们将看到，哪怕种群中所有的个体都是一个可存活的启动个体的后代，GEP 的种群仍然能够有很高的进化效率（相关讨论见第 7 章）。这种控制机制，与最佳个体复制相结合，防止 GEP 中的运行失败情况的出现。所以如果初始种群中所有个体的适应度值都为 0，就会随机产生另一个初始种群。该过程重复执行直到至少产生一个可存活的个体为止。

3.1.2.后续代和精英策略

图 3.3 所示的初始种群中可存活基因的后代如图 3.5 所示。这一代中最佳个体的表达式如图 3.6 所示。再次，我仅使用变异和单点重组作为引入基因修饰的来源，是为了简化种群进化历史的分析。

注意，第 1 代的 0 号染色体是初始种群中最佳个体的原样副本（图 3.3 中 8 号染色体）。的确，代与代之间，最佳个体（或者最佳个体之一）被原样复制到下一代中。如果种群中有多个个体具有最佳适应度，那么选择最后一个进行复制。在第 1 代中，5 号染色体将进行无修饰的复制并占据下一代的 1 号位置（如图 3.7）。这时，原来占据该位置的个体就消失了。我们在分析某些算子，例如重组算子的结果时，必须牢记这一点，因为某个染色体的子代染色体也许并不存在于下一代中。最佳个体复制操作，也称为精英策略，保证至少有一个后代是可存活的（显然，只有在代与代之间的选择环境保持不变的时候），同时在逐渐适应过程中保持个体的最优特性。精英策略还有另外一个作用，它允许使用相对较高比例的遗传算子

```
Generation N: 1
01234560123456
OOcacabNNNbbcb-[0] = 5
AbNbbbcaaAbcab-[1] = 6
AbNbbaaOOAaabb-[2] = 6
OcObcaaAabbcac-[3] = 6
aaOccaaOOOaabb-[4] = 6
AbOccabOOAbabb-[5] = 6
OANbbacNaAbcac-[6] = 4
aOcaccbNNNbbcb-[7] = 4
AbNbbaaAaAbcac-[8] = 5
AbNbbacANNcbcc-[9] = 2
```

图 3.5.图 3.3 中所示初始种群的后代。其中有 5 个优于前一代的最佳个体。

a.
```
01234560123456
NbbbcccAaOabbc
```

b.

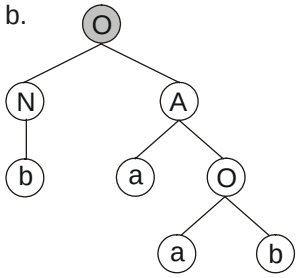


图 3.6.第 1 代中某个最佳个体的表达。**a)**由 2 个基因构成的染色体。**b)**编码在染色体中的程序。连接函数用灰色标识。该布尔函数正确求解 8 个适应度样本中的 6 个。

而且不会有大规模灭绝的危险。

在自然界中，至少在停滞阶段，变异概率是严格受控的，而且通常变异概率都非常小，并且适应和进化过程非常平缓。在 GEP 中，种群总是在很高的创造概率下进化的，Eldredge 和 Gould 在他们的断点平衡理论中称之为创造性的时期（Eldredge 和 Gould 1972）。所以，在人工进化系统中，进化发生的速度较快而且能够在不断的混乱情况下进行，在有限的时间里面找到非常好的解。

```
Generation N: 2
01234560123456
AbOccabOOAbabb-[0] = 6
AbOccabNOAbabb-[1] = 4
ANNbbaaAaAbcac-[2] = 3
AbObaaaAOAbabb-[3] = 6
ObNbbcaAaAbcac-[4] = 4
OcObcaaANNcbba-[5] = 4
AbOccabOOAbabb-[6] = 6
AbNbbbcaaAbcab-[7] = 6
AcObaaaAabbcac-[8] = 8
OOAacabNabbaac-[9] = 4
```

图 3.7.用来求解 Majority 函数问题的下一代计算机程序。这一代的个体是上一代中选中的个体的直接后代。注意这里找到了一个具有最大适应度值的完美程序（8 号染色体），因此该程序比起祖先要好很多。

a. 01234560123456
AcObaaaAabbabb

b.

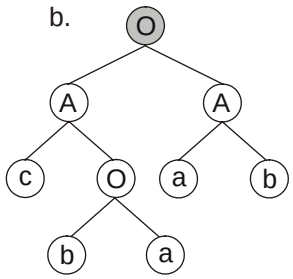


图 3.8. Majority 函数的一个完美而且简洁的解。a)对 OR 连接的两个基因编码的染色体。b)编码在染色体中的多数函数的完美解。连接函数用灰色标识。

图 3.7 为该进化过程中产生的下一代（第 2 代）。这一代的最佳个体（8 号染色体）具有最大适应度因此它对一个多数函数的完美解进行编码。事实上，如图 3.8 所示，该染色体所编码的解是多数函数最简洁的解之一。

值得注意的是，在本次运行中，某些个体含有对单元素表达式树进行编码的基因（第 1 代的 1，4，7 号染色体；第 2 代的 7 号染色体），因为这些染色体的某个基因的起始位置都是一个终点。这些基因都只能通过变异进化，因为只有变异能在子表达式树的根部引入一个终点（关于 GEP 中使用的遗传算子的详细分析，参见 3.3 节）。

3.2.适应度函数与选择

本节中将描述 3 种常用的适应度函数。其中的 2 种适应度函数可以用来解决任何符号回归问题。第一种基于绝对误差，另一种基于相对误差。第三种适应度函数是前面几节用来解决多数函数问题的适应度函数的一个变体，它可以用来解决大部分逻辑合成问题的。其它一些奇特的适应度函数将在本书的其他章节给出。我们将看到，一个问题的成功解决很大程度上取决于适应度函数的设计方法。因此，在设计适应度函数的时候，目标必须清楚并且定义准确，只有这样才能保证系统朝着正确的方向进化。

3.2.1.适应度函数和选择环境

GEP 的一个重要应用就是符号回归或者函数发现，其目的是寻找一个对误差在一定范围内的所有适应问题都能够运行良好的符号表达式。对很多数学上的应用来说，用较小的相对误差或者绝对误差来发现一个非常良好解是很重要的。但是，如果选择域过小，仅仅选择那些误差非常小的个体，则群体会进化得非常慢，而且不太可能找不到一个满意解。另一方面，如果反之，选择范围太大，将会出现许多具有最大适应度 $f_{\max}$ 的解，而这些与最优解相差甚远。所以，仅仅通过采用或者不采用选择误差来区别不同的适应度样本的简单适应度函数肯定是不够的，所以需要一个更加成熟的办法。

为了解决这个两难的问题，我设计出了无需中断进化过程就能找到非常好的解的进化策略（Ferreira 2001）。采用这种策略，系统只需要在某个误差范围之内找到最可能的节。为此，先给出一个很广的选择范围，例如大小为 20%或者 100%的相对误差，使其不仅允许进化过程能够启动，而且能够在想要的最小误差附近进行细微的调整。观察的结果是，在早期的几代中的那些个体通常非常“ 不适合 ”，但是它们的后代一直通过遗传算子的作用不断变化，使整个种群适应得非常好，渐渐发现一些靠近非常优良解的更好的解，更正式一点说来，当误差类型选为绝对误差的时候，个体程序 i 的适应度 f_i 可以用方程（3.1a）来表示，当误差类型选为相对误差的时候，个体程序 i 的适应度 f_i 可以用方程（3.1b）来表示：

$$f_i = \sum_{j=1}^{C_t} \left(M - \left| C_{(i,j)} - T_j \right| \right) \tag{3.1a}$$

$$f_i = \sum_{j=1}^{C_t} \left(M - \left| \frac{C_{(i,j)} - T_j}{T_j} \cdot 100 \right| \right) \tag{3.1b}$$

其中 M 是选择范围，$C_{(i,j)}$ 是染色体个体 i 对于适应度样本 j （来自 C_t 集合中）的返回值。而 T_j 是适应度样本 j 的目标值。请注意，两个方程中的绝对值项都对应各自的误差（前一个方程中对应于绝对误差，后一个方程中对应于相对误差）。该项称为精度 p。因此，对于一个完美适应的情况，$C_{(i,j)} = T_j$ 且 $f_i = f_{\max} = C_t M$ 。

需要强调的是，这种适应度函数有如下几种优势：该适应度函数不仅自己找到最优的可能解，而且发现用来启动进化过程所需的可存活的奠基个体的过程非常简单。为了更清楚地说明问题，考虑一个基于标准 R-平方的适应度函数。能够根据程序的 R-平方值来选择程序的话，将是十分有用的。但是，对于复杂的问题而言，要让随机产生的奠基个体具有合法的 R-平方值（不管这个值多么小），简直是一个艰巨的任务（而且对于一个实际的复杂问题来说是不可能的），而且为了找到一个可存活的个体而产生上千个初始种群的话，将浪费极大的资源。我们可以很轻易地通过采用一个精心设计的选择范围来避免这样的死胡同。另一方面，至少在开始阶段，精度的最小值可以为 0，从而使系统能够有效地为选择范围所驱使。

我们在3.1节看到了适应度函数的另一个例子。其中个体的适应度是正确计算的适应度样本的个数的一个函数。对于更加复杂的布尔问题而言，为了正确解决大约50%的适应度样本，对个体进行惩罚是十分必要的，因为这很可能只反映正确解决一个二元布尔问题的50%的可能性。所以比较合理的办法是只选择那些能够正确解决大约50%的适应度样本的个体。低于这个指标的时候我们可以对适应度赋予一个符号值，例如 $f_i = 1$，或者通过使其不可存活的方法将这些个体从种群中全部清除（ $f_i = 0$ ）。通过这种办法，进化的过程通常从适应性很差的个体开始，因为这些个体很容易在初始种群中产生。然而在后续代中，适应度很高的个体开始出现，并在种群中迅速传播。对于像上一节中分析的Majority(a,b,c)这样的简单函数而言，这样的方法并不是非常关键的。但是对于一些更复杂的问题，选定一个选择的底线就会方便许多。对于这些问题，可以使用以下的适应度函数：

$$\text{If } n \geq \tfrac{1}{2} C_t, \text{ then } f_i = n; \text{ else } f_i = 1 \tag{3.2}$$

这里，n是正确求得的适应度样本的个数，而C_t使所有的适应度样本的数目。

本书中我们还将碰到其它适应度函数的例子，我们还将看到一个问题的成功很大程度上取决于适应度函数的设计。适应度函数设计很糟糕的一个经典的例子就是蛋白质三维结构预测中所使用的（Schulze-Kremer 1992）。虽然进化得到的解适应度很高，但是这些解与目标蛋白质在结构上相去甚远。

3.2.2.选择

在 GEP 中，个体通过赌盘轮采样策略根据其适应度进行选择（Goldberg 1989）。每个个体用圆形赌盘的一块来代表其适应度的比例。赌盘按照群体中个体数的值进行相应次数的旋转，从而始终保持群体的大小不变。本书中的所有问题都将一直使用这种选择方法和 3.1.2

节给出的简单精英策略。采用这种选择策略，确实有时候会丢失一些最佳个体，而一些普通个体进入到了下一代。但是这样也不一定就不好，因为种群会一代一代地向前推进。而且，因为我们使用每一代中的最佳个体复制策略，所以最佳个体的存在和复制能够得到保证。这样，至少最优的特性从来没有丢失过，并且能够达到不断学习的目的。

在本书中我们还会看到一些其它的选择策略，但是几种最常用的－－赌盘选择、确定性选择、锦标赛选择－－将在第 7 章详细分析（第 7.7 节）。我们还将看到，如果采用精英策略的话，这几种选择策略之间不存在明显的差别。它们中的一种效果很好的话，那么其它的几种的效果也会很好。其它需要考虑的就是任何进化计算的本质问题。至少对于 GEP 来说，本质问题不是所采用的选择策略而是所采用的遗传算子的效力问题。下一节我们将看到，遗传算子才是适应度图中真正的“鹰”。

3.3.有修饰的复制

根据适应度函数原理和赌盘原理，被选中的个体在经过修饰以后进行复制，从而产生能够使系统进行漫长运行所需要的基因多样性。

在自然界中，基因组的复制过程中引入了多种修饰操作（例如，变异，小的删除/插入等）；其它一些，比如复制之后的同源重组和变异，在复制之后出现。因此，在自然界中，我们不太可能知道修饰操作出现在什么时候。但是，在 GEP 中，我们就可以确切地知道修饰操作是什么时候发生的。另一方面，遗传算子顺序执行，由复制开始，接下来是变异，转座和重组。然而，基因组中后续的修饰操作的顺序对最终的结果则不是十分重要。

除了复制，即所有被选中的基因组被一丝不苟地原样复制以外，其它的遗传算子都是随机地挑选染色体，并对它们进行一些修饰操作。然而除了变异算子以外，任何一个遗传算子都不能够对一个染色体进行一次以上的修饰操作。比如，对于某个含有 10 个染色体的种群，交叉概率为 0.8，意味着每 10 个个体中有 8 个将被随机地选中进行重组。

所以，每个被随机选中的染色体可以每次被一个以上的遗传算子来修饰。因此，在复制的过程中，不同的遗传算子所施加的修饰在染色体中积累起来，使得新的种群通常与其祖先存在非常明显的不同。

下面讨论的是 GEP 中所采用的一些遗传算子，按照它们所使用的顺序进行介绍，显然应该从复制和选择开始。

3.3.1.复制和选择

虽然不可或缺，但是复制是最没有新意的遗传算子：因为只有它对遗传变化毫无贡献。它与选择一起会产生遗传漂变，导致某些个体所占的比例随时间变化。但是，它同选择和修饰一起，允许适应和进化的发生。

选择算子根据个体的适应度和赌盘的偶然性来选择个体。个体的适应度越高，它留下后代的可能性就越大。如果采用赌盘轮采样策略来进行选择，那么赌盘按照群体中个体数的值进行相应次数的旋转，从而始终保持群体的大小不变。所以，复制算子原样拷贝被选择操作选中的个体的染色体。拷贝的染色体由下一代中的个体的染色体构成。但是繁殖操作并没有完成。在此之前被复制的染色体必须经过修饰。这种修饰操作由变异、转座、重组（搜索算子）等遗传算子完成。但是现在我们暂时集中讨论选择和复制。

图 3.9 显示了被选中个体是如何复制的（在这里暂时不使用搜索算子和精英策略是为了更好地理解复制和选择操作）。例如，0 号染色体，第 0 代的最佳染色体之一，留下了两个后代（第 1 代的 1 号和 3 号染色体）；1 号染色体也是这一代的最佳染色体之一，只留下

一个后代（第 1 代的 9 号染色体）；7 号染色体，也是最佳染色体之一，留下了两个后代（第 1 代的 2 号和 4 号染色体）；但是 8 号染色体，虽然也是最佳个体之一，却没有留下后代，次优的染色体（3 号染色体）也留下了两个后代（第 1 代的 7 号和 8 号染色体）；而且虽然第 0 代中适应度最不好的染色体没有繁殖，但是一个普通的染色体，6 号染色体，留下的后代却最多（第 1 代的 0 号、5 号、6 号染色体）。该进化过程的结果如图 3.10 所示，我们可以看到当到达第 13 代的时候，所有的个体都是一个个体的后代：即第 0 代中 0 号染色体的后代。的确，复制和选择操作一起作用，只会引起遗传漂变。而且，遗传漂变虽然本身并不产生遗传多样性，但是对于适应度图的搜索却是有用的。只有搜索算子具有这种能力。

```
Generation N: 0                 Generation N: 1
01234560123456                  01234560123456
OOOaaabAAAcabb-[0] = 6          OONbcaaAaAaacc-[0] = 4
AAcbbabNONaaac-[1] = 6          OOOaaabAAAcabb-[1] = 6
ANaccbcNAAcbbc-[2] = 2          AcOaccbAbNbabc-[2] = 6
OAOccbaAOAbcab-[3] = 5          OOOaaabAAAcabb-[3] = 6
AAAbbabNcObcca-[4] = 3          AcOaccbAbNbabc-[4] = 6
NbacabbNbccbbc-[5] = 2          OONbcaaAaAaacc-[5] = 4
OONbcaaAaAaacc-[6] = 4          OONbcaaAaAaacc-[6] = 4
AcOaccbAbNbabc-[7] = 6          OAOccbaAOAbcab-[7] = 5
AaOacccAbbbaca-[8] = 6          OAOccbaAOAbcab-[8] = 5
AOAcaaaNaNbaab-[9] = 4          AAcbbabNONaaac-[9] = 6
```

图 3.9.复制和选择示例。仅采用复制和赌盘原理以使这些算子更容易理解。注意，例如第 0 代的 8 号染色体（最佳个体之一）没有留下后代，而 6 号染色体（一个中等个体）留下的后代数量最多。

```
Generation N: 13
01234560123456
OOOaaabAAAcabb-[0] = 6
OOOaaabAAAcabb-[1] = 6
OOOaaabAAAcabb-[2] = 6
OOOaaabAAAcabb-[3] = 6
OOOaaabAAAcabb-[4] = 6
OOOaaabAAAcabb-[5] = 6
OOOaaabAAAcabb-[6] = 6
OOOaaabAAAcabb-[7] = 6
OOOaaabAAAcabb-[8] = 6
OOOaaabAAAcabb-[9] = 6
```

图 3.10.遗传漂变示例。在这个极端的情况下，种群在第 13 代以后完全丧失遗传多样性，其所有成员都是一个染色体的后代，在这里，该染色体是第 0 代的 0 号染色体（见图 3.9）。

3.2.2.变异

变异是所有具有修饰能力的算子中最高效的算子（见第 7 章）。在变异存在的情况下，个体的适应过程将非常有效，它几乎能够对所有的问题进化得到很好的解。一般来说，我采用变异概率p_m来对应每个染色体中的两个点变异。考虑GEP基因组的长度，这种变异概率比

自然界中所能见到的变异概率要高很多（如Futuyma 1998）。的确，由于精英策略，我们可以让GEP种群在经历过度变异的情况下，仍然能够非常高效地进化。作为比较，人类的基因组长度大约为 6×10^9个基对，但是每代基因组中仅引入大约 120 个新的变异。

变异可以发生在染色体内的任何位置。然而，染色体的结构组织必须保持完整。所以在头部中，任何符号都可以变成符号或者终点；在尾部中，终点只能够变成终点。通过这种方法，染色体的结构组织得以保持，而且由变异产生的新个体是结构上正确的程序。

值得强调的是，GEP 点变异完全不受任何限制。这意味着，在头部中，函数可以被其它的任意函数替换，而不必考虑每个函数所需要的参数个数；函数也可以被其它的终点取代。的确，变异操作可以完全自由的在适应度图中“漫步”，而不存在任何限制。

图 3.11 分析了变异的工作机制。在分析的时候，仅仅采用变异作为遗传变化的来源。变异概率p_m仍对应每个染色体中的两个点变异，在此，我们设定为$p_m=0.143$。图 3.11 所示的种群是通过运行一次我们已经熟悉的Majority(a,b,c)函数问题得到的。注意第 6 代的 1 号染色体对一个多数函数的完美解编码，这个解具有最大适应度。

```
Generation N: 0
01234560123456
NabbabbAAccbcb-[0] = 3
NAabbcaNbbbcca-[1] = 2
OcOcaaaNaOabaa-[2] = 4
AaAcccbAbccbbc-[3] = 7
AObbabaAOcaabc-[4] = 7
AAAbaacONOaabc-[5] = 4
AAccbcaNNcbbac-[6] = 6
NOccabaOcbabcc-[7] = 4
NOAcbbbAaNabca-[8] = 2
NacbbacAbccbbc-[9] = 3

...
```

```
Generation N: 5
01234560123456
AabbabcAOcaabc-[0] = 7
babbabcAOcaabc-[1] = 7
AOAacbcOAOcaac-[2] = 6
ANbbabcAOcaabc-[3] = 6
AOAbabacbcaaba-[4] = 7
AabcaccAONaabc-[5] = 6
AOAccbaAbaabbc-[6] = 6
AObcabaAbNcaba-[7] = 6
NAAbbacONOacca-[8] = 3
AONbabacbcaaba-[9] = 5
```

```
Generation N: 6
01234560123456
AOAbabacbcaaba-[0] = 7
AabbabcAOcbabc-[1] = 8
AabbabcAccaabc-[2] = 7
NAAbaacONaaacc-[3] = 4
AOAbabacbcaaba-[4] = 7
AabbbbcAONaabb-[5] = 6
AOAbabacNcabba-[7] = 7
AOAccbaAbaabbc-[6] = 6
NAAbbacONOacaa-[8] = 4
AObbabaAAcaabc-[9] = 7
```

图 3.14.求解 Majority(a,b,c)函数的一个初始种群及通过变异产生的后代。由 OR 连接的子表达式树编码的染色体。注意后代与其初始种群中的第 0 代祖先完全相同。第 6 代（1 号染色体）发现的完美解和其假定的祖先之一用黑体标识。注意第 5 代的 1 号和 3 号染色体也可能成为完美解的祖先，在两种情况下，重组向出现两个点变异。

这个解的前身最可能是第 5 代中的 0 号，1 号和 3 号染色体。在第一种情况下，复制过程中只出现了一个点变异，而在后两种情况中，出现了两个点变异。图 3.12 比较了这些推定的前身所对应的表达式树与子表达式树，即变异前和变异后的情况。这里，复制过程中 3 号染色体上出现了两个点变异：一个是将 2 号基因上的 1 号位上的“ N ”变成“ a ”，另一个是将 2 号基因上的 3 号位上的“ a ”变成“ b ”。注意，第一个变异引起表达式树的明显变化，并将子表达式树缩短一个节点的长度。还要注意，虽然第二个变异没有改变子表达式树，但是子表达式树中所编码的表达式却不相同。

让我们更进一步地分析图 3.11 所示的种群。一方面，我们可以看到有些变异产生中性的效果。例如，第 6 代的 7 号染色体是第 5 代 4 号染色体的后代。这两个染色体仅在 2 号基因的 1 号位和 4 号位上不同。这些染色体的表达式给出的表达式树是相同的（图 3.13），因为变异出现在ORF_2的终止点之后。如我们已经看到的，出现在基因非编码区的变异操作具有中性的作用，因为它们在有机体里面没有表达式。

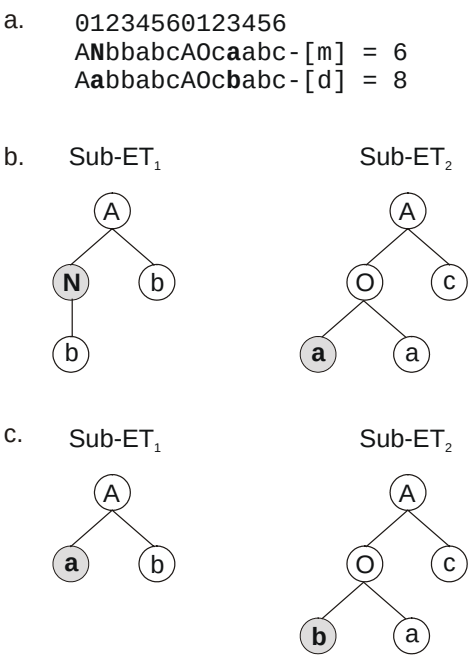


图 3.12.变异示例。**a)**含有黑体标识的变异点的母代染色体和子染色体。**b)**由母代染色体编码的子表达式树（变异前）。**c)** 由子代染色体编码的子表达式树（变异后）。变异节点用灰色标识。注意第一次变异明显改变了$Sub\text{-}ET_1$，将原始的子表达式树缩短了一个节点。

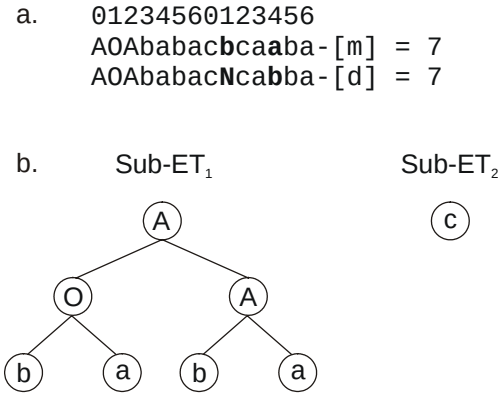


图 3.13.中性变异示例。**a)**含有黑体标识的变异点的母代染色体和子染色体。**b)**两个染色体所编码的子表达式树。注意**(a)**中所示的变异在最终的表达式树中没有得到表达，因为它出现在终结点之后。

另一方面，我们将看到，在 GEP 中，编码序列中的变异操作将产生非常深刻的影响，大部分情况下都会剧烈地改变表达式树的形状。然而这种剧烈地改变表达式树形状的能力对进化能力来说是必不可少的（遗传算子的比较见第 7 章）。的确，本书中给出的结论明显地说明我们希望不破坏小的功能块，让其小心地重组（像 GP 中的那样）的人类化的思想是十分保守的并且效果不佳。基因型/表现型系统会找到更加有效的方法来产生并控制它们自己的基因块。这些基因块与数学家的选择大不相同，但是它们的效率要高出许多。

最后，值得强调的是，在 GEP 中既没有变异种类的限制，也没有一个染色体中变异次数的限制：在所有的情况中，新生的个体都是在句法上正确的程序。这种重要的特点使 GEP 和那些点变异操作会导致非法程序的 GP 风格的系统之间存在明显不同。如果某个进化算法不能采用这种功能强大的算子，这种算法就极其受限制，因为变异是遗传多样化最重要的代理（见第 7 章）。

3.3.3.转座和插入序列元素

GEP 的转座元素是基因位的片断，它们可以被激活后跳到染色体中别的位置。在 GEP 中有三种转座元素(Transposable Elements)：i) 位于第一位的有函数和终结点的短片段转座到基因的头中，但不能转座到根部（Insertion Sequence Elements, IS 元素）；ii) 位于第一位的有函数和终结点的短片段转座到基因的根部（Root Insertion Sequence Elements, RIS 元素）；iii)整个基因转座到染色体的起始位置。

IS 和 RIS 的元素的存在是 GEP 发展过程中的残余物，因为第一个基本基因表达式算法仅仅使用单基因染色体，而且在这样的系统中在根部具有终点的基因没有什么作用。因此，严格限制控制转座到根部的情况，只有起始位置上存在一个函数的转座子才能被选中转座到根部。当多基因染色体引入后这种特点被保留下来，因为这些算子对理解基因变化的机理十分重要。对遗传算子变化能力的比较说明设计遗传算子的时候不必过于谨慎，抛弃那些任何可能导致修饰过于剧烈的东西。实际上，根转座操作——GEP 中最具破坏性的算子——能够非常高效地找到解。而且，IS 转座和 RIS 转座都可以在基因中重复产生简单的序列，这本身并没有什么奇怪的，而且这对进化来说很重要。

3.3.3.1. 插入序列元素的转座

任何基因组中的序列都可以成为 IS 元素，而这些元素从整个染色体中随机选出，生成一个转座子的拷贝，然后将其插入到一个基因除起始位以外的头中任何位置。作为代表，这

```
Generation N: 0
01234567890123456789001234567890123456789O
AbOabcOAOOaacaccacbccANaObOAbNaabcbcccbcbb-[0] = 6
AaAOaOOaAAbbacabcabbcOAAcONaOAccabbbcabbbc-[1] = 6
OaAbcNbaNaabccbbbcaaaNbcbNbAANbacabacacbcb-[2] = 5
ObbbAONNNAccaccbabcbcAAcacObOAaabaaaacbaca-[3] = 7
ONNAbObbONabcabbcbbcbAObaNbOAcbcccbcbaaabb-[4] = 4
ANabONNNcabbcccabcbacAbObOcAaAbabacacaabcb-[5] = 6
AONNObONOOaaacbaaababOAOAANaObacbbcccbbcaa-[6] = 4
NNbabcNbNNcbaacaabcccAbNcANNbbObababbcabaa-[7] = 6
NONNAaAbAOcacaaababccObaabbOabababacaacbab-[8] = 6
NONbcANNONcabbacaccabAbaOcOAacNbbccbacacba-[9] = 6
```

...

```
Generation N: 12
01234567890123456789001234567890123456789O
ANabcaabbcbbcccabcbacAbOcAaAbaaabacacaabcb-[0] = 7
AaANbAaANcbbcccabcbacAbOOcAaAbaabacacaabcb-[1] = 7
AAaANcaabbbbcccabcbacAbcccOOcAaabacacaabcb-[2] = 6
ANababbcbObbcccabcbacAbOcAaAbaaabacacaabcb-[3] = 7
ANacaabbcbbbcccabcbacAbOcAaAbaaabacacaabcb-[4] = 6
AAbaaANcaabbcccabcbacAbOOcAaAbaabacacaabcb-[5] = 7
ANababbcbObbcccabcbacAAbaabbObcabacacaabcb-[6] = 6
ANababbcbObbcccabcbacAbOcaabcAaabacacaabcb-[7] = 7
ANcababbcbbbcccabcbacAbccabObOcabacacaabcb-[8] = 5
ANaabbcbObbbcccabcbacAbOcAaAbaaabacacaabcb-[9] = 7

Generation N: 13
01234567890123456789001234567890123456789O
ANaabbcbObbbcccabcbacAbOcAaAbaaabacacaabcb-[0] = 7
ANabcaabbcbbcccabcbacAbOcbcAaAbababacacaabcb-[1] = 6
ANacbabbcbbbcccabcbacAbOcaabcAaabacacaabcb-[2] = 6
ANababbcbObbcccabcbacAbObOccaababacacaabcb-[3] = 6
AaANbAaANcbbcccabcbacAbOAOcAaAbababacacaabcb-[4] = 6
ANacababbcbbcccabcbacAbOcAaAbaaabacacaabcb-[5] = 6
ANcababbcbbbcccabcbacAbcabObOcaabacacaabcb-[6] = 5
ANcababbcbbbcccabcbacAbccabObOcabacacaabcb-[7] = 5
AcabAbaaANbbcccabcbacAbOOcAaAbaabacacaabcb-[8] = 8
AccaAbaaANbbcccabcbacAbOOcAaAbaabacacaabcb-[9] = 7
```

图 3.14.求解 Majority(a,b,c)函数的一个初始种群及通过 IS 转座产生的后代。由 OR 连接的子表达式树编码的染色体。注意后代与其初始种群中的祖先均不完全相同。还要注意后代的基因组中出现了重复序列。第 13 代找到的完美解（8 号染色体）和其母体（第 12 代的 5 号染色体）用黑体标识，导致该完美解的转座见图 3.15。

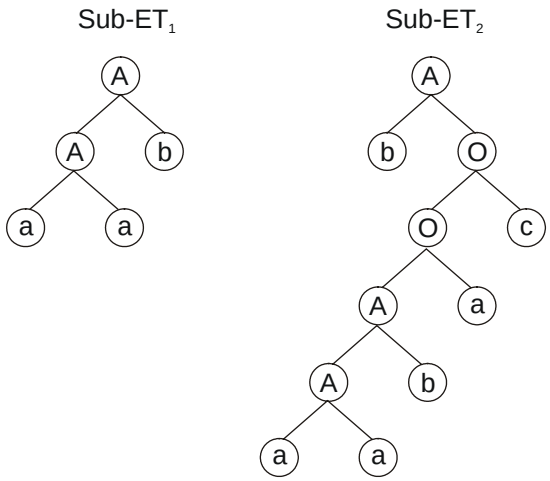
里使用 0.1 的转座概率和由 3 个不同长度的 IS 元素构成的集合。转座算子随机地选取染色体，IS 元素，目标位置，以及转座子的长度。

图 3.14 所示为双基因个体构成的两个种群，作为示例，我们只采用IS转座来引入变化，我们选择的p_{is}为 1.0。进一步，采用 3 组长度分别为 2，3，5 的IS元素。如图 3.14 所示，仅仅采用该算子系统就可以进化并找到解。的确，第 13 代的 8 号染色体就是多数函数问题的一个完美解。该染色体是第 12 代的 5 号染色体的后代。转座操作引起这个完美解的产生的过程见图 3.15。

如图 3.15 所示，在转座过程中，IS 元素被复制到目标基因的头部，一个小的序列块（1 号基因中的 14 到 16 位）被激活，并跳跃到插入位置，1 号基因的 1 号池（0 位与 1 位之间）。其结果是，一个转座子的拷贝出现在插入位置。注意，头部中最后位置上删除了与 IS 元素长度相同的符号序列（这里，原序列被删除）。因此，虽然进行了插入操作，但是染色体的结构组织并没有发生变化，然而，IS 转座所产生的所有个体都是语法上正确的程序。

a.
```
01234567890123456789001234567890123456789O
AAbaaANcaabbcc**cab**cbacAbOOcAaAbaabacacaabcb-[m] = 7
A**cab**AbaaANbbcc**cab**cbacAbOOcAaAbaabacacaabcb-[d] = 8
```

b.



c.

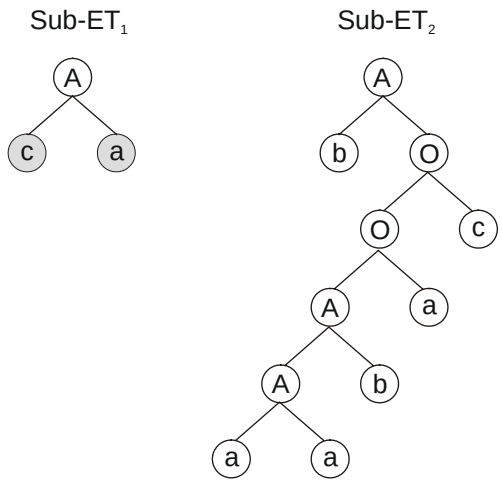


图 3.15.IS转座及其效果示例。**a)**一次IS转座，其中转座子用黑体标识。注意目标基因头部的最后有包含 3 个元素的序列被删除了。还要注意，这里转座子最终在子染色体中被复制。**b)**编码在母代染色体中的子表达式树（IS转座之前）。**c)** 编码在子代染色体中的子表达式树（IS转座之后）。转座子节点现在用灰色标识。注意插入使sub-ET$_1$发生了很大变化，产生的一个新的子表达式树比其母体少 2 个节点。

需要注意，转座也是一个宏变异算子，会导致表达式树深刻的修饰。显然，插入位置越靠前，变化越明显。例如，图 3.15 所示的子表达式树由于转座而缩短了两个节点。

你肯定已经注意到我所选择的用来解释遗传算子的机理和效果的例子，所产生后代的适应度总是更好。然而，请记住，在绝大多数的情况下，它们的作用总会导致适应度较差的个体甚至存在缺陷的个体。但是，在自然界中，进化正是因为这些极其稀有，非常不可能的事情才会发生。

3.3.3.2.根转座

所有的 RIS 元素都从一个函数开始，因此是选自头部分中的序列。因此在头中任选一点，沿基因向后查找，直到发现一个函数为止。该函数成为 RIS 元素的起始位置。如果找不到函数，则变换不作任何操作。该算子随机选取染色体，需要修饰的基因，RIS 元素及其

长度。一般地，可以采用 0.1 的根变换概率和由三组不同长度的 RIS 元素集合。

在图 3.16 所示的例子里，采用p_{ris}＝1.0，并且 3 组RIS元素的长度分别为 2，3，5。为了将分析RIS转座影响的分析过程简化，我们暂时不使用其它的搜索算子。从图 3.16 所示的结果我们可以看到，仅仅采用这种算子就可以使得种群进化。这里，到第 17 代的时候就找到了一个Majority(a,b,c)函数的完美解（8 号染色体）。该染色体是第 16 代的 5 号染色体的后代。这里，RIS元素 1 号基因中的“ AcAaa ”转座到基因的起始位置（图 3.17）。像IS转座中一样，为了保持染色体的结构组织不变，头部中最后的位置上删除了与转座子长度相同的RIS元素。这里，删除的序列是“ AcAaa ”。

需要注意的是，这个破坏性极强的算子也能形成重复地形成简单的序列，例如图 3.16 所示的后续代中的序列（Aa）$_n$和（Oc）$_n$。有趣的是，DNA中充满了重复的序列。实际上，在某些真核细胞中，40%以上的DNA都是由小的重复序列构成的。这些序列中的大部分甚至没有被转录过，但是有些基因中也点缀着一些重复基因的胰岛。

3.3.3.3.基因转座

在基因转座中，整个基因起到转座子的作用，而且自己转座到染色体的开始位置。与其它类型的转座不同，在基因变换中原始位置处的转座子（基因）被删掉。该方法能够保证染色体的长度不变。

显然，基因转座只能改变基因的位置，因为子表达式树是由可交换的函数连接的，这在较短的运行中对适应过程没有任何贡献。但是，需要注意，当子表达式树的连接函数是非交换函数的时候，基因的顺序就有影响，这时基因转座也是一个宏变异算子，大部分情况下产生的个体适应度变差，甚至是不可存活的个体。但是，当基因转座和重组相结合的时候就变得非常有趣，因为它不仅允许基因的复制，还允许基因或者更小的基因块的概化重组。

因此，为了演示的需要，同时也因为进化过程中子表达式树是由可交换的函数连接的，我们准备将基因转座和基因重组结合起来使用（见后面的 3.3.4.3 节）。图 3.18 所示的种群是 9 代适应过程的产物，其中仅采用p_{gt}=0.2 的基因转座和p_{gr}=0.2 的基因重组作为遗传变化的来源。这意味着新事物来源就是将已经存在于初始种群中的基因搬来搬去（作为比较，图 3.18 也给出了初始种群）。请注意，这些基因是如何在染色体中间分散开来，占满所有可以占用的位置的。只有基因转座能够将基因在染色体中移动。的确，基因转座在很多时候的出现是为了将基因调换位置。下面的例子就是这样的情况：

```
012345678900123456789001234567890
AOabccbbabcAOabccbbabc ANAANacbaba -[m] = 7
ANAANacbaba AOabccbbabcAOabccbbabc -[d] = 7
Generation N: 0
012345678901234567890012345678901234567890
NAccbAaOccaaaaabcccccbOAAaANObAbababcacccab-[0] = 4
NOONAONbAAcbbaabbbbccAOONaOacAOaccbaacacac-[1] = 4
AbabNcOcacaaaccbbbcccNcAbOaObcOabbbccbbcaa-[2] = 3
NNAcaNNANbccbcbbbacbcOacAabaNccbbbacbcbbcb-[3] = 6
ONbcaNNcAAccbcbbcaabcAaOaabbbAbaabccabbbca-[4] = 5
NObcNNaaNbabccccaccacNbcAbcNaAbacaaabcabab-[5] = 2
NOAacOcbbAacbaacaaabcAOacNaaabOcacccabbcbb-[6] = 4
ONaAcaNcOacbbbacbccacOcOAOcNcANcbcaabbbaba-[7] = 4
AOAONbAObbbbcbbacbbcbOONAbANAcbbcbabcccaab-[8] = 4
ANbbcbaaccaaaaccbabcbAOaaacObaObabccaacaba-[9] = 6
```

```
...

Generation N: 16
01234567890123456789001234567890123456789O
AOcAcAOccNaaaaabccccbAbaabAbaOAababcacccab-[0] = 7
AaAcAaOAaOaaaaabccccbAOAbaObAObababcacccab-[1] = 7
OcOccAaOccaaaaabccccbAaAAbaabAaababcacccab-[2] = 7
AaOccAcAaOaaaaabccccbAOObAAOAObababcacccab-[3] = 7
OAaaaAaAaAaaaaabccccbObAObAAbaOababcacccab-[4] = 6
OAaOaAaAcAaaaaabccccbAAbObAObAAababcacccab-[5] = 7
AaAaAcAaOAaaaaabccccbObAObAAbaOababcacccab-[6] = 7
OccAcAaOccaaaaabccccbObAAOAObAAababcacccab-[7] = 6
OccAaOccAcaaaaabccccbObAAOAObAAababcacccab-[8] = 6
OccOccAaOcaaaaabccccbAaAAbaabAaababcacccab-[9] = 7

Generation N: 17
01234567890123456789001234567890123456789O
OccOccAaOcaaaaabccccbAaAAbaabAaababcacccab-[0] = 7
AaOAaAaAcAaaaaabccccbObAObAAbaOababcacccab-[1] = 6
OAAaAaAcAaaaaaabccccbObAObAAbaOababcacccab-[2] = 6
AaOcOccAaOaaaaabccccbAaAAbaabAaababcacccab-[3] = 7
OaAaAOAaOaaaaaabccccbAAbObAObAAababcacccab-[4] = 7
OccAaOccOcaaaaabccccbAaAAbaabAaababcacccab-[5] = 7
AaAaAcAaOAaaaaabccccbAbObAObAAbababcacccab-[6] = 7
OcOccAaOccaaaaabccccbAAAaAAbaababababcacccab-[7] = 7
AcAaaOAaOaaaaaabccccbAAbObAObAAababcacccab-[8] = 8
AaOccAcAaOaaaaabccccbAAOAOObAAOababcacccab-[9] = 6
```

图 3.16.求解 Majority(a,b,c)函数的一个初始种群及通过 RIS 转座产生的后代。由 OR 连接的子表达式树编码的染色体。注意后代与其第 0 代中的祖先均不完全相同。还要注意后代的基因组中出现了重复序列。第 17 代找到的完美解（8 号染色体）和其母体（第 16 代的 5 号染色体）用黑体标识，导致该完美解的转座见图 3.17。

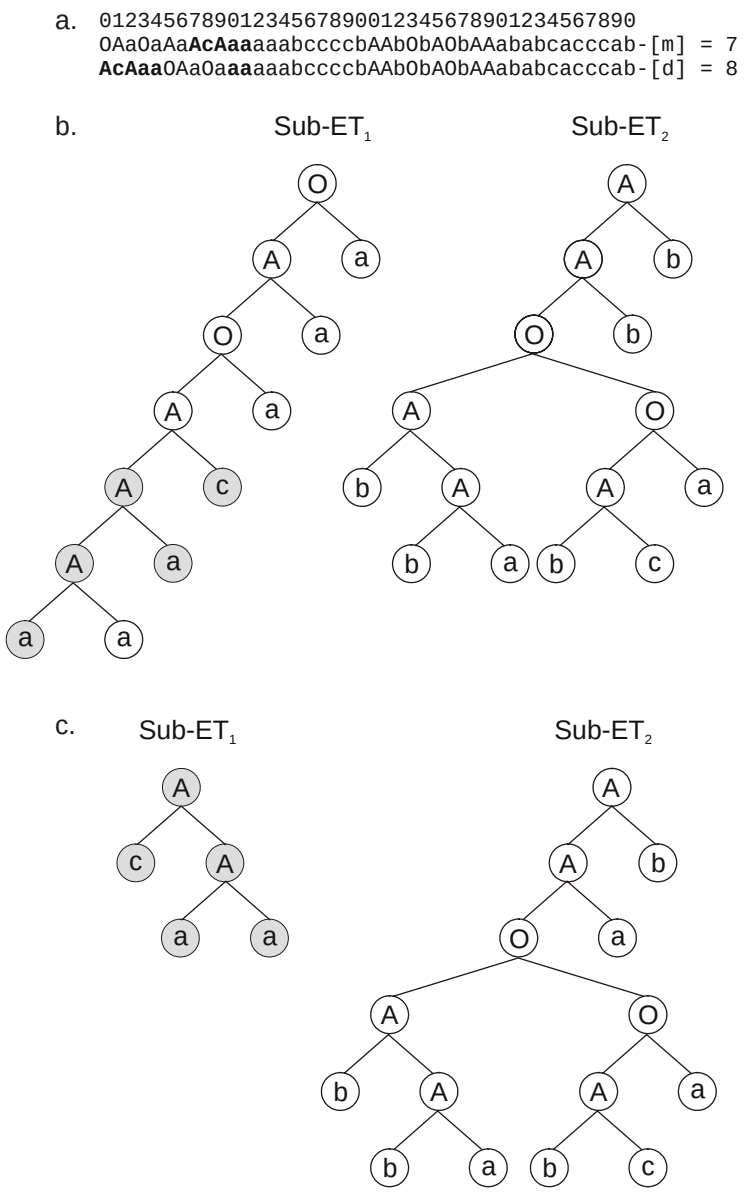


图 3.17.RIS转座及其效果示例。**a)**一次IS转座，其中转座子用黑体标识。注意目标基因头部的最后有包含 3 个元素的序列被删除了。还要注意，这里转座子中只有一部分最终在子染色体中被复制。它的其它元素被删除。**b)**编码在母代染色体中的子表达式树（RIS转座之前）。**c)** 编码在子代染色体中的子表达式树（RIS转座之后）。转座子节点现在用灰色标识。注意插入使sub-ET_1发生了很大变化，产生的一个新的子表达式树比其母体少 8 个节点。

这里，母代染色体中的 3 号基因（第 7 代的 3 号染色体）移到了染色体的开始位置，构成第 8 代的 5 号染色体（见图 3.18）。从结构上说，这个染色体与原染色体并不相同，但是从数学的角度来说，它们编码的表达式是等价的。重要的是在不同位置存在相同基因的染色体可以重组，也许可以用两个相同的基因产生新的染色体。例如，第 8 代找到的一个完美解（3 号染色体）内部就有两个相同的基因。的确，如图 3.18 所示，还有其它一些染色体（第 7 代的 0 号和 3 号染色体，第 8 代的 1 号，3 号和 5 号染色体）内部也含有两个相同的基因。这些染色体中有一个就是多数函数问题的完美解。其表达式见图 3.19。

虽然我所选择的用来演示基因转座和基因重组效果的例子也找到了多数函数的一个正确解，但是这些算子的变换能力还是非常有限的，特别是当种群规模较小的时候，更是如

```
Generation N: 0
01234567890012345678900123456789O
ObOaAbcabbcOcbccbbccbcAOANacccabc-[0] = 5
AbObNbbbbaaAAbaNbccbbaOANNObabcab-[1] = 4
OaaAOaaaaacOAbObcbccacNaAObaccbac-[2] = 4
AOabccbbabcAcOAbacbcbbAcNAAcababb-[3] = 7
AaAAbcbcbabOabbbbbaaccNbObNabbcbb-[4] = 4
ANAANacbabaOOcNNcbbcacOAOabaabbbb-[5] = 4
AAcaObaacaaAAbbAbaabcaNabObcbcccc-[6] = 5
AbNNccacabbNaccNbcaabaOOAaNcbabaa-[7] = 4
OcNaccabaacAacAAbbbccbANNAAbcbaac-[8] = 3
AacbbbbccaaAObbacacabcOcaabcbbcab-[9] = 5

                    ...

Generation N: 7
01234567890012345678900123456789O
AOabccbbabcAOabccbbabcANAANacbaba-[0] = 7
AacbbbbccaaAObbacacabcOcaabcbbcab-[1] = 5
AcNAAcababbAcOAbacbcbbAOabccbbabc-[2] = 7
AOabccbbabcAOabccbbabcANAANacbaba-[3] = 7
AcOAbacbcbbAOabccbbabcAcNAAcababb-[4] = 7
AacbbbbccaaAObbacacabcOcaabcbbcab-[5] = 5
AcNAAcababbOAOabaabbbbAcOAbacbcbb-[6] = 6
AcOAbacbcbbAOabccbbabcAcNAAcababb-[7] = 7
AOabccbbabcAcOAbacbcbbAcNAAcababb-[8] = 7
AacbbbbccaaAObbacacabcOcaabcbbcab-[9] = 5

Generation N: 8
01234567890012345678900123456789O
AOabccbbabcAcOAbacbcbbAcNAAcababb-[0] = 7
AOabccbbabcAOabccbbabcANAANacbaba-[1] = 7
AcNAAcababbAcOAbacbcbbAOabccbbabc-[2] = 7
AOabccbbabcAOabccbbabcAcOAbacbcbb-[3] = 8
AcOAbacbcbbAOabccbbabcAcNAAcababb-[4] = 7
ANAANacbabaAOabccbbabcAOabccbbabc-[5] = 7
AcNAAcababbOAOabaabbbbANAANacbaba-[6] = 6
AOabccbbabcAcOAbacbcbbAcNAAcababb-[7] = 7
OcaabcbbcabAacbbbbccaaAObbacacabc-[8] = 5
AcOAbacbcbbAOabccbbabcAcNAAcababb-[9] = 7
```

图 3.18.求解 Majority(a,b,c)函数的一个初始种群及通过基因转座产生的后代。由 OR 连接的子表达式树编码的染色体。注意第 8 代中找到的完美解（3 号染色体）有一对重复基因（1 号基因和 2 号基因）。该个体的祖先基因一直跟踪到第 0 代，而且其后代均已给出。注意这些基因是如何分散到整个基因组中去的，这些基因跳跃到所有可能的位置而且经常被复制。

a. 012345678900123456789001234567890
AOabccbbabcAOabccbbabcAcOAbacbcbb-[3] = 8

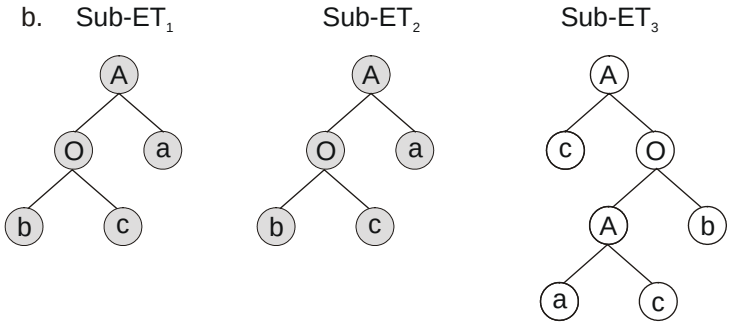


图 3.19.一个 Majority(a,b,c)函数的包含一对重复基因（用黑体标识）的完美解。所有这些重复基因都是在基因转座和重组的联合作用下产生的。**a)**该染色体。**b)**编码在染色体中的子表达式树。

此（例如共有 500 个个体的时候）。值得强调的是，这些算子都不能产生新的基因，它们只是通过不同的方式来将基因移动或者重组而已。通过这种方式产生多样性的系统只能在采用巨大的种群的情况下才能找到解，因为这样的话，所有的基因都已经存在于初始种群中。

3.3.4.重组

在 GEP 中有三种重组：单点重组，两点重组，基因重组。在所有的情况中，两个随机选中的父代染色体配对并相互交换部分成分。

3.3.4.1.单点重组

进行单点重组的时候，父代染色体相互配对并在相同的位置切断，两个染色体相互交换重组点之后的部分。在 GEP 中，重组操作总是涉及到两个父代染色体并产生两个新的个体。通常来说，这两个新的染色体互不相同，与父代染色体也不相同。

为了演示单点重组的工作机理，图 3.20 给出了一次运行中得到的 3 个种群，包括 1 个初始种群和 2 个较晚代的种群。在这个试验中，为了清楚地分析单点重组算子的效果，仅仅采用 $p_{1r} = 0.8$ 的单点重组作为基因变化的来源。第 13 代的 1 号和 6 号染色体是第 12 代的 6 号和 7 号染色体重组的后代。这里，交叉发生在染色体的 1 号基因的 1 号点处（0 号位和 1 号位之间），染色体相互交换 1 号基因的部分和整个 2 号基因（图 3.21）。

从这 4 个染色体的表达式（父代染色体和后代染色体）可以看出单点重组算子的影响有多么大。这里，第一个基因在紧跟开始位置的地方被切断，剧烈地改变了子表达式树的形状，而第二个基因没有受影响。其中一个新产生的染色体（6 号染色体）和它的父代个体中较差的个体一样普通，另一个新产生的个体则比其父代染色体的适应度要好，实际上它就是当前问题的完美解。

值得强调的是，GEP 染色体可以在基因组中的任何位置交叉，持续破坏旧的基因块，并生成新的基因块。进一步，由于 GEP 染色体的多基因本性和绝大部分基因中的非编码区域的存在，这个基因和完整的 ORF 可以在父代染色体之间相互交换。因此，单点重组算子的破坏性倾向（切断基因块）与它的保守性倾向相互依存（基因和 ORF 的互换），使得单点重组成为一个非常平衡的遗传算子。进一步说，像所有其它的重组算子一样，当它与基因转座一起使用的时候，它也能够产生重复的基因。

```
Generation N: 0
012345678012345678
OccbccbacAOAccaaaa-[0] = 6
AcbcacabcAcAAcacac-[1] = 7
AbNNabbccOaacaacbc-[2] = 6
OcbObcacaAAObbabbc-[3] = 6
ONbNaabbcOaOaaaccb-[4] = 6
AcaccbcaaNNcAcabcb-[5] = 6
AOcNaabacONOcccaac-[6] = 4
NOacccacaOOONbbbcb-[7] = 4
AAcNaabbcAONOccaca-[8] = 4
ObcbcbacbNbObbbaaa-[9] = 4

                ...
```

```
Generation N: 12
012345678012345678
AcbcacaccAcAAcacbc-[0] = 7
AcbcacacaAAObbabbc-[1] = 6
AcbcacacaAcAAcacbc-[2] = 7
AcbcacaccAcAAcacbc-[3] = 7
AcbcacacaAcAAbabbc-[4] = 6
AcbcbcacaAcAAcacbc-[5] = 7
NObcacacaAcAAcacbc-[6] = 4
AcacccacaAAOAcacbc-[7] = 7
AcbcacacaAAObcacbc-[8] = 6
AcbcacacaAcAAcacbc-[9] = 7
```

```
Generation N: 13
012345678012345678
AcbcacacaAcAAcacbc-[0] = 7
AObcacacaAcAAcacbc-[1] = 8
AcbcacaccAcAAcacbc-[2] = 7
AcbcacaccAcAAcacbc-[3] = 7
AcbcacaccAcAAcacbc-[4] = 7
AcbcacaccAcAAbabbc-[5] = 6
NcacccacaAAOAcacbc-[6] = 4
AcbcacacaAAObbabbc-[7] = 6
AcbcacacaAcAAcacbc-[8] = 7
AcbcacacaAAObbabbc-[9] = 6
```

图 3.20.求解 Majority(a,b,c)函数的一个初始种群及通过单点重组产生的后代。由 OR 连接的子表达式树编码的染色体。第 13 代找到的完美解（1 号染色体）是前一代的 6 号染色体和 7 号染色体的一个后代（也用黑色标识）。它们的另一个后代也加亮显示。注意后代与其第 0 代中的祖先均不完全相同。导致该完美解的单点重组见图 3.21。

3.3.4.2.两点重组

进行两点重组的时候，父代染色体相互配对，在染色体中随机选择两个点，将染色体切断。两个染色体相互交换重组点之间的部分，形成两个新的子代染色体。

图 3.22 所示为两点重组的工作过程。该图显示了一次运行的初始种群和紧接下来的一代种群后代。在这个试验中，为了清楚地分析单点重组算子的效果，仅仅采用 $p_{2r} = 0.8$ 的单点重组作为基因变化的来源。请注意，早在第 1 代就已经找到了一个完美解（3 号染色体）。的确，当采用重组作为唯一的基因变化的来源的时候，完美解要么在运行的早期就可以找到，要么就根本找不到，因为随着时间的推移，种群的多样性越来越差（见第 7 章）。请注意，第 1 代的 1 号和 3 号染色体是初始种群中 4 号和 6 号染色体的后代。

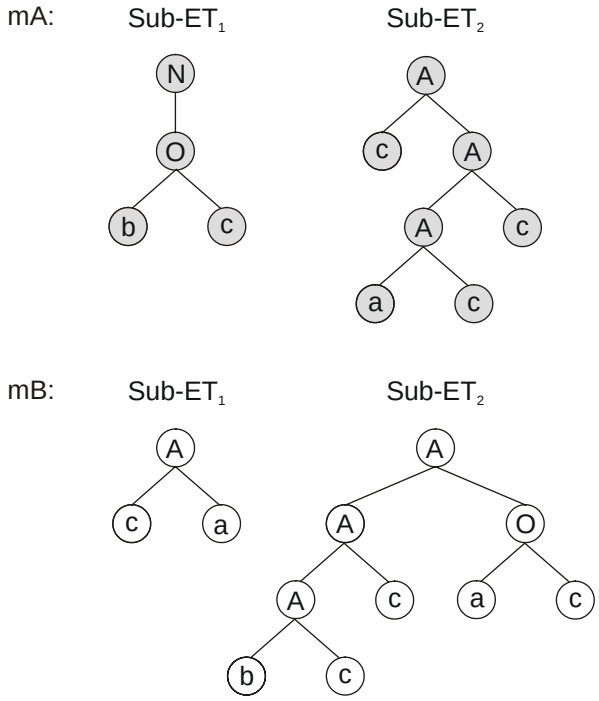
产生完美解的过程如图 3.23 所示。这里，父代染色体相互交换 7 号点（1 号基因的第 6 位和第 7 位之间）和 14 号点（2 号基因的第 2 位和第 3 位之间）之间的遗传元素。注意，在两个父体中的 1 号基因都在终点后被切断。事实上，GEP 染色体的非编码区域是能够被

a.

```
012345678012345678
NObcacacaAcAAcacbc-[mA] = 4
AcacccacaAAOAcacbc-[mB] = 7

012345678012345678
AObcacacaAcAAcacbc -[dA] = 8
Ncacccaca AAOAcacbc -[dB] = 4
```

b.



c.

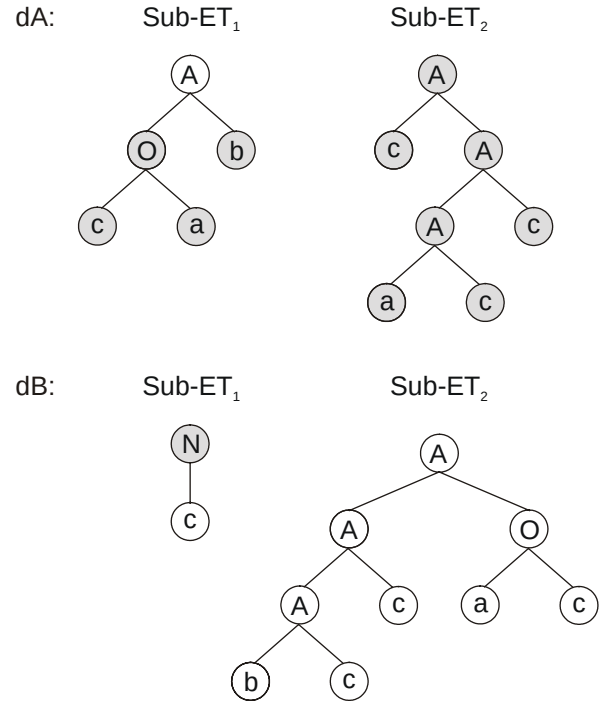


图 3.21.单点重组及其效果示例。**a)**一次单点重组。子代染色体是在 1 号点（1 号基因的 0 号位和 1 号位之间）杂交的结果。注意母代个体和子代个体均不相同。**b)**由母代染色体编码的子表达式树（重组之前）。**c)**子代染色体编码的子表达式树（重组之后）。注意子代个体 dA 是多数函数问题的一个完美解。

```
Generation N: 0
012345678900123456789O
ONOANbbcaaaNcONcaabccc-[0] = 4
ANANbaacbbcNANOAabbcbc-[1] = 5
NNNcOacbbabONcbacbbabb-[2] = 4
ANAcbbababcNcbNbabcaab-[3] = 2
AOaObbcccacObaNOacabab-[4] = 6
OOAcOaabccaAacbNcbaccb-[5] = 5
OcbbAcccbccAAOObacbaab-[6] = 6
NbAANccbacaAbNNNbbcbbc-[7] = 2
AaONbaababcAbNNAcacaaa-[8] = 6
NOAcObaccbaAObbObcacca-[9] = 4

Generation N: 1
012345678900123456789O
AaONbaababcAbNNAcacaaa-[0] = 6
OcbbAccccacObaObacbaab-[1] = 5
AOaObaabccaAacbNacabab-[2] = 6
AOaObbccbccAAONOacabab-[3] = 8
ONOANbbcaaaNcONcaabccc-[4] = 4
OOAcOaabccaAaNONcbaccb-[5] = 5
OOAcObcccacObaNOcbaccb-[6] = 5
ANANbaacbbcNAcbAabbcbc-[7] = 3
ANANbaacbbcNANOAabbcbc-[8] = 5
OaOANbbcaaaNcONcaabccc-[9] = 5
```

图 3.22.求解 Majority(a,b,c)函数的一个初始种群及通过两点重组产生的后代。由 OR 连接的子表达式树编码的染色体。第 1 代找到的完美解（3 号染色体）是第 0 代的 4 号染色体和 6 号染色体的一个后代（用黑色标识）。它们的另一个后代也加亮显示。注意 1 号染色体的适应度比两个母体都差而 3 号染色体的适应度则明显高于两个母体。

截断以完成交叉而又不影响 ORF 的理想区域。我们还看到，这些区域也是中性变异积累的理想区域。然而 6 号染色体的 2 号基因在终点之前被切断，从而导致子表达式树发生明显改变。进一步说，这些染色体重组时，4 号染色体的非编码区域被激活，从而构成第 1 代中找到的完美解（3 号染色体）。

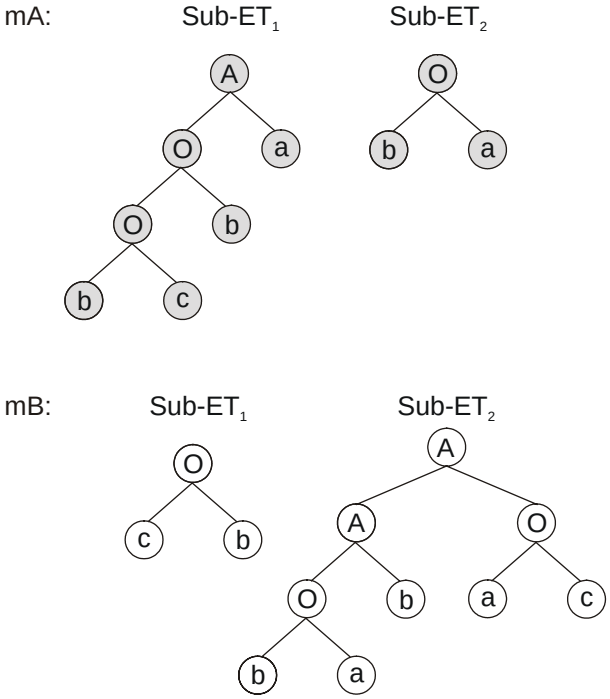
需要强调的是，两点重组的比单点重组的破坏性更强，因为它对遗传元素的重组更加彻底，持续破坏旧的基因块，并生成新的基因块（关于两点重组与其它重组算子的比较见第 7 章）。但是与单点重组类似，两点重组也有保守的方面，并且更加善于交换整个基因和 ORF。最后，当两点重组与基因转座一起使用的时候，也能够产生重复的基因。

尽管如此，如果我们的目的是要进化得到一个优良解，那么就不能仅仅使用单点和两点重组作为遗传变化的来源，因为它们容易导致种群同质化（详见第 7 章的讨论）。但是，在与转座、变异同时使用的时候，这些算子可以称为遗传变化的绝佳来源，并且几乎足以解决所有的问题。

a.
```
01234567890012345678900
AOaObbcccacObaNOacabab-[mA] = 6
OcbbAcccbccAAOObacbaab-[mB] = 6

01234567890012345678900
AOaObbccbccAAONOacabab-[dA] = 8
OcbbAccccacObaObacbaab-[dB] = 5
```

b.



c.

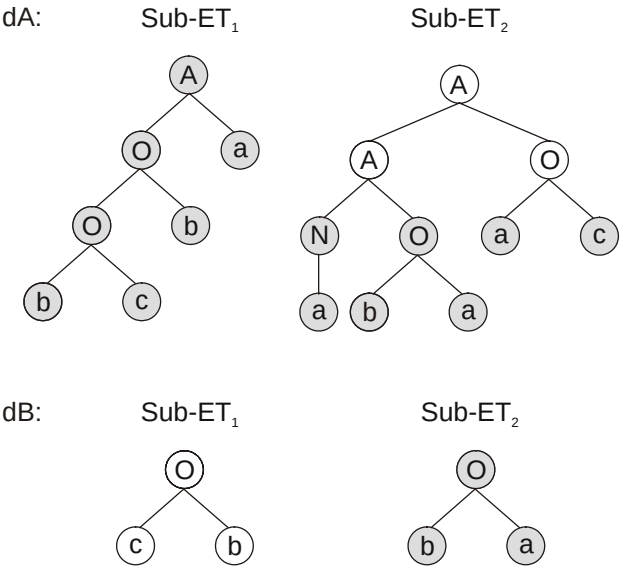


图 3.23.两点重组及其效果示例。**a)**一次两点重组。子代染色体是在 7 号点（1 号基因的 6 号位和 7 号位之间）和 14 号点（2 号基因的 2 号位和 3 号位之间）杂交得到的。注意母代个体和后代个体均不相同。**b)**由母代染色体编码的子表达式树（重组之前）。**c)** 子代染色体编码的子表达式树（重组之后）。注意子代个体 dA 比起母代都好很多，事实上它是多数函数问题的一个完美解。

3.3.4.基因重组

在 GEP 的第三种重组中，两个染色体中的整个基因相互交换，形成的两个子代染色体

含有来自两个父体的基因。

```
Generation N: 0
01234567801234567801234567
OAOcaacacAOaAcbbcbAaAAbabbb-[0] = 6
NNcaabbabONbcbacacOAOObccbb-[1] = 4
NcaacaaacNcAaaccbcOcaaacbbb-[2] = 4
OcbaacbccAOaccbccbNONNbcbbb-[3] = 6
ONAObcbcaNANccacabNAbObacab-[4] = 4
AOaaaabacAbbObcabaNcbAacacb-[5] = 5
NONOabaaaNaObccbbbNccabcbba-[6] = 2
NAaAaacbbNANbcbacaAAONacbaa-[7] = 3
NaAAbcbbaNaOAcbacbNcAaababb-[8] = 2
AcaabbcabNAcacaacbNNOAcbbbc-[9] = 4

Generation N: 1
01234567801234567801234567
OcbaacbccAOaccbccbNONNbcbbb-[0] = 6
OcbaacbccAOaccbccbNcbAacacb-[1] = 4
NONOabaaaAOaccbccbNONNbcbbb-[2] = 7
AcaabbcabNaObccbbbNccabcbba-[3] = 4
AcaabbcabAOaAcbbcbNNOAcbbbc-[4] = 6
NONOabaaaNaObccbbbNccabcbba-[5] = 2
OAOcaacacNAcacaacbAaAAbabbb-[6] = 4
NONOabaaaNAcacaacbNNOAcbbbc-[7] = 4
AOaaaabacAbbObcabaNONNbcbbb-[8] = 6
OAOcaacacAOaAcbbcbAaAAbabbb-[9] = 6

Generation N: 2
01234567801234567801234567
NONOabaaaAOaccbccbNONNbcbbb-[0] = 7
OAOcaacacAOaAcbbcbAaAAbabbb-[1] = 6
AOaaaabacAbbObcabaNONNbcbbb-[2] = 6
AcaabbcabAOaAcbbcbNONNbcbbb-[3] = 8
NONOabaaaAOaccbccbAaAAbabbb-[4] = 7
AOaaaabacAbbObcabaNNOAcbbbc-[5] = 5
OAOcaacacAOaAcbbcbNccabcbba-[6] = 4
NONOabaaaNaObccbbbAaAAbabbb-[7] = 4
NONOabaaaNaObccbbbNONNbcbbb-[8] = 3
NONOabaaaAOaccbccbNccabcbba-[9] = 4
```

图 3.24 求解 Majority(a,b,c)函数的一个初始种群及通过基因重组产生的直接后代。由 OR 连接的子表达式树编码的染色体。第 2 代找到的完美解（3 号染色体）是前一代的 4 号染色体和 8 号染色体的一个后代（用黑色标识）。它们的另一个后代也加亮显示。注意 5 号染色体的适应度比两个母体都差而 3 号染色体的适应度则明显高于两个母体。事实上 3 号染色体是多数函数的一个完美解。

a.

```
012345678012345678012345678
AcaabbcabAOaAcbbcbNNOAcbbbc-[mA] = 6
AOaaaabacAbbObcabaNONNbcbbb-[mB] = 6

012345678012345678012345678
AcaabbcabAOaAcbbcbNONNbcbbb-[dA] = 8
AOaaaabacAbbObcabaNNOAcbbbc-[dB] = 5
```

b.

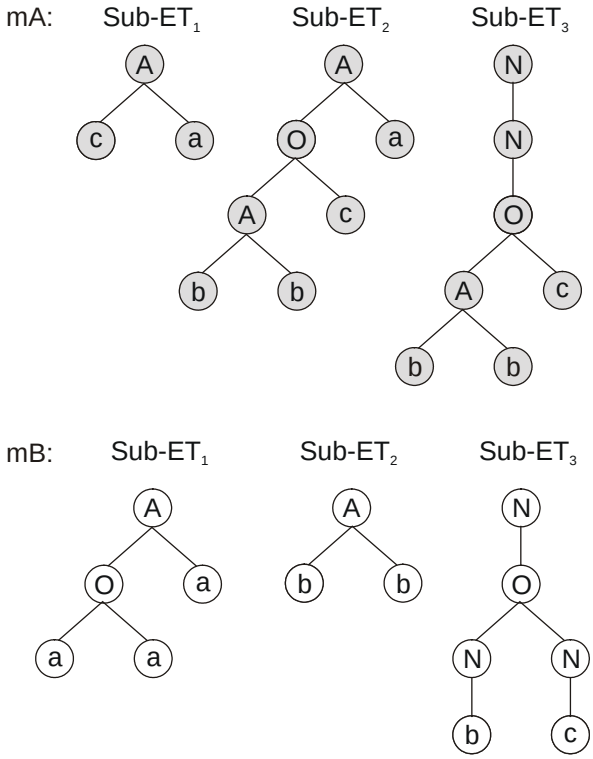


图 3.25.基因重组及其效果示例。**a)**一次基因重组。在此，母代染色体交换 3 号基因，形成两个新的子代染色体。注意母代个体和后代个体均不相同。**b)**由母代染色体编码的子表达式树（重组之前）。**c)** 子代染色体编码的子表达式树（重组之后）。

图 3.24 所示的最后的两个种群是进化过程的结果，其中仅采用p_{gr}=0.8 的基因重组作为引入变化的唯一来源。对于这个简单的问题，同样是这个多数函数问题，不可能仅仅通过调换那些存在于初始种群中的基因的位置就能够找到问题的正确解。显然，对于更加复杂的问题，除了采用巨大的种群以外，我们只能碰运气。还要注意，这种算子的作用下，染色体中的基因从来不变换位置，而总是占有它们原来的位置。

图 3.25 所示的就是产生图 3.24 中完美解的基因重组过程。这里，第 1 代中的 4 号基因和 8 号染色体中的 3 号基因相互交换，得到两个新的子代染色体（第 2 代的 3 号和 5 号染色体）。注意，5 号染色体的适应度要比其父体的适应度差，而 3 号染色体的适应度则远远好于其父代染色体。的确，这个染色体具有最大适应度并且就是多数函数的一个完美解。

值得强调的是，该算子不能产生新的基因：基因重组所产生的个体是已经存在的基因的不同排列。显然，如果仅仅采用基因重组作为引入遗传变化的唯一来源的话，要解决更加复杂的问题将需要使用非常巨大的种群，这样才能保证必要的基因多样性（讨论见第 7 章）。然而。GEP 的进化能力不仅仅建立在基因的重新排列的基础上（由基因重组和基因转座实现），还建立在新的遗传物质不断产生的基础上，这一过程基本上由变异和转座操作（IS 转座和 RIS 转座）完成，在较小程度上也由重组操作（单点重组和两点重组）完成。

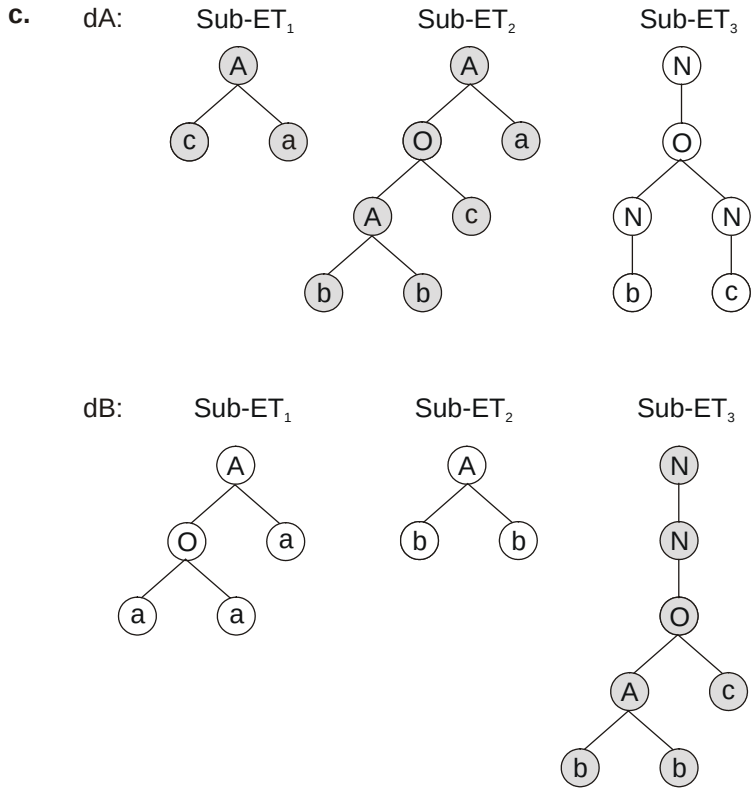


图 3.25.续

3.4.用 GEP 解决一个简单的问题

这一章主要是详细分析一次成功运行的全过程，以使读者了解种群如何逐步适应，并在这个过程中逐渐找到完美解或较优的解。

符号回归或函数发现的目标是为了找到一个表达式可以很好地解释因变量。系统的输入是一组适应度样本，其形式为 $\left(a_{(i,0)}, a_{(i,1)}, \ldots, a_{(i,n-1)}, y_i\right)$，其中 $a_{(i,0)} - a_{(i,n-1)}$ 是自变量，y_i 是因变量。我们已经看到适应度样本集 C_t 构成解的进化环境，最终发现问题的解（计算机程序或者系统输出）。

在本节这个简单的例子中，适应度样本由计算机通过一个测试函数产生。那么，本问题中我们确切知道所要解决的问题的解是什么形态。这是一个好事情，因为我们可以对遗传算子的盲目行为所产生的结果保持应有的敬意。但是，请记住，现实问题的目标函数显然是不可能预先知道的。

那么，假设我们给定曲线

$$y = \frac{a^2}{2} + 3a \tag{3.3}$$

在[-10,10]区间上随机选择的 10 个实数值，我们要找到一个在一定误差范围内适应这些值的一个函数。这里，我们有 10 组样本数据值，其形式为 (a_i, y_i)，其中 a_i 是给定区间上

的自变量的值，y_i 是相应的因变量的值（表 3.2）。这 10 组值就是适应度样本（系统的输入），它们作为选择环境，而且若干代以后适应性较好的个体很可能成为我们的问题的一个完美解。一个特定程序的适应度好坏取决于它在选择环境中的表现。

表 3.2 用于基本符号回归问题的由计算机生成的 10 个随机的适应度样本

a	f(a)
6.9408	44.91
-7.8664	7.341
-2.7861	-4.477
-5.0944	-2.307
9.4895	73.494
-9.6197	17.41
-9.4145	16.073
-0.1432	-0.419
0.9107	3.147
2.1762	8.897

要准备使用 GEP 主要需要以下五个步骤。其中第一步是适应度函数选择。对本问题，我们将采用方程（3.1a）来计算适应度，采用 100 的绝对误差作为选择范围，精度的误差为 0.01。因此，对于 10 个适应度样本而言，最大适应度值 $f_{\max}=1000$。

第二步是选择终点集 T 和函数集 F。在该问题中，显然终点集由自变量构成，即 $T=\{a\}$，要选择恰当的函数集则不那么显而易见，为了所有必须的数学运算符可以作一个不错的猜测。在此，我们将采用 4 个算术运算符。即 $F=\{+,-,*./\}$。

第三个步骤是选择染色体的组织结构：即头部和尾部的长度。本问题中我们将采用头部长度为 $h=7$，每个染色体中含 3 个基因。

第四步是选取连接函数，本问题中采用加法来连接子表达式树。

最后，也就是准备使用 GEP 的第五步是选择遗传算子集并决定其参数大小。在此，我们将综合使用前面所描述的所有遗传算子（变异，三种转座和三种重组）。

表 3.3 基本符号回归问题的参数

Number of generations	50
Population size	20
Number of fitness cases	10 (Table 3.2)
Function set	+ - * /
Head length	7
Number of genes	3
Chromosome length	45
Mutation rate	0.044
One-point recombination rate	0.4
Two-point recombination rate	0.2
Gene recombination rate	0.1
Gene transposition rate	0.1
IS transposition rate	0.1
IS elements length	1,2,3
RIS transposition rate	0.1
RIS elements length	1,2,3
Selection range	100
Precision	0.01

每次运行所使用的参数如表 3.3 所示。对于本问题，我们选择大小为 20 个个体的较小种群是为了能够全面分析进化过程中产生的所有的个体，又不必让整本书被个体编码占满。

但是，我们将看到，GEP 的一个优势就是能够采用较小的种群规模去解决较为复杂的问题，由于 Karva 表示方法非常简洁，所以我们可以分析一次运行过程中的每个个体。

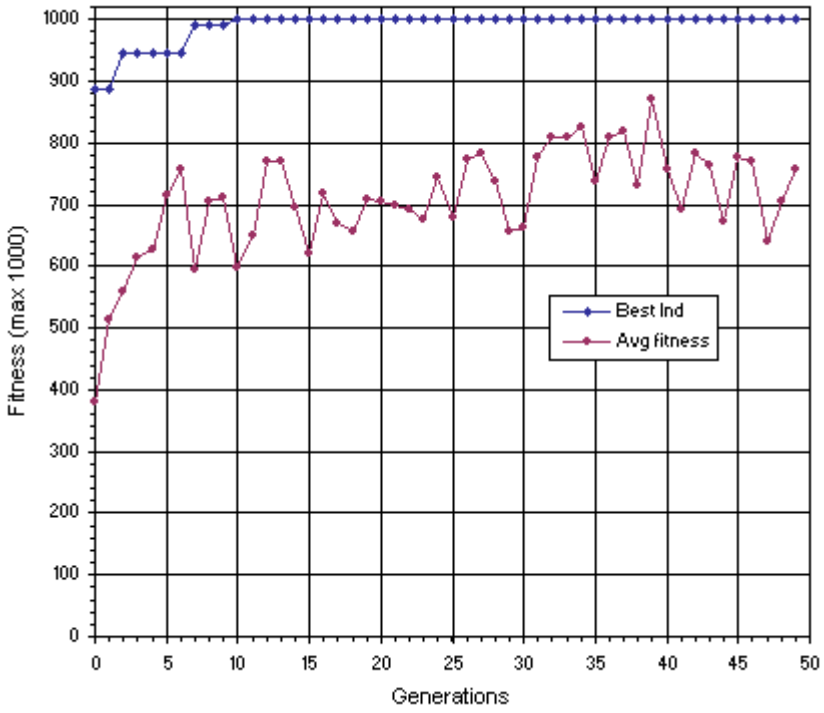


图 3.26.表 3.3 所给出的实验的一次成功运行中种群的平均适应度和最佳个体的适应度进展情况。

图 3.26 所示为我们将要分析的一次成功运行中的平均适应度和最佳适应度的变化过程。在这次运行中，在第 10 代找到一个完美解。这次运行的初始种群和特定环境下每个个体的适应度如下表 3.2 所示（本代的最佳个体用粗体标识）：

```
Generation N: 0
012345678901234012345678901234012345678901234
-/a+*+*aaaaaaaa*+++*/a+aaaaaaaa+++a*--aaaaaaaa-[ 0] = 285.2363
+a-*/+aaaaaaaaa+/*+/+*aaaaaaaa+/++++/aaaaaaaa-[ 1] = 324.4358
--a+a/*aaaaaaaa+a-/aa/aaaaaaaa***-+a+aaaaaaaa-[ 2] = 697.6004
++//a*aaaaaaaaa/aa/+*+aaaaaaaa-++*/+-aaaaaaaa-[ 3] = 711.1687
/*+-*a/aaaaaaaa*+a/-*aaaaaaaaa*////-aaaaaaaaa-[ 4] = 730.5334
-a/a/+*aaaaaaaa*+-*a*-aaaaaaaa/*/**aaaaaaaaaa-[ 5] = 256.4514
/*+*/++aaaaaaaa-/*-a-*aaaaaaaa*++--/*aaaaaaaa-[ 6] = 821.6851
**+//++aaaaaaaa*aa+++/aaaaaaaa+++a/aaaaaaaaaa-[ 7] = 816.0122
/*-/+++aaaaaaaa//-+aa/aaaaaaaa+a+--a-aaaaaaaa-[ 8] = 0
-+/a-/*aaaaaaaa-a-a+a/aaaaaaaa-/+*a//aaaaaaaa-[ 9] = 729.2897
--//+/*aaaaaaaa-***/a+aaaaaaaa+-*/*+/aaaaaaaa-[10] = 376.2071
**+/+*aaaaaaaaa/+*-/-*aaaaaaaa*a**a+aaaaaaaaa-[11] = 0
-aa-+/+aaaaaaaa+a/*-a*aaaaaaaa-a*//++aaaaaaaa-[12] = 0
-a+a+//aaaaaaaa*-aa/+/aaaaaaaa--/*a+-aaaaaaaa-[13] = 0
+/*a+/-aaaaaaaa-/*--/-aaaaaaaa**a/**aaaaaaaaa-[14] = 0
```

```
-a+*a**aaaaaaaa+-***a*aaaaaaaa-/+-+a/aaaaaaaa-[15] = 294.7556
/+-***-aaaaaaaa+a/a/+*aaaaaaaa*-*-a*-aaaaaaaa-[16] = 886.7593
/a++/--aaaaaaaa*-a/a-/aaaaaaaa*/+*+*/aaaaaaaa-[17] = 392.185
+a-+/+/aaaaaaaa*--a**/aaaaaaaa/---+/-aaaaaaaa-[18] = 311.389
/*/aa*-aaaaaaaa/-+--+*aaaaaaaa*/-+*a/aaaaaaaa-[19] = 0
```

注意 20 个个体中有 6 个都是不可存活的个体，所以它们的适应度为 0，这意味着它们或者不能在选择范围内解决一个适应度样本，或者因为除零返回了运算错误。实际上，所有的运算错误都可以采用类似的方法处理：每当一个程序给出运算错误，就将它判定为不可存活的。这意味着它将不能够将其基因传到下一代。这是对 GP 中采用可疑的保护数学运算符来产生无用程序的一种很好的替代方法。

如上面的初始种群所示，该代的最佳个体，即 16 号染色体的适应度为 886.7593。让我们来进一步分析它的效果。表 3.4 对程序的返回结果和目标值进行比较。注意，在选择的精度即 0.01 以内没有解决任何适应度样本。但是对于所有的适应度样本而言，绝对误差在选定的选择误差以内。

表 3.4 一个模型的适应度评价（第 0 代的最佳个体）

Target	Model	Error	Fitness
44.91	33.0282	11.8818	88.1182
7.341	25.0737	17.7327	82.2673
-4.477	3.09508	7.57208	92.42792
-2.307	9.88206	12.18906	87.81094
73.494	56.5148	16.9792	83.0208
17.41	38.6496	21.2396	78.7604
16.073	36.9019	20.8289	79.1711
-0.419	1.86705	2.28605	97.71395
3.147	3.32539	0.17839	99.82161
8.897	6.54412	2.35288	97.64712

这一代中最佳个体的表达式和相应的数学表达式如图 3.27 所示。注意，3 号基因没有作任何工作，因此可以将它视为一个中性基因。还要注意作为简单算术运算结果的子表达式树sub-ET_1和sub-ET_2中的数值常数的产生过程。

下面给出的是初始种群的后代的个体（本代的最佳个体用粗体标识）：

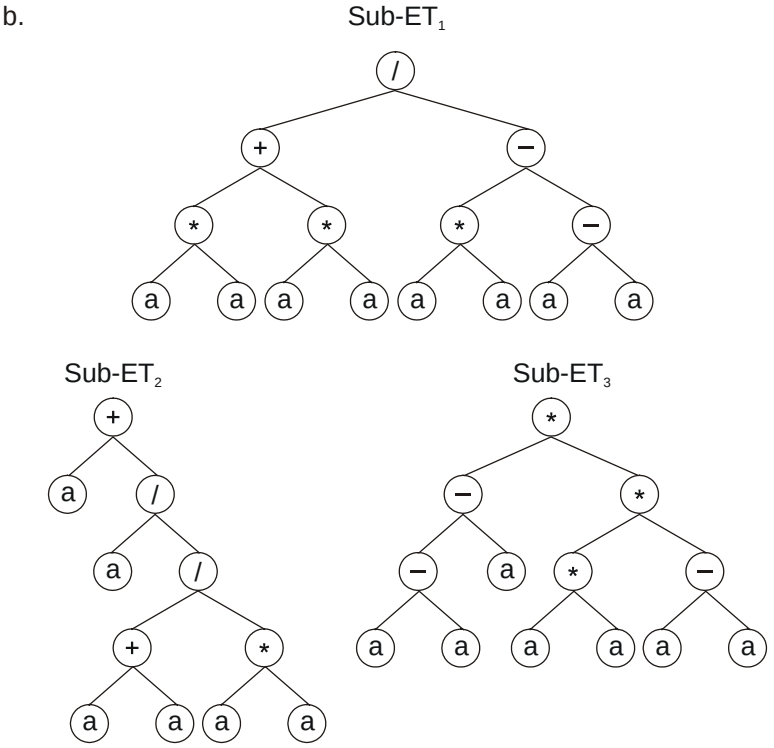
```
Generation N: 1
012345678901234012345678901234012345678901234
/+-***-aaaaaaaa+a/a/+*aaaaaaaa*-*-a*-aaaaaaaa-[ 0] = 886.7593
++//a*aaaaaaaaa-a-++a/aaaaaaaa*-/+*a/aaaaaaaa-[ 1] = 366.9803
+a/+/+/aaaaaaaa*--a**/aaaaaaaa/----/*aaaaaaaa-[ 2] = 313.1345
-+/a-/*aaaaaaaa/aa/+*+aaaaaaaa-++*/+/aaaaaaaa-[ 3] = 650.7792
-a+*a**aaaaaaaa+-***a*aaaaaaaa-/+-*aaaaaaaaaa-[ 4] = 297.1939
++//a*aaaaaaaaa**+//*/aaaaaaaa/--*+/-aaaaaaaa-[ 5] = 462.7996
*aa+++/aaaaaaaa/aa/+++aaaaaaaa-/+*a//aaaaaaaa-[ 6] = 756.707
--a+a/*aaaaaaaaaa-/a*/aaaaaaaa***--a+aaaaaaaa-[ 7] = 697.6004
**+*+*/aaaaaaaa/a++/--aaaaaaaa*-a/a-/aaaaaaaa-[ 8] = 199.4485
```

```
/*+*-*+aaaaaaaa-/*-a-*aaaaaaaa*-+-+/-aaaaaaaa-[ 9] = 794.29
-/a+*+*aaaaaaaa*++*/a+aaaaaaaa+++a*--aaaaaaaa-[10] = 285.2363
/+-***-aaaaaaaa+a/a/+*aaaaaaaa*-*-a*-aaaaaaaa-[11] = 886.7593
++//a*-aaaaaaaa/aa/+*+aaaaaaaa-++*/+-aaaaaaaa-[12] = 0
/a++/--aaaaaaaa*/-a/a-aaaaaaaa*/+*+*/aaaaaaaa-[13] = 446.7045
-+a-/*aaaaaaaaa-a-a+a/aaaaaaaa-/+*a//aaaaaaaa-[14] = 730.5334
/*+**a/aaaaaaaa*+a--*aaaaaaaaa*////-aaaaaaaaa-[15] = 334.3612
-a/a/+*aaaaaaaa*+-*a*-aaaaaaaa/*/*+a/aaaaaaaa-[16] = 314.1658
-+/a-/*aaaaaaaa-a-a+a/aaaaaaaa+++a/aaaaaaaaaa-[17] = 716.5283
+a-+/+/aaaaaaaa*-+a**+aaaaaaaa-++*/+aaaaaaaaa-[18] = 361.1277
/+-***-aaaaaaaa+a/a/a*aaaaaaaa-**-*-aaaaaaaaa-[19] = 738.8981
```

注意，虽然种群在整体上有所进步（通过比较图 3.26 中的两个种群的平均适应度），但是后代个体中没有任何个体超过前一代中的最佳个体。实际上，第 1 代中的几个最佳个体和第 0 代中的最佳个体在基因构成上完全相同。其中的第一个，也就是 0 号染色体是通过精英策略产生的。另外一个，即 11 号染色体是在没有变化的情况下繁殖出来的。

a.
```
01234567890123401234567890123401234567890123 4
/+-***-aaaaaaaa+a/a/+*aaaaaaaa*-*-a*-aaaaaaaa
```

b.



c.

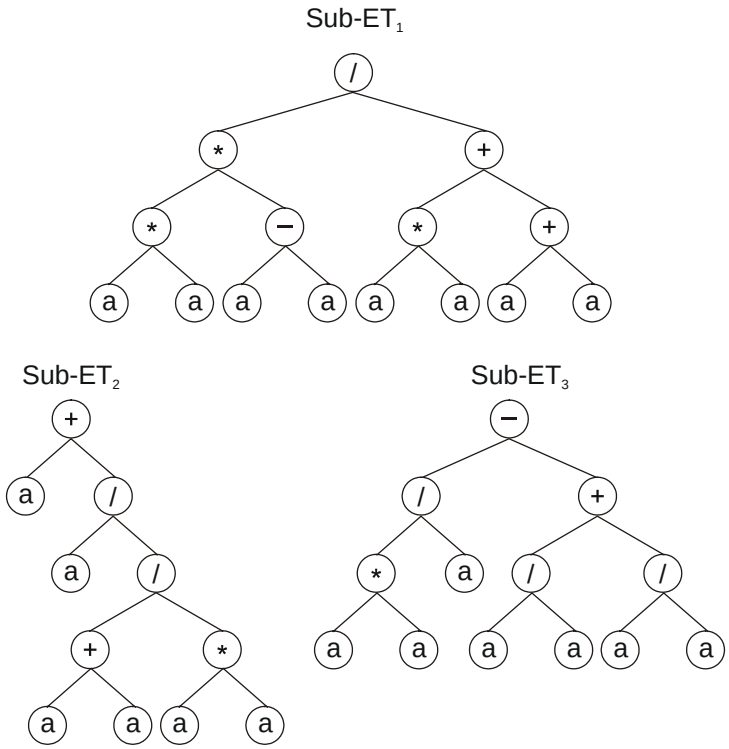
$$y = (2) + \left(a + \frac{a^2}{2} \right) + (0)$$

图 3.27.第 0 代的最佳个体（16 号染色体）。它在表 3.2 给出的适应度样本集上算得得适应度为 886.7593。**a)**该个体的染色体。**b)**每个基因所编码的子表达式树。**c)**加号连接后的相应的数学表达式（每个子表达式树的贡献用括号标识）。注意 3 号基因是一个中性基因。

在下一代中产生了一个新的个体，8 号染色体，它明显优于前一代中的最佳个体。它的表达式如图 3.28 所示。完整的种群如下图所示（本代的最佳个体用粗体标识）：

a.
```
012345678901234012345678901234012345678901234
/*+*-*+aaaaaaaa+a/a/+*aaaaaaaa-/+*a//aaaaaaaa
```

b.



c.

$$y = (0) + \left(a + \frac{a^2}{2} \right) + (a - 2)$$

图 3.28.第 2 代的最佳个体（8 号染色体）。它在表 3.2 给出的适应度样本集上算得得适应度为 945.8432。**a)** 该个体的染色体。**b)**每个基因所编码的子表达式树。注意sub-ET$_2$已经在第 0 代的最佳个体中出现过。**c)**加号连接后的相应的数学表达式（每个子表达式树的贡献用括号标识）。请再次注意存在一个中性基因（1 号基因）。

```
Generation N: 2
012345678901234012345678901234012345678901234
/+-***-aaaaaaaa+a/a/+*aaaaaaaa*-*-a*-aaaaaaaa-[ 0] = 886.7593
/+-***-aaaaaaaa+a/a/+*aaaaaaaa*-*-a*-aaaaaaaa-[ 1] = 886.7593
/+-**+-aaaaaaaa+a/a/+*aaaaaaaa*---a*-aaaaaaaa-[ 2] = 361.2925
/+a/**-aaaaaaaa+a/a/+*aaaaaaaa*-**a//aaaaaaaa-[ 3] = 0
*-*-a*-aaaaaaaa/+-*a+aaaaaaaaa***--a+aaaaaaaa-[ 4] = 822.5905
+a/+/+/aaaaaaaa*--a**/aaaaaaaa/----/*aaaaaaaa-[ 5] = 313.1345
/a++*--aaaaaaaa*/-a/a-aaaaaaaa*/+*+*/aaaaaaaa-[ 6] = 449.5403
*+-***-aaaaaaaa+a/a/a*aaaaaaaa-**-*-aaaaaaaaa-[ 7] = 256.8001
/*+*-*+aaaaaaaa+a/a/+*aaaaaaaa-/+*a//aaaaaaaa-[ 8] = 945.8432
```

```
-+a-/*aaaaaaaaa---a+a/aaaaaaaa-/+*a//aaaaaaaa-[ 9] = 611.6426
*/*/**+aaaaaaaa/a++/--aaaaaaaa*-a/a-/aaaaaaaa-[10] = 475.9649
/+-***-aaaaaaaa+a/a/+*aaaaaaaa*-a-a*-aaaaaaaa-[11] = 372.5987
-+a-/*aaaaaaaaa-a-a+a/aaaaaaaa*-+-+/-aaaaaaaa-[12] = 611.6426
/a++/--aaaaaaaa*--a/a/aaaaaaaa*/+*+*/aaaaaaaa-[13] = 458.072
*aa+/+/aaaaaaaa*/-a/a-aaaaaaaaa/+*+//aaaaaaaa-[14] = 471.6405
aa-/aa/aaaaaaaa--a+**-aaaaaaaa+**a/a/aaaaaaaa-[15] = 804.8516
++//aaaaaaaaaaa**+//*/aaaaaaaa//-a+/-aaaaaaaa-[16] = 723.8981
/+-***-aaaaaaaa+a/a/+*aaaaaaaa*-*-a*-aaaaaaaa-[17] = 886.7593
/+-***-aaaaaaaa//*-a-*aaaaaaaa*-*-a*-aaaaaaaa-[18] = 0
/+-***-aaaaaaaa/a+a/a/aaaaaaaa*-*-a*-aaaaaaaa-[19] = 831.525
```

在接下来的 4 代中，最佳个体都是第 2 代的最佳个体的原样复制品，或者仅仅是在个体上产生了很小的变化（这些染色体用粗体标识）：

```
Generation N: 3
01234567890123401234567890123401234567890123 4
/*+*-*+aaaaaaaa+a/a/+*aaaaaaaa-/+*a//aaaaaaaa-[ 0] = 945.8432
/+-***-aaaaaaaa+a/a/+*aaaaaaaa*-*aa*-aaaaaaaa-[ 1] = 886.7593
*/*/**+aaaaaaaa/a/a/a/aaaaaaaa*/*-a-/aaaaaaaa-[ 2] = 789.7097
/+-***-aaaaaaaa/a+-++/aaaaaaaaa-a-a*+aaaaaaaa-[ 3] = 830.1644
/+-***-aaaaaaaa-a/a/+*aaaaaaaa*aa-a*-aaaaaaaa-[ 4] = 886.7593
/a++/--aaaaaaaa*--a/a/aaaaaaaa/++*a//aaaaaaaa-[ 5] = 558.8105
/+-***-aaaaaaaa+a/a/+*aaaaaaaa*-*/a*/aaaaaaaa-[ 6] = 352.5538
-+a-/*aaaaaaaaa-a-a+a/aaaaaaaa//-a+/-aaaaaaaa-[ 7] = 691.6004
+a/+/+/aaaaaaaa*--a-*/aaaaaaaa--/----aaaaaaaa-[ 8] = 0
/*+*-*+aaaaaaaa+a/a/+*aaaaaaaa-/+*a//aaaaaaaa-[ 9] = 945.8432
/a-*/*-aaaaaaaa+a/a/+*aaaaaaaa*-*--*-aaaaaaaa-[10] = 836.6745
/+-***-aaaaaaaa+a/a/+*aaaaaaaa*-a-a*-aaaaaaaa-[11] = 372.5987
/a++*--aaaaaaaa*/-a/a-aaaaaaaa+**a/a/aaaaaaaa-[12] = 766.7137
--a+**-aaaaaaaaaa-/aa/aaaaaaaa*/+*+*/aaaaaaaa-[13] = 420.6074
+a/+/+/aaaaaaaa*--a-*/aaaaaaaa*/+*+*-aaaaaaaa-[14] = 353.6146
-+a-/*aaaaaaaaa---a+a/aaaaaaaa-/+*a//aaaaaaaa-[15] = 611.6426
*-*-a*-aaaaaaaa/+-*a+aaaaaaaaa+**--a+aaaaaaaa-[16] = 466.7956
/*+*-*+aaaaaaaa+a/a/+*aaaaaaaa-----/*aaaaaaaa-[17] = 569.2436
/+-/*+-aaaaaaaa+a/a/+*aaaaaaaa*---a*-aaaaaaaa-[18] = 356.1858
-a-a+a/aaaaaaaa-+a-/*aaaaaaaaa+-+-+/-aaaaaaaa-[19] = 611.6426

Generation N: 4
01234567890123401234567890123401234567890123 4
/*+*-*+aaaaaaaa+a/a/+*aaaaaaaa-/+*a//aaaaaaaa-[ 0] = 945.8432
-+-***-aaaaaaaa-a/a/+*aaaaaaaa-----/*aaaaaaaa-[ 1] = 569.2436
/+-*-*+aaaaaaaa/a/+a/aaaaaaaaa/----/*aaaaaaaa-[ 2] = 817.2341
*/*/**+aaaaaaaa/a/a/a/aaaaaaaa-/+-a-/aaaaaaaa-[ 3] = 779.7097
```

/*+***-aaaaaaaa/a+--+/aaaaaaaaa-a-a*+aaaaaaaa-[4] = 750.3948
/*+*-*+aaaaaaaa+a/a/+*aaaaaaaa-/+*a*/aaaaaaaa-[5] = 585.3681
/*+--*+aaaaaaaa+a/a/+*aaaaaaaa*/**a/-aaaaaaaa-[6] = 891.116
/a+-*--aaaaaaaa*/-a/a-aaaaaaaa+**a/a/aaaaaaaa-[7] = 766.7137
--/a-aaaaaaaa/a/+*--aaaaaaaa*/+/+aaaaaaaa-[8] = 0
/+-***-aaaaaaaa-a/a/+*aaaaaaaa*aa-a*-aaaaaaaa-[9] = 886.7593
/*+*-a+aaaaaaaa+a/a/+*aaaaaaaa-/+*a//aaaaaaaa-[10] = 945.8432
/*+/-*+aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa-[11] = 945.8432
*/-a-a/aaaaaaaaaa-/aa/aaaaaaaa+**a/a/aaaaaaaa-[12] = 0
-+a-/*aaaaaaaaa-a-a+a/aaaaaaaa//-a+//aaaaaaaa-[13] = 0
+-a+-+/aaaaaaaa-a/a+a/aaaaaaaa-+a-/*aaaaaaaaa-[14] = 763.2632
/a++/--aaaaaaaa*--a-a/aaaaaaaa/+/*a/-aaaaaaaa-[15] = 0
/a++*--aaaaaaaa*/-a/a-aaaaaaaa+**a/a/aaaaaaaa-[16] = 766.7137
/*+*-*+aaaaaaaa+a/a/+*aaaaaaaa*aa-a*-aaaaaaaa-[17] = 565.8646
/*+*-*+aaaaaaaa---a+a/aaaaaaaa-//*a+/aaaaaaaa-[18] = 738.811
a+a-/*aaaaaaaaa---a+a/aaaaaaaa-/+*a//aaaaaaaa-[19] = 816.363

Generation N: 5
01234567890123401234567890123401234567890123401234
/*+/-*+aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa-[0] = 945.8432
a+a-/--aaaaaaaa-a/a/+*aaaaaaaa*aa-a*-aaaaaaaa-[1] = 945.5575
/*+*-*+aaaaaaaa+a/a/+*aaaaaaaa*aa-a*-aaaaaaaa-[2] = 565.8646
/*+*-*+aaaaaaaa---a+a-aaaaaaaa*-a-a*-aaaaaaaa-[3] = 367.2566
-+-***-aaaaaaaa-a/a/+*aaaaaaaa---*a++aaaaaaaa-[4] = 564.0838
aa/a/+*aaaaaaaa/+-***/aaaaaaaa///-*/*aaaaaaaa-[5] = 819.3729
/*+*-*+aaaaaaaa/a/+a/aaaaaaaaa*aa-a*-aaaaaaaa-[6] = 758.7488
+-a+-+/aaaaaaaa-a/a+a/aaaaaaaa-+a-/*aaaaaaaaa-[7] = 763.2632
/*+*-*+aaaaaaaa++/a-+*aaaaaaaa-/+aa//aaaaaaaa-[8] = 798.3864
/a+-*--aaaaaaaa*/-a/aaaaaaaaaa+**a/a/aaaaaaaa-[9] = 799.0481
/a+-*--aaaaaaaa*/-aaa-aaaaaaaa*aa-a*-aaaaaaaa-[10] = 747.9244
-/+*a//aaaaaaaaa*+*-*+aaaaaaaa+aa/+*aaaaaaaaa-[11] = 754.3026
/+-***aaaaaaaaa---a+a/aaaaaaaa-/+*a//aaaaaaaa-[12] = 803.7867
//**/aaaaaaaa/a/a/a/aaaaaaaa-/+-a-/aaaaaaaa-[13] = 816.5905
a+a-/*aaaaaaaaa---a+a+aaaaaaaa-/+*a//aaaaaaaa-[14] = 785.1168
/*+*-*+aaaaaaaa+a/a/+*aaaaaaaa--+*a//aaaaaaaa-[15] = 568.9833
/*+*-*+aaaaaaaa/a/a/+*aaaaaaaa*aa-a*-aaaaaaaa-[16] = 721.8472
a+a-/*aaaaaaaaa---a+//aaaaaaaa+*-/+*aaaaaaaaa-[17] = 752.1143
/*+*-**aaaaaaaa*a/a/+*aaaaaaaa-/+*a//aaaaaaaa-[18] = 408.8309
/+-***-aaaaaaaa-a/a/+*aaaaaaaa+**a/a/aaaaaaaa-[19] = 593.1216

Generation N: 6
01234567890123401234567890123401234567890123401234
/*+/-*+aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa-[0] = 945.8432
/*+*--+aaaaaaaa/a/a/+*aaaaaaaa*aa-a*-aaaaaaaa-[1] = 721.8472

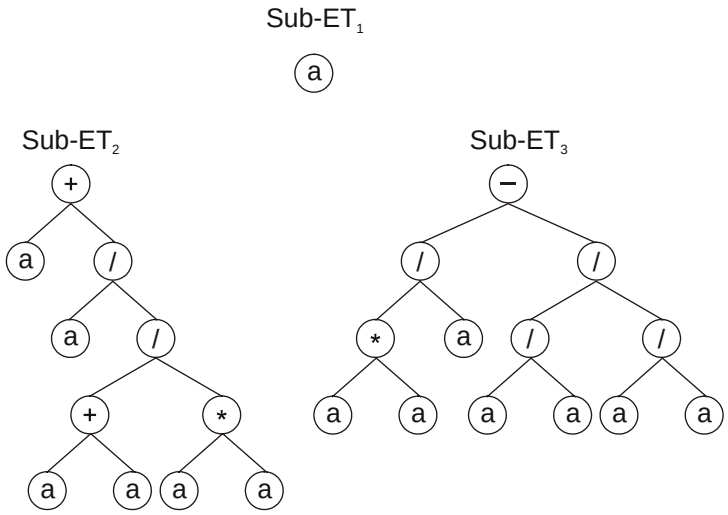
```
---*a++aaaaaaaa-+aa-+*aaaaaaaa*aa-**-aaaaaaaa-[ 2] = 752.1143
a+a-/*aaaaaaaaa---a+a+aaaaaaaa-/a*a//aaaaaaaa-[ 3] = 817.4047
/+-***aaaaaaaaa---a/a/aaaaaaaa+aa/+*aaaaaaaaa-[ 4] = 800.2618
*/+*a//aaaaaaaaa*a*-*+aaaaaaaa-aa/+*aaaaaaaaa-[ 5] = 789.7097
/*+*-*+aaaaaaaa++/a-+*aaaaaaaa-/+aa//aaaaaaaa-[ 6] = 798.3864
a//a/+*aaaaaaaa/a*a/+aaaaaaaaa/a/-//*aaaaaaaa-[ 7] = 817.1696
a+a-/*-aaaaaaaa-a/a/**aaaaaaaa-a/a/+*aaaaaaaa-[ 8] = 594.5831
-/**a//aaaaaaaaa*+*-*+aaaaaaaa+/+*a//aaaaaaaa-[ 9] = 796.5607
aa/a/+*aaaaaaaa/+-**a/aaaaaaaa///-*/*aaaaaaaa-[10] = 783.2302
*a*a/+*aaaaaaaa-a/aa+aaaaaaaaa---*a++aaaaaaaa-[11] = 778.0015
a+a-/-aaaaaaaaa---a+a+aaaaaaaa-/+*a//aaaaaaaa-[12] = 785.1168
-+-***-aaaaaaaa-a/a/+*aaaaaaaa-/-*a++aaaaaaaa-[13] = 945.5575
/*+*-*+aaaaaaaa/+-***/aaaaaaaa*aa-a*-aaaaaaaa-[14] = 708.8708
a+a-/--aaaaaaaa-a/a/+*aaaaaaaa*a-/--aaaaaaaaa-[15] = 594.5831
/*+*-*+aaaaaaaa/a/a/+*aaaaaaaa//a-a*-aaaaaaaa-[16] = 818.3482
-+-***-aaaaaaaa/*+*-**aaaaaaaa-/+*a//aaaaaaaa-[17] = 760.707
/+-***-aaaaaaaa-a/a/+*aaaaaaaa*+**a/aaaaaaaaa-[18] = 374.7753
a+a-/*aaaaaaaaa---a+a+aaaaaaaa-/+*a//aaaaaaaa-[19] = 785.1168
```

在第 7 代中产生了一个新的个体，该个体的适应度有所提高（12 号染色体）。其表达式如图 3.29 所示。完整的种群如下：

```
Generation N: 7
01234567890123401234567890123401234567890123 4
/*+/-*+aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa-[ 0] = 945.8432
/*+/-*+aaaaaaaa/+-**a/aaaaaaaa///-*/*aaaaaaaa-[ 1] = 800.2618
/+-*-*-aaaaaaaa-a/a*+*aaaaaaaa*+**a/aaaaaaaaa-[ 2] = 356.2026
---*a++aaaaaaaa-a*a/+aaaaaaaaa-/-*a++aaaaaaaa-[ 3] = 701.6004
/a/-//-aaaaaaaaa//a/+*aaaaaaaa/**a/+-aaaaaaaa-[ 4] = 0
a+a-/--aaaaaaaa+a*a/+aaaaaaaaa/+/-a/*aaaaaaaa-[ 5] = 360.1841
a+a-/-aaaaaaaaa---a+a+aaaaaaaa-/+*a//aaaaaaaa-[ 6] = 785.1168
/+-***-aaaaaaaa-a/a/+*aaaaaaaa-/-*a++aaaaaaaa-[ 7] = 610.0637
///a*+*aaaaaaaa-a/a/+*aaaaaaaa/a/-//*aaaaaaaa-[ 8] = 621.5726
/*+*---aaaaaaaa/a/a/+*aaaaaaaa**a-**-aaaaaaaa-[ 9] = 0
/+a/a/+aaaaaaaa-a/a/+*aaaaaaaa-**/-*aaaaaaaaa-[10] = 333.7551
/*+/-*+aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa-[11] = 945.8432
aa/a/+*aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa-[12] = 989.9987
a+-***-aaaaaaaa-+aa-+*aaaaaaaa*aa-a--aaaaaaaa-[13] = 789.7097
/*+*+-+aaaaaaaa/+/+/+*aaaaaaaa*aa-a*-aaaaaaaa-[14] = 388.9994
/+-***-aaaaaaaa-a/a/+*aaaaaaaa+***+**aaaaaaaa-[15] = 343.4301
/*+*-*+aaaaaaaa++/a-+*aaaaaaaa-/+aa//aaaaaaaa-[16] = 798.3864
aa/a/+*aaaaaaaa/+-a*a/aaaaaaaa////*/*aaaaaaaa-[17] = 805.9502
---*a++aaaaaaaa-aa+aa-aaaaaaaa*aa-**-aaaaaaaa-[18] = 471.685
a/-a/+*aaaaaaaa-a/-/+aaaaaaaaa*a-/--aaaaaaaaa-[19] = 816.5905
```

a. 012345678901234012345678901234012345678901234
aa/a/+*aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa

b.



c.

$$y = (a) + \left(a + \frac{a^2}{2} \right) + (a - 1)$$

图 3.29.第 7 代的最佳个体（12 号染色体）。它在表 3.2 给出的适应度样本集上算得得适应度为 989.9987。**a)**该个体的染色体。**b)**每个基因所编码的子表达式树。注意sub-ET_2已经在前面的最佳个体中出现过。**c)**加号连接后的相应的数学表达式（每个子表达式树的贡献用括号标识）。

接下来的两代中适应度没有提高：

```
Generation N: 8
012345678901234012345678901234012345678901234
aa/a/+*aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa-[ 0] = 989.9987
a+a-/-aaaaaaaaa---a+a+aaaaaaaa-/+*a//aaaaaaaa-[ 1] = 785.1168
/*+/-/+aaaaaaaa+a/a/+*aaaaaaaa-//+a//aaaaaaaa-[ 2] = 889.116
aa/a/+*aaaaaaaa/a/a/+*aaaaaaaa-//*a//aaaaaaaa-[ 3] = 782.9793
a+a-/-aaaaaaaaa---a+a+aaaaaaaa-/+*a//aaaaaaaa-[ 4] = 785.1168
/*+/-*+aaaaaaaa+aaa/**aaaaaaaa-//*a++aaaaaaaa-[ 5] = 779.7097
/+-*-**aaaaaaaa-a/a*+*aaaaaaaa////*/*aaaaaaaa-[ 6] = 0
aa/a/+*aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa-[ 7] = 989.9987
++/a-+*aaaaaaaa/*/a/+aaaaaaaaa/a/-//*aaaaaaaa-[ 8] = 794.4986
/*+/-*+aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa-[ 9] = 945.8432
///a*+-aaaaaaaa-a+*-*+aaaaaaaa-/+aa//aaaaaaaa-[10] = 0
/a+/-*+aaaaaaaa/*+/-*+aaaaaaaa+a/a/+*aaaaaaaa-[11] = 886.7593
/*+/-*+aaaaaaaa/+-**a/aaaaaaaa///-*/*aaaaaaaa-[12] = 800.2618
/*+*-*+aaaaaaaa++/a-+*aaaaaaaa-/+aa//aaaaaaaa-[13] = 798.3864
/+-***-aaaaaaaa-////+*aaaaaaaa-a/*a//aaaaaaaa-[14] = 807.1407
/+-***-aaaaaaaa-a/a/+*aaaaaaaa+***+**aaaaaaaa-[15] = 343.4301
a/-a/+*aaaaaaaa-a/-/+*aaaaaaaa*a-/--aaaaaaaaa-[16] = 730.5334
/+-***-aaaaaaaa-a*a/++aaaaaaaa*/*a+**aaaaaaaa-[17] = 355.8565
```

aa/a/+*aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa-[18] = 989.9987
///a*+*aaaaaaaa-a/a/+*aaaaaaaa/a/-//*aaaaaaaa-[19] = 621.5726

Generation N: 9
01234567890123401234567890123401234567890123 4
aa/a/+*aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa-[0] = 989.9987
/*+/-*+aaaaaaaa+aaa/**aaaaaaaa-//*a++aaaaaaaa-[1] = 779.7097
aa/a++*aaaaaaaa/a/a/+*aaaaaaaa-/+/*a/aaaaaaaa-[2] = 788.0954
a+a-/-aaaaaaaaa---a+a+aaaaaaaa/*+/-*+aaaaaaaa-[3] = 816.5905
++/a-+*aaaaaaaa-////+*aaaaaaaa/a/-//*aaaaaaaa-[4] = 819.1696
+aaa/+*aaaaaaaa-/-/a+/aaaaaaaa-/+-a-/aaaaaaaa-[5] = 817.4047
/+-***-aaaaaaaa-//-/+*aaaaaaaa-//*a//aaaaaaaa-[6] = 805.3593
aa/a/+*aaaaaaaa/a/a/+*aaaaaaaa-a/*a/-aaaaaaaa-[7] = 791.2383
/*+/-*+aaaaaaaa+aaa/**aaaaaaaa-//a*++aaaaaaaa-[8] = 789.0481
/aaa+-*aaaaaaaa-a/-/+*aaaaaaaa*a-/--aaaaaaaaa-[9] = 697.6004
/+-***-aaaaaaaa-a/a/+*aaaaaaaa+***+**aaaaaaaa-[10] = 343.4301
aa/a/+*aaaaaaaa/a/a/+*aaaaaaaa-//*a/aaaaaaaaa-[11] = 799.0481
a+a-/-aaaaaaaaa---a+a+aaaaaaaa+/+*a//aaaaaaaa-[12] = 817.6932
/*++-*+aaaaaaaa/+***a/aaaaaaaa///*a/*aaaaaaaa-[13] = 370.8166
/*+*-*+aaaaaaaa++/a-+*aaaaaaaa-/+aa//aaaaaaaa-[14] = 798.3864
/*+/-*+aaaaaaaa/+-*-a*aaaaaaaa///-*/*aaaaaaaa-[15] = 812.4845
a+a-/-aaaaaaaaa---a+a+aaaaaaaa//-*a//aaaaaaaa-[16] = 0
a*/a/+*aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa-[17] = 989.9987
-//*a//aaaaaaaa/a//a/+aaaaaaaa/a/a/+*aaaaaaaa-[18] = 327.7421
/*+/-/+aaaaaaaa+a/a/+*aaaaaaaa-//+a//aaaaaaaa-[19] = 889.116

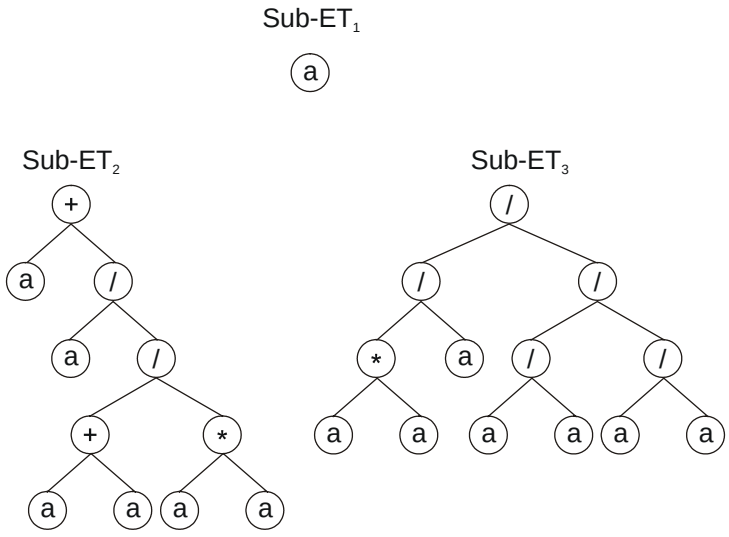
最后，在第 10 代中产生了一个具有最高适应度的个体：
Generation N: 10
01234567890123401234567890123401234567890123 4
a*/a/+*aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa-[0] = 989.9987
+/+a-//aaaaaaaa/*+*-*+aaaaaaaa++/a-+*aaaaaaaa-[1] = 0
/*+/-*+aaaaaaaa+a///+*aaaaaaaa/a/-//*aaaaaaaa-[2] = 800.3348
/*+/-*aaaaaaaaa+-aa/**aaaaaaaa-//-a++aaaaaaaa-[3] = 785.1168
/+aa/a/aaaaaaaa+a/a/+*aaaaaaaa///-*/*aaaaaaaa-[4] = 886.7593
/+-*-**aaaaaaaa-/+-/+*aaaaaaaa-//*a//aaaaaaaa-[5] = 0
-//*a++aaaaaaaa/*+/-*+aaaaaaaa*+aaa/*aaaaaaaa-[6] = 474.2741
a+a-/-aaaaaaaaa---a+a+aaaaaaaa-//*a/aaaaaaaaa-[7] = 796.9538
aa/a/+*aaaaaaaa/a/a/**aaaaaaaa--/*a/-aaaaaaaa-[8] = 0
a+a/+/+aaaaaaaa+a-a/+*aaaaaaaa//+/-*+aaaaaaaa-[9] = 0
a*/a/+*aaaaaaaa+a/a/+*aaaaaaaa-//*a//aaaaaaaa-[10] = 989.9987
-*/a/+*aaaaaaaa+a/a/+*aaaaaaaa-///-*+aaaaaaaa-[11] = 0
a*/a/+*aaaaaaaa+a/a/+*aaaaaaaa///*a//aaaaaaaa-[12] = 1000
a+a-/*aaaaaaaaa-+-a+a+aaaaaaaa/*+*a//aaaaaaaa-[13] = 381.2412
aa/a/+*aaaaaaaa+a/a/+*aaaaaaaa-//*a+/aaaaaaaa-[14] = 891.116

```
a+a-/-aaaaaaaaa---*+a+aaaaaaaa+/+*a//aaaaaaaa-[15] = 738.8981
aa/a/+*aaaaaaaa/aaa/**aaaaaaaa-//*a++aaaaaaaa-[16] = 804.8516
++/a-+*aaaaaaaa-//a/+*aaaaaaaa-/+*a//aaaaaaaa-[17] = 769.7097
/*+/-*+aaaaaaaa/+-*-a*aaaaaaaa-//*a//aaaaaaaa-[18] = 798.2679
/*+-/-aaaaaaaaa---a+a+aaaaaaaa-//+a-/aaaaaaaa-[19] = 829.525
```

注意，本代中的最佳个体是前一代中的最佳个体通过变异得到的：它们的染色体仅仅有一个位置的不同之处（第 3 代中的 0 位上的“ － ”被“ / ”所取代）。这个具有最高适应度的染色体的表达式表明它是当前问题的一个完美解（图 3.30）。

a.
```
012345678901234012345678901234012345678901234
a*/a/+*aaaaaaaa+a/a/+*aaaaaaaa///*a//aaaaaaaa
```

b.



c.

$$y = (a) + \left(a + \frac{a^2}{2} \right) + (a) = \frac{a^2}{2} + 3a$$

图 3.30.基本符号回归问题的完美解。该程序是在第 8 代找到的，并且具有最大适应度。**a)**该个体的染色体。**b)**每个基因所编码的子表达式树。注意sub-ET_1和sub-ET_2已经在前面的最佳个体中出现过（见图 3.29），而且sub-ET_2的“ 家族历史 ”更长，可以追溯到初始种群。**c)**加号连接后的相应的数学表达式（每个子表达式树的贡献用括号标识）。注意它与目标函数(3.3)完全吻合。

所以，我们看到一组随机产生的染色体如何在最初的试探性脚步迈入恶劣环境的情况下逐渐变成成熟的程序。这些“ 卑微的 ”个体被选中进行修饰，产生能够很快适应环境新的个体，而且其适应程度令人难以置信。而这种奇妙的进化过程是通过遗传算子和无情的选择之手在基因组上完成的盲目修饰操作成为可能的。下一章我们将看到如何有效地采用这些原则去解决大量的复杂问题。

用基本基因表达式算法求解问题 4

本章中我们将看到如何采用基本基因表达式算法来解决源自不同领域的问题，包括符号回归，优化，数据挖掘，时间序列分析，逻辑合成和元胞自动机。我们还将进一步看到如何通过 ADF 和 UDF 来使 GEP 的进化工具箱更加丰富。我们最后还将看到 GEP 染色体如何通过域的引入来增加其复杂性。本章将使用含有附加域的染色体组织来对数值常数进行显式操作。的确，这种染色体组织结构作为 GEP 推断神经网络的基石，也被用来进行参数优化，进化 Kolmogorov-Gabor 多项式或数值属性的决策树。

4.1.符号回归

我们在 3.4 节的简单例子中已经看到如何采用 GEP 来进行符号回归。这里，我们将分析更加复杂的符号回归问题。第一个简单的测试函数被采用本算法精确的求解出来，但是该过程很适合演示算法基本参数的工作原理。第二个问题具有一个复杂的测试函数，它演示如何运用 GEP 高效地对问题进行高精度的建模。最后一个问题说明如何采用 GEP 从噪声数据中高效地挖掘出相关信息。

4.1.1.一维参数空间上的函数发现

本节的目标函数是一个简单的多项式:

$$y = a^3 + a^2 + a + 1 \qquad (4.1)$$

选择这个函数仅仅是因为它可以在几秒钟内运行上百次并通过该算法精确求解。而且可以用它来说明如何对一个问题进行合理的设置，包括适应度函数，基因个数，头部长度，函数集和连接函数。这种分析有助于我们从直观上理解算法的基本参数，该问题的参数选择见表 4.1。下面讨论如何及为什么这样选择。

表 4.1 多元函数问题的设置

Number of runs	100
Number of generations	50
Population size	30
Number of fitness cases	10(表 4.2)
Function set	+-*/
Terminal set	a
Head length	6
Number of genes	4
Linking function	+
Chromosome length	52
Mutation rate	0.0385
One-point recombination rate	0.3
Two-point recombination rate	0.3
Gene recombination rate	0.1

IS transposition rate	0.1
IS elements length	1,2,3
RIS transposition rate	0.1
RIS element length	1,2,3
Gene transposition rate	0.1
Selection range	100%
Precision	0%
Success rate	100%

假设由测试函数（4.1）在给定的区间[-10,10]上随机选取的 10 个点上的一组数值。我们希望找到一个函数在 0.01 的误差范围内适应这些值。

首先，必须选择函数集和终点集。在此，我们可以选取 F={+,－,*,/}和 T={a}，然后必须选择染色体的组织结构，即头部长度 n 和基因个数，比较明智的选择是从短的单基因染色体开始，然后逐步增加 h 的值。图 4.1 给出这个问题的分析图，注意成功率如何在开始突然提高，由最紧凑的组织结构（基因长度为 13）到稍有冗余的组织结构，其成功率由 29%变成 86%。还要注意的从这一点开始，成功率开始逐步降低。如读者所看到的，对每个搜索图而言，我们可能或多或少地猜到最佳的染色体长度，以使整个图能够被完全搜索到。

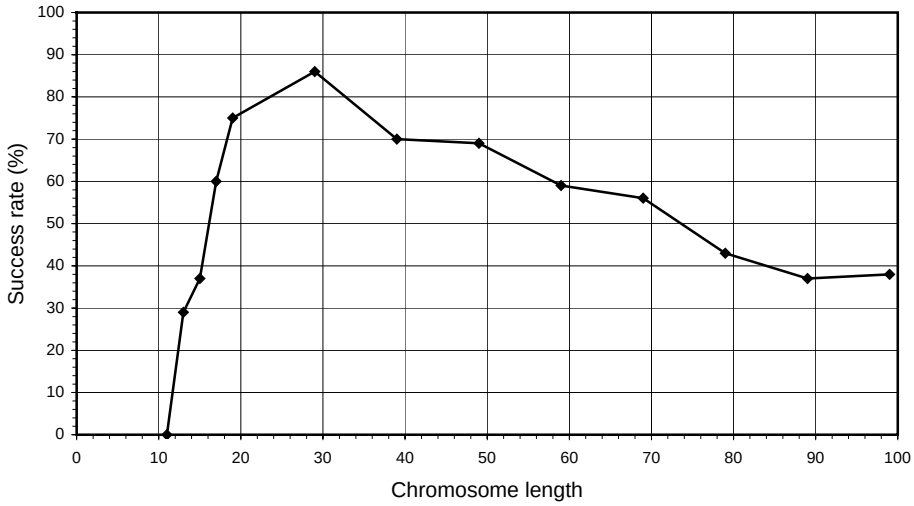


图 4.1.成功率随染色体长度变化。对于该分析，染色体由单基因构成，G=50，P=30。成功率由 100 次相同的运行计算得到。

如图 4.1，对本问题的分析，所采用种群大小为 30 ，进化代数为 50 代。在所有的实验中，为了简化分析，仅采用变异和两点重组。变异概率等效于每个染色体中发生两个单点变异，重组概率为 0.7。本问题的 10 个随机适应度样本选自区间 [-10,10]（表 4.2），适应度由方程（3.1a）求得；选择范围为 100，精度为 0.01。因此对 10 个适应度样本而言，f_{max}=1000。

表 4.2 多元函数问题的适应度样本集

a	f(a)
5.7695	232.107
-4.1206	-56.1063
4.652	127.968

-5.6193	-150.481
-3.2971	-27.2686
-0.0599	0.943473
1.1835	5.24187
-8.2814	-506.651
4.4342	112.282
4.1843	95.9529

注意 GEP 可以用来搜索问题最简单的解。如图 4.1 所示，对该问题而言，采用头部长度为 5 的时候不能找到一个正确解，但是只要 h=6，就可能进化得到一个完美解。这里，这些完美解也是当前问题的最简洁的解。例如，如下的染色体均含有 13 个节点，对问题的完美解也即简洁解进行编码：

```
0123456789012
*++/*/aaaaaaa-[1]
*++*//aaaaaaa-[2]
```

需要注意，h 取值较大的时候 GEP 的进化效率也很高，也就是说，GEP 能够处理非压缩的或者高冗余的信息。如图 4.1 所示，对于每个问题，存在一个能够使进化效率最高的染色体长度。而且，至少对当前这个简单测试函数而言，可以找到最理想的染色体长度。还请注意，最紧凑的基因组并不是最高效的。这表明一定程度的冗余对进行有效的进化来说使十分重要的（讨论见第 7 章）。

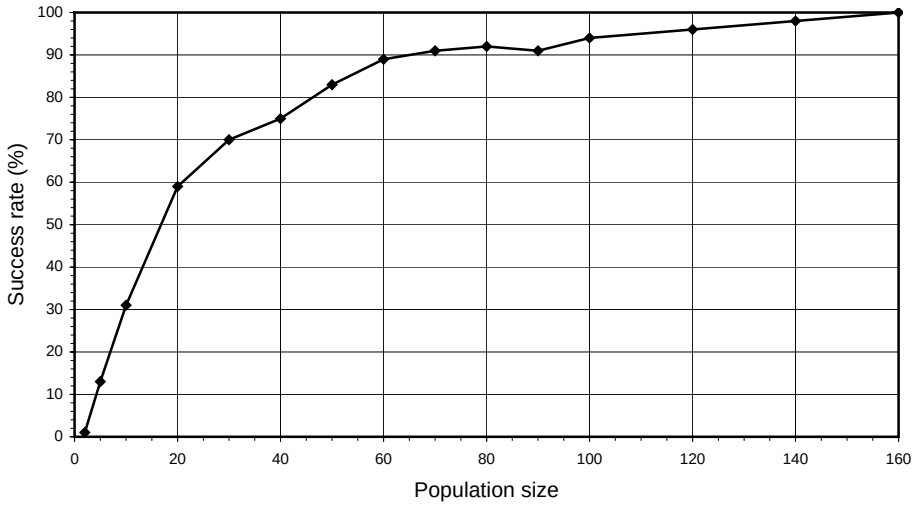


图 4.2.成功率随单基因系统种群大小变化。对于该分析，G=50，P=30，使用染色体的长度为中等值 39（h＝19）。成功率由 100 次相同的运行计算得到。

在下一个试验中，我们将分析成功率与种群规模之间的关系（图 4.2），注意，这里为了使图形具有较好的区分度，我们采用的头部长度为 19。的确，这里不仅头部长度没有选择最好的，单基因染色体也不是最佳选择。因此，请记住 GEP 远比简单基因系统复杂，因为 GEP 染色体可以对多基因进行编码。

假设我们在分析图 4.1 和图 4.2 后找不到满意解或者系统的进化效率不佳。那我们可以选择多基因系统，显然还要选择如何连接这些表达式树。例如，我们可以选择基因的头部长

度为h=6，连接函数为加法。图 4.3 给出了一个这种问题的分析图。变异概率等效于每个染色体中发生两个单点变异，而且随染色体长度变化，$p_{1r}=p_{2r}=0.3$，$p_{gr}=p_{gt}=p_{is}=p_{ris}=0.1$转座子长度分别为 1，2，3。注意，GEP能很好处理基因数量较大的情况，10-基因系统的成功率

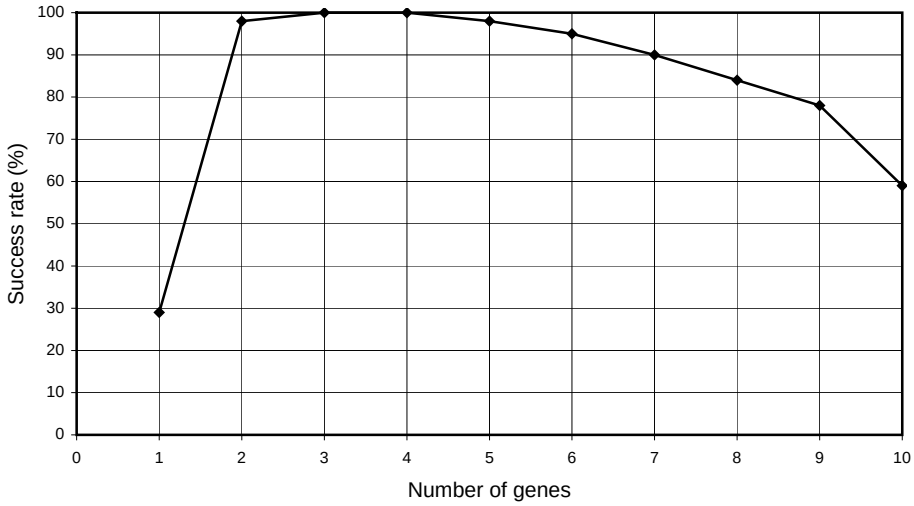


图 4.3.成功率随基因个数变化。对于该分析，G=50，P=30，h=6，(基因长度为 13)，子表达式树由加号连接。成功率由 100 次相同的运行计算得到。

仍然相当高（59%)。我们再次看到一定程度的冗余使系统的效率更高。这里，当染色体所编码的表达式树的个数为 2 到 6，采用加法作为连接函数的时候，在这个特定的解空间中的搜索极其成功，图 4.4 说明，在 GEP 中，能够无限地适应和进化，因为基因池中一直有新物质引入。

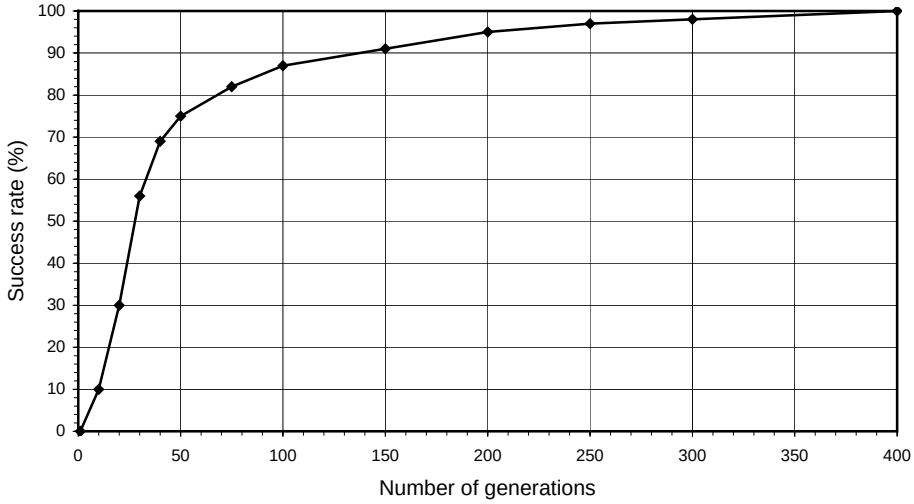


图 4.4.成功率随代数变化。对该分析 P＝30，染色体长度为 79（单基因染色体 h=39)。成功率由 100 次相同的运行计算得到。

最后，假设我们采用由加法连接子表达式树的多基因系统仍然找不到问题的解。我们可以选择其它的连接函数并让系统自己对连接函数进行进化。图 4.5 的分析采用乘法作为连接函数，其它参数与图 4.3 中完全相同。

如我们所料，对于该多项式函数而言，该算法中子表达式树由加法连接的时候，效果最好。该测试过程反复进行，直到发现一个优良解或者当我们觉得已经发现最佳的染色体结构和组成为止。然后选择一组恰当的设置让系统自己去进化最优的可能解。

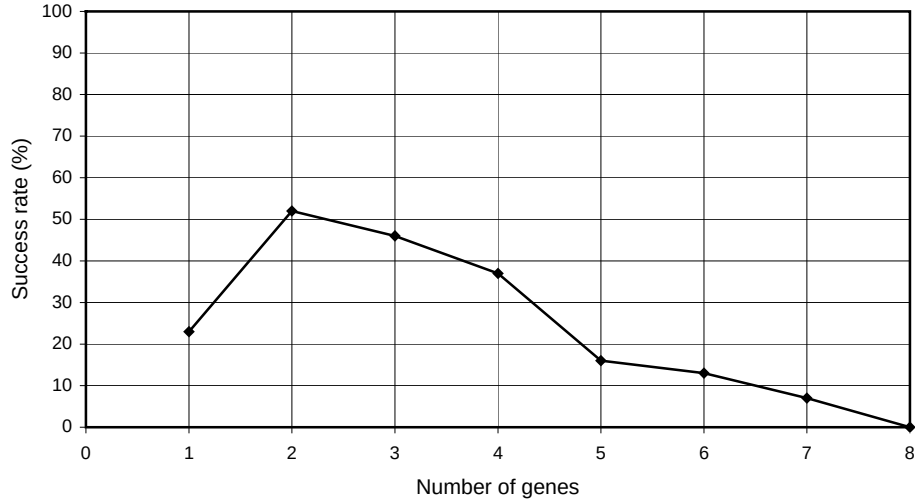


图 4.5.成功率随基因个数变化。对该分析 G＝30，P=30，h=6（基因长度为 13）。子表达式树由乘号连接。成功率由 100 次相同的运行计算得到。

例如，假设有一个多基因系统由 4 个基因构成，用加法作为连接函数。如图 4.3 所示，成功率最大值为 100%。这意味着每次运行都能够找到一个完美解。让我们更仔细地分析该实验的一次成功运行，每次运行的参数如表 4.1 所示。

这次成功运行的进化动力学图示见图 4.6。我们看到，在第 19 代找到了一个完美解。

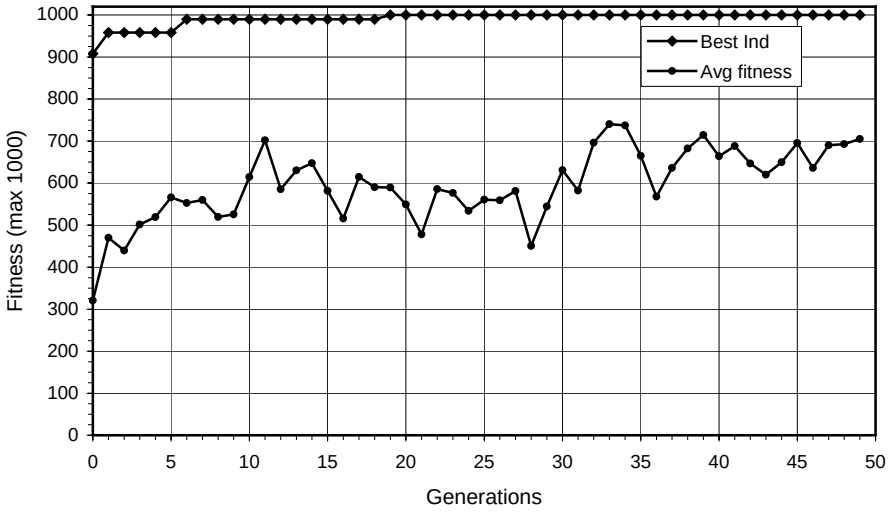


图 4.6.表 4.1 所给出的实验的第 0 次运行的种群平均适应度和最佳个体适应度进展图示。

本次运行的初始种群个体染色体和适应度如下图所示（本代的最佳个体用粗体标识）：

```
GenerationN:0
0123456789012012345678901201234567890120123456789012
**//++aaaaaaa/a//**aaaaaaa++/-**aaaaaaa+/+-aaaaaaaaa-[0]=479.9631
```

```
*/-//+aaaaaaa/*+a/aaaaaaaa--aa//aaaaaaa-+/++*aaaaaaa-[1]=354.6601
-+a/a/aaaaaaa/-/+-*aaaaaaa*-*/+/aaaaaaa*/a-/aaaaaaaa-[2]=373.0389
---//+aaaaaaa*+/a--aaaaaaa++aa+aaaaaaaa**+++-aaaaaaa-[3]=193.9366
/--aa/aaaaaaa+-a+--aaaaaaa//+a/-aaaaaaa-/+a*+aaaaaaa-[4]=287.6966
-*/a/+aaaaaaa*+/+--aaaaaaa/-*++*aaaaaaa+++-a/aaaaaaa-[5]=355.6841
*aaaa*aaaaaaa*a-/-*aaaaaaa/--*+/aaaaaaa++*-*aaaaaaaa-[6]=408.3963
--/++aaaaaaaa*a//++aaaaaaa///a-aaaaaaaa*+a++*aaaaaaa-[7]=0
---**aaaaaaaa*-++++aaaaaaa*/aa+aaaaaaaa--aa/aaaaaaaa-[8]=319.8496
/-/+--aaaaaaa+---/*aaaaaaa*--*-+aaaaaaa/*aa+*aaaaaaa-[9]=0
-**/-+aaaaaaa*+aa-/aaaaaaa-/-a*+aaaaaaa*+-+++aaaaaaa-[10]=324.3444
+a+*--aaaaaaa-+*/+-aaaaaaa/-/a/-aaaaaaa+-++-aaaaaaaa-[11]=0
+a+*--aaaaaaa-+-*+-aaaaaaa+a///-aaaaaaa+//a*-aaaaaaa-[12]=412.0092
/a/-/-aaaaaaa*+/*-*aaaaaaa**a/*+aaaaaaa*--+aaaaaaaaa-[13]=215.5287
***/*/aaaaaaa-*//-aaaaaaaa//-a-aaaaaaaa/*/-+-aaaaaaa-[14]=0
-a+/a+aaaaaaa--*a/+aaaaaaa-**-+aaaaaaaa-/+++/aaaaaaa-[15]=420.3384
//+a-*aaaaaaa/a-a*+aaaaaaa**+/-aaaaaaaa/+-a/-aaaaaaa-[16]=0
+-aa/+aaaaaaa-*-**aaaaaaaa*//*-/aaaaaaa*+*a/aaaaaaaa-[17]=0
*/-+a*aaaaaaa+*-+-/aaaaaaa/a-//aaaaaaaa-+-/+aaaaaaaa-[18]=0
+-++a-aaaaaaa+-a*-aaaaaaaa/*a---aaaaaaa+*a+a*aaaaaaa-[19]=789.56
+*aa++aaaaaaa*++a+/aaaaaaa+*/**/aaaaaaa*/*/*-aaaaaaa-[20]=171.441
/a**a/aaaaaaa*+-++/aaaaaaa*//-++aaaaaaa----/-aaaaaaa-[21]=403.2058
+a*--aaaaaaaa+/-//-aaaaaaa--++a*aaaaaaa+***aaaaaaaaa-[22]=779.5309
*////*aaaaaaa-//-a/aaaaaaa/*/a-+aaaaaaa+-/*-*aaaaaaa-[23]=296.2591
-a-++*aaaaaaa+-**a*aaaaaaa*+-a*aaaaaaaa-/+aa-aaaaaaa-[24]=873.554
*a++/aaaaaaaa/-a--*aaaaaaa***a+/aaaaaaa*/*+a-aaaaaaa-[25]=313.3738
-++/+/aaaaaaa*+a+*/aaaaaaa/+-+-/aaaaaaa-+//a+aaaaaaa-[26]=908.0693
*/***/aaaaaaa+-*---aaaaaaa/+a/a*aaaaaaa*+*+/*aaaaaaa-[27]=195.4805
+*a+/aaaaaaaa**a+-*aaaaaaa/*+a+/aaaaaaa+***a-aaaaaaa-[28]=748.5801
-/---+aaaaaaa//+*+-aaaaaaa-/---aaaaaaaa+-++*-aaaaaaa-[29]=0
```

注意这里随机产生的 30 个个体中有 8 个是不可存活的个体，适应度值为 0。这意味着它们或者是不能在选择范围内解决一个适应度样本，或者因为除零返回了运算错误。这一代的最佳个体，26 号染色体的适应度为 908.0693，如表达式所示，目标函数的 4 项中有 2 项已经存在于这个解中（第 1 项和第 2 项）。

下一代中产生了一个比第 0 代中的最佳个体性质更好的新个体：

```
012345678901201234567890120123456789012345678901201234567890120123456789012
 +a+*-aaaaaaaa+/-//-aaaaaaa--++a*aaaaaaa+***aaaaaaaaa
```

（4.2）

在表 4.2 中所示的适应度样本上，该个体的适应度为 958.3982，其表达式显示这个中间解有 3 项与目标函数完全一致；实际上，只有第 3 项不一致。

在图 4.6 中，读者可以看到，紧接的后 4 代中最佳个体的适应度没有什么改进，但在第 6 代中发现一个适应度为 990 的更好的解：

```
012345678901201234567890120123456789012345678901201234567890120123456789012
 +-/-a-aaaaaaa+-a**aaaaaaaa*+-a*aaaaaaaa+*a+a*aaaaaaa
```

（4.3）

其表达式显示这个中间解有 3 项与目标函数完全一致；实际上，只有最后一项不一致。

接下来的 12 代中，最佳个体的适应度也没有变化，但是第 19 代中发现一个具有最大

适应度的完美解：

```
012345678901201234567890120123456789012345678901201234567890120
+*a/-aaaaaaaa*-/++aaaaaaaa-/***-aaaaaaa+***/*aaaaaaa      (4.4)
```

对这些每代中最佳程序的分析说明其中有些行为是多余的或者是中性的，如加零和乘零操作等，但是这些多余簇的存在不管是对子表达式树中的中性簇还是像上面染色体(4.4)中 2 号基因那样的整个中心基因，它们对于适应度更好的个体的进化而言都很重要（比较图 4.1 和图 4.3 中 h=6 的紧凑的单基因系统和不够紧凑的系统，和基因更多和头部长度大于 6 的基因的成功率）。

显然对于现实世界中的问题进行如上的详细分析是不切实际的，对于这样的问题一般采用两三种染色体组织结构和不同的测试函数进行实验，通过观察诸如最佳适应度或者平均适应度这样的指标，很容易看出系统进化效率的高低，然后可以选择恰当的设置，剩下的就是让系统去找当前的问题的可能解。

4.1.2.在 5 维参数空间上进行函数发现

本节的目标是演示如何用 GEP 对复杂的现实问题精确地建模，所选择的测试函数是下面这个含有 5 个变量的函数：

$$y = \frac{\sin(a).\cos(b)}{\sqrt{\exp^{c}}} + \tan(d-e) \qquad (4.5)$$

其中 a,b,c,d,e 是自变量，*exp* 是无理数 2.71828183。

假设有测试函数（4.1）在给定的区间[0,1]上随机选取的 50 个点上的一组数值。我们希望找到一个函数在 0.01%的误差范围内适应这些值。适应度由方程（3.1b）求得；选择范围为 100，精度为 0.01。因此对 50 个适应度样本而言，f_{max}=5000。

这个复杂的问题中所选择的 50 个适应度样本的集合很可能并不能很好地代表问题域，GEP 进化所得到的程序所建模的现实或许也并不是函数的现实。为了解决这一难题，通常采用一组规模合理的样本数据作为测试集。在进化过程中不使用该数据，我们用它来检验模型的精度和模型的概化能力。对于该问题而言，由于测试集不会减慢进化速度，所以我们可以产生一个样本数为 200 的随机测试集。

该问题的问题域说明，除了基本算术函数以外，函数集中还应使用 sqrt(x)，exp(x)，sin(x), cos(x)，tan(x)，在 Karva 表达方式中可以相应地用 Q, E, S, C, T 来对应。所以对本问题而言，函数集由 F＝{+,-,*,/,Q,E,S,C,T}构成，终点集显然由自变量构成，即 T={a, b, c, d, e}。

对本问题，我们选择双基因染色体对子表达式树进行编码，子表达式树最多可以由 25 个节点。子表达式树翻译后由加法连接。每次运行所使用的参数如表 4.3。

由于问题比较复杂，我们采用 *Automatic Problem Solver* (APS)来对该函数建模。因为它允许对中间解进行简单的优化，而且还允许在测试集上对模型进行简单的检验，可以计算一些标准统计参数，如相关系数 R-平方，在一次运行中发现一个优良的解。它在测试集中的 200 个点上的 R-平方值为 0.999964。

```
012345678901234567890123401234567890123456789012345678901234
*S*aCCQbSc+aabadedbadeecaT-deECd+/+Ccacdadddabceaa      (4.6a)
```

表 4.3 5-参数函数问题

Number of runs	100
Population size	500

Number of fitness cases	50
Function set	+-*/ QESCT
Terminal set	a b c d e
Head length	12
Number of genes	2
Linking function	+
Chromosome length	50
Mutation rate	0.044
One-point recombination rate	0.3
Two-point recombination rate	0.3
Gene recombination rate	0.1
IS transposition rate	0.1
IS elements length	1,2,3
RIS transposition rate	0.1
RIS element length	1,2,3
Gene transposition rate	0.1
Selection range	100%
Precision	0%

它对应的数学表达式为：

$$y = \cos\sqrt{\sin(c)} \cdot \cos(b) \cdot \sin(a) + \tan(d - e) \qquad (4.6b)$$

采用 APS 可以自动将进化得到的 Karva 程序转化为更传统的计算机程序。这里，上面的模型 4.6 被自动翻译成如下的 C++函数：

```
double APSCfunction(double [])
{
 double dblTemp = 0;
 dblTemp += tan((d[3]-d[4]));
 dblTemp += (sin(d[0])*(cos(sqrt(sin(((d[1]-d[1])+d[2]))))*cos(d[1])));
 return dblTemp;
}
```
(4.6c)

其中d_0-d_4分别对应a到e。注意该模型的 3 项与目标函数完全一致。但是请注意GEP发现的$\cos\sqrt{\sin(c)}$是$1/\sqrt{\exp^c}$的一个很好的近似。的确，我们在本书中将一再看到，GEP可以用来找到非常复杂问题的非常优良的解。

4.1.3.从噪声数据中挖掘有意义的信息

随着世界上的数据持续增长，从数据中挖掘知识的工具显得越来越重要，数据量如此巨大，以至于要在数据的海洋中发现有意义的因素成为一个不可能的任务。人们开发了许多技术来从数据中提取相关知识。GEP 是这些众多崭露头角的新技术中的一种。它是区别良莠的理想的工具。这一节我们将演示如何在 10 个变量中的 9 个无意义的情况下进行函数发现。

测试函数是我们在 4.1.1 节已经熟悉的函数。区别在于必须从 10 个变量中找到那个有意

义的变量。图 4.4 为该实验中每次运行所需要的参数，如较高的成功率所示，GEP 并没有被大量的无关数据所征服，它仍然按照自己的方式高效地向前推进。第一个完美解出现在首次运行的第 6 代。其染色体如下（子表达式树用加法连接）

```
0123456789012
*a*aa-hgadadc
-ah*d-gcfjcbd
/--gcgciijeeg
h+eeehbeddbfd
*aadaabcecfgb                    (4.7)
```

其中 a 代表有意义的变量，b-j 代表剩下的无意义变量，如其表达式所示，该染色体所编码的函数与目标函数等价。

表 4.4 10 维数据挖掘问题的设置

Number of runs	100
Number of generations	1000
Population size	50
Number of fitness cases	100
Function set	+-*/
Terminal set	a b c d e f g h I j
Head length	6
Number of genes	5
Linking function	+
Chromosome length	65
Mutation rate	0.044
One-point recombination rate	0.3
Two-point recombination rate	0.3
Gene recombination rate	0.1
IS transposition rate	0.1
IS elements length	1,2,3
RIS transposition rate	0.1
RIS element length	1,2,3
Gene transposition rate	0.1
Selection range	100%
Precision	0%
Success rate	77%

4.2.符号回归和数值常数的产生

这一节，我们将分析解决符号回归问题所采用的两种不同常数产生方法。第一种方法是对随机常数进行显式操作，另一种则是通过从头产生数值常数或者发明新的方法来表示这些常数。

有人认为采用进化计算处理符号回归问题的时候必须产生浮点型的常数（例如 Koza 1992，Banzhaf 1994）。GP 采用一种称为“ 中间随机常数 ”的特殊终点来解决常数产生的问题（Koza 1992）。对于每个初始种群中树所使用的中间随机常数，产生一个特定范围内的特

殊数据类型的随机数。这些随机常数通过杂交算子的作用在树与树之间移动。

GEP 采用不同的方法来解决常数产生的问题（Ferreira 2001）。GEP 采用一种附加终点“？”和由用来表示随机常数的符号构成的附加域 Dc。对每个基因而言，这些随机常数产生于初始种群产生的时候，并被保存在一个数组中。每个随机常数仅在其基因表达的过程中赋值。进一步说来，用一个特殊的算子来向随机常数池中引入遗传变化。该算子直接对随机常数进行变异。而且 GEP 的常规算子加上针对 Dc 域的转座操作保证了种群中的随机常数能够有效地在开始运行的时候产生恰当的多样性，而且通过遗传多样化的作用在以后很容易保持这种多样性。

尽管如此，我们将会看到，如果由进化计算本身来处理常数产生的问题，那么进化计算在符号回归的时候将会更有效；换言之，处理随机常数的这种便利方法实际上对于解决符号回归问题来说并不是十分必要。但是，对于参数优化或者用 GEP 设计神经网络和数值属性的决策树这样的问题而言，随机常数则是十分必要的。

4.2.1.GEP 中随机常数的控制

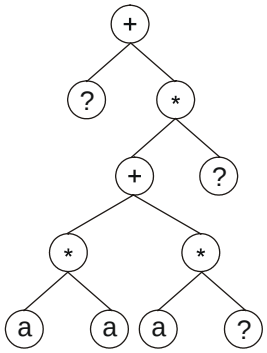
要在 GEP 中实现数值常数很容易。为此在 GEP 基因中引入一个附加域 Dc。从结构上讲，Dc 在尾部的后面，和 t 的长度相等，由代表中间随机常数的符号构成。因此，在基因中生成了具有单独边界的，有自身字符集的另一个区域。

对每个基因来说，这些常数在一次运行的初期随机生成。但是它们的循环由变异、转座、重组等常用的遗传算子保证。另外，一种特殊的变异算子允许向随机常数集中一直引入变化，一种针对域的 IS 转座操作保证常数能够进行更广泛的交换。

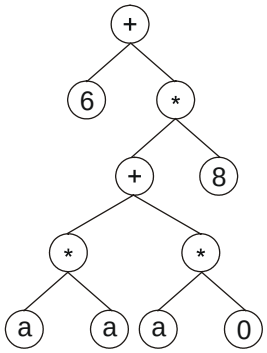
考虑下面这个单基因的染色体，其头部长度为 7（Dc 用黑体标识）：

```
012345678901234567890 12
+?*+?**aaa??aaa68083295
```
(4.8)

其中终点“？”代表随机常数。这种染色体的表达与以前完全相同：



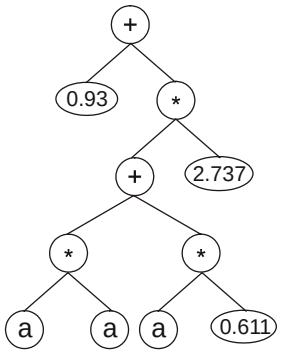
然后表达式树中的“？”从左到右，从上到下，由 Dc 中的符号替换，得到：



这些符号对应的值保存在一个数组中。为了简单起见，数值所代表的为数列中数的顺序。

```
A = {0.611,1.184,2.449,2.98,0.496,2.286,0.93,2.305,2.737,0.755}
```

由染色体（4.8）得到：



在这一章的后面部分我们将看到，对这种域进行编码的基因在参数优化和多项式推断时会有很大的优势。但是这种漂亮的结构也可以用来进化神经网络和阈值。该方法的具体操作见第 5 章。但是，现在我们将看到如何采用这些随机常数的域来进行符号回归。

4.2.2.解决常数产生问题的两种方法

这一节我们将通过比较两种不同算法的效果来分析符号回归中两种不同的常数产生办法。第一种方法采用对随机常数进行显式操作的便利条件，第二种方法则不含这种便利条件。两种方法的比较将在 3 个不同的问题上进行。第一个问题是人工的序列推断问题，它需要整型常数，第二个问题是一个函数发现的问题，它需要浮点型常数，第三个问题是现实世界的时间序列预测问题，也需要浮点型常数。

对序列推断问题，选择如下的测试序列：

$$a_n = 4n^4 + 3n^3 + 2n^2 + n \tag{4.9}$$

其中 n 由非负整数构成。选择这个序列的原因是因为两个算法都能精确地求解这个问题。这样就可以从成功率的角度来对它们的效果进行精确的度量。

对于函数发现问题，选择如下的 V 形函数：

$$y = 4.251a^2 + \ln(a^2) + 7.243e^a \tag{4.10}$$

其中 a 是自变量，e 是无理数 2.71828183。这种问题不能通过进化计算获得精确的解，然而我们可以通过比较最佳运行的平均适应度和最佳运行的平均 R-平方来比较两种方法的效果。

对时间序列预测的问题，采用 Wolfer 太阳黑子序列（见表 4.5）的观测值，其隐含的维数是 10，延时为 1（详见 4.4 节）。同样，我们可以通过比较最佳运行的平均适应度和最佳运行的平均 R-平方来比较两种方法的效果。

表 4.5 Wolfer 太阳黑子序列（按行读取）

101	82	66	35	31	7	20	92
154	125	85	68	38	23	10	24
83	132	131	118	90	67	60	47
41	21	16	6	4	7	14	34

45	43	48	42	28	10	8	2
0	1	5	12	14	35	46	41
30	24	16	7	4	2	8	17
36	50	62	67	71	48	28	8
13	57	122	138	103	86	63	37
24	11	15	40	62	98	124	96
66	64	54	39	21	7	4	23
55	94	96	77	59	44	47	30
16	7	37	74				

对于序列推断问题，将前 10 个正整数n及其相应项作为适应度样本，适应度函数基于相对误差，适应度由方程（3.1b）算得。选择范围为 25%，精度为最大值（误差为 0%），得到f_{max}=250。该实验中两种不同方法的具体设定如表 4.7。

对于V型函数问题采用区间[-1,1]中 20 个随机样本作为适应度样本。但是这里我们采用选择范围是 100%，得到f_{max}=2000。该这个实验中两种不同方法的设定见表 4.9。

对于时间序列预测问题，采用隐含的维数为 10，延时为 1，表 4.5 中的时间序列预测得到 90 个适应度样本（详见 4.4 节）。在此，采用选择范围更大，为 100%，得到f_{max}=90，000。在这个实验中的两种不同方法的设定见表 4.10。

表 4.6 序列推断问题的适应度样本集

n	1	2	3	4	5	6	7	8	9	10
$\mathbf{a_n}$	10	98	426	1252	2930	5910	10738	18056	28602	43210

4.2.2.1 直接对数值常数进行操作

为了利用直接操作随机函数常数来求解序列推断问题，函数集采用 F={+,-,*}，终点集选为 T={a,?}。另外，采用数值 0～9 来作为一组整型随机常数。即 R={0,1,2,3,4,5,6,7,8,9}。中间随机常数“ ？ ”的范围在 0,1,2,3 内，每次运行的参数如表 4.7 第一列所示。下图所示为第 0 次运行第 99 代发现的完美解：

```
Gene 0: ***a+*aaaa?a?4044044
     A0:{0,1,3,0,2,0,1,1,0,1}

Gene 1: **+*a?a?aa???3858227
     A1:{3,2,2,1,3,1,1,0,3,2}

Gene 2: /?a-?-a????aa5011303
     A2:{3,2,0,0,1,0,2,1,3,1}

Gene 3: -?-*-aa??a?aa5485938
A3:{3,0,1,2,0,3,3,1,3,3}

Gene 4: a**/*aaa?aaaa2416267
A4:{0,2,0,2,2,2,1,0,3,3}                    (4.11)
```

如其表达式所示，该程序与目标序列完全匹配。

如表 4.7 所示，对于该问题的成功的概率为 24%，明显低于第 2 种方法得到的 98%的

成功率。奇怪的是，将数值常数包括到进化工具箱中竟然导致效果变差。因此，如果不作任何假设，进化计算能够更容易地找到这些数值常数，值得指出的是，在这里，只有关于解的前提知识才能让我们知道如何正确选择随机常数的类型和范围。

表 4.7 序列推断问题中含随机常量（SI*）和不含随机常量（SI）的总体设置

	SI*	SI
Number of runs	100	100
Number of generations	100	100
Population size	100	100
Number of fitness cases	10	10
Function set	+ - * /	+ - * /
Terminal set	a, ?	a
Random constants array length	10	--
Random constants range	{0, 1, 2, 3}	--
Head length	6	6
Number of genes	7	7
Linking function	+	+
Chromosome length	140	91
Mutation rate	0.044	0.044
One-point recombination rate	0.3	0.3
Two-point recombination rate	0.3	0.3
Gene recombination rate	0.1	0.1
IS transposition rate	0.1	0.1
IS elements length	1,2,3	1,2,3
RIS transposition rate	0.1	0.1
RIS elements length	1,2,3	1,2,3
Gene transposition rate	0.1	0.1
Random constants mutation rate	0.01	--
Dc specific transposition rate	0.1	--
Dc specific IS elements length	1,2,3	--
Selection range	20%	20%
Precision	0%	0%
Average best-of-run fitness	179.827	197.232
Average best-of-run R-square	0.977612	0.999345
Success rate	16%	81%

为了通过显式采用随机常数发现 V 形函数，F＝{+,－,×,/,L,E,K,~,S,C}(L 代表自然对数，E 代表e^x，K 代表基数为 10 的对数，~ 代表10^x，S 代表正弦函数，C 代表余弦函数)，T＝{a,?}。有理数随机常数 R={0,1,2,3,4,5,6,7,8,9}，而且“ ？ ”的取值范围在区间[-1,1]上。每次运行的参数如图 4.9 第一列所示。在本实验的最佳运行中，最优解产生于第 3631 代。其基因组如下（子表达式树由加法连接）：

```
Gene 0: ++?~a*???a?0643987
```

```
A0:{0.03,0.019,0.419,0.247,0.367,
    0.689,0.648,0.245,0.252,0.904}

Gene 1: ~S/aSE?a?aaaa5670520
A1:{0.476,0.477,0.786,0.196,0.92,
    0.344,0.934,0.198,0.107,0.575}

Gene 2: ~CCLE*aa?a??a1654524
A2:{0.298,0.083,0.466,0.404,0.912,
    0.782,0.392,0.223,0.548,0.43}

Gene 3: /LE/-*?aaaaa?4324454
A3:{0.947,0.357,0.549,0.338,0.465,
    0.435,0.837,0.776,0.602,0.593}

Gene 4: ~*aCaE???aaa?8125208
A4:{0.189,0.357,0.494,0.38,0.465,
    0.88,0.995,0.971,0.966,0.294}                                    (4.12a)
```

它在 20 个适应度样本上算得的适应度为 1978.177。R-平方为 0.999914。在选自[-1,1]中的 100 个随机点构成的测试集上算得的 R-平方为 0.999216。形式化一点，它对应如下的C++函数：

```
double APSCfunction(double d[])
{
  double dblTemp = 0 ;
  dblTemp += ((pow(10,(d[0]*0.648))+d[0])+0.03);
  dblTemp += pow(10,sin((d[0]/sin(exp(0.344)))));
  dblTemp += pow(10,cos(cos(log(exp((d[0]*d[0]))))));
  dblTemp += (log(((d[0]*d[0])/0.465))/exp((d[0]-d[0])));
  dblTemp += pow(10,(d[0]*cos(d[0])));
return dblTemp;
}                                                                    (4.12b)
```

其中d_0对应自变量。

该模型与目标函数非常近似，从 R-平方目标函数及模型的比较图可以看出。

表 4.8 “ V ”形函数问题的适应度样本集

a	f(a)
-0.2639725157548	3.19498066265276
0.0578905532656938	1.99052001725998
0.334025290109634	8.39663703997286
-0.236334577564462	3.07088976972825
-0.855744382566804	5.87946763695703
-0.0194437136332785	-0.775326322328458
-0.192134388183304	2.83470225774408
0.529307910124627	12.2154726642137

-0.00788974118728459	-2.49803983418635
0.438969804950631	10.4071734858808
-0.107559292698039	2.09413635645908
-0.274556994377163	3.23927278010839
-0.0595333219604528	1.19701284767347
0.384492993958352	9.35580769189855
-0.874923020736333	6.00642453001302
-0.236546636250546	3.07189729043837
-0.167875941704557	2.67440053130986
0.950682181822091	22.4819639844149
0.946979159577362	22.3750161187355
0.639339910059591	14.5701285332337

最后，对采用随机数值常数预测太阳黑子的问题，F={+,-,*,/}，T={a,b,c,d,e,f,g,h,i,j,?}，其中字母分别代表 t-10,t-9,…，t-1。有理数随机常数集 R={0,1,2,3,4,5,6,7,8,9}。且“？”的范围在区间[-1,1]上。每次运行的参数如图 4.10 的第一列所示。在第 90 次运行的第 4945 代所发现的最优解如下图所示（子表达式树由加法连接）

```
Gene 0: /*+i*h+?hfddh?c29898626
A0:{0.387,0.375,0.373,0.606,0.363,
    0.958,0.211,0.062,0.12,0.239}

Gene 1: *j/j++ii?afc?ad03960507
A1:{0.274,0.472,0.153,0.38,0.887,
    0.828,0.378,0.779,0.414,0.065}

Gene 2: /+++a++agee?gjd89794029
A2:{0.433,0.503,0.333,0.092,0.066,
    0.245,0.592,0.43,0.051,0.998}
```
(4.13a)

在 90 个适应度样本上算得的适应度为 86733.21，R-平方为 0.84312712。其对应的数学函数为：

$$y = \frac{0.377hi}{d+f+h} + \frac{j^2}{0.274+2i} + \frac{2a+g}{0.051+2e+g} \tag{4.13b}$$

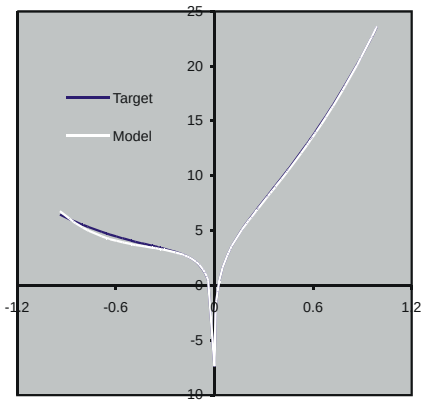


图 4.7.目标函数与利用控制随机常量的 GEP 进化得到的模型之间的比较。R-平方是在 100 个随机点构成的

测试集上得到的，其大小为 0.999216。

表 4.9.“ V ”函数问题中含随机常量（V*）和不含随机常量（V）的总体设置

	V*	**V**
Number of runs	100	100
Number of generations	5000	5000
Population size	100	100
Number of fitness cases	20 (Table 4.8)	20 (Table 4.8)
Function set	+ - * / L E K ~ S C	+ - * / L E K ~ S C
Terminal set	a, ?	a
Random constants array length	10	--
Random constants range	[-1,1]	--
Head length	6	6
Number of genes	5	5
Linking function	+	+
Chromosome length	100	65
Mutation rate	0.044	0.044
One-point recombination rate	0.3	0.3
Two-point recombination rate	0.3	0.3
Gene recombination rate	0.1	0.1
IS transposition rate	0.1	0.1
IS elements length	1,2,3	1,2,3
RIS transposition rate	0.1	0.1
RIS elements length	1,2,3	1,2,3
Gene transposition rate	0.1	0.1
Random constants mutation rate	0.01	--
Dc specific transposition rate	0.1	--
Dc specific IS elements length	1,2,3	--
Selection range	100%	100%
Precision	0%	0%
Average best-of-run fitness	1896.25	1953.057
Average best-of-run R-square	0.95129456	0.99647004

表 4.10.太阳黑子序列预测问题中含随机常量（SS*）和不含随机常量（SS）的总体设置

	SS*	**SS**
Number of runs	100	100
Number of generations	5000	5000
Population size	100	100
Number of fitness cases	90 (Table 4.5)	90 (Table 4.5)
Function set	4 (+ - * /)	4 (+ - * /)
Terminal set	a - j, ?	a - j
Random constants array length	10	--
Random constants range	[-1,1]	--

Head length	7	7
Number of genes	3	3
Linking function	+	+
Chromosome length	69	45
Mutation rate	0.044	0.044
One-point recombination rate	0.3	0.3
Two-point recombination rate	0.3	0.3
Gene recombination rate	0.1	0.1
IS transposition rate	0.1	0.1
IS elements length	1,2,3	1,2,3
RIS transposition rate	0.1	0.1
RIS elements length	1,2,3	1,2,3
Gene transposition rate	0.1	0.1
Random constants mutation rate	0.01	--
Dc specific transposition rate	0.1	--
Dc specific IS elements length	1,2,3	--
Selection range	1000%	1000%
Precision	0%	0%
Average best-of-run fitness	86182.05	89009.66
Average best-of-run R-square	0.706437	0.801144

值得注意的是，采用这种方法，算法实际上在进化的解中整合了某些常数，但是这些常数，至少在某些情况下，与我们所期望的存在很大不同（例如比较 V 形函数的模型和 V 形函数本身）。的确，如果目标函数是有理数系数的简单多项式，GEP（而且我相信所有的进化算法都）可以发现精度达到小数点后 3 位或 4 位的我们所期望的常数。否则，找到的解将会非常有创意，例如对于下面这个简单函数：

$$y = 2.718a^2 + 3.1416a \tag{4.14}$$

就可以通过操作随机常数来精确求解（其效果和参数见表 4.11）。例如，考虑如下这个近似完美的解：

```
Gene 0: *+++*+?aaa?a?4673929
    A0:{0.244,0.966,0.461,0.762,0.409,
        0.567,0.718,0.746,0.993,0.236}

Gene 1: **??aa???a??a8610121
    A1:{0.103,0.086,0.038,0.793,0.307,
        0.062,0.051,0.025,0.046,0.911}
```

(4.15a)

表 4.11 包含有理数系数的多元函数的设置

Number of runs	100
Number of generations	1000
Population size	50
Number of fitness cases	10 (Table 4.12)
Function set	+ - * /

Terminal set	a ?
Random constants array length	10
Random constants range	[-1,1]
Head length	6
Number of genes	2
Linking function	+
Chromosome length	40
Mutation rate	0.044
One-point recombination rate	0.3
Two-point recombination rate	0.3
Gene recombination rate	0.1
IS transposition rate	0.1
IS elements length	1,2,3
RIS transposition rate	0.1
RIS elements length	1,2,3
Gene transposition rate	0.1
Random constants mutation rate	0.01
Dc specific transposition rate	0.1
Dc specific IS elements length	1,2,3
Selection range	25%
Precision	0%
Average best-of-run fitness	243.994
Average best-of-run R-square	0.999826

其 R-平方等于 0.999999999769。因此它几乎与目标函数完美匹配，的确，如其数学表达式所示，算法发现的方程（4.14）的数值常数的精度非常高：

$$y = 2.718a^2 + 3.141636a \qquad (4.15b)$$

表 4.12 多元函数的适应度样本集

a	f(a)
-8.5013	169.7278883
-0.8696	-0.676572453
3.7181	49.25514232
5.0878	86.34118911
-4.313	37.01043094
1.9775	16.84126999
-8.767	181.3638583
-5.5617	66.60191701
-1.4234	1.035098188
6.9014	151.1379353

4.2.2.2.从头产生数值常数

为了不采用对数值常数进行操作的方法来解决序列推断的问题，我们采用与随机常数的

实验中相同的函数集合，终点集显然只含有自变量。

如表 4.7 第 2 列所示，该方法的成功概率为 98%，远远高于用直接控制随机常数方法所得到的 24%。第一个完美解产生于第 0 次运行的第 89 代。如下图所示（子表达式树由加法连接）：

```
0123456789012
**+++*aaaaaaa
*+aaa-aaaaaaa
-a*a/aaaaaaaa
-**a/*aaaaaaa
*--aaaaaaaaaa
```
(4.16)

如其表达式所示，该程序与目标序列相对应，注意该算法是如何通过简单的算术运算从头找到所需常数的，为了不采用对数值常数进行操作的方法来发现 V 形函数，我们选取的函数集与第一种方法完全相同。有了这些函数（其中绝大部分是超越函数），该算法有各种各样的方法来进化高度精确的解，而不需要显式采用数值常数，每次运行的参数如表 4.9 的第 2 列。

该实验的最优解是在第 39 次运行的第 2790 代发现的（子表达式树由加法连接）：

```
0123456789012
+L+*a~aaaaaaa
*aCK+/aaaaaaa
*S*C~Caaaaaaa
*~CEC-aaaaaaa
 S*SC+/aaaaaaa
```
(4.17a)

它在 20 个适应度样本上算得的适应度为 1991.887，R-平方为 0.99992182，与第一种方法使用相同测试集。因此，该模型比采用控制随机常数得到的模型要好，更正式一点，模型(4.17a)可以用以下 C＋＋函数来表示：

```
double APSCfunction(double d[])
{
 double dblTemp = 0 ;
 dblTemp += (log((d[0]*d[0]))+(d[0]+pow(10,d[0])));
 dblTemp += (d[0]*cos(log10(((d[0]/d[0])+d[0]))));
 dblTemp += (sin(cos(d[0]))*(pow(10,d[0])*cos(d[0])));
 dblTemp += (pow(10,exp((d[0]-d[0])))*cos(cos(d[0])));
 dblTemp += sin((sin((d[0]+d[0])))*cos((d[0]/d[0])));
 return dblTemp;
}
```
(4.17b)

其中d_0对应自变量a，目标函数和GEP进化得到的模型比较图如图 4.8 所示。

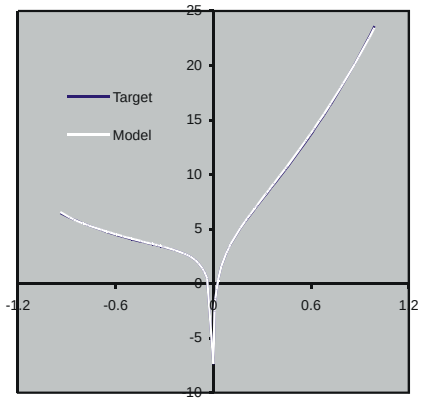


图 4.7.目标函数与不采用显式随机常量的 GEP 进化得到的模型之间的比较。R-平方是在 100 个随机点构成的测试集上得到的，其大小为 0.9998882。

为了不显式采用随机常数来进行太阳黑子预测，我们选取的函数集与第一种方法完全相同。每次运行的参数如表 4.10 的第 2 列。该实验的最优解是在第 45 次运行的第 2284 代发现的（子表达式树由加法连接）：

```
012345678901234
//+*-dehjiecfce
ji--ac/efbhjcjf
/*+-ji+biaifgdf
```
(4.18a)

它在 90 个适应度样本上算得的适应度为 89184.62，R-平方为 0.884351，因此，该模型比采用控制随机常数得到的模型（4.13）要好，更正式一点，模型(4.18a)可以用以下 C＋＋函数来表示：

$$y = \frac{hj}{(d+e)(i-e)} + j + \frac{j(b-i)}{a+2i} \qquad (4.18b)$$

我们通过比较两种方法得到的结果可以得到很多启示。所有的实验中，显式采用随机常数得到的结果相比之下都要差很多。在序列推断问题中，如果由算法从头产生随机常数，则成功率要高很多。更确切的说，两种方法的成功率分别为 98%和 24%（见表 4.7）。在 V 形函数问题中，当允许算法本身去发明表达数值常数的方法时，最佳运行的平均适应度也要明显高出许多，在此，两种方法得到的最佳运行的平均适应度分别为 1953.057 和 1896.25，最佳运行的 R-平方分别为 0.99647004 和 0.95129456。在太阳黑子预测中，两种方法得到的最佳运行的平均适应度分别为 89009.66 和 86182.05，最佳运行的 R-平方分别为 0.801144 和 0.706437(见表 4.10)。因此，在对复杂的现实问题建模时，当数值常数的类型与范围均不可知，而且大部分情况下不可能猜出准确的函数集时，让系统自己去模拟现实则更合适一些，即采用不含有随机常数的工具箱。这样，不仅结果更好，而且系统的复杂度也会更小。

下一节，我们将讨论一些绝对需要随机常数的问题。

4.3 参数优化

函数优化的目的是要发现一组参数值使复杂的多维函数最大化（或最小化）。有时候使函数最小化更有意义。为了将函数最小化，只需要将函数乘以－1，然后求其最大值即可。这一节里面我们将看到 GEP 的用多基因染色体来搜索多维函数的最大值（最小值）。对于这一类问题而言，多基因染色体的每个基因都能够用来发现函数优化任务中不同参数的值。

4.3.1.多基因染色体和多维参数优化

GEP 的多基因结构可以用来有效地搜索以解决函数优化的问题。在此，函数的 N 个参数的值编码在 N 个不同的基因里面。

遗传算法已经被广泛运用于参数优化中（例如 Goldberg 1989，Haupt 和 Haupt 1998）。在其中，GAs 的简单染色体对不同参数的值进行编码。不管是采用二进制或浮点格式，函数的参数都被直接编码在一个简单的 GA 染色体中。但是，在 GEP 中，最佳参数是通过在不停变化的随机数值常数上的数学运算中发现的。在我看来，这种方法提供了新的可能性，因为它使我们能够利用大量遗传算子集来对不同参数进行微调并发现常数的恰当范围。

因此，为了用 GEP 解决这一类问题，我们打算采用前一节介绍的处理随机数值常数的染色体组织结构。对绝大多数优化问题而言，所需的函数集通常都非常简单而且通常由基本算术运算符构成。终点集仅仅由中间随机常数构成，因此 T={?}。随机数值常数集的选取也十分容易，通常，由 10 个随机常数构成的集合对大部分问题来说就能够达到很好的效果，即 R={0,1,2,3,4,5,6,7,8,9}。最后中间随机常数“ ？ ”的范围在区间[-1,1]上。这样的参数设置对于函数优化的问题几乎可以通用，因为算法本身就是用来发现常数的恰当的范围的。

考虑如下染色体，它由两个基因构成：

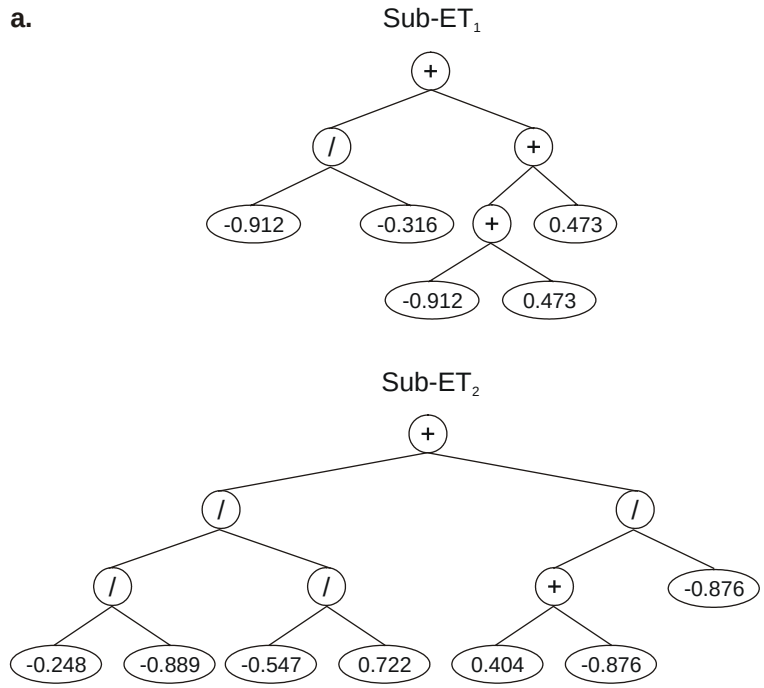
```
Gene 1:+/+??+???????4374796
A1:{-0.698,0.781,-0.059,-0.316,-0.912,
     0.398,0.157,0.473,0.103,-0.756}

Gene 2:+////+???????4562174
A2:{0.104,0.722,-0.547,-0.052,-0.876,
   -0.248,-0.889,0.404,0.981,-0.149}
```
(4.19)

其表达式如图 4.9 所示。读者应该可以看到，第一个基因将第一个参数p_1的值编码为 2.92008，第二个基因将第二个参数p_2的值编码为 0.170599。

这种染色体的适应度由f(p1,p2)构成，其中f是我们要优化的函数。根据函数的不同，某个特定参数集的返回值可能在实际直线的任意位置，而且必须对负适应度和零适应度进行处理。为了解决这个问题，对每一代而言，计算每代中的最差适应度f_{min}，如果f_{min}为负值，则把f_{min}的绝对值加 1 加到适应度上。这种方法能够保证所有的个体的适应度值均是正值，而适应度较差的个体的适应度值为 1。这种变换后的适应度可以用来对个体进行选择，让其参与带修饰的繁殖。

a.



b. $f(p_1, p_2) = f(2.92008, 0.170599)$

图 4.9.一个函数优化问题中对参数进行编码的染色体的表达。**a)**编码在程序(4.19)中的子表达式树。**b)**每个基因中所编码的不同的参数值。该程序的适应度是在点（p_1,p_2）处的函数值。

4.3.2.用 GEP 搜索最大值

这一节我们将寻找两个不同函数的最大值。第一个，即下面这个函数是一个大家广泛研究的二元参数函数：

$$f(x, y) = -x\sin(4x) - 1.1y\sin(2y) \qquad (4.20a)$$

约束条件为：

$$0 \le x \le 10 \text{ 且 } \qquad 0 \le y \le 10 \qquad (4.20b)$$

因为该函数的最大值已知，所以，为了简化，我们将其视为 18.5。

这种解已知的函数虽然简单，但是用它们来精确度量不同算法的效果却十分有用。例如，上面函数（4.20）的全局最小值可以通过采用传统的搜索算法如 Nelder-Meader 或者 Broyden -Fletcher- Golderfarb- Shanno (BFGS)算法（Haupt 和 Haupt 1998）搜索得到。另一方面，一些不太传统的方法例如 GAs 或者 GEP 要找到这个函数的全局最小值也没有任何问题（表 4.13）。的确，考虑 18.5 这个函数（4.20）的最大输出值，GEP 及 h=0 的 GA 模拟（GEP-HZero）都能够在所有的运行中找到准确的参数使函数的返回值等于或大于 18.5。如表 4.13 所示，对于这个简单的问题，GA 模拟的效果比 GEP 的效果稍好一些。如果考虑 GEP 的复杂度较高，显然采用较为简单的 GA 模拟来解决一维或者二维的函数优化问题会更加合理一些。然而，大家知道基本遗传算法容易过早收敛，经常陷入一些局部最优。在这种情况下，GEP 的微调能力就显得必不可少，第二个问题的结果将会说明这一点。

表 4.13 采用 GEP 头部长度为 0 来模拟 GA 的方法（**GEP-HZero**）和基因表达式编程方法（GEP）求解 2-参数函数优化问题的设置

	GEP-HZero	**GEP**
Number of runs	100	100
Number of generations	1000	1000
Population size	30	30
Function set	--	+ - * /
Terminal set	?	?
Random constants array length	1	10
Random constants range	[0, 10]	[-1, 1]
Head length	0	6
Number of genes	2	2
Chromosome length	4	40
Mutation rate	--	0.044
One-point recombination rate	--	0.3
Two-point recombination rate	--	0.3
Gene recombination rate	0.8	0.1
IS transposition rate	--	0.1
IS elements length	--	1,2,3
RIS transposition rate	--	0.1
RIS elements length	--	1,2,3
Gene transposition rate	0.2	0.1
Random constants mutation rate	0.25	0.01
Dc specific transposition rate	--	0.1
Dc specific IS elements length	--	1,2,3
Average best-of-run output	18.5544	18.5351
Success rate	100%	93%

该函数是 4.1.2 中的 5 个参数的函数：

$$f(p_1, p_2, p_3, p_4, p_5) = \frac{\sin(p_1) \cdot \cos(p_2)}{\sqrt{e^{p_3}}} + \tan(p_4 - p_5) \qquad (4.21a)$$

约束条件为：

$$0 \le p_1, \cdots, p_5 \le 10 \qquad (4.21b)$$

该函数的全局最优解目前我们并不知道，因此我们不能从成功率的角度来比较不同算法的效果；因此，我们将采用平均最佳运行输出作为比较。

如表 4.14 所示，在该问题中，GEP的效果明显好于基本遗传算法模拟。其主要原因就是我们前面已经提到的GA容易过早收敛到局部最大值。例如，在表 4.14 第一列所示的实验中，GA所找到的最大值等于 272243，而且GA反复找到这个值（精确地说，在 8 次不同的运行中）。这种情况在GEP中从来没有出现过。令人感到奇怪的是，在所有的 100 次运行中，GEP从来没有找到过GA一直陷入的那个局部最大值。相反，它找到的一些更高的峰值，包括 293686，342089，346903，1.90464×10^6，3.85419×10^6。最高峰值的参数值为：

$$f(p_1, p_2, p_3, p_4, p_5) = (1.5701, 0.00071129, 0, 1.58036, 0.00956748)$$

这一点的函数值为：

$$f(p_1, p_2, p_3, p_4, p_5) = 3.85419 \times 10^6$$

表 4.14 采用 GEP 头部长度为 0 来模拟 GA 的方法（**GEP-HZero**）和基因表达式编程方法（GEP）求解 5-参数函数优化问题的设置

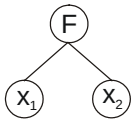
	GEP-HZero	**GEP**
Number of runs	100	100
Number of generations	1000	1000
Population size	30	30
Function set	--	+ - * /
Terminal set	?	?
Random constants array length	1	10
Random constants range	[0, 10]	[-1, 1]
Head length	0	6
Number of genes	5	5
Chromosome length	10	100
Mutation rate	--	0.044
One-point recombination rate	--	0.3
Two-point recombination rate	--	0.3
Gene recombination rate	0.8	0.1
IS transposition rate	--	0.1
IS elements length	--	1,2,3
RIS transposition rate	--	0.1
RIS elements length	--	1,2,3
Gene transposition rate	0.2	0.1
Random constants mutation rate	0.25	0.01
Dc specific transposition rate	--	0.1
Dc specific IS elements length	--	1,2,3
Average best-of-run output	34691.2	98096.8

4.4.时间序列预测

这一节为了进化一些高次多变量多项式，我们将继续探究随机常数域的思想。然后我们将通过比较新的算法 GEP-KGP（含 Kolmogorov-Gabor 多项式的 GEP）和较为简单的 GEA 的性能来讨论这些所谓的 Kolmogorov-Gabor 多项式在进化计算中的重要性。

4.4.1. Kolmogorov-Gabor 多项式的进化

Kolmogorov-Gabor 多项式已经在进化通用非线性模型中被广泛使用（Iba 和 Sato1992，Iba 1994，Ivakhnenko 1971，Kargupta 和 Smith 1991，Nikolaev 和 Iba 2001）。采用 GEP 来进化这样的多项式是非常直接的，只需要生成两个参数的特殊函数。例如，下面的树：



对应：

$$y = a_0 + a_1x_1 + a_2x_2 + a_3x_1x_2 + a_4x_1^2 + a_5x_2^2 \qquad (4.22)$$

它表示一个二元二次多项式函数。系数a_0-a_5可以很容易通过含有随机常数的特殊的Dc域来进化得到。和前面一样，Dc域紧跟在尾部之后，但是，在这里它的长度等于 6h。例如，考虑如下的h＝2 的染色体（Dc域用粗体标识）：

```
01234567890123456
FFabc252175089728
```
(4.23)

对于如下的数值常量集：

```
A={-0.606,-0.398,-0.653,-0.818,-0.047,0.036,0.889,0.148,-0.377,-0.841}
```

其表达式为：

$$y = -0.606 - 0.377y_1 - 0.841a + 0.148y_1a - 0.653y_1^2 - 0.377a^2 \qquad (4.24)$$

其中 $y = -0.653 + 0.036b - 0.653c - 0.398bc + 0.148b^2 + 0.036c^2$。

这里描述的双参数函数“ F ”是STROGANOFF（Ivakhnenko 1971）中所使用的，它由完全的二元多项式构成。在表 4.15 中，该函数由F_9来表示。实现时用来增强STROGANOFF的一些小的变化如不完全的二元多项式也非常简单。所有这些函数见表 4.15。

下面这个函数也很容易实现：

$$y = x_1 + x_2 + x_1x_2 + x_1^2 + x_2^2 \qquad (4.25)$$

它除了系数以外，与函数（4.22）或F_9完全相同。这个函数及类似的函数都是基于表 4.15 中所给出的 16 个二元多项式，我们将把它们包含在GEP的函数工具箱中来分析多项式进化中数值常数的重要性。它们的具体描述见表 4.16。

表 4.15 二元多项式

$F_1 = a_0 + a_1x_1 + a_2x_2 + a_3x_1x_2$
$F_2 = a_0 + a_1x_1 + a_2x_2$
$F_3 = a_0 + a_1x_1 + a_2x_2 + a_3x_1^2 + a_4x_2^2$
$F_4 = a_0 + a_1x_1 + a_2x_1x_2 + a_3x_1^2$
$F_5 = a_0 + a_1x_1 + a_2x_2^2$
$F_6 = a_0 + a_1x_1 + a_2x_2 + a_3x_1^2$
$F_7 = a_0 + a_1x_1 + a_2x_1^2 + a_3x_2^2$
$F_8 = a_0 + a_1x_1^2 + a_2x_2^2$
$F_9 = a_0 + a_1x_1 + a_2x_2 + a_3x_1x_2 + a_4x_1^2 + a_5x_2^2$
$F_{10} = a_0 + a_1x_1 + a_2x_2 + a_3x_1x_2 + a_4x_1^2$
$F_{11} = a_0 + a_1x_1 + a_2x_1x_2 + a_3x_1^2 + a_4x_2^2$
$F_{12} = a_0 + a_1x_1x_2 + a_2x_1^2 + a_3x_2^2$
$F_{13} = a_0 + a_1x_1 + a_2x_1x_2 + a_3x_2^2$
$F_{14} = a_0 + a_1x_1 + a_2x_1x_2$
$F_{15} = a_0 + a_1x_1x_2$
$F_{16} = a_0 + a_1x_1x_2 + a_2x_1^2$

所以，通过选择一个仅由表 4.15 中的F_9构成的函数集，我们就能够准确模拟出一个GEP中的STROGANOFF系统。而且通过选择一个由表 4.15 中的F_1-F_{16}构成的函数集我们就能够准确模拟出增强的STROGANOFF系统。这些系统的效率将在下一节进行分析。

表 4.16 具有单位系数的二元多项式

$F_1 = x_1 + x_2 + x_1x_2$
$F_2 = x_1 + x_2$
$F_3 = x_1 + x_2 + x_1^2 + x_2^2$
$F_4 = x_1 + x_1x_2 + x_1^2$
$F_5 = x_1 + x_2^2$
$F_6 = x_1 + x_2 + x_1^2$
$F_7 = x_1 + x_1^2 + x_2^2$
$F_8 = x_1^2 + x_2^2$
$F_9 = x_1 + x_2 + x_1x_2 + x_1^2 + x_2^2$
$F_{10} = x_1 + x_2 + x_1x_2 + x_1^2$
$F_{11} = x_1 + x_1x_2 + x_1^2 + x_2^2$
$F_{12} = x_1x_2 + x_1^2 + x_2^2$
$F_{13} = x_1 + x_1x_2 + x_2^2$
$F_{14} = x_1 + x_1x_2$
$F_{15} = x_1x_2$
$F_{16} = x_1x_2 + x_1^2$

如前所述，在 GEP 中实现这两个（原始的和增强的）系统的时候都需要一个长度为头部长度 6 倍的 Dc 域，这是由多项式所要求的参数个数决定的。因此，为了更方便，我们采用稍有不同的形式来表示每个系数所对应的随机常数。例如，对于下面这个 h=4 的基因：

```
012345678
NoeEgbcaj                                    (4.26)
```

需要一个长度为 24 的Dc。如果我们选择含有 20 个数字的随机数值常数集并用R={r_0,…,r_{19}}来表示它们，下面的结构就可以成为上面染色体（4.26）的一个合法的Dc域：

```
r19r12r15r9r2r19r13r6r5r19r12r7r17r10r7r15r7r18r9r11r15r16r1r17
```

和前面一样，所有这些随机常数的值均保留在一个数组中，可以在需要的时候取出来。

由于随机常数域的维数巨大，因此控制其中的遗传变化的度则非常重要。所以，我们这里实现一种允许既能在头/尾域上也能在 Dc 域上自主控制变异概率的特殊变异算子。其它算子保持不变，其工作原理和 4.2 节所描述的一样。

4.4.2.用 GEP 模拟 STROGANOFF 和增强的 STROGANOFF

原始的 STROGANOFF 和增强的 STROGANOFF 的效率是在表 4.5 中的太阳黑子数据上计算得到的，如图 4.17 所示。如我们所料，增强的 STROGANOFF 的效果明显好于原始的 STROGANOFF。然而请注意，在 GEP-ESM 实验中采用了一个 3-基因系统，因此我们这里所模拟的系统与 Nikolaev 和 Iba（2001）所描述的增强的 STROGANOFF 并不对应。但是我们所采用的算法更有效，因为它受益于 GEP 的多基因天性。的确，多基因系统的效果明显比单基因系统的效果要好。值得强调的是，在 GP 中多个分列树难以实现，所以 GP 和 GEP-ESM 实验中采用的系统很相似。当然，GP 中所采用的随机常数控制的方法远没有 GEP 中所采用的随机常数控制的方法功能强大，而且，实际上，GP 中的系数仅能发现 posteriori，通常由神经网络使用。这显然提出了一个问题，在这里 GP 到底起到了什么作用，因为在没

有系数的情况下，多项式是无用的。

的确，继续我们关于进化的符号回归中随机常数的重要性的讨论，我们可以作一个简单的实验。我们可以实现表 4.16 中给出的二元多项式，并尝试用它们来进化得到更复杂的多项式模型（表 4.18）。另一方面，图 4.17 中两个实验的比较说明系数对于多项式的进化至关重要。但是真正的带系数的多项式并不比简单的 GEA 高效（对比表 4.10 和表 4.17），而且数值常数在进化的符号回归中的作用并不重要，它只有理论意义。

表 4.17 用 GEP 模拟 STROGANOFF（**GEP-OS**）以及用单基因系统(**GEP-ESU**)和多基因系统(**GEP-ESM**)模拟增强的 STROGANOFF。

	GEP-OS	**GEP-ESU**	**GEP-ESM**
Number of runs	100	100	100
Number of generations	5000	5000	5000
Population size	100	100	100
Number of fitness cases	90	90	90
Function set	$(F9)_{16}$	F1 - F16	F1 - F16
Terminal set	d0 - d9	d0 - d9	d0 - d9
Random constants array length	120	120	40
Random constants range	[-1,1]	[-1,1]	[-1,1]
Head length	21	21	7
Number of genes	1	1	3
Linking function	--	--	+
Chromosome length	169	169	171
Head/tail mutation rate	0.044	0.044	0.044
Dc mutation rate	0.06	0.06	0.06
One-point recombination rate	0.3	0.3	0.3
Two-point recombination rate	0.3	0.3	0.3
Gene recombination rate	--	--	0.1
IS transposition rate	0.1	0.1	0.1
IS elements length	1,2,3	1,2,3	1,2,3
RIS transposition rate	0.1	0.1	0.1
RIS elements length	1,2,3	1,2,3	1,2,3
Gene transposition rate	--	--	0.1
Random constants mutation rate	0.25	0.25	0.25
Dc specific transposition rate	0.1	0.1	0.1
Dc specific IS elements length	5,7,9	5,7,9	5,7,9
Selection range	1000%	1000%	1000%
Precision	0%	0%	0%
Average best-of-run fitness	86069.183	86566.298	86881.997
Average best-of-run R-square	0.4949506	0.6712016	0.7631128

由这里的实验我们可以得到一些结论。第一，采用像 STROGANOFF 这样的系统来搜索二元二次多项式，虽然在数学上看来很有吸引力，但是从进化的角度来说效率十分低下。不仅其效果明显较差而且其结构相对来说要复杂得多。第二，系数的发现，如我们所料，对

于建立基于高次多元变量多项式的模型来说非常重要。最后，包含普通数学函数集的简单 GEP 系统比复杂的、计算代价较高的 STROGANOFF 系统的效率高出许多。

表 4.18 多项式进化中系数的作用。

Number of runs	100
Number of generations	5000
Population size	100
Number of fitness cases	90
Function set	F1 - F16
Terminal set	d0 - d9
Head length	7
Number of genes	3
Linking function	+
Chromosome length	45
Mutation rate	0.044
One-point recombination rate	0.3
Two-point recombination rate	0.3
Gene recombination rate	0.1
IS transposition rate	0.1
IS elements length	1,2,3
RIS transposition rate	0.1
RIS elements length	1,2,3
Gene transposition rate	0.1
Selection range	1000%
Precision	0%
Average best-of-run fitness	73218
Average best-of-run R-square	0.0

下一节我们将采用较优的算法（含有基本算术函数的简单 GEA）来进化得到一个模型对太阳黑子进行预测。

4.4.3 用 GEP 进行太阳黑子预测

这一节我们将探寻时间序列预测中的基本步骤。具体说来，我们将进行太阳黑子预测，但是我们可以预测任何事情，从金融市场到豌豆价格，因为这些任务可以采用相同的时间序列框架来解决。

时间序列预测是符号回归的一个特例，因此，可以采用我们所熟悉的类似的 GEP 框架来实现。的确，我们只需要准备好数据来适应一个正常的符号回归任务就可以。对于本问题，我们将采用表 4.5 的太阳黑子时间序列。该时间序列的一种不同的表示方法见图 4.10。下面让我们来看看应该如何为时间序列分析准备数据。

该数据代表在某个时间间隔基础上进行的一系列观测，这里的时间间隔为一年。时间序列预测背后的思想是根据过去的观测预测未来的情况。这意味着，从实际的角度来说，我们试图找到一个预测模型，该模型是过去一定数量的观测数据的模型。用时间序列分析的术语

来说，这些一定数量的过去的观测数据叫做内嵌维数 d。对于太阳黑子预测任务我们将要采用 d＝10。时间序列分子中还有一个重要的参数－延迟时间 τ－决定数据如何处理。时间延迟为 1 时意味着数据是连续处理的，而 τ 较大则说明数据处理的时候跳过了一些观测数据。例如，采用 d=10，τ=1，表 4.5 的太阳黑子序列得到：

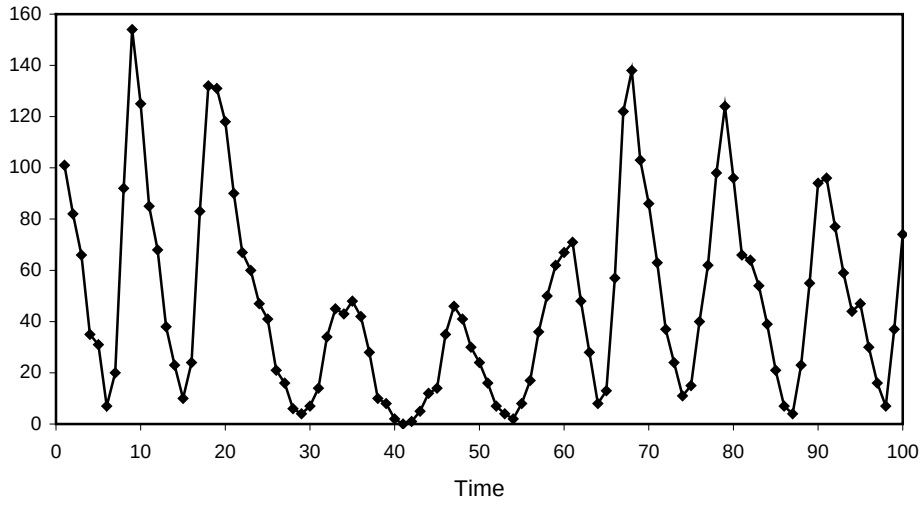


图 4.10.Wolfer 太阳黑子序列（另见表 4.5）

	t-10	t-9	t-8	t-7	t-6	t-5	t-4	t-3	t-2	t-1	t
1.	101	82	66	35	31	7	20	92	154	125	85
2.	82	66	35	31	7	20	92	154	125	85	68
3.	66	35	31	7	20	92	154	125	85	68	38
…						...					
89.	55	94	96	77	59	44	47	30	16	7	37
90.	94	96	77	59	44	47	30	16	7	37	74

在本书的所有时间序列预测问题中，我们将采用内嵌位数大小为 10，延迟时间为 1。如读者所看到的，现在时间序列数据已经可以在符号回归分析中使用了，其中（t-10）到（t-1）是自变量，t 是因变量。

现实世界任务的目标是找到一个模型，然后采用这个模型来进行预测。模型通常用来检验接下来的一天或者一个小时内的情况，这依赖于观测的频率。但是这里我们可以采用 Wolfer 太阳黑子的前 80 个观测数据得到进化模型（训练），用后 10 个数据作预测（检验）来模拟一个现实环境。通过该方法我们可以对 GEP 进化得到的模型的预测精度进行评价。

下面让我们尝试找一个模型来解释并预测太阳黑子的情况。选择太阳黑子实验中能够得到最佳结果的参数将会是一个很好的起点（表 4.10 的第 2 列）。然后，采用 APS 软件，通过一系列的优化运行，我们可以找到当前最好的染色体结构，即直到系统停止改进为止。然后向系统中加入一个中性基因，然后系统再一次尽其所能的进行搜索，如此这样，直到引入一个中性基因的时候系统没有改观，最佳适应度不再有任何改进为止。大体上说，这种过程可以让我们在任何精度要求下找到任何连续函数的近似，只要所需要的项足够多。这里，加入的中性基因就是潜在的新的项，而该软件允许这些基因成功地溶进这些等式中去。例如，下面这个模型就是通过 5 个这样的循环得到的（每次增加一个中性基因）：

```
double APSCfunction(double d[])
{
    double dblTemp = 0;
    dblTemp += (d[9]+((d[8]/(d[5]+d[4]))+((d[9]-d[9])*d[0])));
    dblTemp += (d[9]/(((((d[3]+d[3])+d[2])-d[9])+d[8]));
    dblTemp += (d[9]/(d[4]+d[6]));
    dblTemp += (d[9]/(d[4]+d[2]));
    dblTemp += (d[5]/(d[2]-d[6]));
    dblTemp += (d[3]/(d[1]-d[5]));
    dblTemp += (d[1]/(d[7]-d[0]));
    dblTemp += ((((d[2]-d[8])*d[9])+(d[0]+d[0]))/(d[7]+(d[8]+d[2])));
    return dblTemp;
}
```
(4.27)

其中d_0-d_9分别代表（t-10）到（t-1）。该模型的R-平方等于 0.94974095。如图 4.11 和 4.12 所示，GEP进化得到的模型是一个好的预测工具。注意，在图 4.12 中，最精确的预测是最接近的：在未来中多投入一点点，预测就会变得不准确。

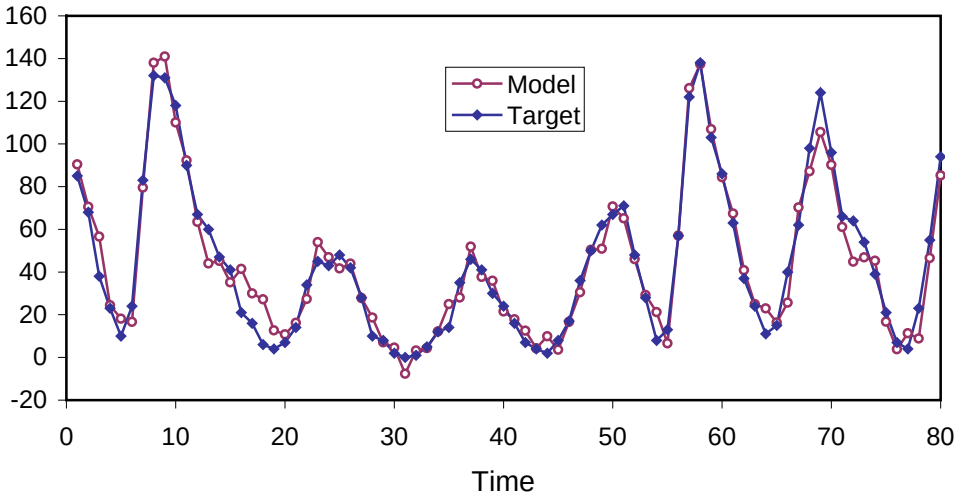


图 4.11.GEP 进化得到的模型(4.27)和目标太阳黑子序列在训练集上的比较。

关于这些时间序列预测模型最重要的是它们都是由含有一个商的简单项构成，而这是算法自身发现的，我个人没有给它任何提示，而且看来这是时间序列预测的一个非常具有创造性又非常有用的框架。

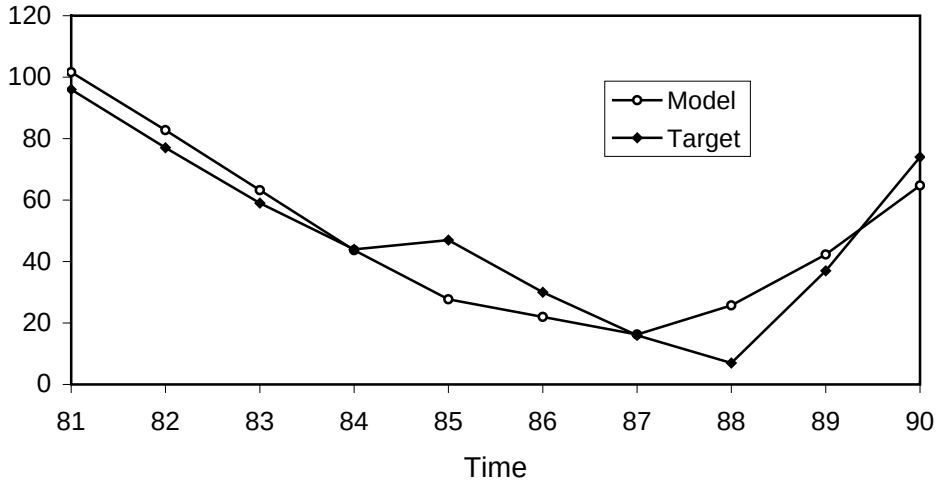


图 4.12.GEP 进化得到的模型(4.27)和目标太阳黑子序列在测试集上的比较。

4.5.分类问题

这一节我们将用 GEP 来对三个现实世界的问题，即乳腺癌诊断，信用卡发放，Fisher 的 iris 分类问题进行建模。前两个数据集来自 PROBEN1——一个神经网络的基准问题和基准规则集（Prechelt 1994）。技术报告和数据集都能够通过匿名的 ftp 从卡内基梅隆大学的神经基准档案馆上（主机名 **ftp.cs.cmu.edu**，文件夹**/asf/cs/project/connect/bench/contrib./prechelt** ）或者从 **ftp.ira.uka.de** 上的文件夹**/pub/neuron** 中获得。两个地方的文件名都是**proben1.tar.gz**。最后一个数据集，iris 数据集（Fisher），可能是数据挖掘领域最著名的数据集，它可以从网上免费获得。

4.5.1.乳腺癌诊断

诊断任务的目标是在 9 个不同的细胞分析（输入属性或终点）的基础上对肿瘤是恶性的（用 0 表示）还是良性的（用 1 表示）进行分类。

这里给出的模型是从 PROBEN1 中的 cancer1 数据集，其中每一位对一个二进制的 m 之一进行编码，它代表 m 个可能的输出类中有一个被一位编码所代替（“0”代表良性，“1”代表恶性）。前 350 个样本用来训练，后 174 个样本用来对实际运用中的模型进行测试。这意味着测试集中样本的信息和测试集的效果在适应过程中都完全不可能预先知道。因此，测试集上的分类错误对于评价概化效果是一个很好的标准。

对该问题，F={+,-,*,/,<,>,:,$,=,!}，但是每个函数的权值被计算两次（后 6 个符号分别代表小于，大于，小于或等于，等于或大于，等于和不等于，它们分别是两个参数的比较符号，当值为真时，返回第一个参数，否则，返回第二个参数）；终点集由问题中出现的 9 个属性构成，由$T=\{d_0,\ldots,d_8\}$表示，分别对应肿块厚度，细胞尺寸的一致性，细胞形状的一致性，边缘粘连，单上皮细胞尺寸，裸露核，无刺激染色质，正常核仁，及有丝分裂。

对于像分类这样输出经常是二元的情况，设定一个将实型数转换成为 0 或 1 的标准十分重要。这个 0/1 舍入阈值R_t在染色体的输出等于或大于R_t时把该输出转换成 1，否则转换成 0。对于该问题，我们将采用$R_t=0$。

分类问题所使用的适应度函数通常非常简单。个体程序的适应度f_i对应击中次数，由下面的等式给出：

$$\text{if } n > C_p \text{，then } f_n = n \text{；else } f_i = 0 \qquad (4.28)$$

其中 n 是计算正确的适应度样本的个数，Cp 是具有较多样本的类（主导类）的适应度样本的个数。

对于本问题，染色体由 h=8 的 3 个基因构成，子表达式树用加法连接。下面的程序是采用个体数为 50 的较小种群发现的：

```
>d3=++$=d8d0d4d1d1d7d0d4d5d3
!+!d5d5>>/d2d2d6d7d3d6d6d3d3
-d1/=*>!+d6d0d4d6d5d2d6d4d6
```

(4.29a)

它在训练集的 350 个适应度样本上算得的适应度为 340，在 174 个测试集样本上算得的适应度为 173。它对应的测试集上的分类错误为 0.575%，分类精度为 97.143%。

注意，要对该染色体进行完整的表达，必须将 0/1 舍入阈值R_t=0 考虑在内。我们可以用APS软件来将模型（4.29）自动转换成完整表达的C＋＋函数：

```
double APSCfunction(double d[])
{
    double dblTemp = 0;
    dblTemp += (d[3]>(((d[4]>=d[1]?d[4]:d[1])+
        (d[1]==d[7]?d[1]:d[7]))==(d[8]+d[0])?((d[4]>=d[1]?d[4]:d[1])+
        (d[1]==d[7]?d[1]:d[7])):(d[8]+d[0]))?d[3]:(((d[4]>=d[1]?d[4]:d[1])+
        (d[1]==d[7]?d[1]:d[7]))==(d[8]+d[0])?((d[4]>=d[1]?d[4]:d[1])+
        (d[1]==d[7]?d[1]:d[7])):(d[8]+d[0])));
    dblTemp += ((d[5]+d[5])!=(((d[7]/d[3])>d[2]?(d[7]/d[3]):d[2])!=
        (d[2]>d[6]?d[2]:d[6])?((d[7]/d[3])>d[2]?
        (d[7]/d[3]):d[2]):(d[2]>d[6]?d[2]:d[6]))?
        (d[5]+d[5]):(((d[7]/d[3])>d[2]?(d[7]/d[3]):d[2])!=
        (d[2]>d[6]?d[2]:d[6])?((d[7]/d[3])>d[2]?
        (d[7]/d[3]):d[2]):(d[2]>d[6]?d[2]:d[6])));
    dblTemp += (d[1]-(((d[0]>d[4]?d[0]:d[4])==(d[6]!=d[5]?
        d[6]:d[5])?(d[0]>d[4]?d[0]:d[4]):(d[6]!=d[5]?
        d[6]:d[5]))/((d[2]+d[6])*d[6])));
    return (dblTemp >= 0 ? 1 : 0);
}
```

(4.29b)

在这种形式下，模型看似非常复杂，但是其分叉结构说明它的子表达式树的确很简单（如图 4.13）。注意，从结果看来所有的属性都和乳腺癌的诊断相关。的确，GEP 进化得到的模型的一个优势就是它们允许知识提取，因为它们不仅容易得到而且容易解释。

a.

```
>d3=++$=d8d0d4d1d1d7d0d4d5d3
!+!d5d5>>/d2d2d6d7d3d6d6d3d3
-d1/=*>!+d6d0d4d6d5d2d6d4d6
```

b.

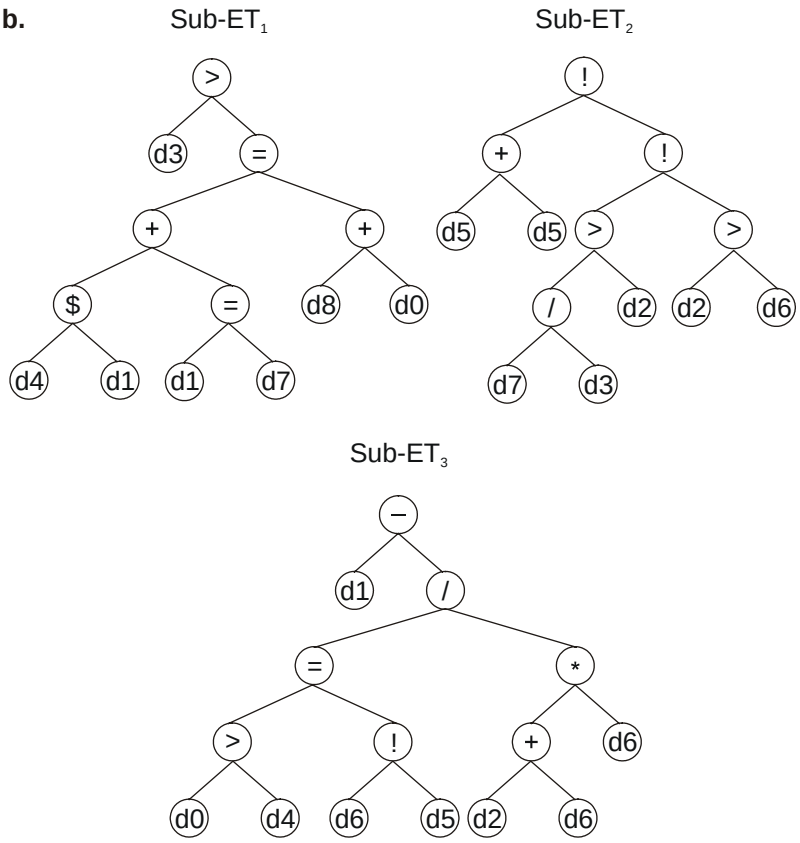


图4.13.采用GEP进行乳腺癌诊断进化得到的模型。**a**)由加号连接的对子表达式树进行编码的3-基因染色体。**b**)每个基因所编码的子表达式树。注意该染色体的表达只有当包含舍入阈值的情况下才完整，这里舍入阈值为0。

4.5.2.信用审查

信用审查的目的就是决定是否批准一个客户的信用卡申请。数据集中的每个样本代表一个真实的信用卡申请，输出描述了银行是否对该申请的客户授予信用卡。该问题含有51个输入属性，出于保密原因，原始数据中的这些属性都没有进行说明。

这里给出的模型是从PROBEN1中的card1数据集，其中二进制的m之一编码再次被一位编码代替（“0”代表批准，“1”代表不批准）。前345个样本用来训练，后172个样本用来进行测试。对该问题，F={+,-,*}，但是每个函数的权值被计算15次，终点集包括所有的51个属性，分别用$d_0,\ldots,d_{50}$来表示。0/1舍入阈值等于1.0，适应度由等式（4.28）给出。

对于本问题，染色体由h=5的5个基因构成，子表达式树用加法连接。下面的程序是采用个体数为50的较小种群在第311代发现的：

```
**+d47-d11d15d27d44d13d4
**+d47-d11d18d29d48d12d38
*d11d40+d32d17d46d0d44d13d7
-d41+*+d18d48d29d4d10d45
*-d11d40*d0d28d28d48d0d43
```
(4.30a)

它在训练集的345个适应度样本上算得的适应度为298，在172个测试集样本上算得的

适应度为 151。它对应的测试集上的分类错误为 12.209%，分类精度为 87.791%。下面是用 APS 软件自动转换得到的完整表达的 C＋＋函数：

```
double APSCfunction(double d[])
{
    double dblTemp = 0;
    dblTemp += ((d[47]*(d[27]-d[44]))*(d[11]+d[15]));
    dblTemp += ((d[47]*(d[29]-d[48]))*(d[11]+d[18]));
    dblTemp += (d[11]*d[40]);
    dblTemp += (d[41]-((d[18]*d[48])+(d[29]+d[4])));
    dblTemp += ((d[40]-(d[0]*d[28]))*d[11]);
    return (dblTemp >= 1 ? 1 : 0);
}
```

(4.30b)

注意，不是所有的属性都出现在 GEP 进化得到的模型中，所有这些属性显然与当前的决策无关。事实上，这个极度精确的模型中值使用了 51 个属性中的 13 个。

4.5.3.Fisher 的鸢尾(Iris)

该分类问题的目的就是在四个度量的基础上对鸢尾进行分类，这四个度量分别是：萼片长度，萼片宽度，花瓣长度，花瓣宽度。Iris 数据中每种鸢尾含有 50 个样本，这三种分别是：Iris setosa，Iris versicolor，Iris virginica。

GEP 也可以解决类别多余两个的分类问题，但是必须对数据进行重新整理。将数据分类成 n 个不同类 C 的时候需要将数据处理成 n 个独立的 0/1 分类问题：

1．C_1 *versus* *NOT* C_1

2．C_2 *versus* *NOT* C_2

…

n．C_n *versus* *NOT* C_n

然后分别根据以上 n 个划分进化得到 n 个不同的模型，然后将这 n 个模型结合起来，就得到了最终的分类规则。

对于 iris 数据，我们将把问题分解成 3 个独立的 0/1 分类问题。第一个是 Iris setosa 与非 Iris setosa；第二个是 Iris versicolor 与非 Iris versicolor；最后一个是 Iris virginica 与非 Iris virginica。

对该问题，F={+,-,*,/}，终点集包括所有的 4 个属性，分别用$d_0,\ldots,d_3$来表示。0/1 舍入阈值等于 0.5，适应度由等式（4.28）给出。

对于每个子问题，我使用的染色体由 h=5 的 5 个基因构成，子表达式树用加法连接。第一个数据集（setosa 与非 setosa）的分类几乎瞬间就能完成，而且没有任何错误，而且我很快就发现要对该数据集进行分类所需的结构也十分简单。下面的模型能够将所有的鸢尾完美地分类成 setosa 与非 setosa：

```
double APSCfunction(double d[])
{
double dblTemp = 0;
dblTemp += (d[1]-d[2]);
return (dblTemp >= 0.5 ? 1 : 0);
}
```

（4.31）

如读者所看到的那样，在区别 Iris setosa 和其它两类鸢尾的时候，只有萼片宽度和花瓣长度

与此相关。

其它的数据集的分类也非常精确，但是两种情况下 150 个样本中都只有 149 个本正确地分类。下面的模型能够将 Iris versicolor 与其它两类鸢尾区分开来：

```
double APSCfunction(double d[])
{
double dblTemp = 0;
dblTemp += (d[3]*(((d[0]*d[3])-d[1])+((d[1]*d[2])-d[2])));
dblTemp += (((d[2]-(d[2]+d[2]))-(d[0]/d[0]))/d[3]);
dblTemp += (((d[0]-(d[2]*d[3]))*(d[2]-d[1]))*d[0]);
return (dblTemp >= 0.5 ? 1 : 0);
}
```
（4.32）

下面一个模型可以将 Iris virginica 与 Iris setosa 和 Iris versicolor 区分开来：

```
double APSCfunction(double d[])
{
double dblTemp = 0;
dblTemp += (d[1]/(d[0]*(d[0]/d[3])));
dblTemp += (d[2]-d[1]);
dblTemp += ((d[2]-(((d[0]+d[3])/d[1])/d[3]))-d[2]);
return (dblTemp >= 0.5 ? 1 : 0);
}
```
（4.33）

所以，将上面三个模型结合起来，用y_1,y_2,y_3来表示它们，就得到如下的分类规则：

IF (y1 = 1 AND y2 = 0 AND y3 = 0) THEN *setosa;*

IF (y1 = 0 AND y2 = 1 AND y3 = 0) THEN *versicolor;*

IF (y1 = 0 AND y2 = 0 AND y3 = 1) THEN *virginica;*　　（4.34）

它能将 150 个鸢尾中的 149 个正确地分类，因此该模型的分类精度为 99.33%，分类错误为 0.667%，这是迄今为止采用机器学习算法得到的最佳模型之一。

4.6.逻辑合成

这一节我们将讨论一类不同的符号回归，我们采用它来发现布尔函数。进一步说，我们将学习如何在 GEP 中实现用户定义函数（UDF）和自动定义函数（ADF）。然后我们将分析具有这些新工具的算法如何执行，这样不仅能帮助我们理解它们是如何工作的，还能够对它们进行透彻的分析。这一节我们将要分析的问题是著名的 n 奇宇称问题。我们将用三种不同的方法发现不同的奇宇称函数：第一种由基本基因表达式算法构成；第二种方法由带 UDF 的基本基因表达式算法；第三种方法由带 ADF 的基本基因表达式算法构成。虽然事实如此，但是本书中所有的实例中，UDF 和 ADF 仅仅在逻辑合成中使用，显然它们也可以运用到本书中所有提到的问题中。

4.6.1.采用基本基因表达式算法发现奇宇称函数的解

当一个数被表达成二进制形式的时候，如果它所含的 1 的个数是奇数，我们就称这个数有奇宇称。有一类函数称为奇宇称函数，当一个 n-位的数是奇数的时候，该函数返回值为“ 1 ”或“ true ”，否则返回“ 0 ”或者“ false”。所以最简单的奇宇称函数就是恒等函数或者 1-奇宇称函数。另一个简单的也是常见的奇宇称函数就是 XOR 函数，也称为 2-奇宇称函数。

我们将就此开始奇宇称函数的发现之旅，然后将继续发现一些更加复杂的问题的解。

让我们就从进化 XOR 函数的解开始。这里，我们将在函数集中采用基本布尔函数，即 F={N,A,O}，并且将使用 AND 作为连接函数。GEP 可以采用个体数为 30 个的较小种群来求解这个问题，成功率为 100%（表 4.19，第一列）。下面这个解是在该实验第一次运行时得到的（子表达式树用 AND 连接）：

```
012345678901234012345678901234
ANOAOOAabaabbbbOOAOAaObababbab
```

(4.35)

如读者所看到的，这个解远不十分简洁，但是值得庆幸的是该问题很简单，所以我们可能找到最简洁的解。下面是一个极其简洁的解，它是采用 h=5 的单基因系统得到的：

```
01234567890
ANOAObababb
```

(4.36)

发现这些基本布尔函数的简洁解极其重要，因为这些常数常常被用来作为构建更加复杂的系统的基础基因块。如读者在表 4.19 中所看到的，XOR 本身就被用来进化更复杂的 n-奇宇称函数的解，事实上，这些实验中观测到的高成功率就是因为把它包含在函数集中造成的（表 4.19，第 2 至 5 列）。还要注意的是，在这些实验中，XOR 也被用来连接子表达式树。将这些解包含在函数集中或者用它们作为连接函数不仅使得进化的效率更高，而且能够使找到的更复杂的问题的解非常简洁。所以，如果有必要将这些基本布尔函数还原，只需要用这些复杂的基因块的简洁表示法来代替这些基因块。

下一节，我们将看一种这种 GEP 工具箱中其它方法，即采用用户定义函数来包含复杂基因块。

表 4.19 采用基本基因表达式算法求解 n-奇宇称函数的参数

	Odd-2	**Odd-3**	**Odd-4**	**Odd-5**	**Odd-6**
Number of runs	100	100	100	100	100
Number of generations	50	50	50	100	200
Population size	30	10	30	30	30
Number of fitness cases	4	8	16	32	64
Function set	A O N	A O N X	A O N X	A O N X	$(A\ O\ N\ X)_2$
Terminal set	a b	a b c	a b c d	a b c d e	a b c d e f
Head length	7	7	7	7	7
Number of genes	2	3	3	3	3
Linking function	A	X	X	X	X
Chromosome length	30	45	45	45	45
Mutation rate	0.044	0.044	0.044	0.044	0.044
One-point recombination rate	0.3	0.3	0.3	0.3	0.3
Two-point recombination rate	0.3	0.3	0.3	0.3	0.3
Gene recombination rate	0.1	0.1	0.1	0.1	0.1
Gene transposition rate	0.1	0.1	0.1	0.1	0.1
IS transposition rate	0.1	0.1	0.1	0.1	0.1
IS elements length	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3
RIS transposition rate	0.1	0.1	0.1	0.1	0.1
RIS elements length	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3
Success rate	100%	100%	98%	93%	91%

4.6.2.用 UDF 发现奇宇称函数的解

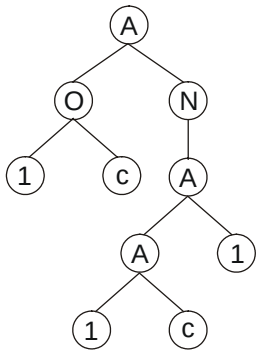
在我们专注于采用用户定义函数来进化 n-奇宇称函数的解之前，让我们先分析一下 UDF 在 GEP 中是如何工作以及如何实现的。

在基因表达式编程中，虽然从表达式的规则上说，UDF 的操作数应该为 0，但是它可以用来表示多个变量的函数。即每个含有 UDF 的节点可以像叶节点一样起作用。所以，在 GEP 中 UDF 至少通过两种方法实现：既可以将它们视为终点在头部和尾部使用，也可以将它们视为函数，仅在头部使用。不管采用那种方法，算法的效果都非常好。因此，可以根据个人的喜好进行选择。我选择后一种，因为这种方法看来更前后一致而且不容易混淆。所以，UDF 只在头中使用，它们可以在初始种群个体的基因中占据一号位置。也可以在 RIS 元素中占据一号位置。

让我们来看含有 UDF 的基因是如何表达的。例如，考虑如下的染色体：

```
0123456789012345678
AON1cAA11ccacbccaba
```
(4.37)

其中“ 1 ”代表一个 UDF。它的表达式导致如下的表达式树：



该程序对一个使用XOR作为UDF的 3-奇宇称函数的解进行编码。该UDF与正常的XOR之间的区别在于UDF的参数在函数定义起见是位置固定的，而XOR的参数则比较灵活，它依赖于每个表达式树的特定的设置。例如，在上面的染色体（4.37）中，UDF_1的参数由a和b的定义决定。只有这个时候我们才能够检查染色体（4.37）所编码的解，并确认它的确是对3-奇宇称函数的一个完美解解进行编码。将这个解与下面这个由正常XOR而不是严格的，用户定义的XOR所得到的解相比：

```
01234
XXabc
```
(4.38)

尽管如此，UDF 还是非常有用的，特别是当它们是根据当前问题而巧妙的设计出来的时候，更是如此。如表 4.20 所示，使用封装在 UDF 中的不可变 XOR 对进化 5-奇宇称函数和 6-奇宇称函数并没有多大的帮助（表 4.20，第 3 列和第 4 列）。然而，事实说明使用其它一些更高维的宇称函数作为 UDF 则非常有效（见表 4.21，特别是第 3 列和第 4 列）。

下一节我们将分析另外一种方法，该方法设计到对基因块的操作，即自动定义函数的使用，我们还将把它和前面讨论的问题进行比较。

表 4.20 采用 XOR 作为 UDF 求解 n-奇宇称函数的参数

	Odd-3	Odd-4	Odd-5	Odd-6
Number of runs	100	100	100	100
Number of generations	50	50	100	300
Population size	10	30	30	30

Number of fitness cases	8	16	32	64
Function set	A O N	A O N	A O N	(A O N) * 2
User defined functions	X	X	X	X
Terminal set	a b c	a b c d	a b c d e	a b c d e f
Head length	7	7	7	7
Number of genes	3	3	3	3
Linking function	X	X	X	(X) * 2
Chromosome length	45	45	45	45
Mutation rate	0.044	0.044	0.044	0.044
One-point recombination rate	0.3	0.3	0.3	0.3
Two-point recombination rate	0.3	0.3	0.3	0.3
Gene recombination rate	0.1	0.1	0.1	0.1
Gene transposition rate	0.1	0.1	0.1	0.1
IS transposition rate	0.1	0.1	0.1	0.1
IS elements length	1,2,3	1,2,3	1,2,3	1,2,3
RIS transposition rate	0.1	0.1	0.1	0.1
RIS elements length	1,2,3	1,2,3	1,2,3	1,2,3
Success rate	95%	100%	2%	1%

表 4.21 采用 n-奇宇称函数作为 UDF 求解 n-奇宇称函数的参数

	Odd-3	**Odd-4**	**Odd-5**	**Odd-6**
Number of runs	100	100	100	100
Number of generations	50	50	100	200
Population size	10	30	30	30
Number of fitness cases	8	16	32	64
Function set	A O N	A O N	A O N	(A O N) * 2
User defined functions	Odd-2	Odd-3	Odd-4	Odd-5
Terminal set	a b c	a b c d	a b c d e	a b c d e f
Head length	7	7	7	7
Number of genes	3	3	3	3
Linking function	X	X	X	X
Chromosome length	45	45	45	45
Mutation rate	0.044	0.044	0.044	0.044
One-point recombination rate	0.3	0.3	0.3	0.3
Two-point recombination rate	0.3	0.3	0.3	0.3
Gene recombination rate	0.1	0.1	0.1	0.1
Gene transposition rate	0.1	0.1	0.1	0.1
IS transposition rate	0.1	0.1	0.1	0.1
IS elements length	1,2,3	1,2,3	1,2,3	1,2,3
RIS transposition rate	0.1	0.1	0.1	0.1
RIS elements length	1,2,3	1,2,3	1,2,3	1,2,3
Success rate	95%	100%	100%	100%

4.6.3.用 ADF 发现奇宇称函数的解

自动定义函数在 2.2.3 节进行过介绍。读者应该还记得，在这个复杂的系统中，正常基因对特殊的 ADF 进行编码，而 homeotic 基因控制每个分子中哪些基因/ADF 被表达。我们还在第二章学到了每个有机体的细胞个数（或者换言之，每个染色体中的 homeotic 基因个数）是任意的，因为每个个体的适应度由最佳细胞的适应度决定，细胞个数越多，进化得到正确的分子或解的可能性就越大。但是这里有一个限度：我们不能无限增加细胞的个数，因为对它们进行表达需要时间和资源。如表 4.22 所示，一个比较好的折中的办法是每个个体中含有 3 个细胞。

如读者在表 4.22 中所看到的，这种方法的效果从成功率上来说几乎和前一种方法不相上下。但是，请记住，这个系统比前一个系统复杂许多，而且需要对很多的参数进行微调。这种方法的一个优势就是子表达式树的连接更加灵活，而且根据进化得到的解的结构，不需要预先作任何假设。

表 4.22 采用 ADF 求解 n-奇宇称函数的参数

	Odd-2	**Odd-3**	**Odd-4**	**Odd-5**	**Odd-6**
Number of runs	100	100	100	100	100
Number of generations	50	50	50	100	200
Population size	30	10	30	30	30
Number of fitness cases	4	8	16	32	64
Function set	A O N	A O N X	A O N X	A O N X	$(A\ O\ N\ X)_2$
Terminal set	a b	a b c	a b c d	a b c d e	a b c d e f
Number of normal genes	3	3	3	3	3
Head length of normal genes	5	5	5	5	5
Linking functions	A O N	A O N X	A O N X	A O N X	A O N X
Number of homeotic genes	3	3	3	3	3
Head length of homeotic genes	3	5	5	5	5
Chromosome length	54	66	66	66	66
Mutation rate normal genes	0.044	0.044	0.044	0.044	0.044
One-point recombination rate	0.3	0.3	0.3	0.3	0.3
Two-point recombination rate	0.3	0.3	0.3	0.3	0.3
Gene recombination rate	0.1	0.1	0.1	0.1	0.1
Gene transposition rate	0.1	0.1	0.1	0.1	0.1
IS transposition rate	0.1	0.1	0.1	0.1	0.1
IS elements length	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3
RIS transposition rate	0.1	0.1	0.1	0.1	0.1
RIS elements length	1,2,3	1,2,3	1,2,3	1,2,3	1,2,3
Success rate	100%	94%	95%	95%	96%

4.7 为密度分类问题进化细胞自动机规则

密度分类任务是一个具有挑战性的问题，人们已经用过各种不同的适应性方法，例如，

GAs 和 GP，来试图进化得到比人工手写得到的更好的规则。GEP 也曾被用来为该任务进化更好的解（Ferrerira 2001），该方法找到的两个解不仅好于任何人工手写得到的规则，也比用 GAs 和 GP 得到的规则。这一节，我将描述这些规则并说明是如何发现这些规则的。

密度分类问题的起点是Gacs-Kurdyumov-Levin(GKL)规则，它是 1978 年为了研究可靠计算和一维空间扩展系统中的逐步过度而由手工设计的（Mitchell 1996）。虽然GKL规则并不是专门为密度分类而设计的，但是它作为该任务效果最好的规则已经被使用了很长一段时间。其无偏效果见表 4.23。1993 年，Lawrence Davis得出一个新的规则来修正GKL规则（Koza 等 1999）。这个新的规则，即Davis规则，得到的精度略优于GKL规则（表 4.23）。与此相似，Rajarshi Das非常聪明地修正了GA得到的规则，得到一种效果比GKL规则和Davis规则略好的规则（Das规则，见Koza等 1999）。GP发现一种规则（GP规则），其效果略好于前面的几种规则（Koza等 1999）。GEP发现的两种规则（GEP_1规则和GEP_2规则）比前面所有的规则都要好（Ferrerira 2001）。最后，Jullié和Pollack（1998）采用共同进化学习，发现两种新的规则（$Coevolution_1$ 和$Coevolution_2$）明显由于以上所有规则，它们的效果如表 4.23 所示。

表 4.23 N=149 时求得的精度

GKL rule	0.815
Davis rule	0.818
Das rule	0.823
GP rule	0.824
GEP1 rule	0.825
GEP2 rule	0.826
Coevolution1 rule	0.851
Coevolution2 rule	0.860

细胞自动机（CA）已经得到了广泛的研究，因为它是能够处理紧急行为的大规模并行、分散计算系统的理想化模型（要全面了解 CA 理论及其应用，请参见 Toffoli 和 Margolus 1987 及 Wolfram 1986）。这些复杂的行为是本地的多点上简单规则的同时执行造成的。在密度分类过程中，一个与小的邻域相关的规则在一维细胞自动机的所有细胞上同时运行，它应该能够使 CA 在初始配置（IC）中 1 的密度较大的情况下收敛到全部为 1 的状态，或者在初始配置（IC）中 0 的密度较大的情况下收敛到全部为 0 的状态。

4.7.1.密度分类任务

简单的密度分类任务使一个由 N 个二元状态细胞构成的围包数组，其中的每个细胞与两侧的 r 个相邻的细胞相互联系。每个细胞的状态根据一个预定的规则不断更新。该规则同时应用在所有的细胞上，该过程迭代 t 次。

在该问题最经常研究的一个版本中，N=149，邻域等于 7（中心细胞用“ u ”表示；左侧的r=3 个细胞分别用“ c ”，“ b ”，“ a ”表示，右侧的r=3 个细胞分别用“ 1 ”，“ 2 ”，“ 3 ”表示）。图 4.14 所示为一个N=11 的CA，其中细胞自动机“ u ”的更新状态如图所示。密度分类任务包括决定IC中所含有的 1 是多数还是 0 是多数，相应地决定系统是收敛到一个全 1 状态（空间-时间图上的黑色或者“ 开 ”细胞）还是全 0 状态（白色或者“ 关 ”细胞）。由于IC的密度是N个函数的系数，在信息和交流有限的情况下，为了能够对IC进行分类，本地细胞的动作必须相互协调。的确，通过手工在 2^{128}个过度状态中搜索得到效果良好的规则，几乎是一个不可能的任务，但是已经有很多算法通过进化得到了比手工书写的规则更好的规则

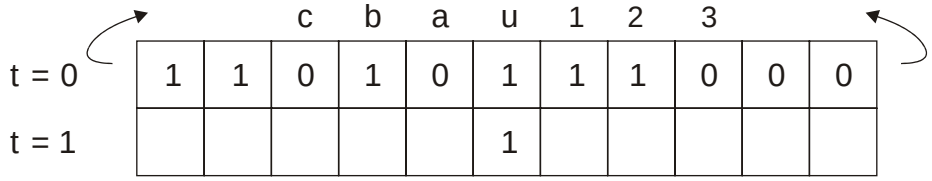


图 4.14.一维，二元状态，r=3，N=11 的细胞自动机。箭头代表周期边界条件。更新的状态仅在中心细胞中显示。同时给出了表示邻域的符号。

（Davis 等 1994；Jullié和 Pollack 1998；Koza 等 1999；Ferrerira 2001）。采用遗传算法的 IC 与规则之间的共同进化策略得到的效果最好的规则的效果为 86%和 85.1%（Jullié和 Pollack 1998）。通过 GEP 进化得到规则比所有的人工手写的规则和 GP 得到的规则及 GA 进化得到的规则都要好（Mitchell 等 1993 及 Mitchell 等 1994），而且 GEP 所使用的资源比 GP 所使用的资源少 60,00 倍。

4.7.2.GEP 发现的两个规则

在一次试验中，F={A,O,N,I}且 T={c,b,a,u,1,2,3}。每次运行的参数如表 4.24 第一列所示。

适应度是在由 25 个无偏IC上算得的（即IC中每个位置上出现 0 或 1 的概率相等）。对于该问题而言，个体程序的适应度f_i是IC的个数n的一个函数，其中n是系统对于在 2×N步之后正确地稳定在某个全 0 或者全 1 状态的一种设置。如此设计的目的是让那些既能够对多数为 1 的IC正确分类，也能对多数为 0 的IC正确分类的个体有较高的特权。因此，如果系统收敛，在所有的情况下，不论配置为 1 或 0，都只归因于一个适应度点；假设在某些情况下，系统正确地收敛到一个配置为 0 的情况或者配置为 1 的情况，则f_i=2；另外，收敛到另一个全部为 1 或者全部为 0 的模式的规则就被去掉，因为它们很容易被发现并入侵种群，导致发现好的规则的过程停止下来；最后，当一个个体程序能够对多数为 1 或者多数为 0 的IC正确分类的时候，就对正确分类的IC的个数加上一个等于IC的个数C作为奖励，在此f_i=n+C。例如，如果一个程序能够正确分类两个IC，其中月个多数为 1，另一个多数为 0，那么它得到 2＋25＝27 个适应度点。

表 4.25GEP 为密度分类任务发现的两个规则的描述。输出二进制位由 000000 开始，以 111111 结束，按词法顺序给出。

GEP1	00010001 00000000 01010101 00000000
	00010001 00001111 01010101 00001111
	00010001 11111111 01010101 11111111
	00010001 11111111 01010101 11111111
GEP2	00000000 01010101 00000000 01110111
	00000000 01010101 00000000 01110111
	00001111 01010101 00001111 01110111
	11111111 01010101 11111111 01110111

本实验一共运行 7 次。在第 5 次运行的第 27 代发现了如下规则（仅给出 K-表达式）：

012345678901234567890123456789

OAIIAucONObAbIANIb1u23u3a12aa （4.39）

该程序在 149×298 的方格内的 100,000 个无偏 IC 上测试得到的精度为 0.82531，因此优于 GP 规则在 149×320 方格测试得到的 0.824 的精度（Jullié和 Pollack 1998，Koza 等 1999）。其规则表见表 4.25。图 4.15 所示为这个新规则的两个空间-时间图。

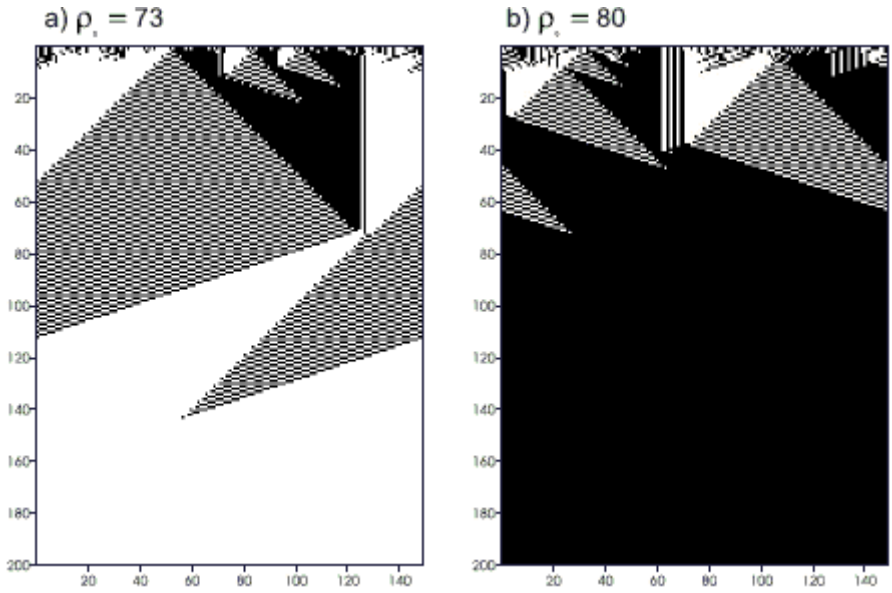


图 4.15.描述GEP_1规则的CA状态进化的时空图。IC中 1 的个数（ρ_0）在每个图的上面显示。在两种情况下，CA都正确地收敛到某种一致模式。

作为比较，GP 采用的种群大小为 51,200，1,000 个 IC，最大运行代数为 51 代（Koza 等 1999），因此一共进行 51,200×1,000×51＝2,611,200,000 次适应度评价，而 GEP 只进行 30×25×50＝37,500 次适应度评价。因此，对于本问题，GEP 不仅比 GP 超出 69,632 倍，而且 GEP 比 GP 发现的规则更多更好。

在另外一个实验中发现的一个规则比GEP_1略好，其精度为 0.8255。同样，它的效果是在 149×298 的方格内的 100,000 个无偏IC上得到的。在这里F={I,M}，而且T显然和前面用到的相同。在此，共使用 100 个无偏IC，其中的 3-基因染色体的子表达式树用IF连接。每次运行的参数见表 4.24 的第 2 列。

在这个试验中，我们对适应度函数稍作修改，引入一个排序系统，其中能够对 2 个 IC 到所有 IC 中的 3/4 进行分类的个体获得一个等于 C；能够对所有 IC 中的 3/4 到 17/20 进行正确分类的个体获得的 2 个奖励 ;能够对所有 IC 中的 17/20 以上进行正确分类的个体获得 3 个奖励。同时，在该实验中，只能够对一种情况进行分类的个体的适应度被区别开来，它们的适应度等于 n。

在 10 次运行的第 43 代，发现如下规则（GEP_2规则，子表达式树用IF连接）：

012345678901201234567890120123456789012

MIuua1113b21cMIM3au3b2233bM1MIacc1cb1aa （4.40）

该程序（GEP_2规则）在 149×298 的方格内的 100,000 个无偏IC上测试得到的精度为 0.8255，因此优于GEP_1规则和GP规则。其规则表见表 4.25。图 4.16 所示为这个新规则的两个空间-时间图。

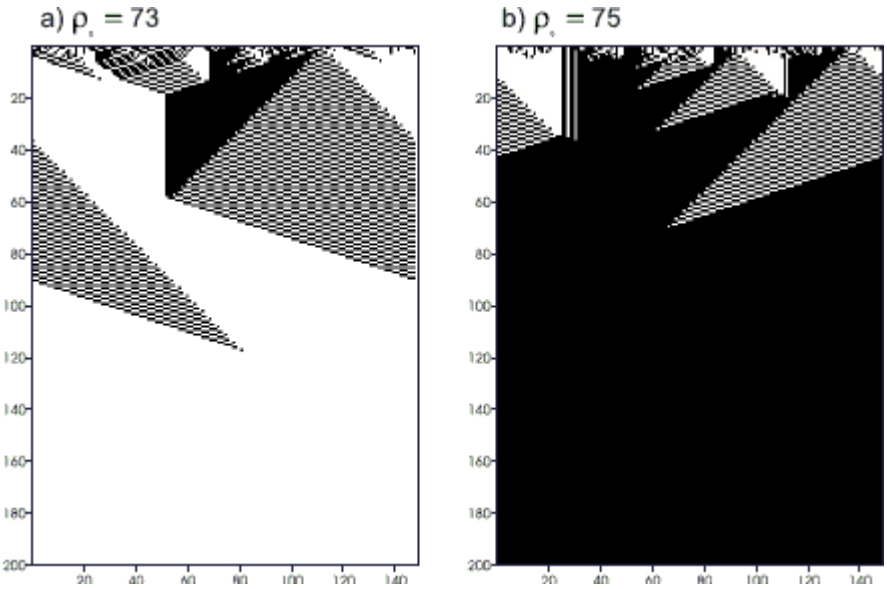


图 4.16.描述GEP_2规则的CA状态进化的时空图。IC中 1 的个数（ρ_0）在每个图的上面显示。在两种情况下，CA都正确地收敛到某种一致模式。

这一章我们学习了如何将基本基因表达式编程运用于一些非常不同的问题域，并引入了一系列新的工具来扩大其应用范围。这些新的工具包括对随机数值常数的操作，一类新的比较函数，二元多项式函数，用户定义函数和自动定义函数。下一章我们将看到如何将 GEP 中的这些为控制随机数值常数而开发出来的优美结构进一步发展，使其允许复杂的神经网络进行进化。

神经网络设计 5

人工神经网络是由许多并行工作的简单互连单元（神经元）构成的一种计算工具。这些单元或者节点之间的联系通常采用实数型的权值来表示。权值是神经网络学习的主要方法，学习算法经常被用来调节这些权值。

从结构上说，一个神经网络具有 3 种不同类型的单元：输入、隐藏和输出单元。输入单元上提供一种激活模式，该模式以向前传播的方式从输入单元经一层或多层的隐藏单元到达输出单元。这个激活模式在从一个单元进入其它单元的时候需要与其传播方向上的连接权值相乘。然后，所有的输入激活值相加，只有当输入结果大于该单元的阈值的时候这个单元才被激活。

总而言之，神经网络的基本元素是单元、单元之间的连接、权值和阈值。我们在模拟一个完整的神经网络的时候必须将这些元素编码在一个线性染色体中，只有这样神经网络才能够在某个特定的选择环境中逐步适应。为此，我设计了一个与前一章所描述的处理随机数值常数时所使用的染色体类似的染色体组织结构。

这一章我们将学习如何修改 GEP 的染色体以使其能够将一个完整的神经网络，包括其结构、权值和阈值，完全编码在一个线性染色体中。进一步，我们将看到这种染色体组织结构如何通过采用 GEP 的选择和修饰机制对神经网络进行训练，并发现那些以神经网络形式表示的解。另外，我们还将看到我们可以对这些神经网络进行透彻的研究，因为我们能够得到所有与其结构有关的信息。

5.1.用于神经网络模拟的多域基因

具有所有必需元素的神经网络是一个相当复杂的结构，要对它进行构造或者进行训练都不是一项简单的任务。所以，一些研究人员采用 GA 等成熟的算法来进化神经网络的某些部分，例如权值、网络结构或者学习算法等（关于神经网络和 GA 的论文集请参考 Whitley 和 Schaffer 1992）。

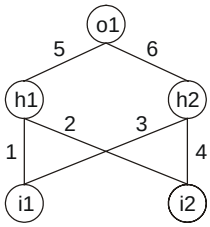
由于 GEP 十分简单而且可塑性很强，所以我们有可能将大小和形状各异的神经网络完全编码在一个固定长度的线性染色体中。进一步，这些复杂的结构功能完善，也就是说它们能够适应某一特殊环境，能够根据适应度被选中。这就意味着可以采用这些实体构成的种群来搜索某个搜索空间，并因此进化得到各种不同问题的解。

在GEP-网中，网络结构编码在一个类似于头/尾域的结构中。头部含有能够激活隐藏和输出单元的特殊函数（在GEP的语境下，称之为功能单元更恰当）和用来表示输入单元的终点。尾部显然只含有终点。除了头部和尾部以外，这些基因还含有两个附加域，Dw和Dt，分别用来对权值和阈值进行编码。从结构上来说，Dw出现在尾部之后，其长度d_w=h*n。Dt的长度d_t＝h。这两个域都由神经网络中代表权值和阈值的符号构成。

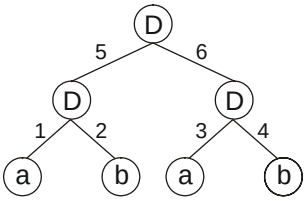
对于每个 NN-基因，其权值和阈值在每次运行的开始产生，其循环由遗传算子来保证。进一步，为了能够向权值集和阈值集中持续引入变化，我们专门设计了特殊的变异算子。

值得强调的是，只要我们能够保持每个区域的边界不变，而且根据每个域的限制正确地使用相应的字符集，像变异、IS 转座和 RIS 转座这样的基本遗传算子将不会受到 Dw 和 Dt 的影响。

考虑下面这个传统方法表示的神经网络，它由两个输入单元（i_1和i_2），两个隐藏单元（h_1和h_2）和一个输出单元（o_1）构成（为了简单，阈值均等于 1，并省略）：



它还可以用一个传统的树形式来表示：



其中a，b分别表示两个输入i_1和i_2，“ D ”表示具有两个连通分量的函数。该函数用参数乘以相应的权值，并对所有的输入激活值求和以得到向前输出。这个输出（0 或 1）依赖于阈值，即如果输入激活值之和等于或者大于这个阈值，则输出为 1，否则为 0。

我们可以将上面的这个 NN-树线性化成如下形式：

0123456**789012**

DDDabab**654321**

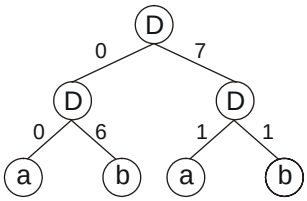
其中粗体结构对权值进行编码。每个权值的值保存在一个数组中并可以随时取出。为了简化问题，Dw 中的数字代表数组中的顺序。

让我们现在来分析一个简单的神经网络，它对一个著名的函数，即异或函数进行编码。例如，考虑下面这个h=3 的染色体，它由一个长度为d_w=6 的域对权值进行编码（Dw用粗体标识）：

0123456**789012**

DDDabab**701160**

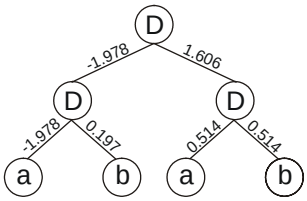
它可以翻译成：



对于权值集合：

W={-1.978,0.514,-0.465,1.22,-1.686,-1.797,0.197,1.606,0,1.753},

上面的神经网络变成：



它是一个异或函数的完美解。

5.2.特殊的遗传算子

这种由不同域和不同字符集构成的复杂实体的进化需要一组特殊的遗传算子。基本基因

表达式算法的算子很容易在对染色体编码的神经网络中稍作转变，只要我们能够保持每个区域的边界不变，而且根据每个域的限制正确使用相应的字符集，我们就可以继续使用这些遗传算子。变异扩展到所有域中，并且仍然是最重要的遗传算子。IS 转座和 RIS 转座也转变到 GEP-网里面，而且它们的作用明显限于头部和尾部之内。然而，我们还在 Dw 和 Dt 的范围内设计了一些特殊的算子来保证权值和阈值能在种群中循环（见下面的详细描述）。进一步，我们还设计了特殊的变异算子——权值和阈值的直接变异——来向现有的权值和阈值中引入修饰。

把重组和基因转座扩展到 GEP-网是非常简单明了的事情，因为它们的作用从来不会混淆域和字符。然而，为了能使其正常工作（即能够使进化有效），我们必须谨慎地决定哪些权值和/或阈值在染色体分裂之后进入哪一块区域，否则系统就不能很有效地进化。在基因重组和基因转座的情况下，要跟踪基因的权值和阈值并不困难，事实上，这些算子容易实现而且效果很好。但是对于单点和两点重组的时候，染色体可以在任何地方分裂也就不可能对权值和阈值进行跟踪。实际上，如果直接使用这些算子，它们将会产生多基因染色体毫无用处的进化怪胎。因此，对于多基因系统，我们设计了一种特殊的基因内两点重组将重组限制在一个特定的基因中（见现面的详细描述）。

5.2.1.针对域的转座

针对域的转座仅限于针对网的域——Dw 和 Dt。然而其机制与 IS 转座机制类似。该算子随机地选择染色体，包含各自 Dw 和 Dt 的基因，转座子的第一个位置，转座子的长度和目标位置（也必须选在 Dw 或 Dt 以内）。然后该算子将转座子从其初始位置移到目标位置。

例如，考虑下面这个 h=4 的染色体（Dw 和 Dt 用不同的阴影显示）：

0123456789012345**7890123456789012**3456
DTQaababaabbaabba**0571745736284668**2867

其中“ T ”代表三个参数的函数，“ Q ”代表 4 个参数的函数。假设选中序列“ 84668 ”作为转座子，其插入位置在 Dw 中的 4 号池（第 30 号位和 31 号位之间）。那么得到下面的染色体：

0123456789012345**7890123456789012**3456
DTQaababaabbaabba**0571846874573622**867

注意初始位置上的转座子被删除，从而保持域的长度不变。

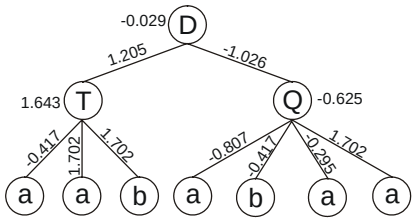
假设下面的数组表示两个染色体的权值和阈值：

W={-1.64,-1.834,-0.295,1.205,-.0807,0.856,1.702,-1.026,-0.417,-1.061}
T={-1.14,1.177,-1.179,-0.74,0.393,1.135,-0.625,1.643,-0.029,-1.639}

如其表达式所示（图 5.1），它们所编码的解非常不同，因为权值会到处移动，而且会测试新的权值和阈值的组合。

a. 012345678901234567890123456789012345678901234567890123456
DTQaababaabbaabba**05717457362**<u>**84668**</u>2867-[m]

W_m = {-1.64, -1.834, -0.295, 1.205, -0.807, 0.856, 1.702, -1.026, -0.417, -1.061}
T_m = {-1.14, 1.177, -1.179, -0.74, 0.393, 1.135, -0.625, 1.643, -0.029, -1.639}



b. 012345678901234567890123456789012345678901234567890123456
DTQaababaabbaabba**0571**<u>**84668**</u>**7457362**2867-[d]

W_d = {-1.64, -1.834, -0.295, 1.205, -0.807, 0.856, 1.702, -1.026, -0.417, -1.061}
T_d = {-1.14, 1.177, -1.179, -0.74, 0.393, 1.135, -0.625, 1.643, -0.029, -1.639}

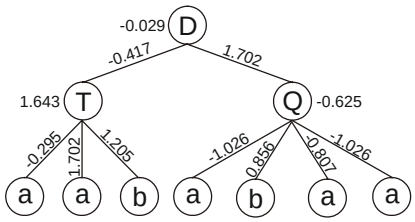


图 5.1.针对Dw的转座。a)神经网络母体。b)由针对Dw的转座产生的子NN。注意母体和子体的网络结构是相同的而且W_m=W_d且T_m=T_d。然而，母体与子体不同，因为这些个体所表达的权值和阈值的组合不同。

5.2.2.基因内两点重组

基因内两点重组是为了在不影响编码在其它基因中的子 NN 的情况下对某个特定的基因进行修饰而设计的。这种重组的机制与前面我们已经熟悉的两点重组的机制完全相同，区别仅仅在于其重组点必须选在一个特定的基因内部。

考虑下面两个由两个基因构成的父代染色体，其中每个染色体包含一个Dw域（W_{ij}表示染色体i的j号基因的权值）：

$W_{0,0}$={-0.78,-0.521,-1.224,1.891,0.554,1.237,-0.444,-0.472,1.012,0.679}
$W_{0,1}$={-1.553,1.425,-1.606,-0.487,1.255,-0.253,-1.91,1.427,-0.103,-1.625}

012345678901234560123456789012345678
TTababaab14393255QDbabbabb96369304-[0]
Qaabbbabb97872192QDbabbaaa81327963-[1]

$W_{1,0}$={-0.148,1.83,-0.503,-1.786,0.313,-0.302,0.768,-0.947,1.487,0.075}
$W_{1,1}$={-0.256,-0.026,1.874,1.488,-0.8,-0.804,0.039,-0.957,0.462,1.677}

假设 0 号基因被选为重组的基因，1 号点（0 号位和 1 号位之间）和 12 号点（11 号位和 12 号位之间）被选为重组点。就会得到如下后代：

$W_{0,0}$={-0.78,-0.521,-1.224,1.891,0.554,1.237,-0.444,-0.472,1.012,0.679}
$W_{0,1}$={-1.553,1.425,-1.606,-0.487,1.255,-0.253,-1.91,1.427,-0.103,-1.625}

01234567890123456012345678901234 56

Taabbbabb97893255QDbabbabb96369304-[0]

QTababaab14372192QDbabbaaa81327963-[1]

$W_{1,0}$={-0.148,1.83,-0.503,-1.786,0.313,-0.302,0.768,-0.947,1.487,0.075}

$W_{1,1}$={-0.256,-0.026,1.874,1.488,-0.8,-0.804,0.039,-0.957,0.462,1.677}

注意这两个后代的权值与父代的权值完全相同。然而由于重组的原因，父代中权值的表达与子代中权值的表达并不相同（比较图 5.2 中它们的表达式）。

a.

01234567890123456012345678901234 56
TTababaab14393255QDbabbabb96369304-[0]
Qaabbbabb97872192QDbabbaaa81327963-[1]

01234567890123456012345678901234 56
Taabbbabb978**93255QDbabbabb96369304**-[0]
Q**Tababaab143**72192QDbabbaaa81327963-[1]

$W_{0,1}$ = {-0.78, -0.521, -1.224, 1.891, 0.554, 1.237, -0.444, 0.472, 1.012, 0.679}
$W_{0,2}$ = {-1.553, 1.425, -1.606, -0.487, 1.255, -0.253, -1.91, 1.427, -0.103, -1.625}

$W_{1,1}$ = {-0.148, 1.83, -0.503, -1.786, 0.313, -0.302, 0.768, -0.947, 1.487, 0.075}
$W_{1,2}$ = {-0.256, -0.026, 1.874, 1.488, -0.8, -0.804, 0.039, -0.957, 0.462, 1.677}

b.

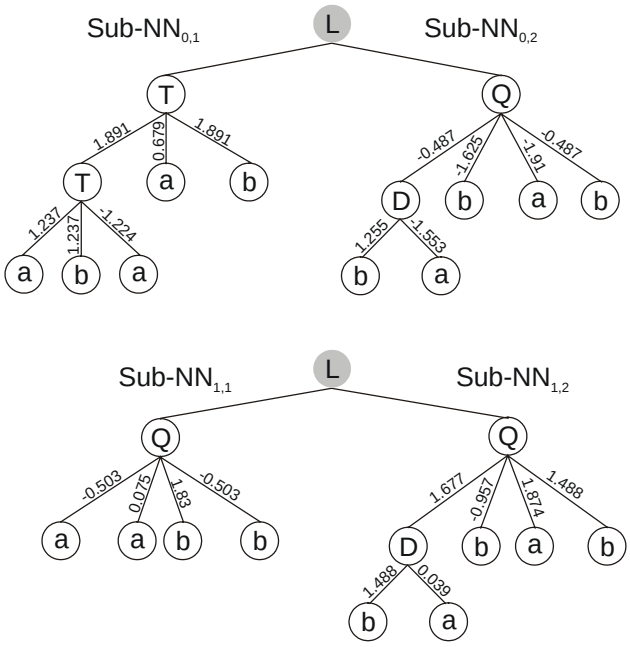


图 5.2.对神经网络进行编码的多基因染色体中的基因内两点重组。**a)** 基因内两点重组的染色体层次示例。**b)**父代染色体所编码的子-NN。**c)**子代染色体编码的子-NN。“ L ”代表一个通用连接函数。

c.

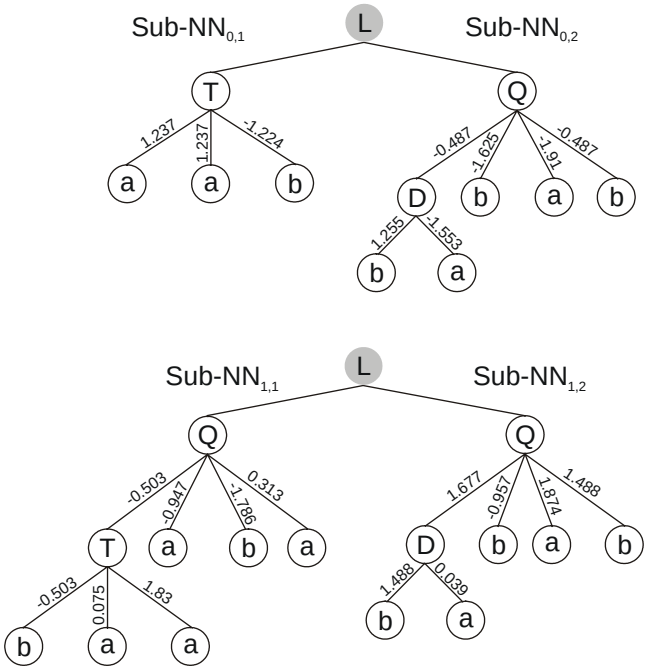


图 5.2.续

值得强调的是，受基因限制的重组允许对修饰机制进行更多的控制，因此，允许对进化过程进行更精确的调整。如果我们要在多基因系统中像在基本基因表达式算法中那样使用两点重组，在任何地方切断染色体，那么多基因系统中对权值的细微调整就会变得几乎不可能。然而将两点重组限制在基因以内的做法保证只对这个基因进行修饰，并因此使得其它基因的权值和阈值不受影响。

但是，请记住，基因内两点重组不是多基因神经网络中重组操作的唯一来源。在这个系统中，基因重组的功能已经比较完善，而且当它和基因转座结合起来的时候，能够进一步推动整个进化过程。而且，在单基因系统中，老的单点和两点重组操作也有很完善的功能，因为它们不需要与权值同步。

5.2.3. 权值和阈值的直接变异

我们已经看到所有的遗传算子都对权值和阈值的移动或多或少地有所贡献。事实上，这种持续的切换对于 GEP-NN 的有效进化来说已经足够，因为在每次运行的开始就会随机产生数目合适的权值和阈值。然而，我们可以实现一些特殊的变异算子来用其它值替换某个特定的权值或阈值。

这些被称为权值和阈值的直接变异算子随机地选择数组中所保存的权值或者阈值作为

一个特定目标，然后用一个随机生成的实数来代替这个目标。例如，考虑数组：

$W_{i,j}$={-0.433,-1.823,1.255,0.028,-1.755,-0.036,-0.128,**-1.163**,1.806,0.083}

该数组编码的是染色体 i 的 j 号基因的权值。假设在 7 号位出现一个变异，是该位上的-1.163 变为-0.494，得到：

$W_{i,j}$={-0.433,-1.823,1.255,0.028,-1.755,-0.036,-0.128,**-0.494**,1.806,0.083}

这种变异可能产生很多种不同的后果：其后果可能是中性的（例如，当基因本身是中性时，或者当权值或阈值在子-NN 上没有表现时），也可能是多重效果。后一种情况只要当所修饰

的权值或阈值恰好被用于一个以上的子-NN 的表达式时就会出现（图 5.3）。

a.

```
01234567890123456789012345678901 2
TDQDbaababababaaaaa7986582527723251-[m]
```

W_m = {**-0.202**, -1.934, **-0.17**, 0.013, 1.905, 1.167, 1.801, -1.719, 1.412, 0.434}

W_d = {**1.49**, -1.934, **1.064**, 0.013, 1.905, 1.167, 1.801, -1.719, 1.412, 0.434}

```
01234567890123456789012345678901 2
TDQDbaababababaaaaa7986582527723251-[d]
```

b.

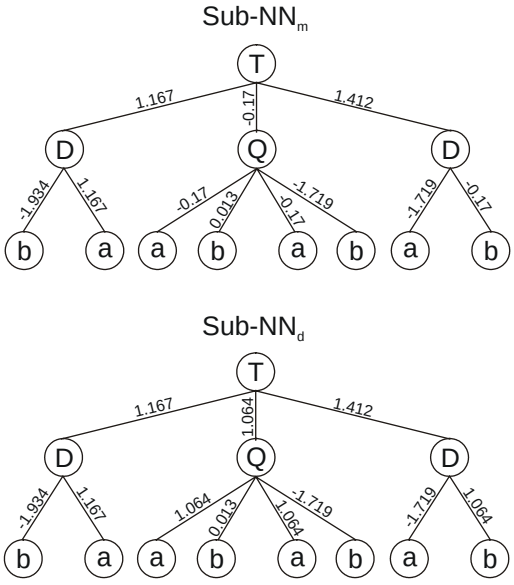


图 5.3.权值直接变异示例。a)母代和子代染色体及其相应的权值。这里，0 号位和 2 号位的权值发生变异 b)编码在母代和子代中的 NN。注意 2 号位的变异点（-0.17）具有折叠效应，因为该权值在神经网络中出现了 4 次。还要注意第 0，4，6，9 号位中的权值变异具有中性影响，因为它们的权值在神经网络中没有表达。

有趣的是，这种变异的重要性看来十分有限，而且不使用这个算子的时候所得到的结果更好。的确，函数发现问题中数值常数的直接变异会产生相同的结果。因此，我们可以断言初始适应度的规模合适的数值常数，不管是数学表达式的数值常数，还是神经网络的权值或阈值，都足以允许其进化过程作调整。一般来讲，对于每个基因，对域长度等于或小于 20 的时候我采用长度为 10 数组来保存权值。对于大一些的域，我们可以增加元素的个数，但是大部分情况下长度为 10 的数组就能够取得很好的效果。

5.3.用 GEP 神经网络求解问题

我们选择用来说明线性编码神经网络的问题是两个著名的逻辑合成问题。选择第一个问题，即异或问题，是出于它在神经网络领域历史上的重要性，而且因为它很简单，能够用它很容易地解读通过进化得到的神经网络。第二个问题，即 6-位多路由器问题，是一个相当复杂的问题，可以用来对这种新算法的效率进行评价。

5.3.1.求解异或问题的神经网络

XOR 是一个有 2 个连通分量的简单布尔函数，它能够很轻易地用线性编码神经网络求解。

用来求解这个问题的函数具有 2 个、3 个、4 个连通分量，它们分别用“ D ”,“ T ”,“ Q ”来表示。该实验的设置如表 5.1 所示，我们选择的 h=4，用它能够发现 XOR 函数的数以百计的不同的正确解。这些解中的大部分都比 186 页所示的传统的含有 7 个节点的解要复杂；还有一些解从总节点个数的角度来说与它的复杂程度相当；然而奇怪的是，还有一些解比上面提到的 XOR 函数的解更简洁。

表 5.1 采用冗余系统（RS）和紧凑系统（CS）求解异或问题的参数

	RS	**CS**
Number of runs	100	100
Number of generations	50	50
Population size	30	30
Number of fitness cases	4	4
Function set	D T Q	D T Q
Terminal set	a b	a b
Weights array length	10	10
Weight range	[-2,2]	[-2,2]
Head length	4	2
Number of genes	1	1
Chromosome length	33	17
Mutation rate	0.061	0.118
One-point recombination rate	0.7	0.7
IS transposition rate	0.1	--
IS elements length	1	--
RIS transposition rate	0.1	--
RIS elements length	1,2,3	--
Dw spesicif transposition rate	0.1	0.1
Dw spesicif IS element length	2,3,5	2,3,5
Success rate	77%	30%

表 5.1 中第一列所给出的实验在第 0 次运行中发现了第一个完美解。其结构如下：

```
012345678901234567890123456789012
TQaTaaababbbabaaa6085977238275036

W={1.175,0.315,-0.738,1.694,-1.251,1.956,-0.342,1.088,-1.694,1.288}
```

如读者在图 5.4 中所看到的，这个解对 XOR 函数来讲颇为复杂。但是，请记住，进化算法在少许的冗余结构下能够非常繁荣（例如，见 7.4 节），而且事实上，对于该问题而言，采用稍有冗余的染色体组织结构所得到的成功率（72%）比 h=2 的较为紧凑的染色体结构得到的成功率（30%）还要高。

但是，如我们早先所说，GEP 对于搜索简洁的解也是很有用的，另一个实验中发现了 XOR 函数的一个非常有趣而又简洁的解。该实验中每次运行的参数如表 5.1 第 2 列所示。注意，我们选择 h=2 的染色体组织结构并不是因为它的效果，而是为了找到比 XOR 函数的

传统解更加简洁的解。下面就是一个这样的解：

```
01234567890123456
TDbabaabb88399837

W={0.713,-0.774,-0.221,0.773,-0.789,1.792,-1.77,0.443,-1.924,1.161}
```

a.
```
012345678901234567890123456789012
TQaTaaababbbabaaa6085977238275036
```

W = {1.175, 0.315, -0.738, 1.694, -1.215, 1.956, -0.342, 1.088, -1.694, 1.288}

b.

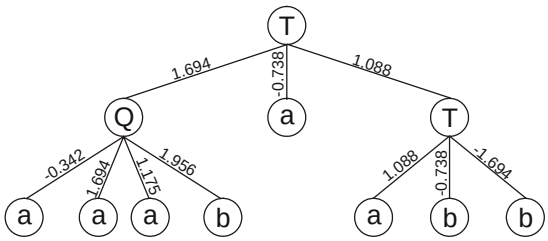


图 5.4GEP 设计的神经网络进化得到的异或问题的一个完美的，稍复杂的解。**a)**其染色体及其对应权值。**b)** 编码在染色体中的完整表达的神经网络。

它就是 XOR 函数的一个极度简洁的完美解。其完整表达式见图 5.5。

a.
```
01234567890123456
TDbabaabb88399837
```

W = {0.713, -0.774, -0.221, 0.773, -0.789, 1.792, -1.77, 0.443, -1.924, 1.161}

b.

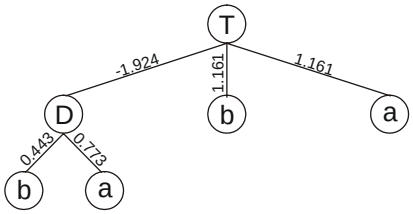


图 5.4GEP-网进化得到的异或问题的一个完美的，稍复杂的解。**a)**其染色体及其对应权值。**b)**编码在染色体中的完整表达的神经网络。

奇怪的是，在这个实验中，XOR 函数的一些其它的解也使用与这个简洁解完全相同的结构。的确，该算法所发现的三个参数的布尔函数不仅不止一个，而且还发明一些新的、出人意料的解。这清楚地说明 GEP 是一个惊人的发明机器，完全不需要先入为主的概念。

5.3.2.求解 6-多路由器的神经网络

多路由器是通信和通过单信道同时转发一定数量的独立信号的输入/输出操作中经常使用的逻辑电路。

6-位布尔多路由器的任务是对一个 2-位二进制地址{00,01,10,11}解码并返回相应的数据寄存器（d_0,d_1,d_2,d_3）。因此，6-多路由器是一个 6 个活性的函数：其中两个，a_0和a_1，决定其地址，另外 4 个，d_0到d_3，决定其回答。

6-多路由器的 6 个参数存在 2^6＝64 个可能的组合，对本问题而言，这 64 个组合构成的全集用来作为选择环境。6-多路由器的规则表如表 5.2 所示。适应度由方程（3.2）算得。因

此，f_{max}=64。

表 5.2 6-多路由器问题的查找表。输出二进制位由 000000 开始，以 111111 结束，按词法顺序给出。

```
00000000 11111111 00001111 00001111
00110011 00110011 01010101 01010101
```

为了简化分析过程，我选择一种较为紧凑的染色体组织结构并把“ Q ”函数排除在函数集之外。因此，对于该问题而言，F={U,D,T}，其中“ U ”表示有一个连通分量的函数；T＝{a,b,c,d,e,f}。分别代表{ a_0,a_1,d_0,d_1,d_2,d_3}；而且W={1,2,3,4,5,6,7,8,9}，每个的取值范围为区间[-2,2]。

表 5.3 的第一列所示的实验中，我们选择单基因染色体是为了更忠实地模拟神经网络。图 5.6 所示是找到的最简洁的解之一，它一共有 32 个节点。

表 5.3 采用单基因系统（US）和多基因系统（MS）求解 6-多路由器的参数

	US	**MS**
Number of runs	100	100
Number of generations	2000	2000
Population size	50	50
Number of fitness cases	64(Table5.2)	64(Table5.2)
Function set	$(U\ D\ T)_3$	$(U\ D\ T)_3$
Terminal set	a b c d e f	a b c d e f
Linking function	--	O
Weights array length	10	10
Weight range	[-2,2]	[-2,2]
Head length	17	5
Number of genes	1	4
Chromosome length	103	124
Mutation rate	0.044	0.044
Intragenic two-point recombination rate	0.6	0.6
Gene recombination rate	--	0.1
Gene transposition rate	--	0.1
IS transposition rate	0.1	0.1
IS elements length	1,2,3	1,2,3
RIS transposition rate	0.1	0.1
RIS elements length	1,2,3	1,2,3
Weights mutation rate	0.002	0.002
Dw spesicif transposition rate	0.1	0.1
Dw spesicif IS element length	2,3,5	2,3,5
Success rate	4%	6%

a. TbDTTTfTTaUDcUUTTafeefebabbdabffddfcfeeeabcabfabdcfe...
...709761631479459597193997465381760511137453583952159

W = {0.241, 1.432, 1.705, -1.95, 1.19, 1.344, 0.925, -0.163, -1.531, 1.423}

b.

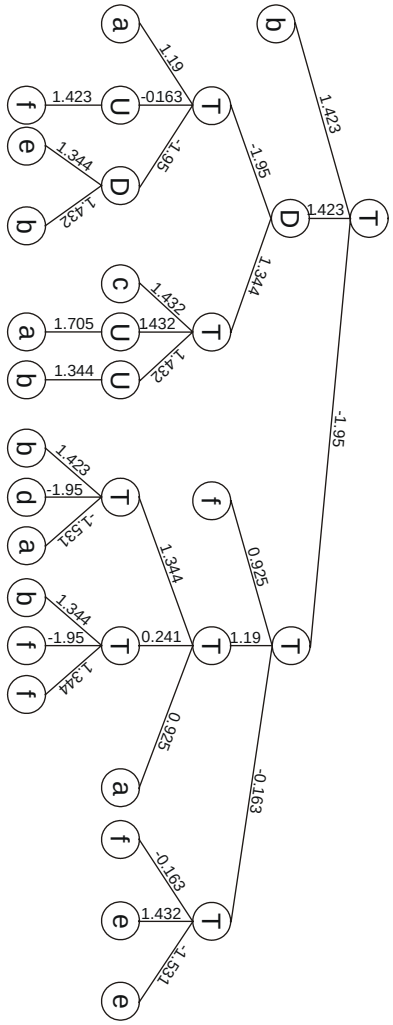


图 5.6.用 GEP 设计的神经网络发现的一个 6-多路由器函数的一个完美解。a)其染色体和相应的权值数组。b)编码在染色体中的完全表达的神经网络。

显然，我们可以探寻 GEP 染色体的多基因本质，并进化出多基因神经网络。然而这些解在结构上限制较多，因为我们必须选择某种连接函数来连接每个基因所编码的子-NN。(如果将 OR 与 “ U ”, “ D ”, “ T ” 函数混用造成混淆，将 OR 设想成具有两个连通量，权值和阈值等于 1 的函数，就能够的一个神经网络 OR)。

表 5.3 的第 2 列所示的实验中，翻译后的 4 个基因用 OR 连接。该实验发现的第一个解如图 5.7 所示。注意 1 号基因和 2 号基因的某些权值相同，而且对 3 号基因和 4 号基因也出现了相同的情况。这很可能说明这些基因有一个共同的祖先。

a.

```
01234567890123456789012345678890
TecTDdfafabdddfa487674791701403-[1]
TDcbTbadddfceacc501702156029560-[2]
TfTTUbadbcdffdce593993321226318-[3]
TDTbaceaaeeacacd072636270049968-[4]
```

W_1 = {1.126, 0.042, 1.588, -0.03, -1.91, 1.83, -0.412, 0.607, -0.294, -0.659}
W_2 = {-1.961, 1.161, 1.588, -0.03, -1.91, 1.762, -0.412, -0.121, -0.294, -0.659}
W_3 = {1.558, -0.69, 0.921, 0.134, 0.468, -1.534, 0.966, 1.399, 0.023, 0.915}
W_4 = {1.558, 0.767, 0.076, 0.071, 0.468, -1.534, 1.387, -1.857, -1.88, 0.331}

b.

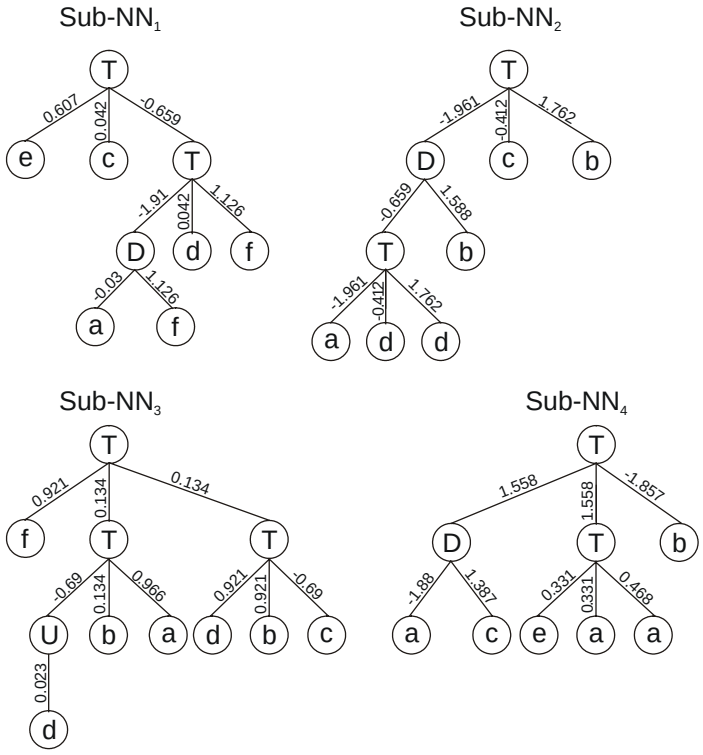


图 5.7.编码在 4-基因染色体中的一个 6-多路由器函数的一个完美解。**a）**基因分开显示的染色体。W1-W4 是含有每个基因权值的数组。**b)**每个基因所编码的子-NN。在该完美解中，子-NN 由 OR 连接。

5.4.GEP-网的进化动力学

本章所讨论的神经网络可能是 GEP 所产生的最复杂的实体。所以，让我们看看这些系统的进化动力学是否与其它一些较为简单的系统表现出某些相同的模式将会是很有趣的。如图 5.8 所示，GEP-NN 系统表现出某些与较为简单系统相同的动力学特点。图 5.8 所示为表 5.3 的第二列所示的实验的一次成功运行的特定的动力学图示。请注意平均适应度的特殊的振荡模式，而且其最佳适应度明显好于其平均适应度。

这些动力学的无处不在说明所有健康的基因型/表现型进化系统很可能都由它们控制。GEP 的动力学将在第 7 章进一步研究，但是在此之前，让我们先分析一下基本基因表达式算法的另一个变型，我们用它来解决调度问题。

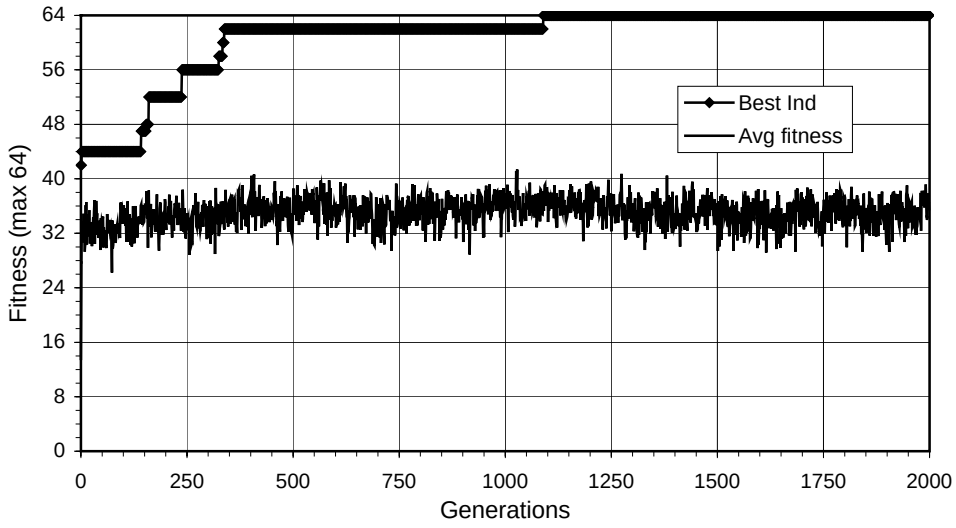


图 5.8.复杂 GEP 系统中，准确地说，即表 5.3 第 2 列所给出实验的第 4 次运行时发现的进化动力学。

组合优化 6

我们已经看到，当 GEP 采用每个基因仅由一个终点构成的最简表示方法时，它与经典的 GA 是等价的。的确，用方程 2.4 来决定尾部的长度，取 h=0 和 n=0，我们得到的基因长度 g=1。而且基因仅仅由终点构成，我们用这种简单的组织结构来求解 11-多路由器问题，其中编码在每个基因重的单元素表达式树翻译后由布尔函数 IF 连接（Ferreira 2001）。然而为了解决组合优化问题，需要另一种连接。例如，在 TSP 问题中连接显然应该由代表城市之间距离的两个相邻基因构成。

进一步，组合优化问题需要针对组合问题的搜索算子，以使备选解种群能够有效进化。的确，为了产生一些针对组合的高效算子，一些研究者创造出变异和重组等遗传算子的变型。但是由于从来未曾对这些算子进行过系统的比较，因此我们并不知道哪些算子效果更好。在这一章，我们将描述一种基于多基因族的新的染色体结构上进行搜索的新算法。我们将会看到这种新的染色体结构与一些针对组合问题的算子一起使得该算法表现出很高的效率，比经典的遗传算法的效率要高出许多，这些算子主要有：倒置，基因删除/插入，序列删除/插入，限制型变异和概化置换。

6.1.多基因族和调度问题

在第二章我们已经介绍了多基因族，这种结构对于发现组合问题的解非常有用，因为多基因族里面可以包含多种不同类的终点/项。例如，一个商人访问的城市就可以构成一个多基因族。下面这个染色体就是一个由多基因族构成的多基因染色体的例子：

```
01234567890123456 78
PMJLFNDOGKHSRQIECAB
```

其中每个字母代表一个城市。我们将在 6.3.1 节采用这种染色体结构来求解 TSP。对含有 N 类终点的组合问题，可以采用由 N 多基因族构成的多基因染色体。在 6.3.2 节的任务分配问题，即 6×6 的组合问题中，我们将采用两种多基因族表达两类不通的元素。例如，染色体：

```
012345012345
632451EDFCBA
```

含有两种多基因族，第一个多基因族由表示“助手”A={1,2,3,4,5,6}的集合 A 中的 6 个元素构成，第二个由表示任务的集合 T={A,B,C,D,E,F}的集合 T 中的 6 个元素构成。显然，必须预先确定一个多基因族中的成员如何与另一个多基因族中的成员相互作用，并在染色体中对其隐式编码。图 6.1 显示了两个多基因族中成员的一种十分直接的相互作用，其中 MGF1

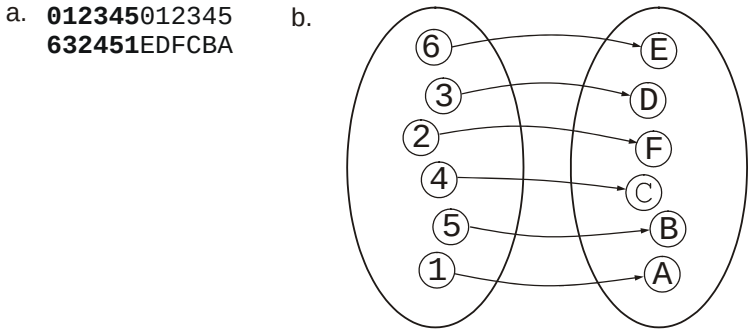


图 6.1.由多基因族构成的染色体的表达。**a**)含有两个多基因族的染色体。**b**)发展完全的个体，其中子表达式树之间的相互作用用箭头表示。

中 1 号成员与 MGF2 中 1 号成员相互作用，其中 MGF1 中 2 号成员与 MGF2 中 2 号成员相互作用，依此类推。

显然，不同的组合问题需要不同的染色体结构及多基因族的特殊的相互作用，但是这种结构几乎是所有调度问题编码的基础。

6.2.针对组合问题的算子：效果和机制

显然，我们不能通过不加修改地使用基本基因表达式算法的遗传算子来解决组合问题。在这些问题中，多基因族的元素都必须且只能表示一次。因此，我们为了在种群中引入遗传变化，必须创造出新的机制。这一节我将以介绍三种高效的，针对组合问题的遗传算子，即倒置，基因删除/插入，序列删除/插入作为开始，以介绍集中效果较差的算子，即限制型变异和概化置换作为结束。这些针对组合问题的算子都能在不打破多基因族平衡的情况下引入遗传变化，因此他们所产生的结构都是合法的。

在我们继续描述它们的机制之前，比较一下它们的效果会很有用（图 6.2）。我们选择 6.3.1 节的 TSP 问题来对该问题进行分析。该 TSP 问题采用 19 个城市，种群大小为 100，进化代数为 200 代。图 6.2 清楚地说明倒置算子的效果远远好于其它算子，接下来是基因删除/插入，而限制型置换的效率很低。

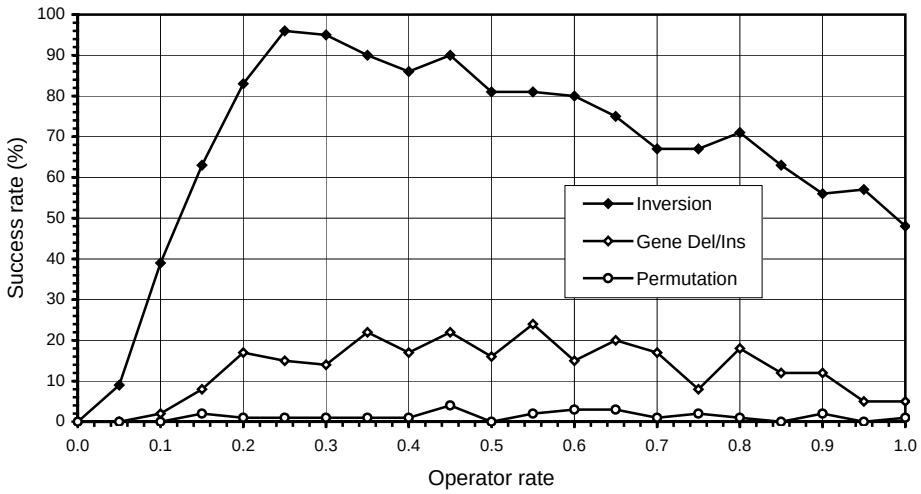


图 6.2.19 个城市的 TSP 问题上倒置（Inversion），基因删除/插入（Gene Del/Ins）以及限制型置换（Permutation）的比较。该分析中 P＝100 且 G＝200。成功率由 100 次相同的运行计算得到。

6.2.1.倒置

倒置算子随机选择染色体、将被修饰的多基因族、MGF 中的倒置点，然后将这些点之间的序列倒置，每个染色体只能被该算子修饰一次。

考虑如下由两个多基因族构成的染色体

```
012345678901234567801234567890123456 78
snpbqhfgkicmjlaedorPRMDCNLEBQFGKJOHIAS          (6.1)
```

假设 MGF2 中的 2 号基因和 7 号基因被选为倒置点，这两点之间的序列倒置后得到

```
012345678901234567801234567890123456 78
snpbqhfgkicmjlaedorPRELNCDMBQFGKJOHIAS          (6.2)
```

注意，可以通过倒置算子将整个 MGF 倒置过来。这种情况在头尾基因被选为倒置点的时候就会出现。例如，染色体中 MGF2 倒置得到：

```
012345678901234567801234567890123456 78
snpbqhf gk icmjlaedor SAIHOJKGFKBELNCDMRP          (6.3)
```

还要注意该算子允许一些小的调整，如两个相邻基因的交换。例如，如果染色体（6.3）中 MGF1 的 7 号基因和 8 号基因被选为倒置点，这两个基因就会交换位置。得到

```
012345678901234567801234567890123456 78
snpbqhf kg icmjlaedorSAIHOJKGFKBELNCDMRP          (6.4)
```

如图 6.2 所示，倒置是针对组合问题的遗传算子中功能最强的，即使将其作为遗传修饰的唯一来源，种群仍然能够以很高的效率进化，的确，仅仅采用该算子得到的结果比结合基因删除/插入或交换时得到的结果还要好。对于大多数问题，采用 20%到 60%的倒置算子就能产生很好的效果。

6.2.2. 基因删除/插入

基因删除/插入时针对组合问题的算子中第二重要的算子（见图 6.2）

该算子随即选择要修饰的染色体和多基因族，移位的基因和插入位置，每个染色体只能被该算子修饰一次。考虑如下由两个多基因族构成的染色体：

```
012345678901234567801234567890123456 78
rpifg hasbdeocjknlqmQSKLHCIGDONPFEJMBRA          (6.5)
```

假设 MGF1 中的 5 号基因被选中移位到 14 号位（13 号基因与 14 号基因之间），然后 5 号基因（“ h ”）在其原始位置上被删去，插入到“ J ”和“ K ”之间，得到：

```
012345678901234567801234567890123456 78
rpifgasbdeocj hknlqmQSKLHCIGDONPFEJMBRA          (6.6)
```

当基因的插入/删除与最强大的算子结合的时候，可能对细微调整有用。但是，对于本章分析的所有问题而言，搜索中仅采用倒置时获得的效果更好。

6.2.3.限制交换

限制交换允许某个特定的多基因族内任意两个位置上的基因交换位置。该算子与倒置算子结合的时候，可能对细微调整有用，但是如果将其作为遗传变化的唯一来源，其效果非常差（见上图 6.2）。限制交换算子随机选择要修饰的染色体和多基因族，要交换的基因，然而每个染色体只能被该算子修饰一次。

考虑如下由两个多基因族构成的染色体：

```
012345678901234567801234567890123456 78
ikmosfghdeqprljncabLNJIHG CDPSRQOBK MFAE          (6.7)
```

假设 MGF2 中的 6 号基因（“ C ”）和 15 号基因（“ M ”）被选中进行交换，然后形成如下染色体：

```
012345678901234567801234567890123456 78
ikmosfghdeqprljncabLNJIHG MDPSRQOBK CFAE          (6.8)
```

限制交换在使用概率很小的时候或者与倒置结合起来使用的时候，可能对微调有用。但是，同样对于本章分析的所有问题而言，当把限制交换与倒置结合起来使用的时候，成功率稍有降低。

6.2.4.其它搜索算子

这一节，我们将分析另一组针对组合问题的算子：序列删除/插入和概化交换。这些算子分别与基因删除/插入和限制交换相关。与倒置相比，这些算子效果极差，但是通过分析他们的效果和原理可以让我们了解一个号的针对组合问题的算子必须具备哪些基因性质。

6.2.4.1. 序列删除/插入

在第 6.2.2 节中我们已经看到基因删除/插入算子只允许基因的移位。换言之，它只允许由单个元素构成的小的序列的移位。我们可以很容易地实现对变长序列进行删除/插入的算子（序列删除/插入算子）。它看起来比基因的删除/插入更由优势，但经验正好相反（见图 6.3）。实际上，在 19 个城市的 TSP 中概算子产生的结果比限制交换算子更差（与图 6.2 比较）。的确，对该算子作同样的分析说明，对 19 个城市的 TSP，采用种群大小为 100，金花代数为 200 的时候序列删除/插入不能解决该问题。因此，我们采用含有 13 个城市的较简单的 TSP 来比较基因删除/插入和序列删除/插入的效果。分析时，采用种群大小为 100，金花代数为 200。这与较复杂的 19 城市的问题所采用的 P 值和 G 值完全相同。

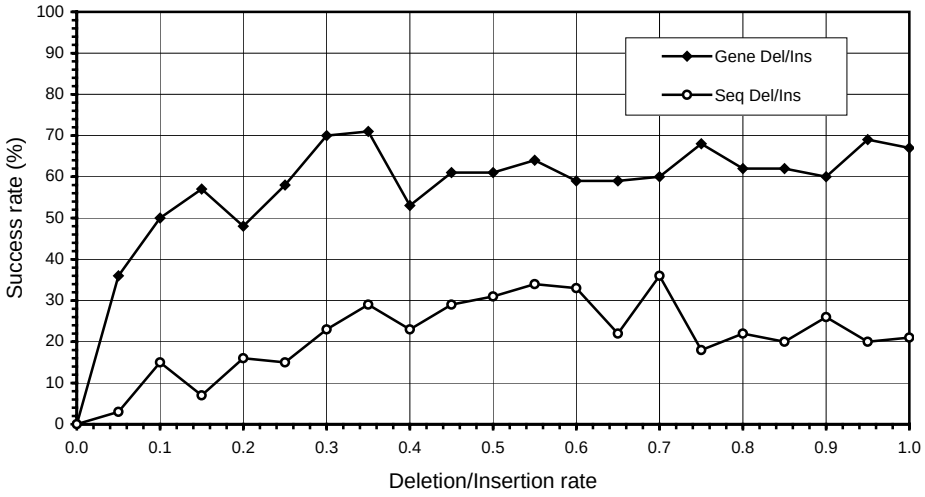


图 6.3.13 个城市的 TSP 问题上基因删除/插入（Gene Del/Ins）和序列删除/插入（Seq Del/Ins），该分析中 P＝100 且 G＝200。成功率由 100 次相同的运行计算得到。

6.2.4.2.概化交换

概化交换时 6.2.3 节中介绍的限制交换的一种变体。我们还记得在限制交换时，每个染色体中只有一对基因相互交换，即限制交换概率P_{rp}由$P_{rp}=N_c/p$来计算，其中N_c代表被修饰的染色体的个数。该算子的概化版很容易实现，其中每个染色体中不同个数的染色体可以按照某个比例交换位置，概化交换概率P_{gp}由$P_{gp}=N_G/(C_{L*}P)$来计算，其中N_G代表被修饰的基因个数，C_L代表染色体的长度。同样，该算子看来应该比上面提到的限制交换更有效，但经验说明限制交换效果稍好一点（见下图 6.4）。例如，在 19 城市TSP中（见上图 6.2）概化交换的效果比限制交换要差。实际上，概化交换找不到问题的完美接。

图 6. 4 所示为简单版 13 个城市 TSP 问题得到的结果。对限制交换算子和概化交换算子进行比较分析时，采用种群大小为 100，金花代数为 200。这与较复杂的 19 城市的问题所采用的 P 值和 G 值完全相同。

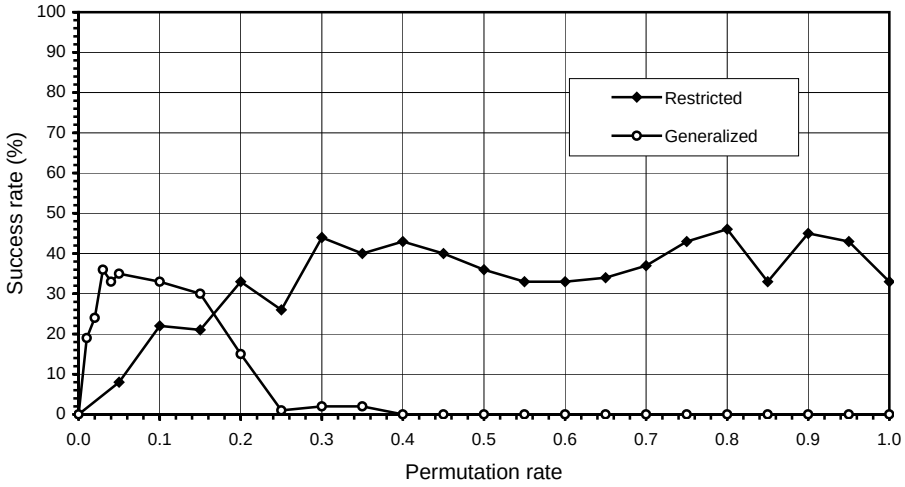


图 6.4.13 个城市的 TSP 问题上限制型交换和概化交换，该分析中 P＝100 且 G＝200。成功率由 100 次相同的运行计算得到。

6.3.两个调度问题

本节的第一个问题是已经讨论的 19 个城市 TSP。我们已经看到该问题只需要一个用来代表商人的城市的基因所构成的多基因族。

第二个问题是一个任务分派问题，它需要两种不同的多基因族：一种含有代理，另一种是分派给代理的任务。

6.3.1.旅行商问题

TSP 问题代表一类经典的优化问题，而且已经有人开发出了许多好的传统的近似算法来解决该问题（例如简介见 Paradimitrion 和 Steiglitz 1982）。但是，对于进化计算机专家来说，TSP 代表一大类与工业调度相关的组合问题（Bonachea 等 2000；Hsu 和 Hsu 等 2001；Johnson 和 McGeoch1997；Katayama 和 Narihisa 1999；Merz 和 Freisleben 1997；Reinelt 1994）。的确，许多进化激励的算法将 TSP 作为战场来开发针对组合问题的搜索算子（例如，见 Ferreira 2002b 中列举的最常用的针对组合问题的算子的清单）。

对 19 城市的TSP，要搜索的组合共有 19！＝1.21645×10^{17}。如果我们将起点固定（相应地终点也被固定，因为商人必须回到起点），可能的组合减少一半，达到 6.0823×10^{6}。进一步而言，我们如果采取某种设置，使所有的城市环绕在如图 6.5 所示的长方形上，我们可以严格地对算法的效果进行评价，因为我们预先知道正确的路径，对 19 城市的路程来说，最短距离应为 20。

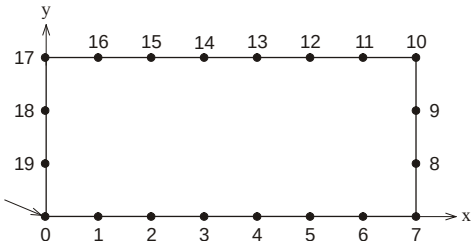


图 6.5.排列在一个长方形上的 19 个城市。箭头表示其实电和终止点。显然，最短路径是环绕长方形，长度为 20。

显然我们不能直接采用路径长度作为适应度函数度量，因为路径越短，个体适应度越好。因此，对于每一代而言，第g代中个体程序i的适应度f_i由以下公式给出：

$$f_i = T_g - t_i + 1 \qquad (6.9)$$

其中t_i是个体i所编码的路径长度，T_g是当前代中染色体所编码的最大路径长度。这样，种群中最差个体的适应度永远为 1。与平时一样，个体由赌盘选择策略根据个体的适应度进行选择，而且每一代中的最佳个体被原样复制到下一代中，每次运行的参数如表 6.1 所示。

表 6.1 19 个城市的 TSP 问题的参数

Number of runs	100
Number of generations	200
Population size	100
Number of multigene families	1
Number of genes per multigene family	19
Chromosome length	19
Inversion rate	0.25
Success rate	96%

如果将 GEP 在 19 城市 TSP 上得到的结果和 GA 在 19 城市 TSP 上得到的结果相比，将会让人非常吃惊。例如，Haupt 和 Haupt（1998）在采用大小为 800 的种群和 200 代进化代数的情况下找不到最短路径，如图 6.2 和表 6.1 所示，这里描述的算法不仅能够在 100 个个体和 200 代的情况下找到最短路径，而且几乎每次都能够找到最短路径（实际上，在 96%的运行中如此）。只得强调的是，在本实验中，采用倒置作为遗传变化的唯一来源。的确，在有其它遗传算子的情况下，即基因删除/插入和限制变换的情况下，成功率会稍微降低一些。因此，这里没有使用这些算子。显然，当采用倒置进行搜索的时候没有必要进行细微调整。

如本章早先所提到的，这里用来解决组合问题的染色体与经典 GA 完全对应。那么，是什么导致这两种算法之间效果的惊人的差别呢？显然，答案在于 GEP 和 GA 中采用的遗传算子集。如上图 6.2，6.3，6.4 所示，倒置算子的效果比变换（包括限制和概化实现）要好很多。而各种不同的变换和形式极其复杂的杂交恰好是 GA 研究者们求解组合问题时所喜爱的搜索算子。不幸的是，在遗传算法的早期，倒置算子被抛弃了，因此即使今天在组合优化中也鲜有使用。

6.3.2.任务分派问题

这一节的任务分派问题（TAP）是 Tank 和 Hopfield（1987）在他们发表在《科学美国人》上的论文中所选用的一个“玩具问题”。该问题用来演示 Hopfield 网络在组合代价优化问题的机制。

在 TAP 中，有 n 个任务必须仅由 n 个工人来完成。每个工人在一方面比较擅长，而在其他方面则不太擅长。显然这些工人在某些任务上比其他工人更擅长。我们的目标就是将完成所有任务的总代价最小化，或者换言之，即从总体上使工人的输出最大化。

假设我们要让图书馆中的 n 个上架助理将 n 个书集放到书架上。每个助理对不同的主题和相应的书集的熟悉程度不同。该任务的输入数据或适应度样本由每分钟书的上架速度构成（图 6.6）。

	A	B	C	D	E	F
1	10	6	1	5	3	7
2	5	4	8	3	2	6
3	4	9	3	7	5	4
4	6	7	6	2	6	1
5	5	3	4	1	8	3
6	1	2	6	4	7	2

图 6.6.任务分配问题。每个助理 (1-6) 应该根据书每分钟上架的速率(适应度样本)被分配一个书集 (A-F)。阴影方块所示为具有最快的上架速率总和，44。

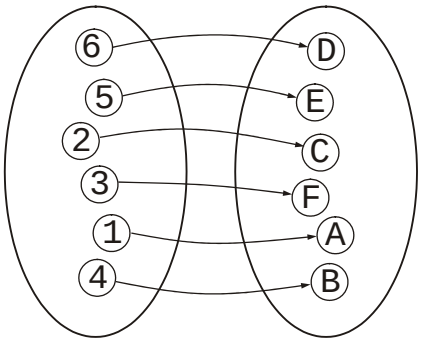
对于这个 6×6 的问题而言，助理和书集已经有 6！=720 种可能的分配方案。最优解的选中的助理的速率之和最快。对某个特定的数据或适应度样本集f_{max}=44。

这种玩具问题对于比较不同算法的效率非常有用，而且在使用的染色体由一个以上的多基因族构成的情况下，我们将进一步在系统中验证倒置算子的潜力。的确，当仅使用倒置作为遗传变化的来源，使用两个多基因族的时候就可以很高效地求解任务分配问题。这两个多基因族对助手编码，另一个对书集进行编码。下面使一个如下的染色体：

012345012345

652314DECFAB

它含有两个不同的 MGF，第一个对助手编码，另一个对书集进行编码，其表达式为：



其中分配过程由箭头表示，读者可以从图 6.6 中看到。该个体的f_i=41。

对于该问题，我们将采用个体数为 30 的较小的种群，进化代数为 50 代，每次运行的参数和以成功率方式给出的算法的效果见表 6.2。

表 6.2 任务分派问题的参数

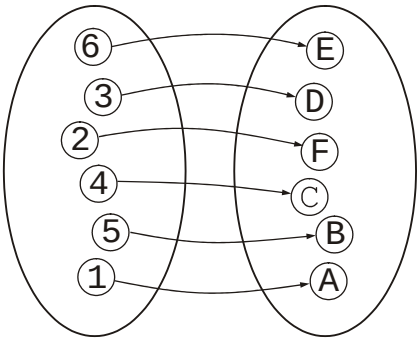
Number of runs	100
Number of generations	50
Population size	30
Number of multigene families	2
Number of genes per multigene family	6
Chromosome length	12
Inversion rate	0.30
Success Rate	69%

在本实验中的首次成功运行中，第 16 代找到一个具有最大适应度的解：

012345012345

536241EDCFBA

其中它对应适应度为 44 的最优分配（同样见上图 6.6）。



6.4.简单 GEP 系统和进化动力学

上一章我们通过分析 GEP 中最复杂的系统之一——线性编码神经网络——作为结束，我们还应注意到 GEP-网表现出与其它较为简单的基因组/表现型组相同的进化动力学特征。

如本章前面提到的，这种用于组合优化的简单的染色体结构与经典 GA 非常相似。因此，我们来看看这些较为简单的 GEP 系统和 GA 种群中所观察到的进化动力学相同将会是很有趣的。

让我们首先分析以下每个染色体中仅含有一个多基因族的最简单的 GEP 系统。其进化动力学如图 6.7。注意该简单系统具有与 GA 种群相似的进化动力学特征。在该图中平均适应度的图形紧挨着最佳适应度的图形，而且平均适应度的图形振荡程度较不明显。

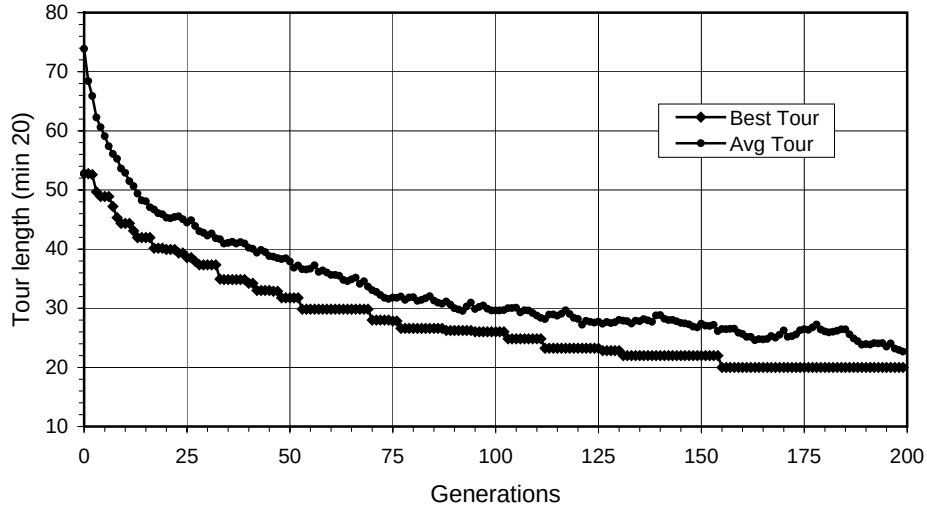


图 6.7.最简单的 GEP 系统中的进化动力学。该动力学是由表 6.1 所给出的实验的首次成功运行得到的（19 个城市的 TSP 问题）。

最后，让我们分析一个稍微复杂一点的系统，其中每个染色体由两个多基因族构成。图 6.8 所示的进化动力学图示是根据表 6.2 所给出的实验的一次成功运行得到的，值得注意的是，这里的进化动力学特征与我们所期望在 GA 中看到的不再一样。事实上，它具有 GEP 动力学的所有特征：平均适应度的振荡模式，最佳适应度和平均适应度之间相当大的差别。的确，我们希望这种动力学当染色体由许多成员之间的复杂的相互作用进行编码的两个多基因族构成时对该染色体进行表达需要更高的复杂度。

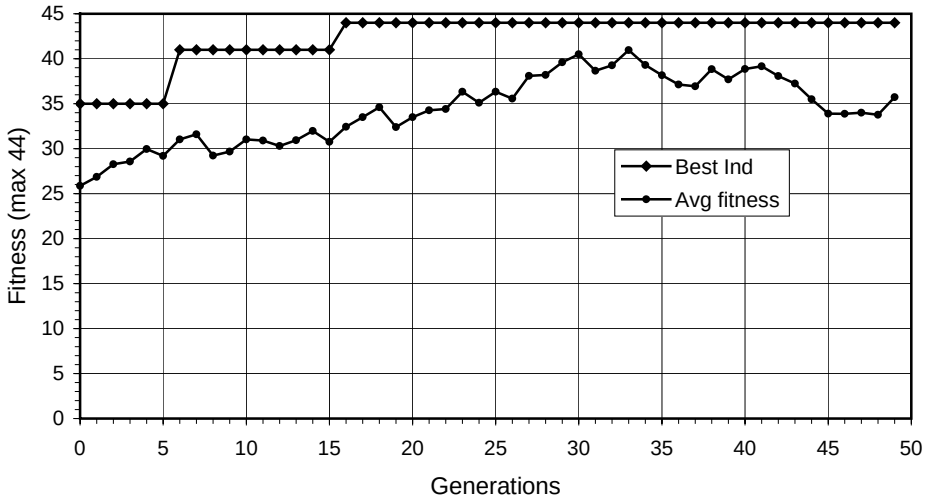


图 6.8.简单的 GEP 系统中的进化动力学。该动力学是由表 6.2 所给出的实验的首次成功运行得到的（6*6 的任务分派问题）。注意，该系统虽然简单，但是它不再表现出典型的 GA 动力学。

下一章我将进行一系列的进化研究，包括对经典的 GEP 种群的进化动力学的详细分析。这些研究的目的时为了讨论进化计算中一些颇具争议的话题，这些话题中大部分是 GEP 的发明本身产生并强化的。

本章的进化学研究既有实际价值也有理论价值。一方面，在进化理论中有许多颇具争议的问题，GEP 的简单基因行/表现型系统能让我们对进化过程有所了解。另一方面，进化计算不仅继承了进化理论中所有有争议的问题，还产生了一些它自身引起的问题，从而为进化理论留下更多问题，而不是解决了这些问题。

这一章我们将讨论进化理论的 3 个基本问题：(1) 遗传算子及其效力；(2) 初始多样性的重要性；(3) 进化过程的中性的重要性。进一步，与这些基本问题相联系的，还有一些进化计算中长期存在的相关问题，都从经验角度给予了明确的回答。这些问题包括，进化中基因块的作用和早熟收敛的问题。

7.1.遗传算子及其效力

每个人都或多或少同意进化依赖于遗传变化与某种形式的选择，并且实际上，所有的进化算法都要经过这些基本过程。然而，对于产生遗传变化的最佳途径却没有一致意见，大多数的研究者分为变异和充足两派。这一事实明确说明所有的人工进化系统相互之间存在根本上的不同。的确，人工进化系统自身仍然在进化，并且在它们中间，虽然伪装成各种不同的表达形式，但是还是能够发现一些简单的复制体系统，基本的基因组/表现型系统，和功能完善的基因组/表现型系统。而所有系统的遗传修饰机制都与它们的表达模式奇妙地相关。

我们已经看到，GEP 不仅采用变异和重组，还采用几种不同的转座，因此，可以用它们来对不同的搜索算子的效力进行严格的分析以了解它们在进化中的作用。我们将看到变异是最重要的遗传算子，性能远远好于重组。实际上，这里所分析的三种遗传重组都明显不及变异，也不及简单染色体间的转座机制。另外，为了理解这些遗传算子的重要性，我们还将详细分析所有遗传算子的进化动力学。

7.1.1.比较变异、转座和重组的性能

为了进行比较，选择了下面这个相对复杂的测试序列:

$$a_n = 5n^4 + 4n^3 + 3n^2 + 2n + 1 \qquad (7.1)$$

其中 n 由非负整数构成。选择该序列有四个原因：(1) 它可以精确求解，因此可以从成功率的角度为性能提供精确度量；(2) 它所需要的种群相对较小，进化代数相对较少，使得问题可行；(3) 它能提供足够的区分度来对性能不同的算子进行比较（比如变异和重组）；(4) 对所有的遗传算子进行研究是合适的，包括针对多基因系统的算子，如基因重组等。

在本节所有实验中，将前 10 个正整数n和相应的a_n作为适应度样本（表 7.1）;适应度由方程（3.1b）算得，采用的选择范围为 20%，精度为最大精度（0%误差）。因此得到f_{max}=200; 种群大小为P＝500，进化代数G＝100；成功率P_s根据 100 次独立实验求得；F＝{+,-,*,/}，终点集由自变量构成；最后，长度为 91 的 7-基因染色体由加法连接（h=6）。实验中不采用转座，3 个转位子得长度分别为 1，2，3。

这一节，我们将只研究 GEP 常用的 7 个算子中的 6 个，暂不考虑基因转座，因为它对于该实验特别选定的条件毫无贡献（即对子表达式书编码的基因由加法连接）。

表 7.1 序列推断问题的适应度样本集

n	1	2	3	4	5	6	7	8	9	10
$\mathbf{a_n}$	15	129	547	1593	3711	7465	13539	22737	35983	54321

所有算子的性能如图 7.1 所示，可以看到，变异是唯一的强大算子，接下来是 RIS 转座和 IS 转座，而重组是所有算子中性能最差的。还要注意，三种算子之间也存在明显的差异，如读者所看到的，两点重组是最有效的，而基因重组是效率最低的。值得指出的是对变异观测得到的手指形状的图形和转座与重组得到的图形非常不同。请注意，只有变异能够爬上“山顶”，而且，的确，经历变异的系统能很容易地调整使种群在峰顶保持平衡（虽然这种平衡并不稳定），因为变异概率上很小的变化对手指形区间有很大的影响。

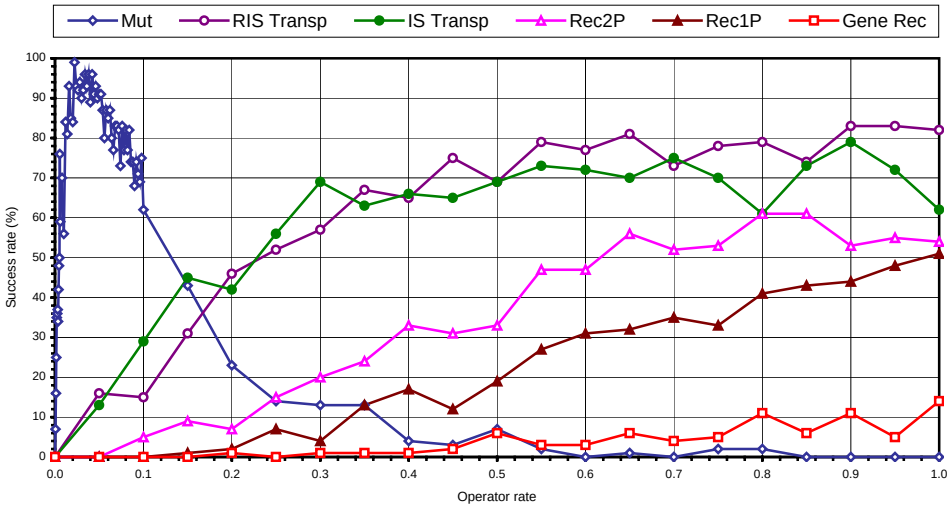


图 7.1.遗传算子及其效果

如图 7.1 所强调的，变异有巨大的创造力，的确，仅使用该算法就足以进化得到几乎所有问题的解。尽管如此，处于实际和理论的原因，GEP 中还是存在并有规律地使用其它遗传算子。例如 RIS 转座有趣是因为它向我们关于剧烈修饰的性能的看法提出了挑战。而且，众所周知的是，变异不足以产生自然界所有的奇迹。例如，考虑由共生产生的真核细胞（Margulis 1970）或较为简单的基因复制现象（Lynch 和 Conery 2000）也可以在 GEP 中看到。

IS 和 RIS 转座的得到的结果同样有趣，特别有趣的事实是这两种简单的基因内转座不仅远远超过所有的重组而且 RIS 转座比 IS 转座性能稍好。值得指出的是，RIS 转座中的基因的 1 号位总是目标位置。这意味着，从表现型的层面来说，子表达式的根发生了变化。的确，这种修饰是最具有破坏性的操作之一，与发生在蛋白质基因的开头的转座变异类似。注意这两种转座的变换能力略差于变异（比较某个算子得到的最大性能）。

还有值得讨论的是三种重组得到的结果。我们还记得两点重组是最具破坏性的算子，如图 7.1 所示，也是最有效的一种重组。毫不奇怪，最保守的重组算子——基因重组——效率最低。另外，值得注意的是这里分析的所有的重组机制，哪怕是最保守的，都比有性繁殖中的同源重组更具有破坏性，因为在 GEP 中，基因的交换很少是同源的。

生物学中一个未解的问题就是进化中性别的作用（Margulis 和 Sugan 1986），而且在很多情况下，生物学上的性别在其具有压倒性地位的多样性下经常与有性繁殖的同源重组相混淆。因此，很多人错误地认为同源重组产生更大的多样性，虽然有许多事实与假设不符

（Margulis 和 Sugan 1997）。GEP 算子的比较，特别是对重组的比较，说明一种较为保守的重组机制如同源重组只对保持停滞现状有用，有趣的是，这种假设进一步受到仅经历重组的种群的进化动力学支持（见 7.1.2 节）。

7.1.2.GEP 种群的进化动力学

图 7.1 所示的种群的进化动力学能够帮助我们了解不同的进化系统的适应性能力。最有趣的是那些性能最佳的系统。显然，我们将首先分析这些系统与之相比，另一些系统则效率较差。

7.1.2.1.变异

在图 7.1 分析的基础上，对经历变异的关键种群的分析可以作为与经历其它遗传修饰的种群进行比较的一个参照。变异图形上升一侧的种群在小的变异概率下能够健康地进化。因此，通常称之为健康种群在下降一侧的种群在过度变异下进化，因而被称为不健康种群。图 7.2 可以看到种群的平均适应度如何变化，以及种群的平均适应度如何随图 7.1 的变异曲线运动的最佳适应度图相关。

在第一个进化动力学p_m=0.001，P_s=16%（图a）平均适应度曲线与最佳适应度曲线紧紧相伴，而且平均适应度只有一个很小的振荡。该种群因为它们进化非常缓慢，而且种群中引入的遗传多样性很少而被称为中等创新的。

在第二个进化动力学中，p_m=0.0045，P_s=48%（图b）。可以看到，虽然平均适应度与最佳适应度紧紧相伴，两条曲线间的差距还数逐步变大。而且其平均适应度的振荡模式特征在很小的变化率下已经非常明显。如图 7.1 所示，该种群的成功率为 48%，在通往峰值的半路上，这样的系统虽然能很健康地适应，但效率并不高，因此称之为健康但虚弱。

如图 7.1 所示，当变异概率到达一个接近p_m=0.022 的稳定状态时，成功率出现猛增，从动力学的角度说，这反映了一种平均适应度的更明显的振荡模式，以及平均适应度和最佳适应度之间的差距变大。出于显而易见的原因，性能最好的种群被称之为健康且强壮。紧接下来的 3 个动力学（图b，c，d，e）都来自于效果的平台或峰值。然而，要注意，由峰值继续向前，变异概率增大导致效果下降直到整个种群完全不具有适应能力（见下图）在这些种群中，平均适应度与最大适应度之间的差距。持续增大（见图f到图i）直到平均适应度曲线达到底部，种群变成完全随机，而且不具有适应能力（见图i）。注意某些种群虽然在过度变异下进化，却仍然非常高效。例如，图f所示的进化动力学，p_m=0.046，P_s=93%，仍然极其高效。这样的种群被称为不健康但强壮（不健康是指变异概率过高）。下一个种群，p_m=0.0045，P_s=48%（图g）。不像前一个种群那么高效，因而被称为不健康且虚弱。

最后两个动力学由变异曲线的手指形区域外的种群得到，前一个的p_m=0.45，P_s=3%（图h）,平均适应度的曲线与由完全随机种群得到的最小位置相隔不远（图i）。因此，这样的种群分别叫做几乎随机和完全随机。注意这两个图中的平均适应度的振荡不那么明显。值得注意的是完全随机种群（差一句话）找不到当前问题的完美解。这告诉我们每当找到一个完美解的时候。都是因为有一个强大的搜索机制而不是一个随机的搜索。

7.1.2.2.转座

转座的进化动力学如图 7.3 所示，如读者所看到的，RIS 和 IS 转座所得到的动力学与变异所得到的类似。而且，这里值得指出，因为并非所有的算子都表现出这种动力学（见下面

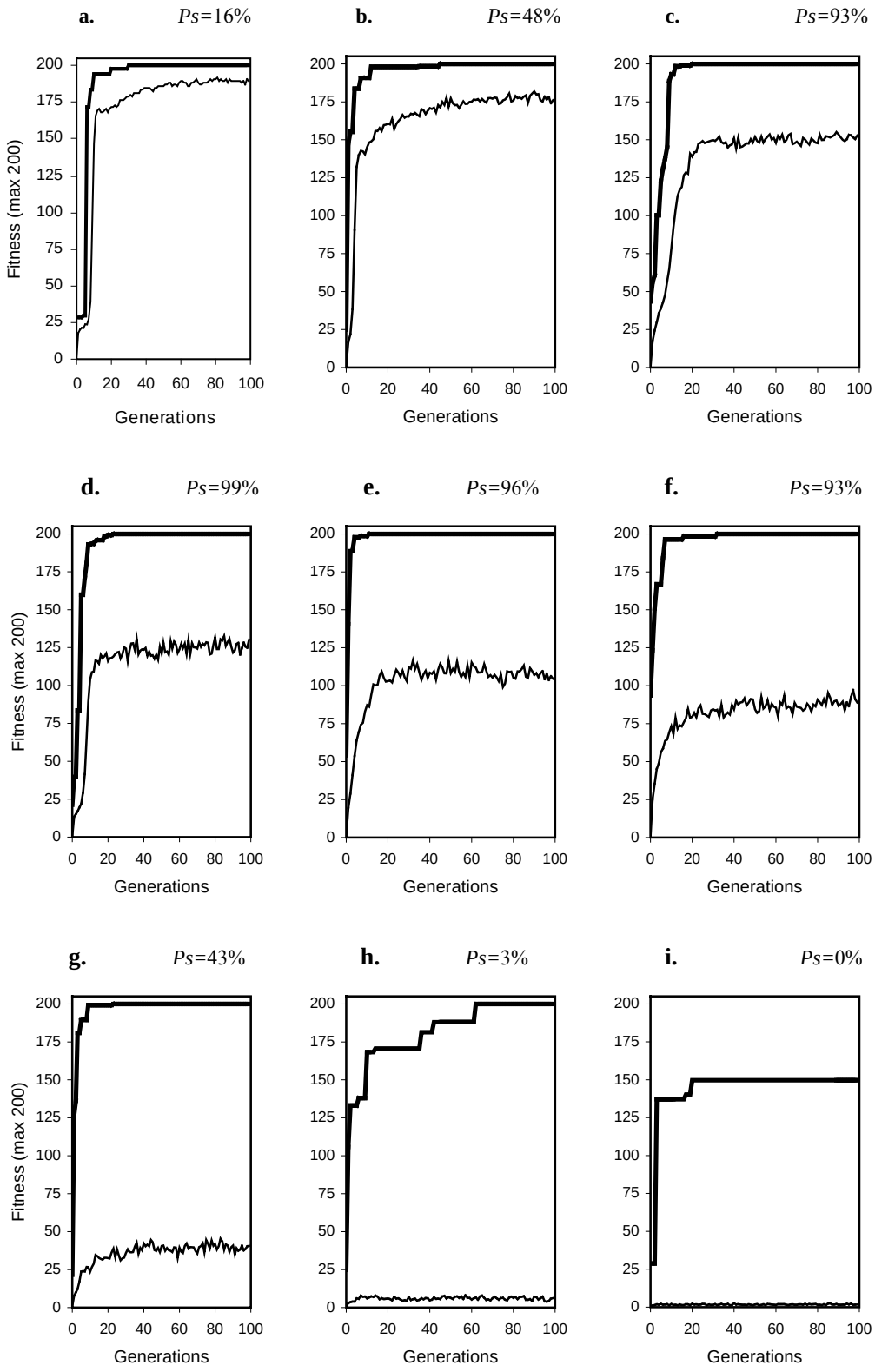


图 7.2.变异的进化动力学。上面每个图形的成功率由图 7.1 所示的实验决定。**a)** 中等创新的（p_m=0.001）。**b)**健康但虚弱的（p_m=0.0045）。**c)**健康且强壮的（p_m=0.016）。**d)** 健康且强壮的（p_m=0.022）。**e)** 健康且强壮的（p_m=0.034）。**f)**不健康但强壮的（p_m=0.034）。**g)** 不健康且虚弱的（p_m=0.15）。**h)**几乎随机的（p_m=0.45）。**i)** 完全随机的（p_m=1.0）。注意除了最后一个动力学以外，只选择了成功的运行以使最佳适应度的图形达到其最大值并使得平均适应度与最佳适应度之间的距离能更好地辨认。

重组的动力学)。像变异中的一样，平均适应度与最佳适应度之间的差距随转座概率的增大而增大。还值得指出的是，种群在P_{ris}=1.0（P_s=3%）的情况下比P_{is}=1.0（P_s=62%）的情况下进化效率更高。因此，平均适应度与最佳适应度之间的差距在最后一种情况下最小（比较图c和f)。更一般地说，RIS转座的平均适应度曲线比相应的IS动力学所占的位置更低。如前所示，对于健康的种群，这是一种更加高效的进化的标志。的确，如图 7.1 所示，RIS转座比IS转座性能稍好一点。

RIS 和 IS 转座在行为上表现上与变异的相似性更进一步强化了重组的唯一性。下一节我们将看到仅经历重组的种群行为非常奇怪 :它们在平均适应度没有明显的振荡的情况下进化得非常平滑，而且一直试图缩小平均适应度与最佳适应度之间的差距。

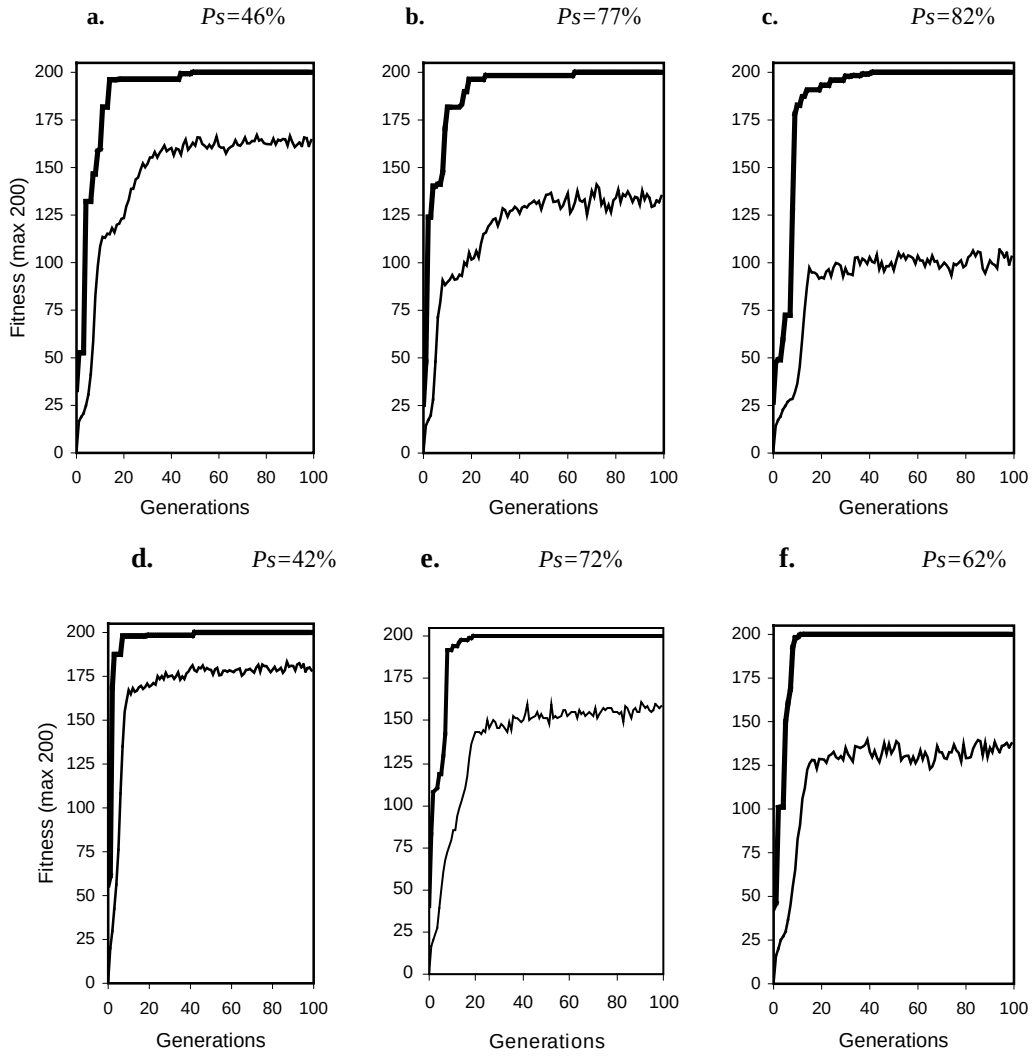


图 7.3.由RIS转座（图**a,b,c**）和IS转座（图**d,e,f**）进化得到的种群的进化动力学特征。上面每个图形的成功率由图 7.1 所示的实验决定。**a)**p_{ris}=0.2。**b)** p_{ris}=0.6。**c)** p_{ris}=1.0。**d)** p_{is}=0.2。**e)** p_{is}=0.6。**f)** p_{is}=1.0。注意该进化行为与变异类似，而与重组由很大不同（见图 7.4)。还要注意RIS转座的平均适应度的图形在图中的位置比相应的IS转座的要低，这表明IS转座是一种波动较大而且进化更加有效的指标。

7.1.2.3.重组

重组的进化动力学说明所有的重组算子，从最保守的（基因重组）到最具破坏性的（两

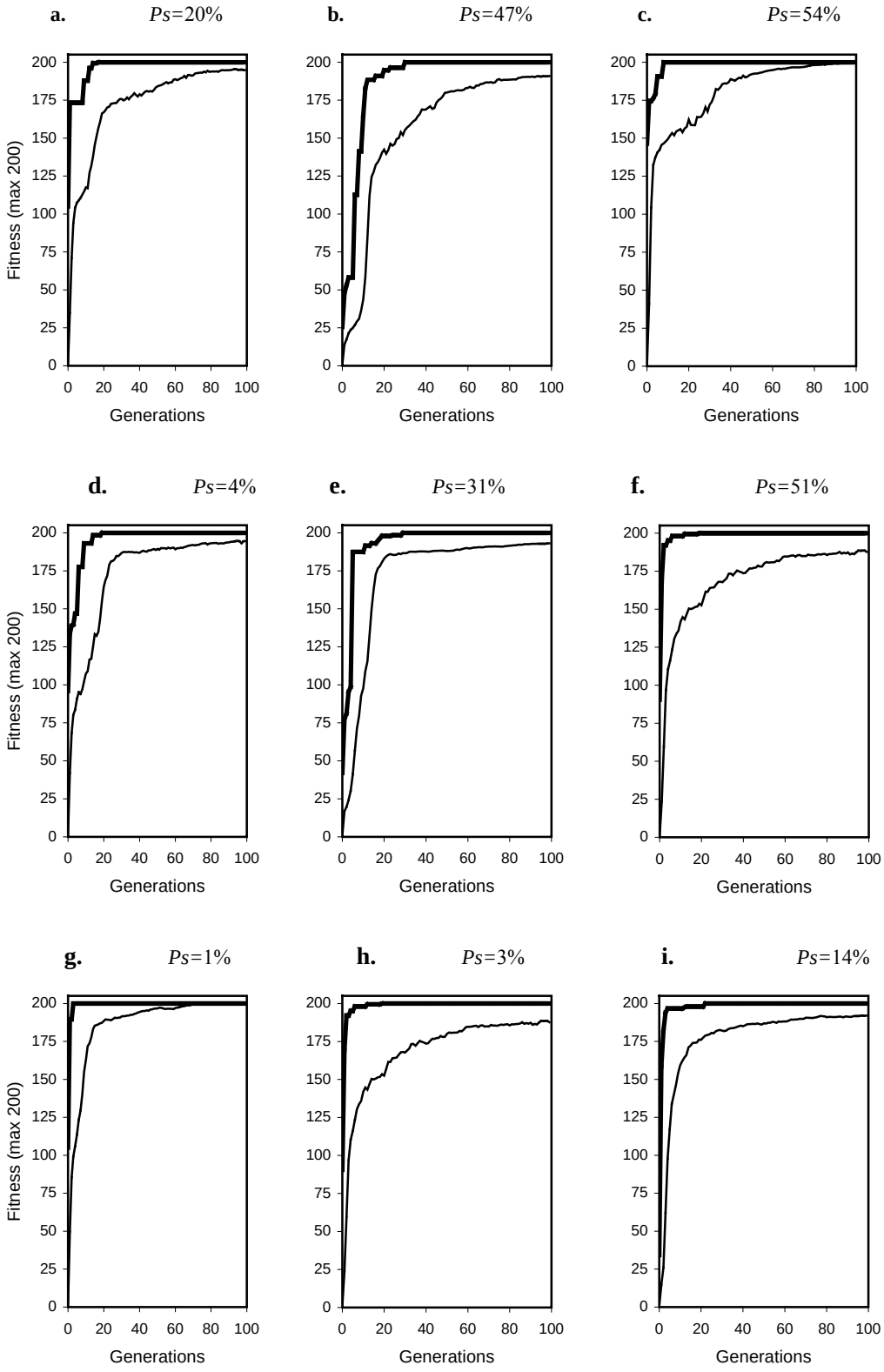


图 7.4.由两点重组（图**a,b,c**）和单点重组（图**d,e,f**）及基因重组（图**g,h,i**）进化得到的种群的进化动力学特征。上面每个图形的成功率由图 7.1 所示的实验决定。**a)** p_{2r}=0.3。**b)** p_{2r}=0.6。**c)** p_{2r}=1.0。**d)**p_{1r}=0.3。**e)** p_{1r}=0.6。**f)** p_{1r}=1.0。**g)**p_{gr}=0.3。**h)** p_{gr}=0.6。**i)** p_{gr}=1.0。注意所有的重组操作产生同种类型的同质动力学。还要注意这些动力学与变异和转座的动力学大不相同（见图 7.2 和图 7.3）。

点重组），表现出一种同质效应（图 7.4）。由于显而易见的原因，这些针对重组的动力学称为同质动力学。注意，在所有的情况下，平均适应度的曲线与最佳适应度的曲线紧密相伴。而且，只要有足够的时间，由于种群丧失所有的遗传多样性，曲线很容易重叠（例如，见图 c，g，h）。注意，这种情况的发生与重组概率和所涉及的重组算子的类型无关，而且，事实上，仅仅通过这些图，我们不仅能区别这些重组操作，也不能将高效系统与低效系统越来越差，最后变得完全不具备适应能力。显然，如果种群在找到一个好的解之前收敛到这个阶段，如果没有其它的非同质算子的话，它们就不可避免地陷入这一点。如重组所得到的较低成功率所强调的那样（见图 7.1）。当种群在只有重组的情况下进化时。大多数情况下都会在找到一个好的解之前收敛。这说明，如果想要系统有效地适应，绝对不能将重组作为遗传变化的唯一来源。

还要注意，同质动力学中平均适应度的振荡不那么明显。尽管如此，振荡随重组概率变化稍有增加。说明重组较多的种群具有更好的弹性，而且在陷入停滞之前需要更长时间。

总之，重组是一种同质算子，因此，不适合用来产生遗传多样性，在运行代数较长的时候，只经历变异的种群会变得停滞不前。因此，重组被从“ 进化风暴 ”的中心移走，而且该赋予它新的作用。一方面，如果它和其它遗传算子如基因转座相结合，它能在基因复制中起到重要作用。另一方面，它有保持现状的作用，通过不断使种群的遗传构成同质化而保持种群稳定。的确，物种从它们出生到灭亡的过程中都是稳定的实体（Eldredge 和 Gould 1972）。而且像同源重组这样的算子对于保持现状使不可或缺的。

另一方面，我们也观察到系统的性能不仅随变异概率明显变化，而且仅使用变异也能达到性能的峰值。因此，很容易对变异概率进行调整使系统以最大效率进化。实际上，变异概率需要严格地控制。而且在自然界中受到选择压力的限制，这也说明是变异而非重组处在进化风暴的中心。

最后，我们还观察到转座算子所显示的动力学与变异非常相似（即非同质动力学）。而且经历转座的种群在进化时明显好于只采用重组的种群。这进一步强调了重组的唯一的同质效应。

7.2.奠基效应

前一节我们已经看到一个系统的进化能力在很大程度上依赖于用来产生遗传修饰的遗传算子的种类。初始种群的规模和种类也与该问题密切相关。

在所有的进化算法中，一次进化进程或运行始于一个初始种群。虽然产生初始种群的方法多种多样，但不同算法的性能和代价（按照 CPU 时间来说）很大程度上依赖于初始种群的特征。最简单而且最省时的初始种群是完全随机得到的初始种群。但是，很少有进化算法能够采用这种初始种群，其原因一方面是出于结构上的限制，另一方面是由于用来产生遗传修饰的遗传算子有限。GEP 的初始种群是完全随机的。由种群个体的线性基因组构成。

在人工进化系统中，初始多样性的问题与两大因素相关。首先，对于一些复杂的问题，可存活的个体的产生（即有正的适应度的个体）可能是非常罕见的事件。在这种情况下，较有优势的作用是让进化过程从一个或几个奠基个体开始；这种情况是否可能，将取决于系统所能够使用的修饰机制。其次，由于以上原因，用来产生遗传修饰的机制的种类变得至关重要。如果采用像点变异这样的非同质算子来产生遗传修饰，那么种群就能适应并进化。但是，如果采用像重组这样的同质算子来产生遗传变化。那么进化要么会在只有一个奠基个体的情况下完全停顿，要么在奠基个体过小的情况下严重地折中。

进化中初始多样性的重要性是 E.Mayr 强调的，当时他称之为奠基效应物种形成（Mayr

1954，1963）。这一过程可以看成是一个新的物种的建立，这个新的种群的产生是由遗传漂变和自然选择所造成的。奠基效应的一种极端的例子就是单个受孕雌性对一个无人居住区域的统治。在自然界中，除了重组以外，还有其它的遗传算子来产生遗传修饰和种群，使种群能够跨越瓶颈，得以进化，甚至有些时候产生新的物种。

同样，在人工进化系统中，奠基种群的进化能力也很大程度上依赖于用来产生遗传变化的机制的种类。的确，如果仅仅采用同质算子来产生遗传修饰，在极端的情况下，如果只有一个奠基个体，种群将不能高效地进化或者根本不进化。

这一节，我们将分析两种不同系统中进化过程中初始多样性的重要性。第一个系统在变异的情况下进化，具有高效适应的非同质动力学特征，另一个系统在重组的情况下具有较差进化系统的同质动力学特征。

7.2.1.选择非同质和同质种群来研究奠基效应

为了对不同的种群如何对初始种群中的实际奠基个体的个数的反映进行精确的量化，必须选择一个简单的可以精确求解的问题。该问题必须允许对遗传算子性能的不同之处进行比较，例如点变异的高性能和重组的较低性能。另外，用来进行比较的种群必须服从不同的进化动力学以使这里讨论的结果不仅在理论上有用，而且能够帮助我们对不同的人工进化系统所选择的进化策略有所理解。

对于这里的分析，我们将采用已经熟悉的 4.2.2 节的检验序列（4.9）。在所有的实验中，前 10 个正整数n和它们相对应的项作为适应度样本（见表 4.6）适应度函数为方程（3.1），选择范围为 20%，采用最大精度（0%的误差），得到最大适应度为f_{max}=200；种群大小\P为 50 个个体，进化代数G＝100 代；成功率P_s根据 100 次独立实验求得；F＝{+,-,*,/}，终点集由自变量构成；最后，长度为 78 的 7-基因染色体由加法连接（h=6）。

我们已经看到变异是唯一的最重要的遗传算子，经历编译的种群表现出非同质化的动力学。进一步，变异是唯一能够到达性能峰值的算子，对于每个系统而言，都可以找到这个峰值。对于该问题，性能峰值如图 7.5 所示。在此，在变异概率p_m=0.05 附近达到最大性能峰值。因此，在本研究中我们将一直使用该值。图 7.6 所示健康且强壮的非同质动力学是由表 7.2 中给出的实验的一次成功运行得到的。

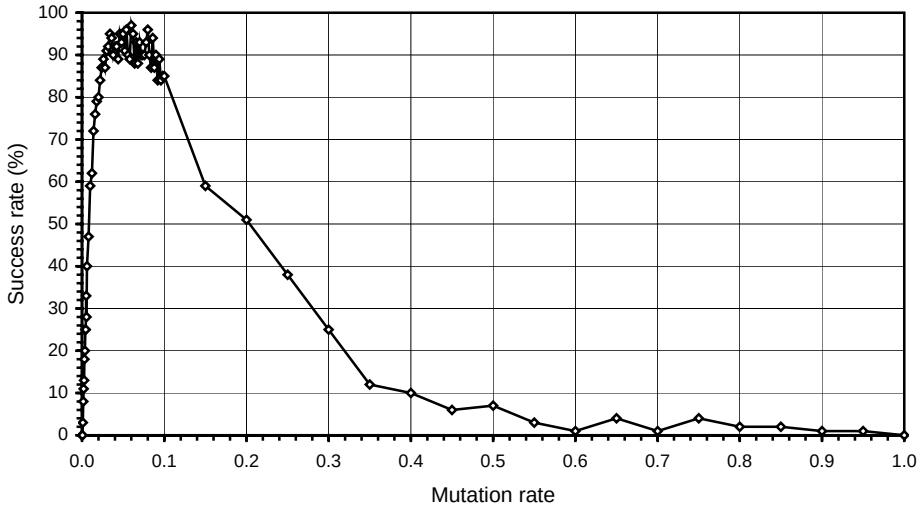


图 7.5.仅采用变异来决定性能峰值。成功率由 100 次相同的运行计算得到。

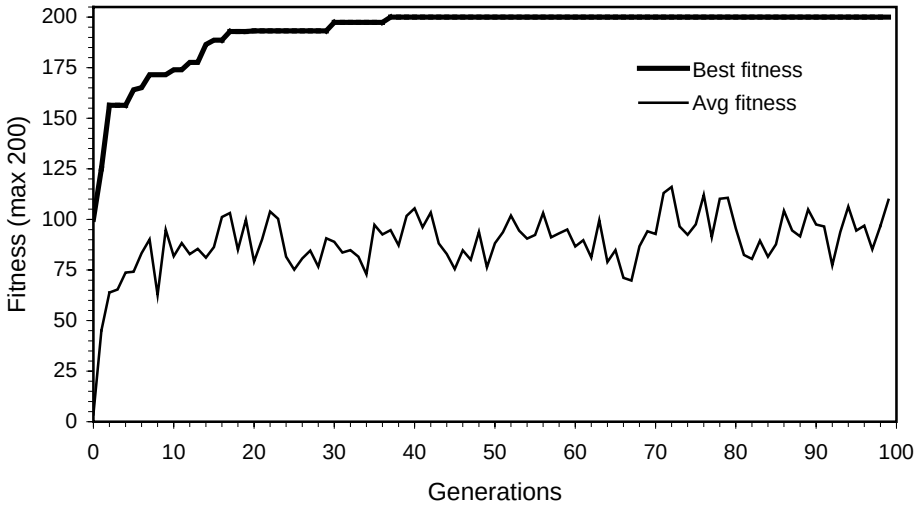


图 7.6.非同质化系统的进化动力学特征。该种群是在变异概率为 0.05 的情况下得到的。注意该平均适应度的振荡模式和最佳适应度与平均适应度之间存在很大的差异。

对于重组，该算子是所有 GEP 算子中性能较差的算子，仅经历重组的种群表现出同质的动力学。进一步，在前一节我们还看到 GEP 的三种重组（两点、单点和基因重组）在取 1.0 的最大概率时性能最好，而且两点重组是三种重组算子中最强大的算子，而基因重组是效果最差的。然而，对于本分析中的特定设置，当单独使用的时候，三种重组算子的效果非常差，所以必须将三个算子联合使用以使算法的效果能够有较小的提高（见表 7.2）。注意成功率与重组算子单独工作得到的单个的性能相比稍有提高。因此，对于本研究，我们将选择

表 7.2.经历变异（**Mut**）的非同质化系统与经历两点重组（**Rec2P**），单点重组（**Rec1P**），基因重组（**RecG**）及三种不同类型的重组（**RecMix**）的同质化系统的成功率和参数。

	Mut	**Rec2P**	**Rec1P**	**RecG**	**RecMix**
Number of runs	100	100	100	100	100
Number of generations	100	100	100	100	100
Population size	50	50	50	50	50
Number of fitness cases	10	10	10	10	10
Head length	6	6	6	6	6
Number of genes	6	6	6	6	6
Chromosome length	78	78	78	78	78
Mutation rate	0.05	--	--	--	--
Two-point recombination rate	--	1.0	--	--	0.8
One-point recombination rate	--	--	1.0	--	0.8
Gene recombination rate	--	--	--	1.0	0.8
Selection range	20%	20%	20%	20%	20%
Precision	0%	0%	0%	0%	0%
Success rate	96%	0.04%	0.03%	0.0%	0.13%

表 7.2 第 5 列的大致设定。尽管如此，该种群表现出与一次仅经历一种类型的重组的种群相同的同质动力学特征（图 7.7）。这进一步强化了关于重组是保守的假设，说明重组在保持现

状方面发挥了重要的作用。注意，在这个特定的情况下，在 54 代的时候平均适应度和最佳适应度的曲线重合，所有的个体从遗传的角度来说完全相同。如果所有的个体变得等价并且

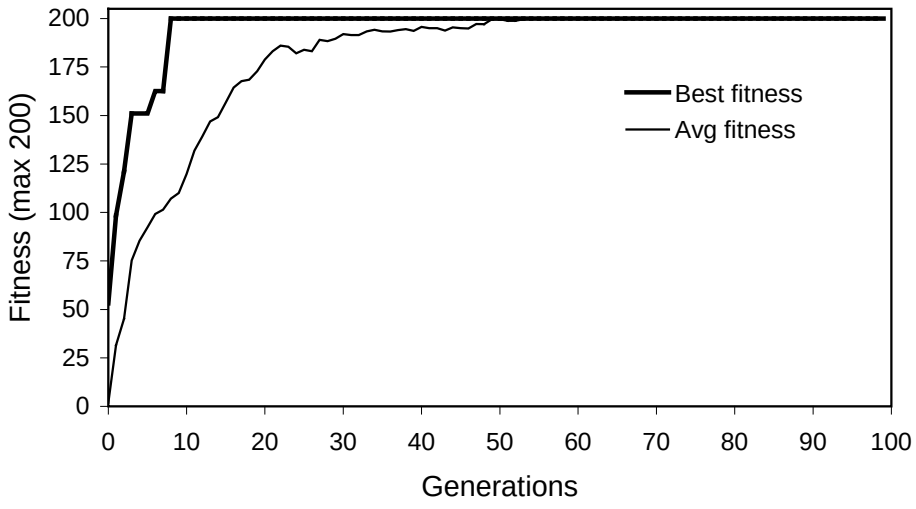


图 7.7. 经历大量重组的同质化种群。该动力学来源于表 7.2 第 5 列所示实验的一次成功运行。

完美的时候，我们可以把这看作一件好事。但是，请记住，在复杂的现实问题中，如同自然界中一样，不存在这样的完美，总可能有小的改进。当平均适应度在找到一个完美解或较好的解以前达到最佳适应度的时候，这样的进化策略的坏处就会变得比较明显。图 7.8 所示就是这种情况，其中种群停滞在一个普通的解上。这里，第 86 代以后，适应变得不可能，因为所有的个体有相同的遗传构成。的确，仅仅经历重组的种群的典型的较低成功率说明，在大多数情况下，同质种群会在找到较好的之前收敛，因为它们不可逆转的陷入一个局部点，而不一定是最优点。

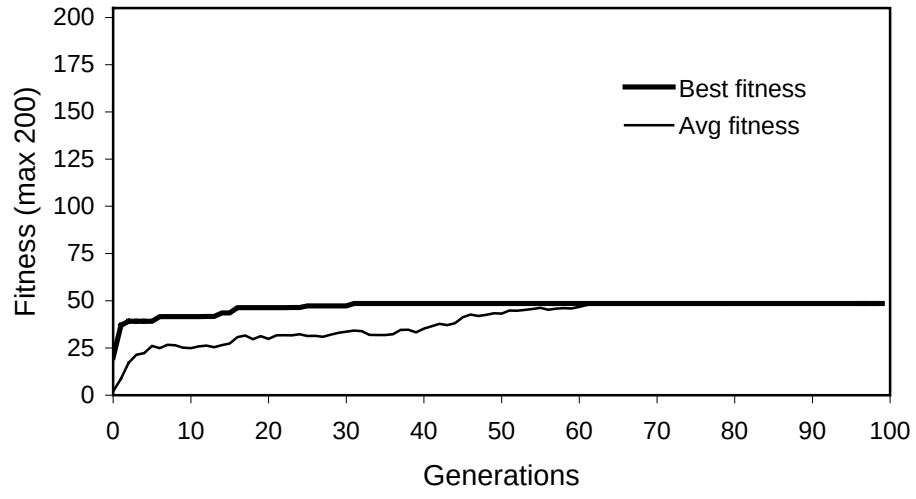


图 7.8.收敛到一个中等解的同质化种群，所使用的参数与图 7.7 完全一致。

值得注意的是，表 7.2 中给出的实验采用完全随机的初始种群，因此，那些初始种群中可存活的个体的数目不受控制。下一节，我将演示如何对初始种群中的可存活的个体的个数进行严格的控制来分析人工进化系统中的奠基效应。

7.2.2.分析模拟进化过程中的奠基效应

对于本分析，产生一个由一定数量可存活个体构成的小的，完全随机的“ 初始 ”种群（初始种群）。也就是说，只有在奠基种群中的所有成员都是可存活的或者，换言之，都具有正的适应度。这些奠基个体然后被选中并进行繁殖，留下后代的个数与实际种群大小 P 相当。

如图 7.9 所示，对于非同质种群而言，成功率与初始多样性之间不存在什么关联。的确，由于种群中能够持续引入遗传修饰，在非同质种群中，一段时间以后，奠基效应就被完全抹去，而种群能够如常地高效进化。

当把杂交作为唯一的遗传多样性来源时，种群会发生一种非常不同的情况，而且进化动力学的效果是同质的。在这些情况中，成功率与初始多样性之间存在很强的关联。注意，在重组之下种群的进化效果非常差，当奠基个体的数量为 2－5 个的时候实际上不具有适应能力（显然，对于只有一个奠基个体的时候，同质种群完全不具有适应能力）。值得强调的是，在这些系统中，甚至当奠基种群的大小等于 P 的时候，成功率仍明显比仅含有一个个体的奠基种群在经历变异的种群得到的成功率要低。

同样值得考虑的是为了保证产生大的奠基种群所需要的计算资源是非常昂贵的。因此，像 GEP 这样的可以在初始多样性最小的情况下高效地进化的系统是最有优势的。进一步，对于一些复杂的问题，例如，4.7 节中密度分类任务的细胞自动机规则发现问题，要随机产生一个可存活的个体，或者甚至是一个普通的个体，来让运行启动都是非常困难的。在这些情况下，像 GEP 这样的系统能够采用这个个体作奠基个体，并从这里继续下去，而仅仅依赖于重组的系统将它们聚集动能的时候停滞很长一段时间。的确，在 GEP 中，由于非同质遗传算子的种类多样，不需要巨大的奠基种群，因为只要能够随机产生一个可存活的个体，进化过程就能开始。

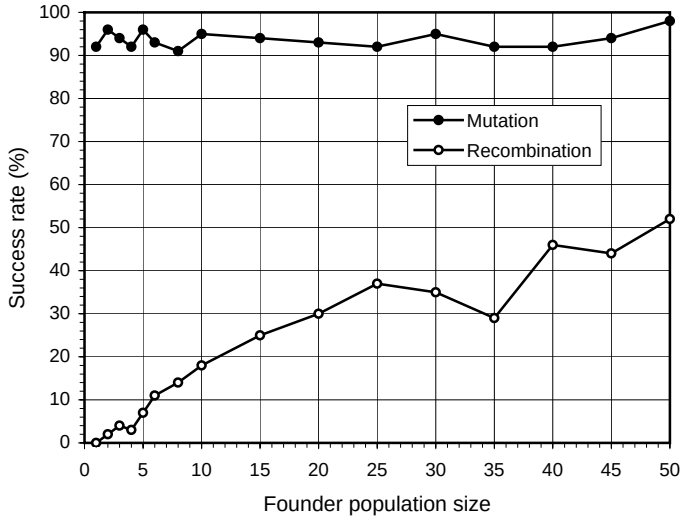


图 7.9.仅经历变异（变异概率为 0.05）的非同质化种群和仅经历重组（两点、单点、和基因重组的概率等于 0.8）的同质化种群中成功率对奠基种群大小的依赖。

7.3.验证基因块假设

前一节我们已经看到了基因块重组的种种局限性。当系统紧紧依赖于现存的基因块而且不能通过变异或者其它机制产生新的基因块的时候，这些基因块就变得极其受限。这里，我们将进一步追究这个问题，比较三种不同的系统：一种不断像种群中引入变化，另外两种只

能对某种特定的基因块，即 GEP 基因，进行重组。我们还记得 GEP 基因有定义的边界，而且通过基因重组和基因转座，有可能在不破坏这些基因块的情况下验证这些基因块的新的组合。

对于本分析我们将采用与前一节相同的序列推断问题，采用表 7.3 中给出的大致设置。对于第一个实验，我们将混合采用所有的遗传算子；对于第二个问题，我们将仅仅采用p_{gr}＝1.0 的基因重组；对最后一个实验，我们将通过融合基因重组（p_{gr}＝1.0）和基因转座（p_{gt}＝0.5）来允许基因块进行更加概化的换位。

表 7.3.经历多种遗传修饰的一个健康且强壮的系统（**All Op**）和另外的两个仅仅通过重组基因进化的系统（**GR** 和 **GR+GT**）。

	All Op	**GR**	**GR + GT**
Number of runs	100	100	100
Number of generations	50	50	50
Number of fitness cases	10	10	10
Function set	+ - * /	+ - * /	+ - * /
Head length	6	6	6
Number of genes	4	4	4
Linking function	+	+	+
Chromosome length	52	52	52
Mutation rate	0.0384	--	--
One-point recombination rate	0.3	--	--
Two-point recombination rate	0.3	--	--
Gene recombination rate	0.1	1.0	1.0
IS transposition rate	0.1	--	--
IS elements length	1,2,3	--	--
RIS transposition rate	0.1	--	--
RIS elements length	1,2,3	--	--
Gene transposition rate	0.1	--	0.5
Selection range	25%	25%	25%
Precision	0%	0%	0%

在本分析中，我们将研究成功率随种群规模的变化情况（图 7.10），而不像前一节那样生成一个奠基种群。如读者所看到的，该实验更加强调前一节得到的结果：不能向基因池中持续引入新的遗传物质的系统的进化效果很差。当只有基因块来回移动的时候，只有采用相对较大的种群时才有可能解决这个简单的问题，虽然其效率很低。请记住，现实问题要比我们这里分析的问题复杂得多，因此，像变异这样能力较强的算子必须一直使用。

总而言之，基因块的移动（Holland 1975）只有有限的进化影响力：没有变异（或者其它的非同质算子），适应过程非常缓慢而且需要很多的个体使得适应本身变得无效。

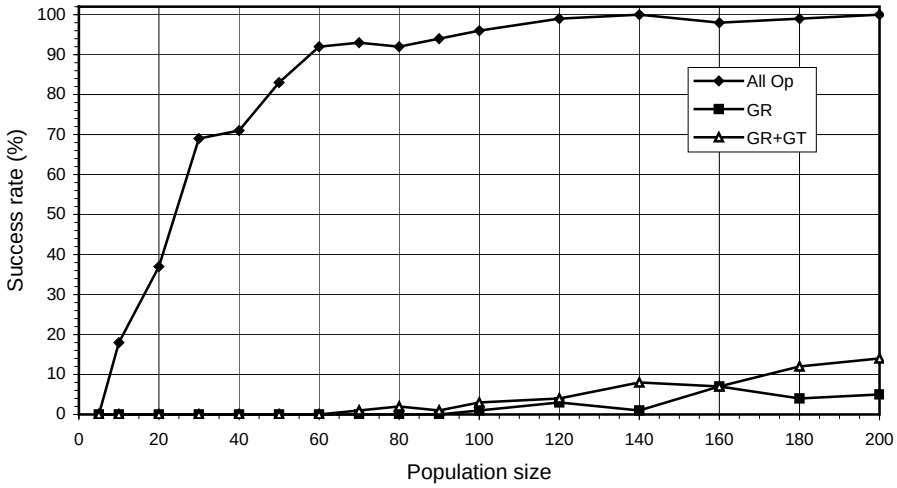


图 7.10.在多个算子混合作用下进化得到的健康且强壮的种群（**All Op**）和仅仅通过重组基因进化的到的种群（**GR:p_{gr}=1.0；和GR+Gt:p_{gr}=1.0 且p_{gt}=0.5**）中成功率对中群大小的依赖。成功率由 100 次相同的运行计算得到。

7.4.进化内中性的作用

我们已经知道计算机程序的自动进化只有在真正的基因型/表现型映射的情况下才能平滑、有效地进行。这种映射的产生需要一些创造性的想法，因为蛋白质和计算机程序时非常不同的两种东西。值得庆幸的是，计算机程序比蛋白质要容易理解，而且要创造一个能够进化计算机程序的简单的基因型/表现型系统，不需要知道诸如决定蛋白质的三维结构的规则之类的知识。

那么，DNA/蛋白质系统和特别用来进化计算机程序的人工系统之间本质的共同之处何在？显然，第一点是基因组和程序的产生；第二，不管如何，基因组总是必须产生合法的程序。那如何达到这个目的呢？

我们回到自然界去寻找灵感也许会有帮助。DNA/蛋白质系统如何处理复杂度？基因组中的信息是以某种片断形式存在的么？那么基因中基因组的片断也对简单的人工进化系统来说也会有用。那么对于自然界中的表达又是怎样的呢？编码在基因组中的信息全部都能表达吗？怎样才能将用来表达的信息和沉默的信息区分开来？为什么这种区分是重要的？这对人工进化系统有帮助么？虽然对于这些问题仍未找到答案，但是我们现在已知的是，在自然界中，基因组是十分冗余的，有很多很多所谓的垃圾 DNA 从来没有表达过：高度重复的序列，基因内区，伪基因，等等。所以，最可能的情况是，在人工的基因组中引入垃圾序列也可能是有用的。

GEP 中所采用的遗传表达不仅探索基因中基因组的片断，还包括基因组中垃圾序列和非编码区域的存在性。如 Kimura 假设的那样（Kimura 1983），中性变异的积累在进化中起到重要的作用。而 GEP 的非编码区域是这些中性变异积累理想场所。这一节，我们将分析基因组内中型区域的重要性，相应地，采用 GEP 这一功能完善的基因型/表现型系统进行进化时中性变异的重要性。

对于本分析，选择两个简单的，可精确求解的问题。这两个问题既可以采用单基因系统求解，也可以采用多基因系统求解。一方面，单基因系统中非编码区域的范围可以很容易地通过增加基因的长度来增加。另一方面，多基因系统中非编码区域的个数可以通过增加基因

的个数来增加。

本分析选择的第一个问题是 4.1 节用过的函数发现问题，即测试函数（4.1）第二个问题是一个较为困难的序列推断问题，即测试序列（4.9）（该序列在第 7.2 节和第 7.3 节中也曾使用过）。

对于函数发现问题，采用区间[-10,10]之间随机选择的 10 个适应度样本；适应度函数为方程（3.1b），采用的选择范围为 25%，精度误差为 0.01%。因此得到最大适应度f_{max}=250; 种群大小为P为 30 个个体，进化代数G为 50 代。

对于序列推断问题，前 10 个正整数n和相应的a_n作为适应度样本；适应度函数为方程（3.1b），采用的选择范围为 25%，精度为最大精度（0%误差）。因此得到最大适应度f_{max}=250; 种群大小和进化代数分别增加为 50 和 100，因为该问题比函数发现问题稍微困难一些。

在所有的实验中，F＝{+,-,*,/}，终点集仅由自变量 a 构成，得到 T={a}。遗传修饰的引入时采用的变异概率为 0.03，IS 和 RIS 转座概率为 0.1，转位子集的长度分别为 1，2，3，两点重组和单点重组概率为 0.3；多基因系统中还采用基因重组和基因转座作为遗传修饰的来源，其概率分别为 0.1，连接函数为加法；如往常一样，选择采用赌盘策略和简单精英策略，成功率由 100 次单独运行算得。

7.4.1.单基因系统中的遗传中性

单基因系统中的遗传中性的重要性可以很容易地通过增加 GEP 中的基因长度来进行分析（图 7.11）。在发现能够找到当前问题的完美的，最简洁的解所需要的最紧凑的组织结构以后，只要增加基因的长度都会通过进化得到完美的，不够简洁的解，而其中会相应地出现中性块和非编码区域。

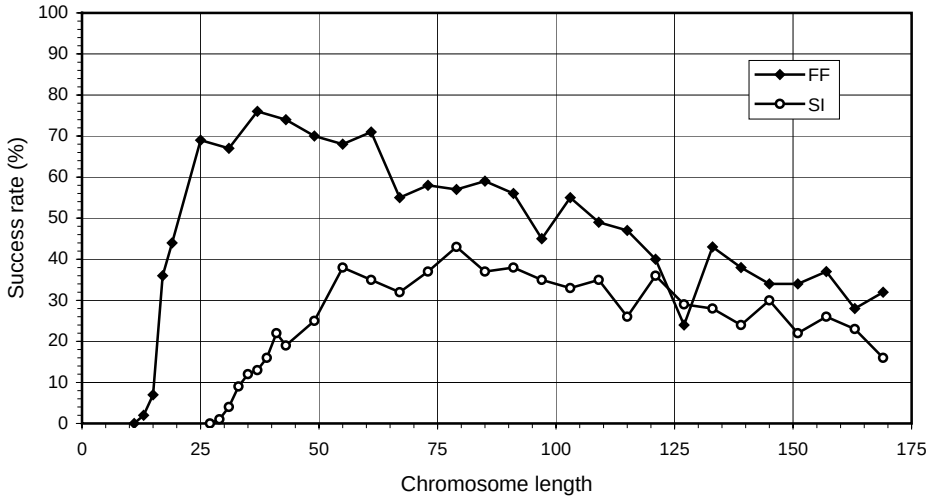


图 7.11.对于函数发现问题（FF）和序列推断问题（SI），染色体长度与成功率变化。

对于算法能够精确求解的问题，所需的最紧凑的组织结构在绝大多数情况下都能找到。如图 7.11 所示，测试函数（4.1）能够用头部长度为 6 的基因进行紧凑的编码（相应的 g=13）。下面是一个 13 个节点的这样的解。

```
0123456789012
*++*//aaaaaaa
```

注意，这里，基因中所有的元素都被表达出来，不存在非编码区域。

而测试函数 4.9，要使用选定的函数集得到正确并且简洁的表达式就需要更多的节点。如图 7.11 所示，h=14（相应的 g=29）是解决该问题所需的最小头部长度。下面两个解是完美解，同时也是简洁的解，它们分别采用 23 个和 25 个节点:

```
012345678901234567890123456 78
+*aa+*++**++/+aaaaaaaaaaaaaaa

012345678901234567890123456 78
**a+a*a++/+/+/aaaaaaaaaaaaaaa
```

注意，前一个基因有一个由 4 个元素构成的小的非编码区域，而第二个含有一个较大的非编码区域，它由 6 个元素构成。

如图 7.11 所强调的，最紧凑的组织结构并不是最高效的。例如，在函数发现问题中，最紧凑的组织结构（g=13）所得到的成功率仅为 2%，而最高的成功率 76%，是在染色体长度为 37 的时候得到的。在序列推断问题中也观察到了相同的情况，最紧凑的组织结构（g=29）所得到的成功率仅为 1%，而最佳染色体长度（g=79）所得到的成功率为 43%。因此，一定程度的冗余对于进化的高效进行是非常必要的。的确，在两个问题中，都发现了一个使系统进化效率最高的平台区域。还请注意，高度冗余的系统的适应性明显好于高度紧凑的系统，说明进化系统能很好的处理遗传冗余。例如，在函数发现实验中，冗余量最大的系统的染色体长度为 169，其成功率为 32%，明显高于 g=13 的最紧凑的组织结构所得到的 2%的成功率。序列推断实验中也观察到了相同的情况，冗余最大的结构（g=169）和最简洁的结构（g=29）所得到的成功率分别为 16%和 1%。

对这些紧凑的和不够紧凑的组织结构的分析也能帮助我们了解进化中冗余的作用。例如，下面这些函数（4.1）的完美解所采用的头部长度分别为 6，18 和 48（只给出 K-表达式）：

(a)
```
0123456789012
*++*//aaaaaaa
```

(b)
```
012345678901234567890123456
+-*+/+a*/**a*/a++-aaaaaaaaa
```

(c)
```
012345678901234567890123456789012345678 9...
/+aa++*+aa+*-a--+-*+*+/**a*a-*--+*/-/-/a...

...012345678901234567890123456789012345678 9
...aa//-/*-aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

注意第一个 h=6 的解用 13 个节点来表达，因此整个基因都用来进行表达；第二个 h=18 的解用 31 个节点来表达，因此存在一个 6 个元素的非编码区域；第一个 h=48 的解只使用 97 个元素中的 77 个来表达，因此存在一个 20 个元素的非编码区域。还要注意，从最紧凑的解到不够紧凑的解的变化，不仅非编码区域的长度增加，冗余或者中性主题也增加。例如，一共由 14 个节点构成的两个中性主题可以被看作使中等紧凑的解，由 38 个节点构成的 7 个中性主题则可以被看作使不紧凑的解。这种现象在 GP 中称为代码膨胀，许多人在争论它的进化功能（Angeline 1994,1996；Nordin 等 1995）。像所有类型的遗传冗余一样，只要能够使用适度，中性主题可能是其中最有益的。这个度能够用 GEP 严格地确定，虽然这样的分析需要花一些时间。然而，我们从图 7.11 的分析所学到的和我们所知道的关于蛋白质或者技术制品的进化都说明进化中中性区域具有重要的作用。没有它们或者过度使用它们都很有可能导致进化效率低下，而适度地使用它们则是十分有益的，就像我们这里所看到的一样。

7.4.2.多基因系统中的遗传中性

另一个分析 GEP 中遗传中性的方法是增加基因的个数。为此，我们选择一个能够精确解决当前问题的紧凑的单基因系统作为起点，因此，函数发现问题的起点是一个 h=6（基因长度等于 13）的单基因系统，对于序列推导问题 h=14（基因长度等于 29）（见图 7.11）。如图7.12所示，多基因系统得到的结果进一步强化了进化里面中性的重要性。要注意，在两个实验中，系统的效率随第二个基因的引入剧烈增加，而且紧凑的单基因比不够紧凑的多基因系统的效率要低很多。例如，在函数发现问题中，紧凑的单基因系统（染色体长度c等于13）的成功率为2%，而2-基因系统（c=26）的成功率为94%。在序列推断问题中，紧凑的单基因系统（c=29）的成功率仅为1%，而不够紧凑的2-基因系统（c=58）的成功率为73%。这里，我们再次发现对于效率最高的系统，也存在一个平台区域，说明一定程度的冗余对于进化的高效进行是非常必要的。的确，我们的直觉理解是对于某个实验来说一定的空间，包括通过生成新的基因块和抛弃现有的一些基因块，对于产生一些新的、有用的东西是非常必需的。如果没有足够的空间，那么要得到（如果能够的话）当前问题的一个好的解就需要很大的代价。同样，请注意高度冗余的系统的的效率明显好于高度紧凑的系统。例如，例如，在函数发现问题中使用的10-基因系统（c=130）的成功率为61%，而最紧凑的组织结构得到的成功率仅为2%。在序列推断问题中也有同样的情况出现，染色体长度为290的高度冗余的10-基因系统的性能比最紧凑的系统稍好（c=29）（成功率分别为1%和11%）。

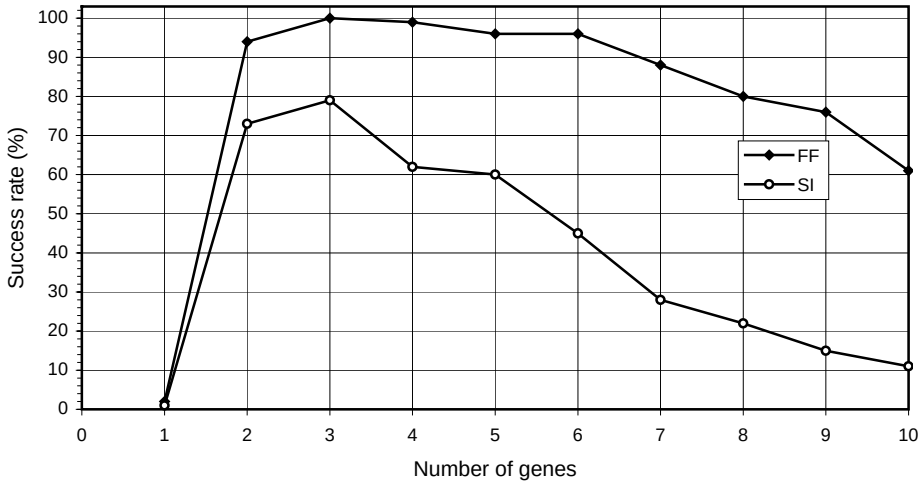


图7.12.对于函数发现问题（FF）和序列推断问题（SI），基因个数与成功率变化。

图7.11和7.12的比较还说明多基因系统明显优于单基因系统（见7.5节）。例如，在函数发现问题中，从2-基因系统到6-基因系统（相应的染色体长度从26到78），成功率一直高于94%，而且最好的系统的成功率为100%（见图7.12），而单基因系统中最佳染色体长度（c=37，h=18）的成功率仅为76%（见图7.11）。在序列推断问题中也有同样的情况出现，从2-基因系统到5-基因系统（相应的染色体长度从58到145）成功率一直高于60%，而且最好的系统的成功率为79%（见图7.12），而单基因系统中最佳染色体长度（c=77，h=39）的成功率仅为43%（见图7.11）。

对这些紧凑的和不够紧凑的组织结构的分析也能帮助我们了解进化中冗余的作用。例如，下面得到的这些序列推断问题的完美解，所对应的系统分别包含1，3，5个基因：

(a) `01234567890123456789012345678`

```
**a+a*a++/+/+/aaaaaaaaaaaaaaa
```

(b)
```
012345678901234567890123456 78
**a+/*+a//++//aaaaaaaaaaaaaaa
/*a+/*+a//++//aaaaaaaaaaaaaaa
-*a+/-*a//++aaaaaaaaaaaaaaaaa
```

(c)
```
012345678901234567890123456 78
-aa+*///-+aa/*aaaaaaaaaaaaaaa
--/*a/+/**/-/-aaaaaaaaaaaaaaa
+/-*-*a-/a+*a-aaaaaaaaaaaaaaa
aa/a/*a+*+/+/+aaaaaaaaaaaaaaa
/-/*aa++**/+/+aaaaaaaaaaaaaaa
```

还要注意，从最紧凑的解到不够紧凑的解的变化，不仅非编码区域的总长度增加，中性主题的个数也增加。特别的是，单基因解中非编码区域的长度仅为6，没有中性主题；而对于3-基因的解，非编码区域的总长度为16，三个中性主题一共涉及12个节点；而对于5-基因的解，非编码区域的总长度为66，三个中性主题一共涉及31个节点。

鉴于其清晰性，这一节所给出的结果对于理解人工和自然进化中的遗传中性非常有用。如这里所显示的，GEP中有两种不同类型的中性区域：ORF内部的中性主题和ORF尾部的非编码区域。在像GP和GAs这样的简单复制体系统中，只存在前一种类型，而在像DNA/蛋白质系统或者GEP这样的基因型/表现型系统中，这两种类型的都存在。基因型/表现型系统中非编码区域的存在肯定与这些系统的高效率密切相关。例如，DNA中的基因内区被认为是杂交的绝佳目标，它允许在不破坏这些不同基因块的情况下对它们进行重组（例如，Maynard Smith和 Szathmáry 1995）。GEP中的非编码区域也可以作为这种用途，的确，只要杂交位置选在这些区域以内，整个ORF就会发生变化。进一步，GEP基因的非编码区域也是中性变异累计的理想场所，这些中性变异在稍后将被激活并整合到编码区域中去。中性变异是一种基因变化的绝佳来源而且肯定会对冗余系统的效率提高有一定的贡献。

但是，至少在GEP中，非编码区域还起到另外一种更加重要的作用：它们允许大量的高性能遗传算子对基因组进行修饰。这里我采用“高性能”是指那些总是产生合法结构的遗传算子。这个合法结构的问题仅仅适用于金花系统，因为在自然界中没有非法蛋白质这样的事情存在。我们迄今为止仍然不知道DNA和蛋白质如何以及为什么会这样，但是肯定存在一种选择压力来排除这些有残缺的基因型/表现型映射。GEP的非编码区域允许产生完美的基因型/表现型映射的这一事实进一步强化了进化内中性的重要性，因为一种好的映射关系对于跨越表现型阈值至关重要。

另一方面，结构中的中性主题，不管它们是分列树还是蛋白质，能够促进进化的原因很难理解，尽管我个人认为这是小的基因块的重组和验证这两种相同现象的另一种表现形式。在此，小的基因块并不是有明确边界的整个基因，而是基因内部的一些小的域。的确，在自然界中，大多数蛋白质都有很多的变型，其中会产生各种不同的蛋白质替换。这些蛋白质的替换大部分都出现在酶的激活位置这样的蛋白质的关键域之外，因此，蛋白质的变体效果相当，而且在功能上差别很小。在分子层，这些变体构成了实际的遗传多样性，即进化的原始物质。GEP的中性主题恰好起到完全相同的作用，允许不同的基因块重组和验证，而且同时，允许产生中性变体，这些中性变体最后会分布到适应性更好的结构中或者产生适应性更好的结构。

7.5.多级因系统的更高级别的组织结构

GEP 唯一一种把基因作为独立的实体连接起来构成一个更加复杂的结构，即染色体，的遗传算法。前面一节的分析很明显的说明多基因系统远远优于单基因系统。这里我们将对染色体长度相同的单基因系统和多基因系统进行一个更加系统的对比分析。本分析所选用的问题及其参数设定与前一节完全相同（见表7.4和7.5）。

表7.4.在函数发现问题上对单基因和多基因系统的性能进行比较

	1G	3G	5G
Number of runs	100	100	100
Number of generations	50	50	50
Population size	30	30	30
Number of fitness cases	10	10	10
Function set	+ - * /	+ - * /	+ - * /
Head length	37	12	7
Number of genes	1	3	5
Linking function	--	+	+
Chromosome length	75	75	75
Mutation rate	0.03	0.03	0.03
One-point recombination rate	0.3	0.3	0.3
Two-point recombination rate	0.3	0.3	0.3
Gene recombination rate	--	0.1	0.1
IS transposition rate	0.1	0.1	0.1
IS elements length	1,2,3	1,2,3	1,2,3
RIS transposition rate	0.1	0.1	0.1
RIS elements length	1,2,3	1,2,3	1,2,3
Gene transposition rate	--	0.1	0.1
Selection range	25%	25%	25%
Precision	0.01%	0.01%	0.01%
Success rate	58%	93%	98%

表7.5.在函数发现问题上对单基因和多基因系统的性能进行比较

	1G	3G	5G
Number of runs	100	100	100
Number of generations	100	100	100
Population size	50	50	50
Number of fitness cases	10	10	10
Function set	+ - * /	+ - * /	+ - * /
Head length	37	12	7
Number of genes	1	3	5
Linking function	--	+	+
Chromosome length	75	75	75
Mutation rate	0.03	0.03	0.03

One-point recombination rate	0.3	0.3	0.3
Two-point recombination rate	0.3	0.3	0.3
Gene recombination rate	--	0.1	0.1
IS transposition rate	0.1	0.1	0.1
IS elements length	1,2,3	1,2,3	1,2,3
RIS transposition rate	0.1	0.1	0.1
RIS elements length	1,2,3	1,2,3	1,2,3
Gene transposition rate	--	0.1	0.1
Selection range	25%	25%	25%
Precision	0%	0%	0%
Success rate	41%	79%	96%

对于本分析，采用对于三种不同的染色体组织结构我们选择的染色体长度均为75：单基因染色体的h=37；3-基因染色体的长度为h=12；5-基因染色体的h=7。这些系统的性能以成功率的形式来度量，详见表7.4和7.5。

如我们所料，多基因系统效率明显高于单基因系统，因此应该成为我们的首选。尽管这样可能会有问题，因为基因中染色体的分块没有什么好处。例如，当我们试图进化一个问题的解来对一个复杂表达式的一个平方根函数进行建模的时候，多分子系统就不是我们的最佳选择。但是，即使是在这些情况下，系统那仍然能够有办法将不必要的基因转化成中心的基因，那么仍然有可能出现高效的适应过程。但是当然，在多分子系统中，各种函数的建模显然会从多基因中受益，因为这些系统不会限于某一特定的连接函数。

7.6.GEP种群的自由进化

我们已经看到GEP的种群能够用来进行高效的进化，因为遗传算子能够一直向基因池中引入新的物质。这里我们将进一步讨论这个问题，分析成功率如何随进化代数发生变化。

在这一节，将再次使用4.2.2节的序列推断问题，因为它是一个相当有难度的问题，而且有精确的解。

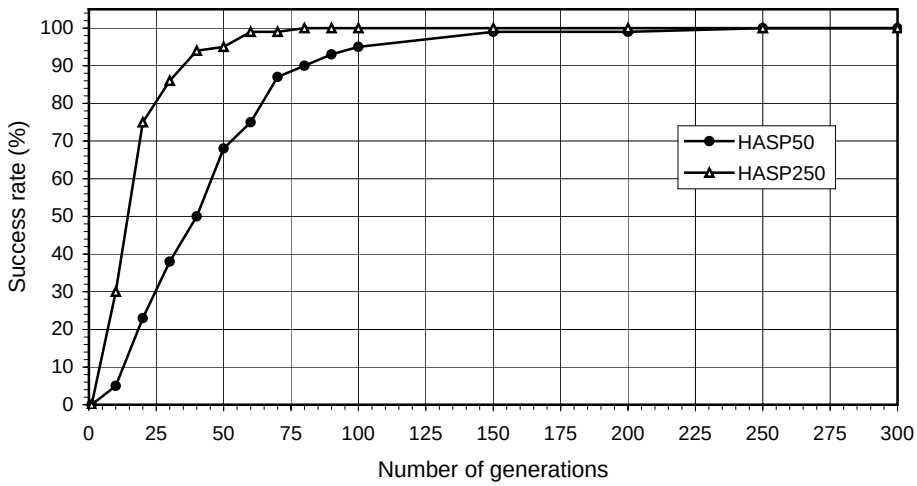


图 7.13.分别由 50 个个体构成（**HASP50**）和 250 个个体构成（**HASP250**）的健康且强壮的种群的成功率随进化时间变化。成功率由 100 次相同的运行计算得到。

在第一个分析中，成功率对进化代数的依赖在健康且强壮的种群上进行分析，种群大小分别为50和250（图7.13）在本实验中，使用所有的遗传算子。其各自的比例见表7.6的第一列。

表7.6.健康且强壮的种群和同质化种群中所使用的遗传变化的来源

	HASP	HP
Number of runs	100	100
Number of fitness cases	10	10
Function set	+ - * /	+ - * /
Head length	7	7
Number of genes	5	5
Linking function	+	+
Chromosome length	75	75
Mutation rate	0.044	--
One-point recombination rate	0.3	1.0
Two-point recombination rate	0.3	1.0
Gene recombination rate	0.1	1.0
IS transposition rate	0.1	--
IS elements length	1,2,3	--
RIS transposition rate	0.1	--
RIS elements length	1,2,3	--
Gene transposition rate	0.1	--
Selection range	25%	25%
Precision	0%	0%

注意当进化倒数达到150代的时候成功率均达到最大值。如我们所料，当性能进京以成功率来计的时候，较大的种群占优势。然而，如果我们比较每个系统所需的CPU时间，我们将发现，50个染色体的系统更加有效。例如，对于50个染色体的系统，每次运行150代，100次运行需要1'30"，对于250个个体的运行需要3'55"。然而如果我们不使用停机准则（即系统在找到完美解之前不会停止）而让系统运行完150代，种群大小为50 时运行时间为4'14"，种群大小为250时运行时间为29'42"。这并不是无意义的比较，因为显示世界的问题不一定总是存在完美解（即误差为0%或0.01%的解），因此，系统在完成规定的代数以前不会停止。

总之，在GEP中，只要存在变异，采用大小为30-100的较小种群就是非常有优势的，因为这已经能够保证系统在有限的时间里进行高效的进化。

这些事实进一步被采用重组作为遗传多样性的第二个实验所证明（图7.14）。所采用的重组的类型和它们的相应比例将表7.6的第二列。

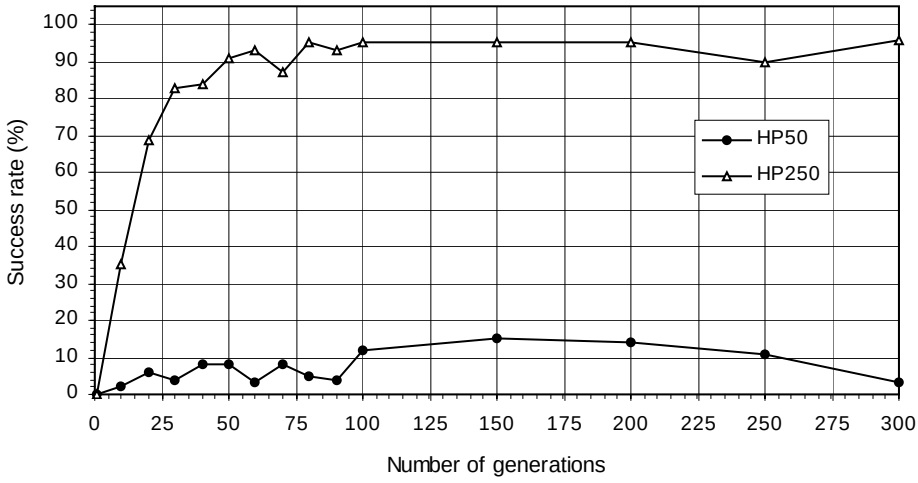


图7.14.分别由50个个体构成（**HP50**）和250个个体构成（**HP250**）的同质化种群成功率随进化时间变化。成功率由100次相同的运行计算得到。

该分析明确说明，通过变异和类似的同质化算子如RIS和IS转座不断引入遗传变化的重要性。请注意，在这种情况下，50个个体的种群进化效率非常低。宿命论者经常说的GP在50代之后不能进化评论就完全能够理解了。如图7.14所强调的，同质系统在50代以后成功率没有明显的增加。请记住，GP几乎仅使用针对GP的重组操作作为遗传变化的唯一来源。这里我们可以看到即使是比GP中作使用的重组更具有破坏性的重组机制（即单点和两点重组）也不足以使系统高效进化。这显然是GP优于GEP的一方面的原因，当然，最重要的还是基因型/表现型表达方式以及这种表达方式所带来的一切。250个个体种群的曲线表明在这些系统中只有使用巨大的种群规模才能使系统高效进化。然而，请注意，即使使用如此规模巨大的种群，要超过使用变异和转座算子的仅有50个个体的种群所得到的结果也是不可能的（见图7.13）。

7.7不同选择策略的分析

众所周知，选择的结果要经历一段时间才能看得出来。因此，我们将用两个问题来通过计算成功率随进化代数的变化来分析其性能。

我所选择的用来研究的选择策略包括本书中所有问题中所使用的赌盘选择，两名选手的锦标赛选择和一种确定性选择策略。对于所有的策略，甚至使确定性选择，为了允许在不丧失最佳特性的情况下采用一定程度的遗传修饰，我们还是进行最佳个体复制。

有精英策略的锦标赛选择工作方式如下：随机选取两个个体，选择较好的一个复制一次，如果这两个个体恰好有同样的适应度，那么从中随机选择一个个体进行复制。因此，每一代，对于N个个体的种群，进行N次锦标赛选择，从而使种群规模保持不变。

在确定性选择策略中，个体按照其适应度进行比例选择，适应度较差的个体被排斥在生命的盛宴之外。你可以指定排斥在盛宴之外的个体的数目的多少，我个人采用不同的排斥程度验证了几种确定性选择策略。在大多数情况下，E为1.5的排斥因子是最佳的折中。

为了应用这个排斥因子，每个个体的可存活性（适应度值除以平均适应度）乘以E。设想一个圆桌，每个盘子被排名和可存活性的比例逐个占据，从最佳个体开始，直到桌子上的位置全部占满。排斥因子越高，死亡时不留下后代的个体的数目越多，只有种群中的精华才

能繁殖。显然，我们必须小心选择排斥的程度，否则遗传多样性将急剧减少，进化严重地受限。

这里研究所选用的问题是两个序列推断的检验问题。第一个是4.2.2节的序列(4.9),第二个是复杂一些的序列（7.1）.两个实验的大致设置见表7.7所示。

表7.7.用来比较不同选择策略色序列推断实验的大致设置

	SI1	**SI2**
Number of runs	100	100
Population size	50	50
Number of fitness cases	10	10
Function set	+ - * /	+ - * /
Head length	7	6
Number of genes	5	7
Linking function	+	+
Chromosome length	75	91
Mutation rate	0.044	0.044
One-point recombination rate	0.3	0.3
Two-point recombination rate	0.3	0.3
Gene recombination rate	0.1	0.1
IS transposition rate	0.1	0.1
IS elements length	1,2,3	1,2,3
RIS transposition rate	0.1	0.1
RIS elements length	1,2,3	1,2,3
Gene transposition rate	0.1	0.1
Selection range	25%	25%
Precision	0%	0%

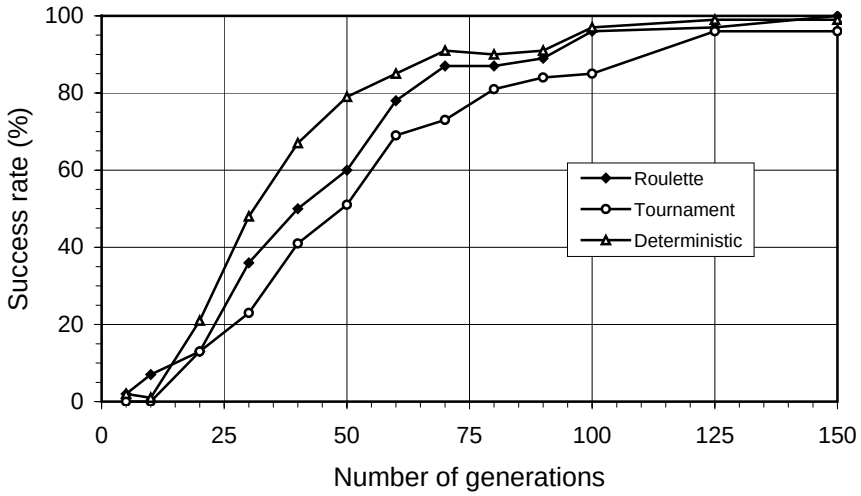


图7.15.赌盘、锦标赛、确定性选择在更简单的测试序列上的比较。

图7.15和图7.16对很长时间间隔上的三种选择策略的成功率变化进行了比较。在两个实验中，锦标赛选择策略比赌盘选择和确定性选择稍差一点。要在赌盘策略和确定性选择之间做出选择是一个非常困难的问题，因为对于更加复杂的问题两种策略的曲线会更加纠缠在一起（图7.16）。

我们选择赌盘策略有几个理由。(1) 现实世界的问题比这里分析的问题要复杂很多。(2) 理想的排斥因子依赖于种群的大小和问题的复杂程度。(3) 确定性选择需要更多的CPU时间，因为必需对个体排序，对于大的种群，这是一个需要慎重考虑的问题。(4) 确定性选择不适合仅仅使用重组的系统，因为它会显著降低种群中的遗传多样性。(5) 赌盘选择易于实现，而且更忠实地模拟自然因而更有吸引力。

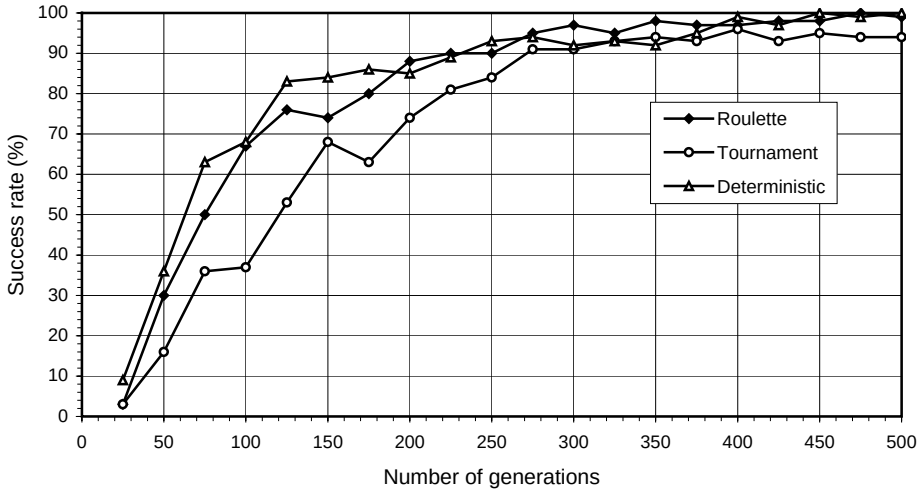


图7.16.赌盘、锦标赛、确定性选择在更困难的测试序列上的比较。

本书在这里就算完成了，我希望向读者展示了GEP不仅易于实现而且易于理解。虽然自然进化看来超出我们的控制，为了揭示GEP这个简单的人工系统的所有秘密，我们可以对它进行细致入微的分解。

最后，我希望已经说服大家GEP可以帮助读者找到现实世界中困难问题的好的解，而这些问题采用传统的数学方法或者统计方法可能很难得到满意的解答。

Bibliography

Angeline, P. J., 1994. Genetic Programming and Emergent Intelligence. In K. E. Kinnear Jr., ed., *Advances in Genetic Programming*, chapter 4, pages 75-98, MIT Press, Cambridge, MA.

Angeline, P. J., 1996. Two Self-adaptive Crossover Operators for Genetic Programming. In P. J. Angeline and K. E. Kinnear, eds., *Advances in Genetic Programming 2*, chapter 5, pages 89-110, MIT Press, Cambridge, MA.

Banzhaf, W., 1994. Genotype-Phenotype-Mapping and Neutral Variation: A Case Study in Genetic Programming. In Y. Davidor, H.-P. Schwefel, and R. Männer, eds., *Parallel Problem Solving from Nature III*, *Lecture Notes in Computer Science* 866: 322-332, Springer-Verlag.

Banzhaf, W., P. Nordin, R. E. Keller, and F. D. Francone, 1998. *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and its Applications*, Morgan Kaufmann.

Bonachea, D., E. Ingerman, J. Levy, and S. McPeak, 2000. An Improved Adaptive Multi-Start Approach to Finding Near-Optimal Solutions to the Euclidean TSP. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, eds., *Proceedings of the Genetic and Evolutionary Computation Conference*, 143-150, Las Vegas, Nevada, Morgan Kaufmann.

Cramer, N. L., 1985. A Representation for the Adaptive Generation of Simple Sequential Programs. In J. J. Grefenstette, ed., *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, 183-187, Erlbaum.

Das, R., M. Mitchell, and J. P. Crutchfield, 1994. A Genetic Algorithm Discovers Particle-based Computation in Cellular Automata. In Y. Davidor, H.-P. Schwefel, and R. Männer, eds., *Parallel Problem Solving from Nature III*, Springer-Verlag.

Dawkins, R., 1995. *River out of Eden*, Weidenfeld and Nicolson.

Eldredge, N. and S. J. Gould, 1972. Punctuated Equilibria: An Alternative to Phyletic Gradualism. In T. J. M. Schopf, ed., *Models in Paleobiology*, Freeman.

Ferreira, C., 2001. Gene Expression Programming: A New Adaptive Algorithm for Solving Problems. *Complex Systems* 13 (2): 87-129.

Ferreira, C., 2002a. Mutation, Transposition, and Recombination: An Analysis of the Evolutionary Dynamics. In H. J. Caulfield, S.-H. Chen, H.-D. Cheng, R. Duro, V. Honavar, E. E. Kerre, M. Lu, M. G. Romay, T. K. Shih, D. Ventura, P. P. Wang, Y. Yang, eds., *Proceedings of the 6th Joint Conference on Information Sciences, 4th International Workshop on Frontiers in Evolutionary Algorithms*, pages 614-617, Research Triangle Park, North Carolina, USA.

Ferreira, C., 2002b. Combinatorial Optimization by Gene Expression Programming: Inversion Revisited. In J. M. Santos and A. Zapico, eds., *Proceedings of the Argentine Symposium on Artificial Intelligence*, pages 160-174, Santa Fe, Argentina.

Fisher, R. A., 1936. The Use of Multiple Measurements in Taxonomic Problems. *Annual Eugenics* 7 (2): 179-188. (Reprinted in *Contributions to Mathematical Statistics*, 1950. New York, John Wiley.)

Futuyma, D. J., 1998. *Evolutionary Biology*, 3rd ed., Sunderland, MA: Sinauer Associates.

Goldberg, D. E., 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley.

Haupt, R. L. and S. E. Haupt, 1998. *Practical Genetic Algorithms*, Wiley-Interscience.

Holland, J. H., 1975. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, University of Michigan Press (second edition: MIT Press, 1992).

Hsu, W. W. and C.-C. Hsu, 2001. The Spontaneous Evolution Genetic Algorithm for Solving the Traveling Salesman Problem. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, eds., *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 359-366, San Francisco, California, Morgan Kaufmann.

Iba, H. and T. Sato, 1992. Meta-level Strategy for Genetic Algorithms Based on Structured Representations. In *Proceedings of the Second Pacific Rim International Conference on Artificial Intelligence*, 548-554.

Iba, H., T. Sato, and H. de Garis, 1994. System Identification Approach to Genetic Programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation*, Vol. I: 401-406, Piscataway, NJ, IEEE Press.

Ivakhnenko, A. G., 1971. Polynomial Theory of Complex Systems. *IEEE Transactions on Systems, Man, and Cybernetics* 1 (4): 364-378.

Johnson, D. S. and L. A. McGeoch, 1997. The Traveling Salesman Problem: A Case Study. In E. H. L. Aarts and J. K. Lenstra, eds., *Local Search in Combinatorial Optimization*, 215-310, Wiley & Sons, New York.

Juillé, H. and J. B. Pollack, 1998. Coevolving the “Ideal” Trainer: Application to the Discovery of Cellular Automata Rules. In J. R. Koza, W. Banzhaf, K. Chellapilla, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. L. Riolo, eds., *Genetic Programming 1998: Proceedings of the Third Annual Conference*, Morgan Kaufmann, San Francisco, CA.

Kargupta, H. and R. E. Smith, 1991. System Identification with Evolving Polynomial Networks. In R. K. Belew and L. B. Booker, eds., *Proceedings of the Fourth International Conference on Genetic Algorithms*, 370-376, San Mateo, California, Morgan Kaufmann.

Katayama, K. and H. Narihisa, 1999. Iterated Local Search Approach Using Gene Transformation to the Traveling Salesman Problem. In W. Banzhaf, ed., *Proceedings of the Genetic and Evolutionary Computation Conference*, 321-328, Morgan Kaufmann.

Kimura, M., 1983. *The Neutral Theory of Molecular Evolution*, Cambridge University Press, Cambridge, UK.

Keith, M. J. and M. C. Martin, 1994. Genetic Programming in C++: Implementation Issues. In K. E. Kinnear, ed., *Advances in Genetic Programming*, MIT Press.

Keller, R. E. and W. Banzhaf, 1996. Genetic Programming Using Genotype-Phenotype Mapping from Linear Genomes into Linear Phenotypes. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, eds., *Genetic Programming 1996: Proceedings of the First Annual Conference*, MIT Press.

Koza, J. R., F. H. Bennett III, D. Andre, and M. A. Keane, 1999. *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco: Morgan Kaufmann.

Koza, J. R., 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, MA: MIT Press.

Lynch, M. and J. S. Conery, 2000. The Evolutionary Fate and Consequences of Duplicated Genes, *Science* 290: 1151-1155.

Margulis, L., 1970. *Origin of Eukaryotic Cells*, Yale University Press.

Margulis, L. and D. Sagan, 1986. *Origins of Sex: Three Billion Years of Recombination*, Yale University Press.

Margulis, L. and D. Sagan, 1997. *What is Sex?*, Simon and Schuster, New York.

Mathews, C. K., K. E. van Holde, and K. G. Ahern, 2000. *Biochemistry*, 3rd ed., Benjamin/Cummings.

Maynard Smith, J. and E. Szathmáry, 1995. *The Major Transitions in Evolution*, W. H. Freeman.

Mayr, E., 1954. Change of Genetic Environment and Evolution. In J. Huxley, A. C. Hardy, and E. B. Ford, eds., *Evolution as a Process*, 157-180, Allen and Unwin, London.

Mayr, E., 1963. *Animal Species and Evolution*, Harvard University Press, Cambridge, Massachusetts.

Merz, P. and B. Freisleben, 1997. Genetic Local Search for the TSP: New Results. In T. Bäck, Z. Michalewicz, and X. Yao, eds., *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation*, 159-164, Piscataway, NJ, IEEE Press.

Mitchell, M., 1996. *An Introduction to Genetic Algorithms*, MIT Press.

Mitchell, M., P. T. Hraber, and J. P. Crutchfield, 1993. Revisiting the Edge of Chaos: Evolving Cellular Automata to Perform Computations. *Complex Systems* 7: 89-130.

Mitchell, M., J. P. Crutchfield, and P. T. Hraber, 1994. Evolving Cellular Automata to Perform Computations: Mechanisms and Impediments. *Physica D* 75: 361-391.

Nikolaev, N. I. and H. Iba, 2001. Accelerated Genetic Programming of Polynomials. *Genetic Programming and Evolvable Machines* 2: 231-257.

Nordin, P., F. Francone, and W. Banzhaf, 1995. Explicitly Defined Introns and Destructive Crossover in Genetic Programming. In J. P. Rosca, ed., *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 6-22, Tahoe City, California.

Papadimitriou, C. H. and K. Steiglitz, 1982. *Combinatorial Optimization*, Prentice Hall.

Prechelt, L., 1994. PROBEN1 – A Set of Neural Network Benchmark Problems and Benchmarking Rules. *Technical Report* 21/94, Karlsruhe University, Germany.

Reinelt, G., 1994. The Traveling Salesman: Computational Solutions for TSP Applications. *Lecture Notes in Computer Science* 496: 188-192, Springer-Verlag, Berlin, Germany.

Ryan, C., J. J. Collins, and M. O'Neill, 1998. Grammatical Evolution: Evolving Programs for an Arbitrary Language. In *Proceedings of the First European Workshop on Genetic Programming*, *Lecture Notes in Computer Science* 1391: 83-95, Springer-Verlag.

Schulze-Kremer, S., 1992. Genetic Algorithms for Protein Tertiary Structure Prediction. In R. Männer and B. Manderick, eds., *Parallel Problem Solving from Nature II*, North-Holland.

Tank, D. W. and J. J. Hopfield, 1987. Collective Computation in Neuronlike Circuits. *Scientific American* 257 (6): 104-114.

Toffoli, T. and N. Margolus, 1987. *Cellular Automata Machines: A New Environment for Modeling*, MIT Press.

Whitley, L. D. and J. D. Schaffer, eds., 1992. *COGANN-92: International Workshop on Combinations of Genetic Algorithms and Neural Networks*. IEEE Computer Society Press.

Wolfram, S., 1986. *Theory and Applications of Cellular Automata*, World Scientific.