

世界著名计算机教材精选

计算理论基础

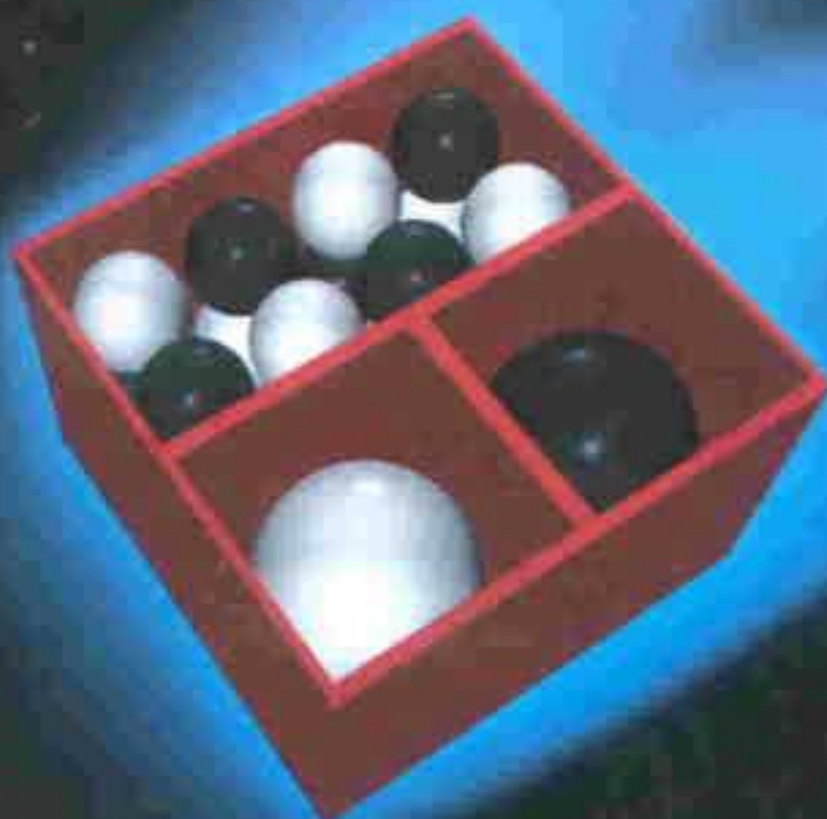
第2版

ELEMENTS OF THE THEORY OF COMPUTATION

SECOND EDITION

Harry R. Lewis, Christos H. Papadimitriou 著

张立昂 刘 田 译



清华大学出版社
<http://www.tup.tsinghua.edu.cn>



PRENTICE HALL
<http://www.prenticehall.com>

制作信息和申明

图书名称：计算理论基础（第二版）

作者：Harry R.Lewis Christos H.Papadimitriou

译者：张立昂 刘田

出版：清华大学出版社

制作版本：1.1

制作时间：2001 年 8 月 15 日

制作形式：pdf 文件

制作声明：

本电子文档及相关文档，只为本人私人阅读方便而制作，以及极小范围作教学交流使用，不供网路传播。如果不慎流传，并有人通过某些特殊途径偶然获取到本文档，为尊重知识产权，特此敬告：版权读物，请勿在网路传播！

否则，凡有未经通知本人、未经同意许可，私自传播者，一概跟本人无关，自负其责，特此申明。

个人联系方式：BookFTP.bbs@bbs.nju.edu.cn or BookFTP@21cn.com

世界著名计算机教材精选

计算理论基础

(第2版)

Harry R. Lewis
Christos H. Papadimitriou 著

张立昂 刘 田 译

清华大学出版社

Prentice Hall

(京)新登字 158 号

内 容 简 介

计算理论是计算机科学的理论基础。本书介绍了计算理论最核心、最基本的内容,包括形式语言与自动机、可计算性和计算复杂性三大部分。全书共分七章,分别为:集合、关系和语言;有穷自动机;上下文无关语言;Turing 机;不可判定性;计算复杂性;NP 完全性。本书突出了算法,从而使计算机专业的学生更易接受,也更有收益。

本书适合作为计算机专业及数学专业本科生或研究生的教材,也可供从事计算机科学的教学与研究人员参考。

Elements of the Theory of Computation(Second Edition)

Harry R. Lewis, Christos H. Papadimitriou

Copyright ©1998 by Prentice Hall, Inc.

Original English Language Edition Published by Prentice Hall, Inc.

All Rights Reserved.

本书中文简体字版由 Prentice Hall 出版公司授权清华大学出版社独家出版、发行。

未经出版者书面许可,不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有清华大学出版社激光防伪标签,无标签者不得销售。

北京市版权局著作权合同登记号: 图字 981320 号

书 名: 计算理论基础(第 2 版)

作 者: Harry R. Lewis Christos H. Papadimitriou 著

出版者: 清华大学出版社(北京清华大学学研楼,邮编 100084)

<http://www.tup.tsinghua.edu.cn>

印刷者: 北京昌平环球印刷厂

发行者: 新华书店总店北京发行所

开 本: 787×1092 1/16 **印张:** 16 **字数:** 367 千字

版 次: 2000 年 7 月第 1 版 2000 年 7 月第 1 次印刷

书 号: ISBN 7-302-03948-8/TP·2310

印 数: 0001~6000

定 价: 29.00 元

译者序

半年前我先后接到清华大学出版社第四编辑室主任薛慧同志和机械工业出版社华章图文信息有限公司总编辑傅豫波同志的电话,前者请我翻译这本书,后者请我翻译 Sipser 的 Introduction to the Theory of Computation (Second edition),我都欣然接受,并且邀请王捍贫、刘田和黄雄三位博士共同翻译。其实,翻译教材并不是一件美差,对我来说,更谈不上是“好事”,由于眼疾,看书时间一长,心里总犯怵。现在总算完成了任务,两本书都即将出版(尽管中间眼睛还是又犯了一次病)。我之所以愿意翻译这两本书,一是因为它们确实是两本关于计算理论的优秀教材;二是因为国内太缺乏这样的书了,尽管计算机类书刊极其繁多,可是计算理论的书却难觅踪影。清华大学出版社和华章图文信息有限公司购买这两本书的版权翻译出版,此乃善举!我怎好推却。

计算理论在计算机科学技术专业的教学计划中的地位是有争议的。国内计算机专业本科几乎没有开计算理论课的,研究生学这门课的好像也不多,倒是那些想去美国读书的学生为了应付 GRE Subject 测试,自己在找有关计算理论的书看。我希望这两本书的出版能有利于提高计算机教育界对计算理论的重视,特别希望立志致力于计算机科学技术事业的青年学生都来认真地学一点计算理论,那将可能是终身受益的。

本书包括形式语言与自动机、可计算性和计算复杂性三大部分,这是计算理论最核心、最基本的内容。本书的一大特色是突出了算法,从语法分析到 NP 难问题的实用算法,算法贯穿全书。这些内容不仅透彻生动地提供了算法设计与分析的基本理论,而且使读者清楚地看到,计算理论不是没有用,也不是离实际很远的。恰恰相反,不仅当今的许多计算机技术可以在计算理论中找到它的思想渊源,而且在许多实际问题中充满着计算理论。

我们按照作者在第二版序言中提供的电子邮件地址得到本书的勘误。在翻译过程中我们又发现了一些错误,也都予以更正。这些错误基本上都是非实质性的,因此没有在书中一一指明。我们已用电子邮件把发现的错误发给作者。

导言和第 1 章到第 3 章由张立昂翻译,第 4 章到第 7 章由刘田翻译。译文中定有错误和不当之处,敬请读者赐教。

张立昂

1999 年 9 月写于
北京大学燕北园

第一版序言

本书在大学本科水平上介绍经典的和当代的计算理论。粗略地说,内容包括自动机理论与形式语言、Turing(图灵)机的可计算性与递归函数、不可计算性、计算复杂性以及数理逻辑,采用数学的处理方法和计算机科学的观点。于是,在关于上下文无关语言的一章中包含语法分析的讨论,在关于逻辑的几章中给出定理归结证明的有效性和完备性。

在本科教学计划中,这门课通常安排在后面和算法设计与分析课同时上。我们的看法是学计算机科学的学生应该早一点接触这些材料,比如,在二年级或三年级。这是因为,其一,对这些材料的深入研究形成计算机科学中的一些专题;其二,它有助于提供必要的数学示范。但是,我们发现由于一些更高级的教科书要求学生具有较好的数学基础,教这门严格的大学生课是一项困难的任务。我们写这本书的目的是用数学的语言、但不要求有专门的数学经验使得这门课的实质易于被大学生们理解。

全书提供一学年的课程。我们都讲过一个学期的课程,包括第1章到第6章的大部分内容,根据情况删去了关于语法分析、递归函数及具体的不可解判定问题的部分内容,其他的选择是可能的。例如,强调可计算性和机械逻辑基础的课程可以很快地跳过第1章到第3章,集中力量讲第4、6、8和9章。不管怎样使用它,我们都强烈地希望本书对下一代计算机科学家的智力发展有所贡献,在他们受教育的早期阶段向他们介绍关于计算问题的新鲜的和有条理的思想。

我们借这个机会感谢所有让我们从他们那儿学到东西的人,包括教师和学生。特别感谢 Larry Denenberg 和 Aaron Temin 校对早期的草稿,Michael Kahl 和 Oded Shmueli 作为助教对我们的帮助和建议。1980年春季,Albert Meyer 在 MIT(麻省理工学院)用这本书的草稿讲课,提出很好的批评和修改意见,对此我们表示衷心的感谢。当然,遗留下任何错误都应该由我们负责。Renate D'Arcangelo 以她特有的、非凡的完美和迅速打手稿和画图。

第二版序言

自《计算理论基础》问世以来,在这 15 年中许多东西变了,也有许多东西没有变。计算机科学现在是一门成熟得多的公认的学科,在这个计算无处不在、信息全球化和复杂性迅速增长的世界里扮演越来越重要的角色,因此有更多的理由关心它的基础。《计算理论基础》的作者们自己现在更加成熟和忙碌——这是何以这么久才出第二版的原因。我们写第二版,是因为我们感觉到有几处可以说得更好些,有几处可以简单些,还有一些可以删掉。更重要的是,我们希望本书反映计算理论和学它的学生在这些年的进步。虽然就绝对而言现在教计算理论的比过去多了,但是它在计算机科学教学计划中的相对地位,例如与算法课程相比,没有得到加强。实际上,算法设计与分析领域现在已经很成熟,有理由认为它的基本原理是关于计算理论的基础课程的一部分。此外,今天的大学生有广泛的和早期的计算经验,清楚地知道自动机在编译程序中的应用。但是,例如在讲像 Turing(图灵)机这样的简单模型,用它们作为一般的计算机模型时,他们的疑问更多。总之,对这些问题的处理需要作某些修改。

具体地说,与第一版的主要区别有:

- 早在第 1 章就形式地介绍算法设计与分析的入门(与闭包有关),并且算法问题的讨论贯穿全书。在第 2、3 章有几节讨论与有穷自动机和上下文无关文法有关的算法问题(包括状态最小化和上下文无关识别)。有关于 NP 完全问题的易解变形的算法,还有一节考察“对付 NP 完全性”的算法技术(特殊情况的算法、近似算法、回溯与分支定界、局部改进以及模拟退火算法)。

- 第 4 章对 Turing 机的处理更形式化,模拟论证更加简单和更加定量。引入随机存取 Turing 机,以帮助跨越 Turing 机的笨拙与计算机和程序设计语言能力之间的鸿沟。

- 第 5 章介绍不可判定性,包括某些递归函数论内容(到 Rice 定理为止)。介绍文法和递归数值函数,证明它们等价于 Turing 机,证明更加简单。与上下文无关文法有关问题不可判定性的证明采用简单的直接论证,而不求助 Post 对应问题。保留铺砖问题,这个问题在 NP 完全性一章还要见到。

- 处理复杂性的方式相当的新颖;在第 6 章,除多项式时间界限之外没有定义其他时间界限——于是, P 是接触到的第一个复杂性类和概念。然后,用对角化方法证明存在不在 P 中的指数问题。同时介绍现实生活中的问题与它们的语言表示(两者的区别被故意地弄模糊了),广泛地考察它们的算法问题。

- NP 完全性是单独的一章。在这一章中,有一组新的、广泛的、我们认为有助于教学的 NP 完全性归约,最后一个是正则表达式的等价问题——回到本书的第一个问题。正如上面所说,本书的最后一节是关于“对付 NP 完全性”的算法技术。

- 在这个新版中删去关于逻辑的几章。这是一个困难的决定。作出这样的决定有两

个原因：种种迹象表明，这几章是本书过去读得最少和教得最少的几章；现在有几本更好的论述这个题目的书。但是，在第 6 章将详细地论述布尔逻辑和它的可满足性问题。

• 总的说来，证明和讲解被简化了，并且在一些关键的地方更加形式化。有几处，如在上下文无关语言与下推自动机的等价性的证明中，把长篇技术性归纳陈述的证明改作习题。每一节的后面有几个习题。

经过这些改动之后，现在有不只一种的方式讲授本书的材料（还有在第一版中提出的方式以及使用中形成的方式）。以讲授计算理论和算法的基础为目标的一学期长的课程可以以从第 2 章到第 7 章中选取的材料为基础。

我们衷心地感谢在这 15 年中所有向我们提供反馈、想法、错误和更正的我们的学生和同事——不可能在这里把他们全部列出来。特别感谢 Martha Sideri 帮助修订第 3 章。还要十分感谢本书的编辑 Alan Apt 和 Prentice Hall 的朋友们——Barbara Kraemer, Sondra Chavez 和 Bayani de Leon——他们都非常有耐心而且乐于助人。

最后，我们乐于收到指出错误的报告和其他评论，最好是通过电子邮件，地址是：elements @ cs.berkeley.edu. 有关本书的错误、更正和其他信息也可以通过这个地址得到。

导 言

看看你的周围,计算无处不在,无时不在,每一个人都计算,计算影响着所有的人。所以能够如此,是因为在过去的几十年中计算机科学家发现了很复杂的方法用来管理计算机资源,实现通讯,翻译程序,设计芯片和数据库,创造更快、更廉价、更容易使用、更安全的计算机和程序。

和所有的主要学科一样,计算机科学的成功实践建立在优美和坚实的基础之上。自然科学有一些基本问题,如物质的本质是什么?有机体生命的基础和起源是什么?计算机科学同样有它自己的基本问题:什么是算法?什么是能计算的?什么是不能计算的?什么时候可以认为一个算法是实际可行的?60多年来(甚至从电子计算机出现之前就开始了)计算机科学家一直在考虑这些问题并且给出充满智慧的回答,这一切对计算机科学产生了深刻的影响。

本书的目的是介绍渗透在计算机科学中的这些基本思想、模型和结果,它们都是该领域的基本范例,它们有很多理由是值得学习的。首先,现代计算机科学中的很多东西直接或间接地以它们为基础——而其余的应该……。其次,这些思想和模型是漂亮、有力的,是数学建模的杰出例子,是有长久价值的。此外,它们更是历史的一部分,是我们这个领域的“共同潜意识”。不首先了解它们,是很难理解计算机科学的。

这些思想和模型在本质上是数学的,这大概不会让人感到奇怪。虽然计算机肯定是一个物理实体,但是同样肯定的是关于它的物理方面,如它的分子和它的形状,能够值得说的很少;对计算机最有用的抽象显然是数学的,论证这些抽象所必需的手段也一定同样是数学的。此外,实际的计算工作需要只有数学才能提供的铁的保证(希望我们的编译程序正确地翻译,应用程序最终停机,等等)。但是,在计算理论中使用的数学与其他应用学科中使用的数学有很大的区别,它一般是离散的,在这里不强调实数和连续变量,而强调有穷集合和序列。它以很少几个基本的概念为基础,通过小心、耐心、仔细、一层一层地处理这些概念,勾画出它的能力和深奥之处——这很像计算机,在第1章将使你回想这些基本的概念和方法(其中有集合、关系和归纳法),介绍在计算理论中使用它们的风格。

第2章和第3章描述某些受限制的计算模型。它们能够完成一些非常特殊的字符串处理工作,例如回答一个给定的字符串是否出现在给定的正文中,如字punk是否出现在莎士比亚文集中,或者检查一个给定的括号字符串是否正好平衡——像()和(())((),而不是)()的样子。这些受限制的模型(分别叫做有穷自动机和下推自动机)竟然作为像电路和编译程序之类很普通的系统的非常有用和高度完善的部分出现在现实生活中。在这里它们为我们探索算法的一般的形式定义提供极好的热身练习。此外,看到由于增加或删除各种特性这些装置的能力如何增强和减弱(或者,更经常地是,保持不变)是有教益的。其中最值得注意的特性是非确定性,计算的这个迷惑人的特性既是重要的,又是(相当荒谬地)非现实的。

在第4章,我们研究算法的一般模型,其中最基本的模型是 Turing 机^①。Turing 机是第2章和第3章中字符串处理装置的相当简单的推广,结果令人吃惊地成为描述任意算法的一般框架。为了证明这一点,即通常所说的 Church-Turing 论题,我们引入越来越复杂的计算模型(Turing 机更强的变种,还有随机存取 Turing 机和递归定义的数值函数),并且证明它们在能力上全都完全等价于基本的 Turing 机模型。

再下一章处理不可判定性。某些自然的明确的计算任务具有这种出人意料的性质,可以证明它们超出了算法能够解决的范围。例如,问你是否能用给定的若干种形状的砖铺整个平面。如果这些形状中包括一个正方形,或者甚至一个三角形,则答案显然是“能够”。但是,如果是几种稀奇古怪的形状,或者规定几种形状必须至少使用一次,那会怎样?这肯定是很复杂的问题,你想用机器给出回答。在第5章我们使用 Turing 机的形式表示证明这个问题和许多其他问题是根本不可能用计算机解决的。

即使当一项计算任务能够用某个算法解决的时候,也可能不存在解决它的适当的、实际可行的算法。在本书的最后两章,我们说明怎么用它们的复杂性对现实生活中的计算问题进行分类;某些问题能够在合理的,即多项式的时间界限内解决,而另一些问题似乎需要以天文速度,即指数地增长的时间。在第7章我们定义一个叫做 NP 完全的问题类,它们是一些普通的、实际的、但难得出名的问题(旅行商问题仅是它们中的一个)。我们证明所有这些问题在下述意义下是等价的:如果它们中的一个有有效的算法,则它们每一个都有有效的算法。普遍地相信所有 NP 完全问题具有固有的指数复杂度;这个猜想是否实际上为真是著名的 $P \neq NP$ 问题,它是当代数学家和计算机科学家面临的最重要和最深刻的问题之一。

本书有很多篇幅是有关算法及其形式基础的。然而,大概你也意识到了,在今天的计算机科学教学计划中,算法课程,包括算法的分析和设计,相当地脱离计算理论的课程。在本书的现行版本中,我们试图恢复这门课程的某种统一性。结果是,本书对算法也提供了相当不错的介绍,只是有些专门和非传统。在第1章形式地介绍算法及其分析,并且在第2章和第3章研究受限制的计算模型和由它们产生的自然的计算问题的时候一再重新提起。这样一来,在后面探索算法的一般模型的时候,读者处在较好的位置,能正确评价这种探索的目标并且断定是能成功的。在讲解复杂性时算法同样担任主角。因为鉴别复杂问题的最好方法是把它与另一个经得起有效算法考验的问题进行比较。最后一章用一节叙述对付 NP 完全性作为结束,给出在遇到 NP 完全问题时已经成功地使用过的算法技术(近似算法、穷举算法、启发式局部搜索等)。

计算是必不可少的、有力的、优美的、具有挑战性和不断发展的——它的理论也是如此。本书仅讲了一个激动人心的故事的开头,它是对从计算理论宝库中精选出的少量基本论题的朴素介绍。我们希望它将激发读者去作进一步的探索,每一章最后的参考文献指出了好的出发点。

^① 以卓越的英国数学家和哲学家 Alan M. Turing (1912—1954) 的名字命名。他在 1936 年发表的开创性的论文标志计算理论的诞生。Turing 也是人工智能和计算机下棋以及生物学中形态形成领域中的先驱者。在第二次世界大战中,他帮助破译德国海军密码 Enigma。想更多地了解他迷人的一生(以及他最后悲惨地死在官方残忍和偏执的手中)请阅读《Alan Turing: The Enigma》, Andrew Hodges 著, New York: Simon Schuster, 1983 年。

目 录

译者序	(Ⅱ)
第一版序言	(V)
第二版序言	(Ⅵ)
导言	(Ⅹ)
第1章 集合、关系和语言	(1)
1.1 集合	(1)
1.2 关系与函数	(4)
1.3 特殊类型的二元关系	(7)
1.4 有穷集合与无穷集合	(11)
1.5 三个基本的证明技术	(13)
1.6 闭包与算法	(17)
1.7 字母表与语言	(25)
1.8 语言的有穷表示	(29)
参考文献	(33)
第2章 有穷自动机	(34)
2.1 确定型有穷自动机	(34)
2.2 非确定型有穷自动机	(39)
2.3 有穷自动机与正则表达式	(47)
2.4 正则语言与非正则语言	(54)
2.5 状态最小化	(58)
2.6 关于有穷自动机的算法	(65)
参考文献	(70)
第3章 上下文无关语言	(72)
3.1 上下文无关文法	(72)
3.2 语法分析树	(79)
3.3 下推自动机	(83)
3.4 下推自动机与上下文无关文法	(87)
3.5 上下文无关语言与非上下文无关语言	(92)
3.6 关于上下文无关文法的算法	(97)
3.7 确定性与语法分析	(102)
参考文献	(115)
第4章 Turing 机	(117)
4.1 Turing 机的定义	(117)
4.2 用 Turing 机计算	(125)

4.3 Turing 机的扩充	(130)
4.4 随机存取 Turing 机	(136)
4.5 非确定型 Turing 机	(144)
4.6 文法	(148)
4.7 数值函数	(151)
参考文献	(158)
第 5 章 不可判定性	(160)
5.1 Church-Turing 论题	(160)
5.2 通用 Turing 机	(161)
5.3 停机问题	(163)
5.4 与 Turing 机有关的不可判定问题	(166)
5.5 与文法有关的不可解问题	(168)
5.6 不可解的铺砖问题	(172)
5.7 递归语言的性质	(174)
参考文献	(178)
第 6 章 计算复杂性	(179)
6.1 P 类	(179)
6.2 若干问题	(181)
6.3 布尔可满足性	(187)
6.4 NP 类	(190)
参考文献	(194)
第 7 章 NP 完全性	(196)
7.1 多项式时间归约	(196)
7.2 Cook 定理	(202)
7.3 其他的 NP 完全问题	(207)
7.4 对付 NP 完全性	(218)
参考文献	(229)
中英对照名词索引	(231)

第 1 章 集合、关系和语言

1.1 集 合

人们说数学是科学的语言,它当然也是我们在本书将要学习的科学学科——计算理论——的语言。数学语言涉及集合以及它们重叠、相交和用集合构成新的集合的各种复杂方式。

集合是对象的汇集。例如,4 个字母 a, b, c 和 d 的汇集是一个集合,把它叫做 L ,记作 $L = \{a, b, c, d\}$ 。构成集合的所有对象叫做它的元素或成员。例如, b 是集合 L 的一个元素,表成 $b \in L$,有时直接说 b 在 L 中,或 L 包含 b 。另一方面, x 不是 L 的元素,记作 $x \notin L$ 。

在集合中,不区分元素的重复。于是,集合 $\{\text{red}, \text{blue}, \text{red}\}$ 与 $\{\text{red}, \text{blue}\}$ 是相同的集合。类似地,所有元素的顺序是无关紧要的。例如, $\{3, 1, 9\}$, $\{9, 3, 1\}$ 和 $\{1, 3, 9\}$ 是相同的集合。总之,两个集合相等(即,相同)当且仅当它们有相同的元素。

一个集合的元素之间不需要有什么关系(除去它们都是同一个集合的元素外)。例如, $\{3, \text{red}, \{d, \text{blue}\}\}$ 是有 3 个元素的集合,其中一个元素的本身是一个集合,集合可能只有一个元素,这样的集合叫做单元集。例如, $\{1\}$ 是以 1 作为它的唯一一个元素的集合, $\{1\}$ 与 1 是完全不同的,还有一个根本没有元素的集合。当然,只可能有一个这样的集合,它叫做空集,用 \emptyset 表示。称除空集以外的任何集合是非空的。

到现在为止,我们用直接列出所有的元素来规定集合,元素用逗号隔开,放在一对花括号内。有些集合是无穷的,不可能用这种方式写出来。例如,自然数集合 N 是无穷的。写成 $N = \{0, 1, 2, \dots\}$,在无穷长的表处用 3 个点可以直觉地示意出它的元素。不是无穷的集合是有穷的。

规定集合的另一个方法是求助别的集合与元素可能具有或不具有的性质,设 $I = \{1, 3, 9\}$ 和 $G = \{3, 9\}$,可以把 G 描述成由 I 的大于 2 的元素组成的集合,写成

$$G = \{x; x \in I \text{ 且 } x \text{ 大于 } 2\}$$

一般地,设集合 A 已经有定义, P 是 A 的元素可能具有、也可能不具有的性质,则可以定义一个新的集合

$$B = \{x; x \in A \text{ 且 } x \text{ 具有性质 } P\}$$

下面是另一个例子,奇自然数集合为

$$O = \{x; x \in N \text{ 且 } x \text{ 不被 } 2 \text{ 整除}\}$$

如果集合 A 的每一个元素都是集合 B 的元素,则称 A 是 B 的子集,记成 $A \subseteq B$ 。于是, $O \subseteq N$,因为每一个奇自然数都是自然数。注意任何集合是它自身的子集。如果 A 是 B 的子集且 A 不等于 B ,则称 A 是 B 的真子集,记作 $A \subset B$ 。还要注意空集是每一个集合的

子集。设 B 是任一集合, 因为 \emptyset 的每一个元素(没有这样的元素)也是 B 的元素, 故 $\emptyset \subseteq B$ 。

要想证明两个集合 A 和 B 相等, 可以证明 $A \subseteq B$ 和 $B \subseteq A$ 。于是, A 的每一个元素一定是 B 的元素, B 的每一个元素也一定是 A 的元素, 从而 A 和 B 有相同的元素, 故 $A = B$ 。

和用算术运算, 比如用加法, 把数结合在一起一样, 可以用各种集合运算把两个集合结合成第三个集合。并是一种集合运算: 两个集合的并是一个集合, 它的元素至少在这两个给定集合中的一个中、也可能在这两个中。用符号 \cup 表示并, 因而

$$A \cup B = \{x; x \in A \text{ 或 } x \in B\}$$

例如,

$$\{1, 3, 9\} \cup \{3, 5, 7\} = \{1, 3, 5, 7, 9\}$$

两个集合的交是这两个集合共有的全部元素组成的集合, 即

$$A \cap B = \{x; x \in A \text{ 且 } x \in B\}$$

例如,

$$\{1, 3, 9\} \cap \{3, 5, 7\} = \{3\}$$

和

$$\{1, 3, 9\} \cap \{a, b, c, d\} = \emptyset$$

最后, 两个集合 A 和 B 的差, 记作 $A - B$, 是 A 的不在 B 中的所有元素组成的集合。即

$$A - B = \{x; x \in A \text{ 且 } x \notin B\}$$

例如,

$$\{1, 3, 9\} - \{3, 5, 7\} = \{1, 9\}$$

这几个集合运算的某些性质容易从它们的定义得到。例如, 设 A, B 和 C 是集合, 下述算律成立。

幂等律	$A \cup A = A$ $A \cap A = A$
交换律	$A \cup B = B \cup A$ $A \cap B = B \cap A$
结合律	$(A \cup B) \cup C = A \cup (B \cup C)$ $(A \cap B) \cap C = A \cap (B \cap C)$
分配律	$(A \cup B) \cap C = (A \cap C) \cup (B \cap C)$ $(A \cap B) \cup C = (A \cup C) \cap (B \cup C)$
吸收律	$(A \cup B) \cap A = A$ $(A \cap B) \cup A = A$
De Morgan 律	$A - (B \cup C) = (A - B) \cap (A - C)$ $A - (B \cap C) = (A - B) \cup (A - C)$

例 1.1.1 证明第一条 De Morgan 律, 令

$$L = A - (B \cup C)$$

$$R = (A - B) \cap (A - C)$$

要证 $L=R$ 。为此要证 (a) $L \subseteq R$ 和 (b) $R \subseteq L$ 。

(a) 设 x 是 L 的任一元素, 则 $x \in A$ 但 $x \notin B$ 且 $x \notin C$ 。因而 x 是 $A-B$ 和 $A-C$ 两者的元素, 故是 R 的元素。所以 $L \subseteq R$ 。

(b) 设 $x \in R$, 则 x 是 $A-B$ 和 $A-C$ 两者的元素, 因而在 A 中但不在 B 和 C 中。因此 $x \in A$ 但 $x \notin B \cup C$, 故 $x \in L$ 。因此 $R \subseteq L$ 。

这就证明了 $L=R$ 。

◇

如果两个集合没有共同的元素, 即它们的交是空集, 则称它们不相交。

可以定义两个以上集合的并和交。设 S 是任一集合的汇集, $\cup S$ 表示 S 中所有集合的全部元素组成的集合。例如, 如果 $S = \{\{a, b\}, \{b, c\}, \{c, d\}\}$, 则 $\cup S = \{a, b, c, d\}$; 如果 S 是所有以自然数作元素的单元集 $\{\{n\}; n \in \mathbb{N}\}$, 则 $\cup S = \mathbb{N}$ 。一般地,

$$\cup S = \{x; \text{对于某个集合 } P \in S, x \in P\}$$

类似地,

$$\cap S = \{x; \text{对于每一个集合 } P, x \in P\}$$

集合 A 的所有子集的汇集本身是一个集合, 叫做 A 的幂集, 记作 2^A 。例如, $\{c, d\}$ 的全部子集是 $\{c, d\}$ 本身、单元集 $\{c\}$ 和 $\{d\}$, 以及空集 \emptyset 。因此

$$2^{\{c, d\}} = \{\{c, d\}, \{c\}, \{d\}, \emptyset\}$$

非空集合 A 的划分是 2^A 的一个子集 Π , 使得 \emptyset 不是 Π 的元素并且 A 的每一个元素在且只在 Π 中的一个集合中。即, 如果 Π 是 A 的子集的集合使得

- (1) Π 的每一个元素非空,
- (2) Π 的不同元素是不相交的,
- (3) $\cup \Pi = A$,

则 Π 是 A 的一个划分。

例如, $\{\{a, b\}, \{c\}, \{d\}\}$ 是 $\{a, b, c, d\}$ 的一个划分, 而 $\{\{b, c\}, \{c, d\}\}$ 不是它的划分。奇自然数集合和偶自然数集合构成 \mathbb{N} 的一个划分。

习 题

1.1.1 判断下述每一条是真是假:

- (a) $\emptyset \subseteq \emptyset$
- (b) $\emptyset \in \emptyset$
- (c) $\emptyset \in \{\emptyset\}$
- (d) $\emptyset \subseteq \{\emptyset\}$
- (e) $\{a, b\} \in \{a, b, c, \{a, b\}\}$
- (f) $\{a, b\} \subseteq \{a, b, \{a, b\}\}$
- (g) $\{a, b\} \subseteq 2^{\{a, b, \{a, b\}\}}$
- (h) $\{\{a, b\}\} \in 2^{\{a, b, \{a, b\}\}}$
- (i) $\{a, b, \{a, b\}\} - \{a, b\} = \{a, b\}$

1.1.2 用花括号、逗号和数字写出下述集合:

- (a) $(\{1, 3, 5\} \cup \{3, 1\}) \cap \{3, 5, 7\}$

- (b) $\cup \{\{3\}, \{3,5\}, \cap \{\{5,7\}, \{7,9\}\}\}$
 (c) $(\{1,2,5\} - \{5,7,9\}) \cup (\{5,7,9\} - \{1,2,5\})$
 (d) $2^{\{7,8,9\} - \{7,9\}}$
 (e) 2^{\emptyset}

1.1.3 证明下述等式:

- (a) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
 (b) $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
 (c) $A \cap (A \cup B) = A$
 (d) $A \cup (A \cap B) = A$
 (e) $A - (B \cap C) = (A - B) \cup (A - C)$

1.1.4 设 $S = \{a, b, c, d\}$ 。

- (a) S 的成员最少的划分是什么? 成员最多的划分是什么?
 (b) 列出 S 的恰好有两个成员的所有划分。

1.2 关系与函数

数学研究关于对象以及它们之间的关系的陈述。例如,说“小于”是某种对象(即,数)之间的关系是很自然,这种关系在 4 与 7 之间成立,但在 4 与 2 和 4 与 4 之间不成立。但是现在可供我们使用的数学语言只有集合的语言,怎么用这种语言表达对象之间的关系?把关系本身看作集合。属于关系的对象在本质上是直观上使得关系成立的个体的组合。因而小于关系是第一个数小于第二个数的所有数对组成的集合。

但是我们走得快了一点。在属于关系的一对数中,必须能够区分这对数的两部分,而我们还没有说明如何区分。不能把这些数对写成集合,因为 $\{4,7\}$ 和 $\{7,4\}$ 是一样的。最简单的办法是引进一个新的组合对象的方法,把它叫做有序对^①。

a 和 b 的有序对记作 (a,b) , a 和 b 叫做它的分量。有序对 (a,b) 不同于集合 $\{a,b\}$ 。首先,顺序是有关联的: (a,b) 与 (b,a) 不一样,而 $\{a,b\} = \{b,a\}$ 。其次,有序对的两个分量不必是不同的, $(7,7)$ 是有效的有序对。注意:仅当 $a=c$ 且 $b=d$ 时两个有序对 (a,b) 和 (c,d) 相等。

两个集合 A 和 B 的笛卡儿积是 $a \in A$ 且 $b \in B$ 的所有有序对 (a,b) 构成的集合,记作 $A \times B$ 。例如

$$\begin{aligned} & \{1,3,9\} \times \{b,c,d\} \\ &= \{(1,b), (1,c), (1,d), (3,b), (3,c), (3,d), (9,b), (9,c), (9,d)\} \end{aligned}$$

两个集合 A 和 B 上的二元关系是 $A \times B$ 的子集。例如: $\{(1,b), (1,c), (3,d), (9,d)\}$ 是 $\{1,3,9\}$ 和 $\{b,c,d\}$ 上的二元关系。 $\{(i,j); i,j \in \mathbb{N} \text{ 且 } i < j\}$ 是小于关系,它是 $\mathbb{N} \times \mathbb{N}$ 的子集(用二元关系关联的两个集合常常是相同的)。

更一般地,设 n 是任一自然数。如果 a_1, \dots, a_n 是任意 n 个对象,不必是不相同的,则 (a_1, \dots, a_n) 是一个有序组。对每一个 $i=1, \dots, n, a_i$ 是 (a_1, \dots, a_n) 的第 i 个分量。有序 m 元

^① 在集合论中不把有序对 (a,b) 看作新的对象类型,它等同于 $\{\{a\}, \{a,b\}\}$ 。

组 (b_1, \dots, b_m) 和 (a_1, \dots, a_n) 相同当且仅当 $m=n$ 且 $a_i=b_i, i=1, \dots, n$ 。于是, $(4, 4)$, $(4, 4, 4)$, $((4, 4), 4)$ 和 $(4, (4, 4))$ 都是不相同的。有序二元组就是前面讨论的有序对。有序 3、4、5 和 6 元组分别叫做有序三元组、四元组、五元组和六元组。此外, 序列是一个有序 n 元组, 其中 n 是某个未指定的数, 叫做该序列的长度。设 A_1, \dots, A_n 是任意 n 个集合, n 重笛卡儿积 $A_1 \times \dots \times A_n$ 是所有有序 n 元组 (a_1, \dots, a_n) 的集合, 其中对于每一个 $i=1, \dots, n, a_i \in A_i$ 。当所有的 A_i 都相同(设为 A)时, A 自身的 n 重笛卡儿积 $A \times \dots \times A$ 也写成 A^n 。例如, N^2 是所有自然数有序对的集合。集合 A_1, \dots, A_n 上的 n 元关系是 $A_1 \times \dots \times A_n$ 的一个子集。1、2 和 3 元关系分别叫做一元、二元和三元关系。

另一基本的数学概念是函数。直观上, 函数是把一类中的每一个对象关联到另一类中的一个唯一的对象。例如, 人与他们的年龄, 狗与它们的主人, 数与它们的后继等等。利用把二元关系看作有序对集合的思想, 可以给出这个直观概念的具体定义。从集合 A 到集合 B 的函数是 A 和 B 上的具有下述特殊性质的二元关系 R : 对于每一个元素 $a \in A$, 在 R 中恰好有一个有序对以 a 为第一个分量,用下面的例子说明这个定义。令 C 是美国的城市集合, S 是州集合,

$$R_1 = \{(x, y): x \in C, y \in S \text{ 且 } x \text{ 是州 } y \text{ 的一个城市}\},$$

$$R_2 = \{(x, y): x \in S, y \in C \text{ 且 } y \text{ 是州 } x \text{ 的一个城市}\}.$$

那么, R_1 是一个函数, 因为每一个城市在且只在一个州内; R_2 不是函数, 因为一些州有一个以上城市^①。

一般地, 用字母 f, g, h 等表示函数, 用 $f: A \rightarrow B$ 表示 f 是从 A 到 B 的函数。把 A 叫做 f 的定义域。设 a 是 A 的任一元素, 用 $f(a)$ 表示 B 中使得 $(a, b) \in f$ 的元素 b 。因为 f 是一个函数, 恰好存在一个 $b \in B$ 具有这个性质, 所以 $f(a)$ 表示一个唯一的对象。 $f(a)$ 叫做 a 在 f 下的象。为了详细说明函数 $f: A \rightarrow B$, 只要对每一个 $a \in A$, 指定 $f(a)$ 。例如, 要详细说明上述函数 R_1 , 只要对每一个城市指出它所在的州。设 $f: A \rightarrow B, A'$ 是 A 的子集, 定义 $f[A'] = \{f(a): a \in A'\}$ (即, $\{b: \text{对某个 } a \in A', b = f(a)\}$)。称 $f[A']$ 是 A' 在 f 下的象。 f 的值域是它的定义域的象。

当函数的定义域是笛卡儿积时, 通常省略一对圆括号。例如, 如果定义 $f: N \times N \rightarrow N$ 使得有序对 (m, n) 在 f 下的象是 m 与 n 的和, 则按照习惯记法写成 $f(m, n) = m + n$, 而不写成 $f((m, n)) = m + n$ 。

设 $f: A_1 \times \dots \times A_n \rightarrow B$ 是一个函数, $f(a_1, \dots, a_n) = b$, 其中 $a_i \in A_i, i=1, \dots, n$ 且 $b \in B$, 有时称 a_1, \dots, a_n 是 f 的自变量, 而 b 是 f 对应的值。于是, 对自变量的每一个 n 元组给出 f 的值可以指定函数 f 。

对某些类型的函数特别感兴趣。如果对任意两个不同的值 $a, a' \in A, f(a) \neq f(a')$, 则称函数 $f: A \rightarrow B$ 是一对一的。例如, 设 C 是美国的城市集合, S 是州集合, 而 $g: S \rightarrow C$ 规定为

$$g(s) = \text{州 } s \text{ 的首府}, \quad \text{对每一个 } s \in S$$

^① 我们认为马萨诸塞州的坎布里奇(Cambridge)和马里兰州的坎布里奇(Cambridge)不是同一个城市, 而是恰巧重名的不同的城市。

那么 g 是一一对一的, 因为没有两个州有相同的首府。如果 B 的每一个元素是 A 的某个元素在 f 下的象, 则称函数 $f: A \rightarrow B$ 满射到 B 。刚才规定的函数 g 不是满射到 C 的, 但上面定义的函数 R_1 满射到 S , 因为每一个州至少有一个城市。最后, 如果函数 $f: A \rightarrow B$ 既是一一对一的, 又是满射到 B 的, 则称 f 是 A 与 B 之间的双射。例如, 如果 C_0 是州首府集合, 和前面一样, 函数 $g: S \rightarrow C_0$ 规定为

$$g(s) = \text{州 } s \text{ 的首府}$$

则 g 是 S 与 C_0 之间的双射。

二元关系 R 的逆是关系 $\{(b, a); (a, b) \in R\}$, 记作 R^{-1} 。显然, 若 $R \subseteq A \times B$, 则 $R^{-1} \subseteq B \times A$ 。例如, 上面定义的关系 R_2 是 R_1 的逆。可见函数的逆不一定是函数。对 R_1 来说, 它的逆不再是一个函数, 因为某些州有一个以上城市, 即存在不同的城市 C_1 和 C_2 使 $R_1(C_1) = R_1(C_2)$ 。设 $f: A \rightarrow B$ 是一个函数, 如果存在 $b \in B$ 使得对于所有的 $a \in A$, $f(a) \neq b$, 则函数 f 的逆也不是函数。然而, 如果 $f: A \rightarrow B$ 是一个双射, 则这些情况都不可能发生, f^{-1} 也是一个函数, 它是 B 与 A 之间的双射。而且, 对每一个 $a \in A$, $f^{-1}(f(a)) = a$; 对每一个 $b \in B$, $f(f^{-1}(b)) = b$ 。

当在两个集合之间规定一个特别简单的双射之后, 有时能够把定义域中的对象与它在值域中的象看作在实质上是不可区分的: 一个是另一个的换名或一种重写方式。例如, 严格地说, 单元集和有序一元组是不同的。但是, 由于对每一个单元集 $\{a\}$, $f(\{a\}) = a$ 是一个明显的双射, 故偶尔地混淆两者的区别也没有多大害处。这样的双射叫做一个自然同构。当然, 这不是一个形式定义, 因为什么是“自然的”, 什么差别是可以混淆的, 要依赖上下文。再举几个稍微复杂一点的例子可能会更清楚一些。

例 1.2.1 对于任意 3 个集合 A, B 和 C , 有 $A \times B \times C$ 到 $(A \times B) \times C$ 的自然同构, 即对任意的 $a \in A, b \in B$ 和 $c \in C$,

$$f(a, b, c) = ((a, b), c) \quad \diamond$$

例 1.2.2 对于任意的 A 和 B , 从 A 和 B 上的所有二元关系的集合 $2^{A \times B}$ 到集合 $\{f: f \text{ 是从 } A \text{ 到 } 2^B \text{ 的函数}\}$

的自然同构, 即, 对任意的二元关系 $R \subseteq A \times B$, 令 $\phi(R)$ 为函数 $f: A \rightarrow 2^B$ 使得

$$f(a) = \{b; b \in B \text{ 且 } (a, b) \in R\}$$

例如, 设 S 是州集合, $R \subseteq S \times S$ 包含所有有共同边界的两个州的有序对, 则自然的相关函数 $f: S \rightarrow 2^S$ 规定为

$$f(s) = \{s'; s' \in S \text{ 且 } s' \text{ 与 } s \text{ 有共同边界}\} \quad \diamond$$

例 1.2.3 有时即使当函数 $f: A \rightarrow B$ 不是一个双射时也把它逆看作一个函数。这个想法是用例 1.2.2 中描述的自然同构把 $f^{-1} \subseteq B \times A$ 看作从 B 到 2^A 的函数。于是, 对于任意的 $b \in B$, $f^{-1}(b)$ 等于使得 $f(a) = b$ 的所有 a 组成的集合。例如, 设 R_1 的定义如前面所述(对于每一个城市, 这个函数的值等于它所在的州), 则对于每一个州 s , $R_1^{-1}(s)$ 等于州 s 中所有城市的集合。 \diamond

设 Q 和 R 是两个二元关系, 它们的合成 $Q \circ R$ (可简写成 QR) 是关系

$$\{(a, b); \text{对于某个 } c, (a, c) \in Q \text{ 且 } (c, b) \in R\}$$

注意到两个函数 $f: A \rightarrow B$ 和 $g: B \rightarrow C$ 的合成是一个从 A 到 C 的函数 h , 使得对于每一个

$a \in A, h(a) = g(f(a))$ 。例如,如果 f 是把每一条狗对应到它的主人的函数,而 g 把每一个人对应到他(她)的年龄,则 $f \circ g$ 把每一条狗对应到它的主人的年龄。

习 题

1.2.1 列出下列各式的所有元素:

(a) $\{1\} \times \{1,2\} \times \{1,2,3\}$

(b) $\emptyset \times \{1,2\}$

(c) $2^{\{1,2\}} \times \{1,2\}$

1.2.2 设 $R = \{(a,b), (a,c), (c,d), (a,a), (b,a)\}$ 。 R 与自身的合成 $R \circ R$ 是什么? R 的逆 R^{-1} 是什么? $R, R \circ R$ 和 R^{-1} 是函数吗?

1.2.3 设 $f: A \rightarrow B, g: B \rightarrow C$, 而 $h: A \rightarrow C$ 是它们的合成。叙述当且仅当 f 和 g 满足什么条件时 h 具有下述各条性质:

(a) 满射

(b) 一对一

(c) 双射

1.2.4 设 A 和 B 是任意两个集合,用 B^A 表示从 A 到 B 的所有函数的集合,描述一个 $\{0,1\}^A$ 与 2^A 之间的自然同构。

1.3 特殊类型的二元关系

在本书中将会一再看到二元关系,有一些表示它们的便利方法和讨论它们的性质的术语是会有帮助的。一个完全“随机的”二元关系没有值得注意的内在结构,而我们将要遇到的许多二元关系出自某种具体的场合,因而具有某些重要的规则性。例如,两个城市属于同一个州的关系具有某种“对称性”以及其他值得注意、讨论和研究的性质。

本节讨论表示这些以及类似规律性的二元关系。下面只考虑一个集合与它自己上的二元关系。设 A 是一个集合, $R \subseteq A \times A$ 是 A 上的一个二元关系。可以用一个有向图表示关系 R 。 A 的每一个元素用一个小圆圈表示(叫做有向图的顶点),从 a 到 b 画一个箭头当且仅当 $(a,b) \in R$ 。这些箭头是该有向图的边。例如,用图 1-1 中的图表示关系 $R = \{(a,b), (b,a), (a,d), (d,c), (c,c), (c,a)\}$ 。特别要注意 c 到自身的环,它对应有序时 $(c,c) \in R$ 。从一个顶点到另一个顶点或者没有边,或者有一条边,这里不允许“平行的箭头”。

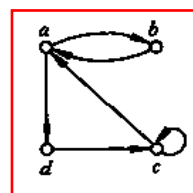


图 1-1

集合 A 上的二元关系与顶点来自 A 的有向图没有形式的区别。当我们想强调对在上面定义二元关系的集合没有单独的兴趣时,使用术语有向图。有向图以及很快就要介绍的无向图作为复杂系统(交通网与通信网、计算结构与过程等)的模型和抽象是有用的。我们将在 1.6 节,尤其是在第 6 章和第 7 章更加详细地讨论与有向图有关的许多有趣的计算问题。

作为二元关系或有向图的另一个例子,定义在自然数集合上的小于等于关系 \leq , 如图 1-2 所示。当然,不可能画出整个有向图,因为它是无穷的。

如果对于每一个 $a \in A$, $(a, a) \in R$, 则称关系 $R \subseteq A \times A$ 是自反的。表示一个自反关系的有向图从每一个顶点到它自身有一个环。例如, 图 1-2 中的有向图表示一个自反关系, 而图 1-1 中的图所表示的关系不是自反的。

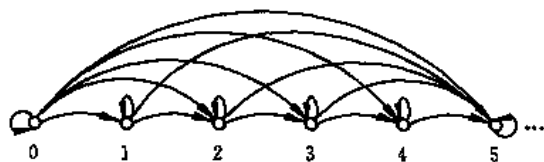


图 1-2



图 1-3

如果只要 $(a, b) \in R$ 就有 $(b, a) \in R$, 则称关系 $R \subseteq A \times A$ 是对称的。在对应的有向图中, 只要两个顶点之间有一个箭头, 则在这两个顶点之间两个方向都有箭头。例如, 图 1-3 中的有向图表示一个对称关系。这个有向图可以描述 5 人中的“朋友”关系。只要 x 是 y 的朋友, y 也是 x 的朋友。朋友关系不是自反的, 因为不把一个人看作他(她)自己的朋友。当然, 一个关系可以既是对称的、又是自反的。例如, $\{(a, b); a \text{ 和 } b \text{ 的父亲是同一个人}\}$ 是这样一个关系。

把没有 (a, a) 形式的有序对的对称关系表示成无向图, 或简称为图。画无向图不使用箭头, 把两个顶点之间一对方向相反的箭头合并成一条不带箭头的边。例如, 图 1-3 给出的关系也可以用图 1-4 中的图表示。

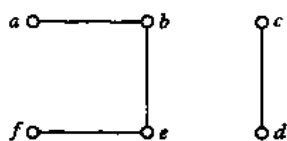


图 1-4

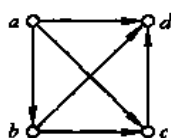


图 1-5

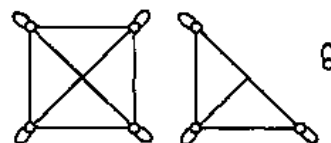


图 1-6

如果当 $(a, b) \in R$ 且 a 与 b 不同时, $(b, a) \notin R$, 则称关系 R 是反对称的。例如, 设 P 是所有的人组成的集合, 则

$$\{(a, b); a, b \in P \text{ 且 } a \text{ 是 } b \text{ 的父亲}\}$$

是反对称的。一个关系可能既不是对称的, 也不是反对称的。例如, 关系

$$\{(a, b); a, b \in P \text{ 且 } a \text{ 是 } b \text{ 的兄弟}\}$$

和图 1-1 表示的关系都既不是对称的, 也不是反对称的。

如果只要 $(a, b) \in R$ 且 $(b, c) \in R$ 就有 $(a, c) \in R$, 则称二元关系 R 是传递的。例如, 关系

$$\{(a, b); a, b \in P \text{ 且 } a \text{ 是 } b \text{ 的前辈}\}$$

是传递的, 因为若 a 是 b 的前辈且 b 是 c 的前辈, 则 a 是 c 的前辈。小于等于关系也是传递的。用有向图表示的时候, 传递性等价于要求只要有从元素 a 引导到元素 z 的箭头序列, 则有从 a 直接到 z 的箭头。例如, 图 1-5 给出的关系是传递的。

把自反、对称和传递的关系叫做等价关系。表示等价关系的无向图由若干集团组成, 在每一个集团中的每一对顶点由一条线连结(见图 1-6)。把等价关系的这种“集团”叫做等价类。当根据上下文不必指明等价关系 R 时, 通常用 $[a]$ 表示包含元素 a 的等价类。即,

$[a] = \{b: (a, b) \in R\}$, 或者 $[a] = \{b: (b, a) \in R\}$ (因为 R 是对称的)。例如, 图 1-6 中的等价关系有 3 个等价类。一个等价类有 4 个元素, 一个等价类有 3 个元素, 还有一个等价类有 1 个元素。

定理 1.3.1 设 R 是非空集合 A 上的等价关系, 则 R 的等价类构成 A 的一个划分。

证: 令 $\Pi = \{[a]: a \in A\}$ 。要证明 Π 中的集合是非空不相交的, 并且并在一起等于 A 。根据自反性, 对于所有的 $a \in A, a \in [a]$, 故所有等价类非空。证明它们是不相交的, 考虑任意两个不同的等价类 $[a]$ 和 $[b]$, 且假设 $[a] \cap [b] \neq \emptyset$ 。于是, 存在元素 c 使得 $c \in [a]$ 和 $c \in [b]$ 。从而 $(a, c) \in R$ 且 $(c, b) \in R$, 由传递性得 $(a, b) \in R$ 。又由 R 是对称的, 有 $(b, a) \in R$ 。现在取任一元素 $d \in [a]$, 则 $(d, a) \in R$, 由传递性得 $(d, b) \in R$ 。因而 $d \in [b]$, 得证 $[a] \subseteq [b]$ 。类似可证 $[b] \subseteq [a]$ 。因此, $[a] = [b]$ 。这与 $[a]$ 和 $[b]$ 不相同的假设矛盾。

只要注意到由自反性, A 的每一个元素 a 在 Π 中的某个集合, 即 $a \in [a]$, 容易看出 $\bigcup \Pi = A$ 。 ■

于是, 给定一个等价关系 R , 总能够构造出对应的划分 Π 。例如, 设

$$R = \{(a, b): a \text{ 与 } b \text{ 是两个人且有相同的双亲}\}$$

则 R 的等价类是同胞兄弟姐妹。注意定理 1.3.1 的构造能够反过去: 任给一个划分, 可以构造一个对应的等价关系。也就是说, 设 Π 是 A 的一个划分, 则

$$R = \{(a, b): a \text{ 和 } b \text{ 在 } \Pi \text{ 中的同一个集合中}\}$$

是一个等价关系。于是, 在集合 A 上的等价关系与 A 的划分之间存在一个自然同构。

自反、反对称和传递的关系叫做偏序。例如,

$$\{(a, b): a \text{ 和 } b \text{ 是两个人且 } a \text{ 是 } b \text{ 的前辈}\}$$

是一个偏序 (假定把每一个人看作自己的前辈)。设 $R \subseteq A \times A$ 是一个偏序。如果对所有的 $b \in A$, 仅当 $b = a$ 时 $(b, a) \in R$, 则称 $a \in A$ 是极小元。例如, 在上面定义的前辈关系中, 只有亚当和夏娃是极小元。有穷的偏序一定至少有一个极小元, 无穷的偏序不一定有极小元。

如果 $R \subseteq A \times A$ 是一个偏序, 并且对所有的 $a, b \in A$, 或者 $(a, b) \in R$ 或者 $(b, a) \in R$, 则称 R 是一个全序。前辈关系不是全序, 因为不是任意两人都有前辈关系 (例如, 兄弟姐妹)。而自然数上的小于等于关系是一个全序。全序不可能有两个或两个以上极小元。

二元关系 R 中的一条通路是一个满足下述条件的序列 (a_1, \dots, a_n) , 对于 $i = 1, \dots, n-1, (a_i, a_{i+1}) \in R$, 其中 $n \geq 1$ 。称这是一条从 a_1 到 a_n 的通路。通路 (a_1, \dots, a_n) 的长度为 n 。如果所有的 a_i 都不相同且 $(a_n, a_1) \in R$, 则称通路 (a_1, \dots, a_n) 是一个圈。

习 题

1.3.1 设 $R = \{(a, c), (c, e), (e, e), (e, b), (d, b), (d, d)\}$ 。画出表示下述关系的有向图:

- (a) R
- (b) R^{-1}
- (c) $R \cup R^{-1}$
- (d) $R \cap R^{-1}$

1.3.2 设 R 和 S 是 $A = \{1, \dots, 7\}$ 上的两个二元关系, 分别如图 E1-1(a) 和 (b) 所示。

- (a) 指出 R 和 S 是否是 (i) 对称的, (ii) 自反的, (iii) 传递的。

(b) 对于关系 $R \cup S$ 回答(a)中的问题。

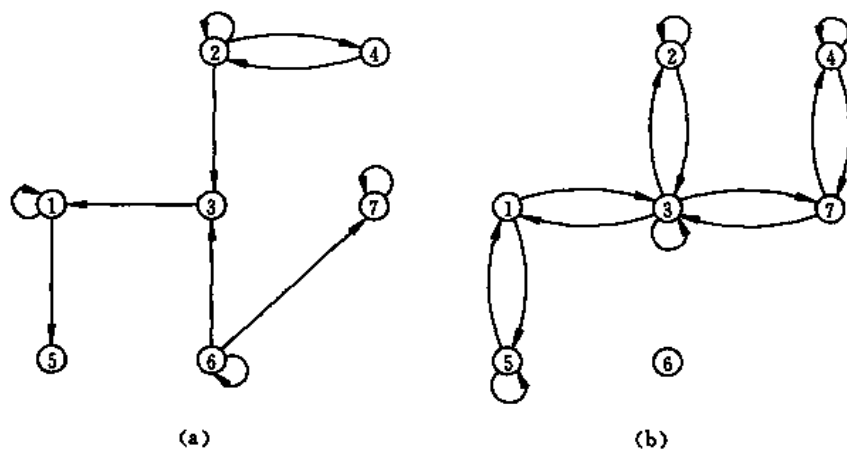


图 E1-1

1.3.3 画出表示下述类型的关系的有向图：

- (a) 自反、传递和反对称的。
- (b) 自反和传递的，但既不对称又不反对称。

1.3.4 设 A 是一个非空集合， $R \subseteq A \times A$ 是空集。 R 具有哪些性质？

- (a) 自反性
- (b) 对称性
- (c) 反对称性
- (d) 传递性

1.3.5 设 $f: A \rightarrow B$ 。证明下述关系 R 是 A 上的等价关系： $(a, b) \in R$ 当且仅当 $f(a) = f(b)$ 。

1.3.6 设 $R \subseteq A \times A$ 是如下定义的二元关系。哪些 R 是偏序？哪些 R 是全序？

- (a) A 为所有的正整数； $(a, b) \in R$ 当且仅当 a 可以整除 b 。
- (b) $A = \mathbb{N} \times \mathbb{N}$ ； $((a, b), (c, d)) \in R$ 当且仅当 $a \leq c$ 或 $b \leq d$ 。
- (c) $A = \mathbb{N}$ ； $(a, b) \in R$ 当且仅当 $b = a$ 或 $b = a + 1$ 。
- (d) A 为所有英语单词的集合； $(a, b) \in R$ 当且仅当 a 不比 b 长。
- (e) A 为所有汉字的集合； $(a, b) \in R$ 当且仅当 a 与 b 相同或在本书中出现的次数比 b 多。

1.3.7 设 R_1 和 R_2 是同一个集合 A 上的任意两个偏序。证明 $R_1 \cap R_2$ 是一个偏序。

1.3.8 (a) 证明：如果 S 是任意集合的汇集，则 $R_S = \{(A, B) : A, B \in S \text{ 且 } A \subseteq B\}$ 是一个偏序。

(b) 设 $S = 2^{\{1, 2, 3\}}$ 。画出表示(a)中定义的偏序 R_S 的有向图。 R_S 的极小元是什么？

1.3.9 在什么情况下有向图表示一个函数？

1.3.10 证明：有穷集合到它自身的任意函数包含一个圈。

1.3.11 设 S 是任一集合， \mathcal{P} 是 S 的所有划分的集合。又设 R 是 \mathcal{P} 上的二元关系定义

如下: $(\Pi_1, \Pi_2) \in R$ 当且仅当对于每一个 $S_1 \in \Pi_1$, 存在 $S_2 \in \Pi_2$ 使得 $S_1 \subseteq S_2$ 。如果 $(\Pi_1, \Pi_2) \in R$, 则称 Π_1 细化 Π_2 。

证明: R 是 \mathcal{S} 上的偏序。 \mathcal{S} 的什么元素是极大的和极小的? 假设 \mathcal{S} 是 2^S 的子集的任意汇集, 这些子集不一定是 S 的划分。 R 一定是偏序吗?

1.4 有穷集合与无穷集合

有穷集合的一个基本性质是它的大小, 即它包含的元素个数。关于有穷集合大小的某些事实是非常明显的, 几乎不需要证明。例如, 如果 $A \subseteq B$, 则 A 的大小小于等于 B 的大小; A 的大小等于 0 当且仅当 A 是空集。

然而, 如果我们打算按照直觉把“大、小”的概念推广到无穷集合上就要产生困难。17 的倍数 $\{0, 17, 34, 51, 68, \dots\}$ 比完全平方数 $\{0, 1, 4, 9, 16, \dots\}$ 多吗? 你可以去猜想哪一个多, 但是经验已经表明只有认为这两个集合有同样的大小是令人满意的约定。

如果存在双射 $f: A \rightarrow B$, 则称集合 A 与 B 等势。回想一下, 如果存在双射 $f: A \rightarrow B$, 则存在双射 $f^{-1}: B \rightarrow A$, 从而等势性是一种对称关系。事实上, 容易证明它是一个等价关系。例如, $\{8, \text{red}, \{\emptyset, b\}\}$ 与 $\{1, 2, 3\}$ 等势, 可以令 $f(8)=1, f(\text{red})=2, f(\{\emptyset, b\})=3$ 。所有 17 的倍数与所有完全平方数也是等势的。对于每一个 $n \in \mathbb{N}$, 令 $f(17n)=n^2$ 给出两者之间的双射。

一般地, 直观上如果对于某个自然数 n , 一个集合与 $\{1, 2, \dots, n\}$ 等势, 则称这个集合是有穷的。(当 $n=0$ 时, $\{1, \dots, n\}$ 是空集, 空集与自身等势, 所以空集是有穷的。)如果 A 与 $\{1, \dots, n\}$ 等势, 则称 A 的基数(记作 $|A|$)为 n 。有穷集合的基数就是它的元素个数。

如果集合不是有穷的, 则称它是无穷的。例如, 自然数集合 \mathbb{N} 是无穷的, 整数集合、实数集合和完全平方数集合等都是无穷的。但是, 不是所有的无穷集合都等势。

称与 \mathbb{N} 等势的集合是可数无穷的, 称有穷的或可数无穷的集合是可数的。不是可数的集合称作不可数的。要证明集合 A 是可数无穷的, 必须给出 A 与 \mathbb{N} 之间的一个双射 f 。等价地, 只需给出一种方式, 能够用这种方式把 A 枚举成

$$A = \{a_0, a_1, a_2, \dots\}$$

这样的枚举立即给出一个双射: $f(0)=a_0, f(1)=a_1, \dots$ 。

例如, 可以证明任意有穷个可数无穷集合的并是可数无穷的。下面给出对于 3 个两两不相交的可数无穷集合的证明, 一般情况的证明可类似进行。设这 3 个集合是 A, B 和 C 。可以像上面那样把它们列成: $A = \{a_0, a_1, \dots\}, B = \{b_0, b_1, \dots\}, C = \{c_0, c_1, \dots\}$ 。于是, 它们的并可以列成 $A \cup B \cup C = \{a_0, b_0, c_0, a_1, b_1, c_1, a_2, \dots\}$ 。这种列表方式是通过在不同集合之间的交替“访问” $A \cup B \cup C$ 中所有元素的一种方法, 如图 1-7 所示。这种交替枚举几个集合的方法叫做“制作棒头”(任何木工在看了图 1-7 之后都能说出这个名字的由来)。

同样的思想可以用来证明可数无穷个可数无穷集合的并是可数无穷的。例如, 证明 $\mathbb{N} \times \mathbb{N}$ 是可数无穷的。注意 $\mathbb{N} \times \mathbb{N}$ 是 $\{0\} \times \mathbb{N}, \{1\} \times \mathbb{N}, \{2\} \times \mathbb{N}$ 等等的并, 这是可数无穷个可数无穷集合的并。制作棒头在这里必须比在上面的例子中更巧妙一些。不能和在那儿一样在访问第一个集合的第二个元素之前访问每一个集合的一个元素, 因为有无穷多个

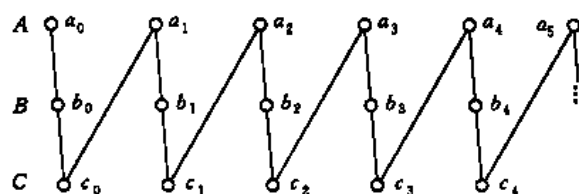


图 1-7

集合要访问,根本不能结束第一轮访问!换成用如下方式进行(见图 1-8)。

- (1) 在第一轮,访问第一个集合的一个元素(0,0)。
- (2) 在第二轮,访问第一个集合的下一个元素(0,1)和第二个集合的第一个元素(1,0)。
- (3) 在第三轮,访问第一个和第二个集合的下一个未访问过的元素(0,2)和(1,1),还有第三个集合的第一个元素(2,0)。
- (4) 一般地,在第 n 轮,访问第一个集合的第 n 个元素,第二个集合的第 $n-1$ 个元素,……,以及第 n 个集合的第一个元素。

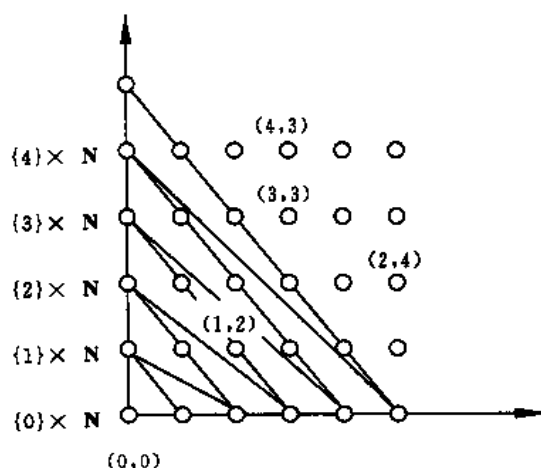


图 1-8

观察这样使用制作棒头的另一种方式是发现第 m 次访问有序对 (i, j) , 其中 $m = \frac{1}{2}[(i+j)^2 + 3i + j]$ 。也就是说,函数 $f(i, j) = \frac{1}{2}[(i+j)^2 + 3i + j]$ 是从 $\mathbb{N} \times \mathbb{N}$ 到 \mathbb{N} 的双射(见习题 1.4.4)。

在下一节的最后,我们要给出证明两个无穷集合不等势的技术。

习 题

1.4.1 证明下述集合是可数的:

- (a) 任意三个可数集合的并,不要求这三个集合是无穷的或不相交的。
- (b) \mathbb{N} 的所有有穷子集的并。

1.4.2 给出下述每一对集合之间的双射:

(a) N 与全体奇自然数。

(b) N 与所有整数的集合。

(c) N 与 $N \times N \times N$ 。

(我们正在寻找尽可能简单的、只含有加法和乘法之类运算的公式。)

1.4.3 设 C 是集合的集合, 定义如下:

1. $\emptyset \in C$ 。

2. 如果 $S_1 \in C$ 且 $S_2 \in C$, 则 $\{S_1, S_2\} \in C$ 。

3. 如果 $S_1 \in C$ 且 $S_2 \in C$, 则 $S_1 \times S_2 \in C$ 。

4. C 的元素均可由(1),(2)和(3)得到。

(a) 详细地说明怎么用(1~4)得到 $\{\emptyset, \{\emptyset\}\} \in C$ 。

(b) 给出一个有序对的集合 S , 使得 $S \in C$ 且 $|S| > 1$ 。

(c) C 包含无穷集合吗? 说明之。

(d) C 是可数的、还是不可数的? 说明之。

1.4.4 证明: 图 1-8 所示的制作棒头的方法在第 m 次访问有序对 (i, j) , 其中

$$m = \frac{1}{2}[(i+j)^2 + 3i + j]$$

1.5 三个基本的证明技术

证明是各不相同的, 因为每一个证明是为了确认各自不同的结果。但是和下棋、打棒球一样, 通过大量的观察发现可以找到一些能够利用的模式、经验和技巧。本节的主要目的是介绍三种基本的原则: 数学归纳法、鸽巢原理和对角化方法。它们以各种不同的形式出现在许多证明中。

数学归纳法的原理 如果 A 是一个自然数的集合使得

(1) $0 \in A$,

(2) 对于每一个自然数 n , $\{0, 1, \dots, n\} \subseteq A$ 蕴涵 $n+1 \in A$,

则 $A = N$ 。

说得非形式一点, 数学归纳法的原理说, 任何一个包含 0 的自然数集合, 如果它具有下述性质: 它只要包含所有小于等于 n 的自然数就包含 $n+1$, 则它实际上就是全体自然数的集合。

直观上这个原理是显然的, 对于每一个自然数, 从 0 开始、每次加 1、在有限步内能够“到达”它, 故每一个自然数都在 A 中。论证这个思想的另一方法是用反证法。假设(1)和(2)成立, 但 $A \neq N$ 。于是, 有一个自然数不在 A 中。特别地, 令 n 是 $0, 1, 2, \dots$ 中第一个不在 A 中的数^①。那么, 由(1), $0 \in A$, 故 n 不是 0; 由 n 的取法, $\{0, 1, \dots, n-1\} \subseteq A$, 故由(2), $n \in A$, 矛盾。

在实践中, 归纳法用来证明下述形式的断言: “对于所有的自然数 n , 性质 P 成立。”用

^① 这里用到另一个原理——最小数原理, 它本质上与数学归纳法原理是等价的, 因此我们实际上没有“证明”数学归纳法原理。最小数原理是: 如果 $A \subseteq N$ 且 $A \neq \emptyset$, 则有唯一的最小数 $n \in A$, 即唯一的数 n 使得 $n \in A$, 而 $0, 1, \dots, n-1 \notin A$ 。下面是一个没有多大意思的命题: 没有不感兴趣的数。可以用最小数原理证明。假设有不感兴趣的数, 则有最小的这样一个数, 比如说是 n 。于是 n 有一个值得注意的性质: 最小的不感兴趣的数。这个性质足以使 n 成为感兴趣的……。

下述方式把数学归纳法原理应用于集合 $A = \{n: \text{对于 } n, P \text{ 为真}\}$;

(1) 在基础步骤中证明 $0 \in A$, 即对于 0, P 成立。

(2) 归纳假设是假设对于任意固定的 $n \geq 0$, P 对于每一个自然数 $0, 1, \dots, n$ 成立。

(3) 在归纳步骤中, 利用归纳假设证明 P 对于 $n+1$ 成立。于是, 根据归纳原理, A 等于 \mathbf{N} , 即, P 对于每一个自然数成立。

例 1.5.1 证明: 对于任意的 $n \geq 0$,

$$1 + 2 + \dots + n = \frac{1}{2}(n^2 + n)$$

基础步骤。 令 $n=0$ 。左边的和等于零, 因为没有要加的数。右边的表达式也等于零。

归纳假设。 假设对于某个 $n \geq 0$, 当 $m \leq n$ 时,

$$1 + 2 + \dots + m = \frac{1}{2}(m^2 + m)$$

归纳步骤。

$$\begin{aligned} & 1 + 2 + \dots + n + (n+1) \\ &= (1 + 2 + \dots + n) + (n+1) \\ &= \frac{1}{2}(n^2 + n) + (n+1) && \text{(由归纳假设)} \\ &= \frac{1}{2}(n^2 + n + 2n + 2) \\ &= \frac{1}{2}[(n+1)^2 + (n+1)] \end{aligned}$$

这正是我们要证明的。 \diamond

例 1.5.2 对于任意的有穷集合 A , $|2^A| = 2^{|A|}$, 即 A 的幂集的势等于 2 的 A 的势次方。用对 A 的势作归纳, 证明这个陈述。

基本步骤。 设集合 A 的势 $n=0$, 则 $A = \emptyset$, $2^{|\emptyset|} = 2^0 = 1$; 另一方面, $2^A = \{\emptyset\}$, $|2^A| = |\{\emptyset\}| = 1$ 。

归纳假设。 设 $n \geq 0$ 并且当 $|A| \leq n$ 时, $|2^A| = 2^{|A|}$ 。

归纳步骤。 设 $|A| = n+1$ 。由于 $n+1 > 0$, A 至少包含一个元素 a 。令 $B = A - \{a\}$, 则 $|B| = n$ 。由归纳假设, $|2^B| = 2^{|B|} = 2^n$ 。现在 A 的幂集可以分成两部分: 包含元素 a 的集合与不包含元素 a 的集合。后一部分正好是 2^B , 而前一部分由在 2^B 的每一个成员中加入 a 得到。于是,

$$2^A = 2^B \cup \{C \cup \{a\} : C \in 2^B\}$$

这样实际上把 2^A 划分成两个不相交的等势的部分, 从而整个的势等于 $2^{|B|}$ 的两倍。根据归纳假设, 得到 $|2^A| = 2 \cdot 2^n = 2^{n+1}$, 这正是我们要证明的。 \diamond

下面用归纳法证明第二个基本原理——鸽巢原理。

鸽巢原理 设 A 和 B 是两个有穷集合且 $|A| > |B|$, 则不存在从 A 到 B 的一对一的函数。

换句话说, 如果我们企图把 A 的元素(“鸽子”)与 B 的元素(“鸽巢”)配对, 迟早不得不把一只以上鸽子放入一个巢里。

证: 基本步骤。 假设 $|B| = 0$, 即 $B = \emptyset$, 则没有任何从 A 到 B 的函数, 更不用说一对一

的函数。

归纳假设。假设如果 $|A| > |B|$, $|B| \leq n$ 和 $f: A \rightarrow B$, 这里 $n \geq 0$, 则 f 不是一对一的。

归纳步骤。设 $f: A \rightarrow B$, $|A| > |B| = n+1$ 。取一个 $a \in A$ (因为 $|A| > |B| = n+1 \geq 1$, A 非空, 故总能取到)。如果有 A 的另一个元素 a' 使得 $f(a) = f(a')$, 则显然 f 不是一对一的, 证明结束。因此, 假设 a 是被 f 映射到 $f(a)$ 的唯一元素。现在考虑集合 $A - \{a\}$ 和 $B - \{f(a)\}$, 以及从 $A - \{a\}$ 到 $B - \{f(a)\}$ 的函数 g 。在 $A - \{a\}$ 上 g 与 f 相同。因为 $B - \{f(a)\}$ 有 n 个元素, $|A - \{a\}| = |A| - 1 > |B| - 1 = |B - \{f(a)\}|$ 。根据归纳假设, $A - \{a\}$ 有两个不同的元素被 g (因而也被 f) 映射到 $B - \{f(a)\}$ 中的同一个元素, 从而 f 不是一对一的。 ■

在很多各种各样的证明中要用到这个简单的事实。这里只给出一个简单的应用, 在后面的各章当用到它时会指出来。

定理 1.5.1 设 R 是有穷集合 A 上的二元关系, $a, b \in A$ 。如果在 R 中有一条从 a 到 b 的通路, 则有一条长度不超过 $|A|$ 的通路。

证: 假设 (a_1, a_2, \dots, a_n) 是从 $a_1 = a$ 到 $a_n = b$ 的最短的通路, 即长度最小的通路, 又假设 $n > |A|$ 。由鸽巢原理, A 有一个元素重复出现在这条通路上, 比如说 $a_i = a_j, 1 \leq i < j \leq n$ 。于是, $(a_1, a_2, \dots, a_i, a_{j+1}, \dots, a_n)$ 是一条更短的从 a 到 b 的通路, 这与假设 (a_1, a_2, \dots, a_n) 是从 a 到 b 的最短通路矛盾。 ■

最后, 我们介绍第三个基本的证明技术——对角化原理。虽然这个原理在数学中不像前面讨论的两个原理使用得那样广泛, 但它似乎特别适合证明计算理论中的某些重要结果。

对角化原理 设 R 是集合 A 上的二元关系, $D = \{a: a \in A \text{ 且 } (a, a) \notin R\}$ 称作 R 的对角线集合。对于每一个 $a \in A$, 令 $R_a = \{b: b \in A \text{ 且 } (a, b) \in R\}$, 则 D 与每一个 R_a 都不同。

如果 A 是有穷集合, 则可以把 R 画成一个正方形阵列。用 A 的元素标记行和列; 正好当 $(a, b) \in R$ 时, 在行标记 a 和列标记 b 的方格内画一个叉。对角线集合 D 对应主对角线上的方格序列的补, 把有叉的换成没有叉的, 把没有叉的换成有叉的。集合 R_a 对应方阵的行。于是, 可以把对角化原理叙述成: 对角线的补与每一行都不同。

例 1.5.3 考虑关系 $R = \{(a, b), (a, d), (b, b), (b, c), (c, c), (d, b), (d, c), (d, e), (d, f), (e, e), (e, f), (f, a), (f, c), (f, d), (f, e)\}$ 。注意: $R_a = \{b, d\}$, $R_b = \{b, c\}$, $R_c = \{c\}$, $R_d = \{b, c, e, f\}$, $R_e = \{e, f\}$ 以及 $R_f = \{a, c, d, e\}$ 。总之, 可以把 R 画成

	a	b	c	d	e	f
a		×		×		
b		×	×			
c			×			
d		×	×		×	×
e					×	×
f	×		×	×	×	

对角线上的方格序列是

	×	×		×	
--	---	---	--	---	--

它的补是

×			×		×
---	--	--	---	--	---

这对应对角线集合 $D = \{a, d, f\}$ 。 D 确实与方阵的每一行都不同。根据 D 的构造,它与第一行在第一个位置不同,与第二行在第二个位置不同,……。 ◇

对角化原理对于无穷集合也成立。理由是相同的,关于 a 是否是元素的问题,对角线集合与集合 R_a 总是不同的。因此对于任意的 a , D 不可能与 R_a 相同。

用 Georg Cantor (1845—1918) 的一个经典的定理说明对角化方法的使用。

定理 1.5.2 集合 2^N 是不可数的

证:假设 2^N 是可数无穷的。即,假设能把 2^N 的所有成员枚举成

$$2^N = \{R_0, R_1, R_2, \dots\}$$

(注意,考虑关系 $R = \{(i, j) : j \in R_i\}$, 这些 R_i 是对角化原理叙述中的集合 R_a 。)现在考虑集合

$$D = \{n \in N : n \notin R_n\}$$

(D 是对角线集合。) D 是一个自然数的集合,因此应该出现在枚举 $\{R_0, R_1, R_2, \dots\}$ 中。但是, D 不可能是 R_0 , 因为 D 与 R_0 在是否包含 0 的问题上不同 (D 包含 0 当且仅当 R_0 不包含 0); D 不可能是 R_1 , 因为 D 与 R_1 在是否包含 1 的问题上不同,等等。矛盾。

重新更形式化地叙述这个论证。假设对于某个 $k \geq 0$, $D = R_k$ (因为 D 是一个自然数的集合,又假设 $\{R_0, R_1, R_2, \dots\}$ 枚举 N 的所有子集,因此这样的 k 一定存在)。问是否 $k \in R_k$, 得到矛盾:

(a) 假设 $k \in R_k$ 。因为 $D = \{n \in N : n \notin R_n\}$, 故 $k \notin D$ 。而 $D = R_k$, 矛盾。

(b) 假设 $k \notin R_k$, 则 $k \in D$ 。而 $D = R_k$, 故 $k \in R_k$, 也矛盾。

我们从假设 2^N 是可数无穷的开始,经过无懈可击的严格数学推理得出这个矛盾,只能得出结论:这个假设不成立。因此, 2^N 是不可数的。 ■

用类似的方法可以证明区间 $[0, 1]$ 上全体实数的集合是不可数的。见习题 1.5.11。

习 题

1.5.1 用归纳法证明

$$1 \cdot 2 \cdot 3 + 2 \cdot 3 \cdot 4 + \dots + n(n+1)(n+2) = \frac{1}{4}n(n+1)(n+2)(n+3)$$

1.5.2 用归纳法证明:对于所有的 $n \geq 0$, $n^4 - 4n^2$ 能被 3 整除。

1.5.3 指出下述“证明”中的错误。该证明声称:所有马的颜色都相同。

对马匹数作归纳证明。

基本步骤。只有一匹马。于是显然所有的马是一种颜色。

归纳假设。任何数量不超过 n 的一群马都是同一种颜色。

归纳步骤。考虑 $n+1$ 匹马。牵走一匹马。根据归纳假设,留下的所有马是同一种

颜色。现在把牵走的马牵回来、再牵走另一匹马。留下的所有马也是同一种颜色。于是,所有的马都与没有牵走过的马的颜色相同,从而是同一种颜色。

- 1.5.4 证明:设 A 和 B 是任意两个有穷集合,则有 $|B|^{|A|}$ 个从 A 到 B 的函数。
- 1.5.5 用归纳法证明:每一个非空有穷集合上的偏序至少有一个最小元。如果取消有穷性的条件,这条陈述一定成立吗?
- 1.5.6 证明:在任意一群至少有两个人的人群中,至少有两个人在这群人中相识的人数相同。(利用鸽巢原理。)
- 1.5.7 假如我们企图用完全类似定理 1.5.2 证明的论证来证明: \mathbb{N} 的所有有穷子集的集合是不可数的。错在哪里?
- 1.5.8 举例说明两个可数无穷集合的交可能是有穷的或者可数无穷的,两个不可数集合的交可能是有穷的、可数无穷的或者不可数的。
- 1.5.9 证明:一个不可数集合与一个可数集合的差是不可数的。
- 1.5.10 证明:设 S 是任一集合,则存在 S 到 2^S 的一对一的函数,但不存在 2^S 到 S 的一对一的函数。
- 1.5.11 证明:区间 $[0,1]$ 上所有实数的集合是不可数的。(提示:众所周知 $[0,1]$ 上的每一个实数都可以用二进制表示写成 0 和 1 的无穷序列,如 $0.0110011100000\cdots$ 。假设能够枚举这些序列,用“翻转”第 i 个序列的第 i 位生成一个“对角线”序列。)

1.6 闭包与算法

考虑图 1-9(a)和(b)中的两个有向图 R 和 R^* 。 R^* 包含 R 并且是自反的和传递的(而 R 都不是)。事实上,容易看出 R^* 是具有这些性质(即,包含 R 并且是自反的和传递的)的最小的有向图,这里“最小”的意思是边数最少的。基于这个原因, R^* 叫做 R 的自反传递闭包。下面形式地定义这个很有用的概念。

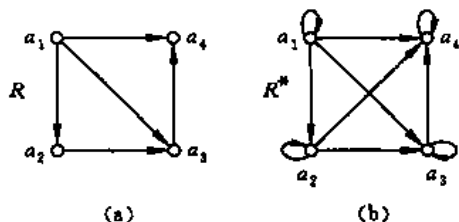


图 1-9

定义 1.6.1 设 $R \subseteq A^2$ 是集合 A 上的有向图。 R 的自反传递闭包是关系

$$R^* = \{(a, b) : a, b \in A \text{ 且在 } R \text{ 中有从 } a \text{ 到 } b \text{ 的通路}\}$$

注意在定义 1.6.1 中给出的“从下面来的”定义与我们引入自反传递闭包时“从上面来的”非形式定义(包含 R 的自反和传递的最小关系)之间有趣的显著差别。这两个定义的等价性在直观上大概是明显的。在本节剩余的部分更多地是研究“从上面来的定义”和它为何总对应于“从下面来的”定义的理由。

1.6.1 算法

定义 1.6.1 直接提供一个算法,任给某个有穷集合 $A = \{a_1, a_2, \cdots, a_n\}$ 上的二元关系 R , 计算 R 的自反传递闭包 R^* :

initially $R^* := \emptyset$

for $i=1, \dots, n$ do

for 每一个 i 元组 $(b_1, \dots, b_i) \in A^i$ do

if (b_1, \dots, b_i) 是 R 中的一条通路 then 把 (b_1, b_i) 加入 R^* 。

在某种意义上,严格地研究算法是全书的全部内容。但是如果在这里给出算法的形式定义,就会把我们应该讲的这个美妙的故事弄糟了。在介绍算法的一般的形式模型之前,我们将非形式地和多少有些不严格地处理算法,逐渐增加对它的了解和熟悉。这样一来,当形式地定义算法时,你会觉得那个定义确实正确地抓住了算法这个重要概念的本质。因此,本节只直观地处理算法,并非形式地介绍它们的分析。

幸运的是,讲述一个算法是容易的。上面描述的计算 R^* 的过程是一个详细明确的指令序列,它产生一个称作 R^* 的结果。它由一些基本的步骤组成,有理由假设这些步骤是容易实现的。它们是:初始化 R^* 为 \emptyset , 把一个新元素加入 R^* , 检查是否 $(b_j, b_{j+1}) \in R$ 。第三种步骤在算法的最后一行检查 (b_1, \dots, b_i) 是否是 R 的一条通路时要使用 $i-1$ 次。我们假设这个算法以某种方式直接对 A 和 R 的元素进行操作,因而不需要规定这些集合和关系用于算法操作的表示方式。

下面证明该算法计算出来的关系 R^* 确实是 R 的自反传递闭包(即,算法是正确的)。理由是:这个算法只是直接地履行定义 1.6.1。 R^* 开始时是空集,算法把 A 中所有由 R 中通路连接的元素对加入 R^* 。按照长度递增的顺序,一条一条地检查所有可能的通路。根据定理 1.5.1,如果 A 的两个顶点之间有一条通路,则有一条连接它们的长度不超过 n 的通路。所以,算法在检查完长度为 n 的序列后停止。

因此,显然这个算法最后停止时给出正确的答案。还有一个问题是算法将在多少步之后停止? 这是我们在本书所关心的最重要、最贴切的问题。本书的最后将要给出一套完整的理论,用来解决如何计算这类问题的答案和什么时候认为结果是满意的。但是,现在我们非形式地进行。

我们需要有一种方法表示当提供一个关系 R 作为输入时,算法需要多少基本步骤,即多少“时间”。由于有理由预计输入的 R 愈大,算法花费的时间愈多,因而答案将与输入 R 的大小有关——更具体地说,将与集合 A 中的元素个数 n 有关。于是,我们寻找一个函数 $f: \mathbb{N} \rightarrow \mathbb{N}$ 使得,当给算法提供一个二元关系 $R \subseteq A \times A$ 时,它在至多 $f(n)$ 步之后停止,其中 $n = |A|$ 。在算法分析中的典型做法是,允许 $f(n)$ 是一个粗略的过头估计,只要它具有正确的增长率。增长率是下一个论题。

1.6.2 函数的增长率

在下述三个从自然数集到自然数集的函数中哪一个最大?

$$f(n) = 1\,000\,000n, \quad g(n) = 10n^2, \quad h(n) = 2^n$$

虽然对于大约到一打为止 n 的所有值,有 $f(n) > g(n) > h(n)$,但是正确的顺序正好相反。这在直觉上应该是清楚的——如果 n 取得足够大,则最终会有 $f(n) < g(n) < h(n)$ 。在这一小节将要给出几个概念,它们对于按照产生大函数值的最终潜力排列函数是必需的。

定义 1.6.2 设 $f: \mathbb{N} \rightarrow \mathbb{N}$ 是一个从自然数集到自然数集的函数。 f 的阶,记作 $\mathcal{O}(f)$,是所有具有下述性质的函数 $g: \mathbb{N} \rightarrow \mathbb{N}$ 的集合:存在正自然数 $c > 0$ 和 $d > 0$ 使得对

于所有的 $n \in \mathbb{N}$, $g(n) \leq c \cdot f(n) + d$ 。如果这个不等式对所有的 n 成立, 则称以常数 c 和 d 有 $g(n) \in \mathcal{O}(f)$ 。

设两个函数 $f, g: \mathbb{N} \rightarrow \mathbb{N}$ 。如果 $f \in \mathcal{O}(g)$ 且 $g \in \mathcal{O}(f)$, 则记作 $f \sim g$ 。显然, \sim 是定义在函数上的等价关系。它是自反的(因为总是以常数 1 和 0 有 $f \in \mathcal{O}(f)$) 和对称的(因为在 \sim 的定义中 f 和 g 的角色是完全可以交换的)。最后, 它是传递的。这是因为假设以常数 c 和 d 有 $f \in \mathcal{O}(g)$, 以常数 c' 和 d' 有 $g \in \mathcal{O}(h)$, 则对于所有的 n ,

$$\begin{aligned} f(n) &\leq c \cdot g(n) + d \leq c \cdot (c' \cdot h(n) + d') + d \\ &= (c \cdot c')h(n) + (d + c \cdot d') \end{aligned}$$

因此, 以常数 $c \cdot c'$ 和 $d + c \cdot d'$ 有 $f \in \mathcal{O}(h)$ 。

于是, 从自然数集到自然数集的全体函数被 \sim 划分成等价类。 f 关于 \sim 的等价类叫做 f 的增长率。

例 1.6.1 考虑多项式 $f(n) = 31n^2 + 17n + 3$ 。我们说 $f(n) \in \mathcal{O}(n^2)$ 。注意 $n^2 \geq n$, 故 $f(n) \leq 48n^2 + 3$ 。于是, 以常数 48 和 3 有 $f(n) \in \mathcal{O}(n^2)$ 。当然, 也有 $n^2 \in \mathcal{O}(f)$, 以常数 1 和 0。因此, $n^2 \sim 31n^2 + 17n + 3$, 这两个函数具有相同的增长率。

类似地, 对于所有非负系数的 d 次多项式

$$f(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0$$

其中所有的 $a_i \geq 0$ 且 $a_d > 0$, 容易证明 $f(n) \in \mathcal{O}(n^d)$, 以常数 $\sum_{i=1}^d a_i$ 和 a_0 。所有次数相同的多项式具有相同的增长率。

现在考虑两个次数不同的多项式, 它们也有相同的增长率吗? 回答是否定的。由于所有次数相同的多项式有相同的增长率, 只需要考虑两个表达式最简单的多项式 n^i 和 n^j , 这里 $0 < i < j$ 。显然, 以常数 1 和 0 有 $n^i \in \mathcal{O}(n^j)$ 。要证明 $n^j \notin \mathcal{O}(n^i)$ 。

假设不然, 以常数 c 和 d 有 $n^j \in \mathcal{O}(n^i)$ 。即, 对所有 $n \in \mathbb{N}$, $n^j \leq cn^i + d$ 。令 $n = c + d$, 容易发现矛盾:

$$c(c+d)^i + d < (c+d)^{i+1} \leq (c+d)^j$$

综上所述, 对于任意两个多项式 f 和 g , 如果它们的次数相同, 则 $f \sim g$ 。如果 g 的次数大于 f 的次数, 则 $f \in \mathcal{O}(g)$, 但 $g \notin \mathcal{O}(f)$, 即 g 的增长率比 f 的高。 \diamond

例 1.6.2 前一个例子表明多项式的次数决定了它的增长率。次数愈大, 增长率愈高。但是, 还有增长率比任何多项式都高的函数。一个简单的例子是指数函数 2^n 。

先证明对于所有的 $n \in \mathbb{N}$, $n \leq 2^n$ 。对 n 作归纳, 当 $n=0$ 时结论确实成立。于是, 假设对于所有小于等于 n 的自然数结论成立, 则有

$$n+1 \leq 2^n + 1 \leq 2^n + 2^n = 2^{n+1}$$

其中, 第一个不等式用到归纳假设, 第二个不等式用到对所有的 n , $1 \leq 2^n$ 。

现在把这个简单的事实推广到所有的多项式。要证对于任意的 $i \geq 1$, $n^i \in \mathcal{O}(2^n)$, 即对于适当的常数 c 和 d

$$n^i \leq c2^n + d \tag{1}$$

取 $c = (2i)^i$ 和 $d = (i^2)^i$ 。分两种情况: 如果 $n \leq i^2$, 则由于 $n^i \leq d$, 不等式(1)成立。如果 $n > i^2$, 则可以证明 $n^i \leq c2^n$, 从而不等式(1)也成立。事实上, 令 m 是 n 除以 i 的商, 即 m 是

使得 $im \leq n < im + i$ 的唯一整数。于是,

$$\begin{aligned} n^i &\leq (im + i)^i && \text{根据 } m \text{ 的定义} \\ &= i^i(m + 1)^i \\ &\leq i^i(2^{m+1})^i && \text{把 } n = m + 1 \text{ 代入不等式 } n < 2^n \\ &\leq c2^{mi} && \text{因为 } c = (2i)^i \\ &\leq c2^n && \text{根据 } m \text{ 的定义} \end{aligned}$$

于是,任何多项式的增长率都不比 2^n 的更快。多项式能与 2^n 有相同的增长率吗? 如果某个多项式与 2^n 具有相同的增长率,则任何次数更大的多项式也与 2^n 具有相同的增长率。而在前一个例子中已经看到,不同次数的多项式具有不同的增长率,矛盾。因此, 2^n 的增长率比任何多项式的高。另外一些指数函数,比如 $5^n, n^n, n!, 2^{n^2}$, 或者更厉害的 2^{2^n} , 具有更高的增长率。◇

1.6.3 算法分析

多项式的和指数的增长率自然地出现在算法分析中。例如,让我们粗略地估计在本节开始介绍的自反传递闭包算法所需要的步数。

算法检查每一个长度不超过 n 的序列 (b_1, \dots, b_i) 是否是 R 中的通路,如果回答“是”则把 (b_1, b_i) 加入 R^* 。每一个这样的运算过程一定能用 n 个或少于 n 个“基本操作”(检查是否 $(a, b) \in R$, 把 (a, b) 加入 R^*) 实现。于是,总的操作数可以不多于

$$n \cdot (1 + n + n^2 + \dots + n^n)$$

它在 $\mathcal{O}(n^{n+1})$ 中。这就给出该算法的时间需求的增长率为 $\mathcal{O}(n^{n+1})$ 。

这个结果是相当令人失望的,因为这是一个指数函数,其增长率甚至比 2^n 还要高。这个算法的效率太低。

于是,产生这样的问题:是否有更快的计算自反传递闭包的算法? 考虑下述算法:

```
initially  $R^* := R \cup \{(a_i, a_i) : a_i \in A\}$ 
while 有 3 个元素  $a_i, a_j, a_k \in A$  使
 $(a_i, a_j), (a_j, a_k) \in R^*$ , 但  $(a_i, a_k) \notin R^*$  do
    把  $(a_i, a_k)$  加入  $R^*$ 。
```

这个算法大概比前一个算法还要更自然和更直观一些。它开始的时候把 R 中的所有有序对和所有 (a_i, a_i) 形式的有序对加入 R^* , 这保证 R^* 包含 R 并且是自反的。然后它反复寻找违反传递性的地方。这种寻找总该能够以某种有条理的方式进行,在算法中没有精确地描述这个方式。比如说,检查 a_i 的所有值,对 a_i 的每一值检查 a_j 的所有值,最后对 a_i 和 a_j 的每一对值检查 a_k 的所有值。如果发现一个违反传递性的地方,把一个适当的有序对加入 R^* 纠正这个违反,并且重新从头开始寻找。如果在某一步检查了所有的三元组并且没有发现违反传递性的地方,则算法结束。

当算法结束时, R^* 肯定包含 R 并且是自反的。由于在 while 循环的最后一次迭代中没有发现违反传递性,故 R^* 一定是传递的。此外, R^* 是具有所有这三条性质的最小的二元关系(从而它是所求的 R 的自反传递闭包)。为了看到这一点,假设 R^* 有一个真子集 R_0 包含 R 且是自反和传递的。考虑第一个不属于 R_0 , 但仍被算法加入 R^* 的有序对。它不可

能在 R 中或者是 (a_i, a_i) 形式的, 因为所有这样的有序对都在 R_0 中。于是, 它一定是一个有序对 (a_i, a_k) 使得在那一步 (a_i, a_j) 和 (a_j, a_k) 都属于 R^* 。由于 (a_i, a_k) 是第一个在 R^* 中且不在 R_0 中的有序对, 故 (a_i, a_j) 和 (a_j, a_k) 都属于 R_0 。而根据假设, R_0 是一个传递关系, 它一定也包含 (a_i, a_k) , 矛盾。因此, 所得到的结果 R^* 是 R 的自反传递闭包, 算法是正确的。

但是, 算法什么时候结束? 回答是至多在 n^2 次 while 循环迭代之后。这是因为每一个成功的迭代把一个原先不在 R^* 中的有序对 (a_i, a_k) 加入到 R^* 中, 而至多有 n^2 个要加入的有序对。因此, 算法至多在 n^2 次迭代之后结束。因为每一次迭代可以在 $O(n^3)$ 时间内实现 (需要搜索 A 的元素的所有三元组), 所以这个新算法的复杂度是 $O(n^2 \times n^3) = O(n^5)$ 。这是多项式的增长率, 因而极大地改进了指数的第一个算法。

例 1.6.3 让我们来看看这个新算法在图 1-10 中如何运行。开始在第一行把所有的环加到图上。假设搜索违反传递性的三元组 (a_i, a_j, a_k) 按照“自然的”顺序进行。

$(a_1, a_1, a_1), (a_1, a_1, a_2), \dots, (a_1, a_1, a_4), (a_1, a_2, a_1), (a_1, a_2, a_2), \dots,$
 (a_4, a_4, a_4)

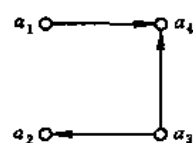


图 1-10

这样, 第一个被发现的违反传递性的三元组是 (a_1, a_4, a_3) , 从而加入边 (a_1, a_3) 。下面从头开始检查所有的三元组, 因为加入 (a_1, a_3) 可能在前面的迭代已检查过的三元组中产生违反传递性。事实上, 发现的下一个违反传递性的三元组是 (a_1, a_3, a_2) , 从而加入 (a_1, a_2) 。再一次从头开始搜索, 这一次发现 (a_4, a_3, a_2) 违反传递性, 从而加入边 (a_4, a_2) 。在下次迭代中, 检查所有的三元组没有发现违反传递性, 因此得出结论: 已经计算出这个图的自反传递闭包。◇

这个例子说明了这个算法效率不高 (表现在它的复杂度为太高的 n^5 增长率) 的原因是: 必须重复检查一个三元组 (a_i, a_j, a_k) 是否违反传递性, 因为新插入的有序对可能在已经检查过的三元组产生新的违反传递性。

能做得更好吗? 具体地说, 能不能有一种方法排列三元组使得新加入的有序对绝不会在已经检查过的三元组再产生违反传递性? 这种排序将会产生一个 $O(n^3)$ 算法, 因为每一个三元组必须检查一次且只检查一次。

原来这样的排序是可能的, 按照 j (中间的下标!) 递增的顺序排列所有待搜索的三元组 (a_i, a_j, a_k) 。首先, 有计划地寻找 (a_i, a_1, a_k) 形式的违反传递性的三元组; 当找到一个之后, 加入适当的有序对纠正这个违反, 并且从暂停的地方开始继续搜索。检查完所有 (a_i, a_1, a_k) 形式的三元组之后, 检查所有 (a_i, a_2, a_k) 形式的三元组, 然后是 (a_i, a_3, a_k) , 等等。在每一组内三元组的检查顺序是不重要的。检查完最后一组, 即所有 (a_i, a_n, a_k) 形式的三元组后, 得到自反传递闭包, 计算停止。整个算法如下:

```
initially  $R^* := R \cup \{(a_i, a_i) : a_i \in A\}$ 
for  $j=1, 2, \dots, n$  do
  for  $i=1, 2, \dots, n$  与  $k=1, 2, \dots, n$  do
    if  $(a_i, a_j), (a_j, a_k) \in R^*$ , 但  $(a_i, a_k) \notin R^*$ 
    then 把  $(a_i, a_k)$  加入  $R^*$ 。
```

这个想法为什么正确? 考虑从 a_{i_0} 到 a_{i_k} 的一条通路 $(a_{i_0}, a_{i_1}, \dots, a_{i_{k-1}}, a_{i_k})$, 其中对所有的 $j, 1 \leq j \leq k$. 定义这条通路的秩是 i_1, \dots, i_{k-1} 中的最大整数, 即中间顶点的最大下标。平凡通路(边和环)的秩等于 0, 因为它们没有中间顶点。

利用这个术语可以论证上述算法的正确性。具体地说, 可以证明下述陈述: 对于每一个 $j=0, \dots, n$, 在执行第 j 次外循环后 R^* 包含所有在 R 中有从 a_i 到 a_k 且秩小于等于 j 的通路的有序对 (a_i, a_k) 。注意, 由于所有通路的秩不超过 n , 因此所有由通路连接的有序对最终都被加入到 R^* 中。

对 j 做归纳来证明这个陈述。当 $j=0$ 时结论确实成立。此时还没有做任何迭代, 相应地 R^* 只包含平凡通路(秩为 0)连接的有序对。假设结论对于所有的 $j \leq m < n$ 成立, 考虑第 $m+1$ 次迭代。要证明第 $m+1$ 次迭代恰好把那些在 R 中由秩为 $m+1$ (即, 中间下标最大的顶点为 a_{m+1}) 的通路连接的有序对加入 R^* 。如果两个顶点 a_i 和 a_k 被这样一条通路连接, 那么它们一定被一条 a_{m+1} 恰好出现一次, 而其余中间顶点的下标都小于等于 m 的通路连接, 这样的通路由一条从 a_i 到 a_{m+1} 秩小于等于 m 的通路接着一条从 a_{m+1} 到 a_k 秩小于等于 m 的通路组成。(如果 a_{m+1} 出现多次, 则删去第一个和最后一个 a_{m+1} 之间的部分。)根据归纳假设, 有序对 (a_i, a_{m+1}) 和 (a_{m+1}, a_k) 此时都一定在 R^* 中。因此, 算法会发现 $(a_i, a_{m+1}), (a_{m+1}, a_k) \in R^*$, 但 $(a_i, a_k) \notin R^*$, 把有序对 (a_i, a_k) 加入 R^* 。反过来, 第 $m+1$ 次迭代只把由秩恰好为 $m+1$ 的通路连接的有序对 (a_i, a_k) 加入 R^* 。所以, 算法是正确的。

例 1.6.3(续) 如果把这个算法用于图 1-10 中的图, 当 $j=1$ 和 $j=2$ 时没有边被加入, 当 $j=3$ 时加入边 (a_4, a_2) , 当 $j=4$ 时加入边 (a_1, a_3) 和 (a_1, a_2) 。◇

1.6.4 封闭性与闭包

关系的传递闭包只是从较小的集合(或关系)出发定义更大的集合(或关系)的一种重要方式的一个例子。

定义 1.6.3 设 D 是一个集合, $R \subseteq D^{n+1}$ 是 D 上的一个 $n+1$ 元关系, 其中 $n \geq 0$ 。又设 B 是 D 的子集。如果只要 $b_1, \dots, b_n \in B$ 且 $(b_1, \dots, b_n, b_{n+1}) \in R$, 就有 $b_{n+1} \in B$, 则称 B 在 R 下是封闭的。任何“集合 B 在关系 R_1, R_2, \dots, R_m 下是封闭的”形式的性质称作 B 的封闭性。

例 1.6.4 有后代的人组成的集合在下述关系下是封闭的:

$$\{(a, b); a \text{ 和 } b \text{ 是两个人且 } b \text{ 是 } a \text{ 的父亲或母亲}\}$$

因为有后代的人的父母亲也是有后代的人。◇

例 1.6.5 设 S 是一个固定的集合。如果 $A \subseteq S$, 则称 S 满足有关 A 的包含性。任何包含性是一种封闭性, 只要把关系 R 取作一元关系 $\{(a); a \in A\}$ (注意应把定义 1.6.3 中的 n 取作 0)。◇

例 1.6.6 有时会说集合 $A \subseteq D$ 在函数 $f: D^k \rightarrow D$ 下是封闭的。它的含义应是不难理解的, 因为函数是特殊类型的关系。例如, 可以说自然数集合在加法下是封闭的。它的意思是, 对于任意的 $m, n \in \mathbb{N}$, 也有 $m+n \in \mathbb{N}$ —— $(m, n, m+n)$ 是自然数集合上的“加法关系”中的一个三元组。 \mathbb{N} 在乘法下也是封闭的, 但是在减法下不封闭。◇

例 1.6.7 因为关系是一个集合, 所以可以谈论一个关系在一个或多个其他关系下

是封闭的。设 D 是一个集合, Q 是 D^2 上的三元关系(即, $(D \times D)^3$ 的子集)如下

$$Q = \{((a, b), (b, c), (a, c)); a, b, c \in D\}$$

于是, 一个关系 $R \subseteq D \times D$ 在 Q 下是封闭的当且仅当它是传递的。因此, 传递性是一种封闭性。另外, 自反性也是一种封闭性, 因为它是有关集合 $\{(d, d); d \in D\}$ 的包含性。◇

一种通用的数学构造类型是从一个集合 A 到包含 A 且具有性质 P 的极小的集合 B 。“极小的集合 B ”的意思是“集合 B 不真包含任何集合 B' 使得 B' 也包含 A 且具有性质 P ”。必须注意, 当使用这种形式的定义时, 集合 B 应该是良定义的, 即只存在一个这样的极小集合。因为集合的集合可能有多个极小元或者根本没有极小元, B 是否是良定义的与性质 P 的性质有关。例如, 设 P 是性质“包含 b 或 c 作为元素”, 而 $A = \{a\}$, 则 B 不是良定义的。因为 $\{a, b\}$ 和 $\{a, c\}$ 都是以 A 为子集且具有性质 P 的极小集合。

然而, 正如下述结果所保证的那样, 如果 P 是一种封闭性, 则集合 B 总是良定义的。

定理 1.6.1 设 P 是由集合 D 上的关系定义的封闭性, A 是 D 的子集, 则有唯一的包含 A 且具有性质 P 的极小集合 B 。

证: 考虑 D 的在 R_1, \dots, R_m 下封闭且包含 A 的所有子集组成的集合。把这个集合的集合叫做 S 。要证明 S 有唯一的极小元 B 。容易看到 S 是非空的, 因为它包含“全集” D 。 D 本身在每一个 R_i 下当然是封闭的并且确实包含 A 。

考虑 S 中所有集合的交

$$B = \bigcap S$$

首先, B 是良定义的, 因为它是集合的非空集合的交。还容易看到它包含 A 。因为 S 中的所有集合都包含 A 。下面证明 B 在所有 R_i 下是封闭的。假设 $a_1, \dots, a_{n-1} \in B$ 且 $(a_1, \dots, a_{n-1}, a_n) \in R_i$ 。因为 B 是 S 中所有集合的交, 故 S 中所有的集合包含 a_1, \dots, a_{n-1} 。又因为 S 中的所有集合在 R_i 下封闭, 故它们也都包含 a_n 。因此, B 一定包含 a_n , 从而 B 在 R_i 下是封闭的。最后, B 是极小的, 不可能有具有这些性质(包含 A 且在所有的 R_i 下封闭)的真子集 B' 。因为如果 B' 具有这些性质, 则 B' 是 S 的一个成员, 从而 B' 包含 B 。■

称定理 1.6.1 中的 B 是 A 在关系 R_1, \dots, R_m 下的闭包。

例 1.6.8 你的所有先辈(在这里约定每一个人是自己的先辈)组成的集合是只包含你一个人的单元集在关系 $\{(a, b); a \text{ 和 } b \text{ 是两个人且 } b \text{ 是 } a \text{ 的父亲或母亲}\}$ 下的闭包。◇

例 1.6.9 自然数集合 N 是集合 $\{0, 1\}$ 在加法下的闭包。 N 在加法和乘法下是封闭的, 但在减法下不是封闭的。整数集合(正的、负的和零)是 N 在减法下的闭包。◇

例 1.6.10 有穷集合 A 上的二元关系 R 的自反传递闭包定义为

$$R^* = \{(a, b); \text{在 } R \text{ 中有从 } a \text{ 到 } b \text{ 的通路}\}$$

(见定义 1.6.1)。它完全应该享有这个名字, 原来它是 R 在自反性和传递性下的闭包, 这两个性质都是封闭性。

首先, R^* 是自反的和传递的, 因为对于任意的元素 a , 有从 a 到 a 的平凡通路(a); 如果有从 a 到 b 的通路和从 b 到 c 的通路, 则有从 a 到 c 的通路。其次, 显然 $R \subseteq R^*$, 因为只要 $(a, b) \in R$ 就有从 a 到 b 的通路。

最后, R^* 是极小的。为此令 $(a, b) \in R^*$ 。由于 $(a, b) \in R^*$, 故有从 a 到 b 的通路($a = a_1, \dots, a_k = b$)。对 k 做归纳可以证明 (a, b) 一定属于包含 R 的任何自反和传递的二元

关系。

自反传递闭包只是几种可能的闭包中的一种。例如,关系 R 的传递闭包,记作 R^+ ,是所有满足下述条件的 (a, b) 的集合:在 R 中有从 a 到 b 的非平凡的通路。它不一定是自反的。任何关系的自反、对称和传递的闭包(没有专用符号)总是一个等价关系。下面将要证明,有计算这些闭包的多项式算法。◇

任何有穷集合上的封闭性能够在多项式时间内计算!假设给定有穷集合 D 上各种元数的关系 $R_1 \subseteq D^{r_1}, \dots, R_k \subseteq D^{r_k}$ 和集合 $A \subseteq D$ 。要求计算 A 在 R_1, \dots, R_k 下的闭包 A^* 。这可以在多项式时间内实现,只需直接推广在上一小节为传递闭包问题设计的 $\mathcal{O}(n^5)$ 算法:

initially $A^* := A$

while 存在下标 $i (1 \leq i \leq k)$, r_i 个元素 $a_{j_1}, \dots,$

$a_{j_{r_i-1}} \in A^*$ 和 $a_{j_{r_i}} \in D - A^*$ 使得 $(a_{j_1}, \dots, a_{j_{r_i}}) \in R_i$ do

把 $a_{j_{r_i}}$ 加入 A^* 。

直接延伸前面关于传递闭包算法的论证可以证明,上述算法是正确的,在 $\mathcal{O}(n^{r+1})$ 步之后结束,其中 $n = |D|$, r 是 r_1, \dots, r_k 中的最大整数。我们把它留作一个习题(习题 1.6.9)。因此,任意给定的有穷集合在用给定的固定元数的关系定义的任何封闭性质下的闭包都能够在多项式时间内计算出来。事实上,在第 7 章将要证明一个非常有趣的逆命题:能够把任何多项式时间算法描述成计算集合在某个固定元数的关系下的闭包。换句话说,上述闭包的多项式算法是所有多项式算法之母。

习 题

- 1.6.1 下述集合在指定的运算下是封闭的吗?如果不是,各自的闭包是什么?
 - (a) 奇整数集合在乘法下。
 - (b) 正整数集合在除法下。
 - (c) 负整数集合在减法下。
 - (d) 负整数集合在乘法下。
 - (e) 奇整数集合在除法下。
- 1.6.2 关系 $R = \{(a, b), (a, c), (a, d), (d, c), (d, e)\}$ 的自反传递闭包 R^* 是什么?画出表示 R^* 的有向图。
- 1.6.3 一个二元关系的对称闭包的传递闭包一定是自反的吗?证明它或者举出一个反例。
- 1.6.4 设 $R \subseteq A \times A$ 是任意一个二元关系。
 - (a) 令 $Q = \{(a, b); a, b \in A \text{ 且在 } R \text{ 中有从 } a \text{ 到 } b \text{ 和从 } b \text{ 到 } a \text{ 的通路}\}$ 。证明 Q 是 A 上的等价关系。
 - (b) 令 Π 是 A 对应等价关系 Q 的划分。令 \mathcal{R} 是关系 $\{(S, T); S, T \in \Pi \text{ 且在 } R \text{ 中有从 } S \text{ 的某个成员到 } T \text{ 的某个成员的通路}\}$ 。证明 \mathcal{R} 是 Π 上的偏序。
- 1.6.5 给出一个二元关系,它不是自反的,但它的传递闭包是自反的。
- 1.6.6 回想在关于增长率的小节的开头的三个函数

$$f(n) = 1\,000\,000n, \quad g(n) = 10n^3, \quad h(n) = 2^n$$

对于包含关系 $f(n) \in \mathcal{O}(g(n))$, $f(n) \in \mathcal{O}(h(n))$ 和 $g(n) \in \mathcal{O}(h(n))$ 的适当的常数 c 和 d 是什么? 使得 $f(n) \leq g(n) \leq h(n)$ 的最小整数 n 等于多少?

1.6.7 按照增长率上升的顺序排列下述函数。指出增长率相同的函数:

$$n^2, 2^n, n^{\lceil \log n \rceil}, n!, 4^n, n^n, n^{\lceil \log n \rceil}, 2^{2^n}, 2^{2^n}, 2^{2^{n+1}}$$

1.6.8 关于一个问题你有 5 个算法, 运行时间分别为

$$10^6 n, \quad 10^4 n^2, \quad n^4, \quad 2^n, \quad n!$$

(a) 你的计算机每秒执行 10^8 步。用各个算法在一秒钟内你能够解的最大规模 n 是多少?

(b) 在一天内呢? (假设一天是 10^5 秒。)

(c) 如果你买了一台快 10 倍的计算机, (a) 和 (b) 中的答案如何变化?

1.6.9 证明: 本节最后给出的算法在 $\mathcal{O}(n^r)$ 时间内正确地计算出集合 $A \subseteq D$ 在关系 $R_1 \subseteq D^{r_1}, \dots, R_k \subseteq D^{r_k}$ 下的闭包, 其中 $n = |D|$, r 是元数 r_1, \dots, r_k 中的最大值。(提示: 直接推广关于 $\mathcal{O}(n^5)$ 传递闭包算法的论证。)

1.7 字母表与语言

在上一节我们非形式地研究过的算法有许多含糊不清的地方。例如, 需要存取和修改关系 R 和 R^* , 但没有精确地规定如何表示和存储它们。实际计算时, 在计算机的存储器中这种数据被编码成位的串或其他适合计算机处理的符号的串。因此, 计算理论的数学研究从了解字符串的数学开始。

从字母表的概念开始, 字母表是一个有穷的符号集合。一个自然的例子是罗马字母表 $\{a, b, \dots, z\}$ 。在计算理论中用得最多的字母表是二进制字母表 $\{0, 1\}$ 。事实上, 任何对象都可以是一个字母表的成员。从形式的角度, 一个字母表就是任意的一个有穷集合。但是, 为了简单起见, 我们只把字母、数字和其他常用的字符(如 \$, #)用作符号。

字母表上的字符串是该字母表中的符号的有穷序列。写字符串的时候不再像写其他的序列那样使用圆括号和逗号, 而是直接地把符号并排在一起。例如, *watermelon* 是字母表 $\{a, b, \dots, z\}$ 上的一个字符串, 0111011 是字母表 $\{0, 1\}$ 上的一个字符串。还用自然同构把只有一个符号的字符串等同于这个符号本身。例如, 符号 a 和字符串 a 是一样的。一个字符串可以根本没有符号, 这时把它叫做空串, 记作 ϵ 。通常用 u, v, w, x, y, z 和希腊字母表示字符串。例如, 可以用 w 作为字符串 abc 的名字。当然, 为了避免混淆, 不得把同一个字母既用作符号又用作字符串的名字。字母表 Σ 上所有字符串(包括空串在内)的集合记作 Σ^* 。

一个字符串的长度是它作为序列的长度。例如, 字符串 $acrd$ 的长度是 4。把字符串 w 的长度记作 $|w|$ 。例如, $|101| = 3$, $|\epsilon| = 0$ 。还可以(通过一个自然同构)把字符串 $w \in \Sigma^*$ 看作一个函数 $w: \{1, \dots, |w|\} \rightarrow \Sigma$ 。对于 $1 \leq j \leq |w|$, $w(j)$ 的值是 w 第 j 位上的符号。例如, 如果 $w = \text{accordion}$, 则 $w(3) = w(2) = c$, $w(1) = a$ 。这种观点说清了可能混淆的地方。自然, 在第三位上的 c 和在第二位上的 c 是一样的。但是, 如果需要区分在字符串中不同位

置上的同一个符号,就要把它们看作这个符号的不同出现。也就是说,如果 $w(j)=a$, 则符号 $a \in \Sigma$ 出现在字符串 $w \in \Sigma^*$ 的第 j 位上。

用连接运算可以把同一个字母表上的两个字符串结合成第三个字符串。字符串 x 和 y 的连接是在字符串 x 的后面接上字符串 y , 记作 $x \circ y$, 或简记作 xy 。形式地, $w = x \circ y$ 当且仅当 $|w| = |x| + |y|$, 并且对于 $j = 1, \dots, |x|$, $w(j) = x(j)$; 对于 $j = 1, \dots, |y|$, $w(|x| + j) = y(j)$ 。例如, $01 \circ 001 = 01001$, $beach \circ boy = beachboy$ 。当然, 对于任意的字符串 w , $w \circ e = e \circ w = w$ 。连接运算是可结合的; 对于任意的字符串 w, x 和 y , $(wx)y = w(xy)$ 。字符串 v 是字符串 w 的子串当且仅当存在字符串 x 和 y 使得 $w = xvy$ 。 x 和 y 都可以是 e , 故每一个字符串是它自己的子串。取 $x = w$ 和 $v = y = e$, 看到 e 是每一个字符串的子串。如果对于某个 x , $w = xv$, 则称 v 是 w 的后缀; 如果对于某个 y , $w = vy$, 则称 v 是 w 的前缀。例如, $road$ 是 $roadrunner$ 的前缀, 是 $abroad$ 的后缀, 是这两个字符串及 $broad$ 的子串。在一个字符串中可以多次出现同一个子串。例如, 在 $ababab$ 中 ab 出现三次, $abab$ 出现两次。

对于每一个字符串 w 和每一个自然数 i , 字符串 w^i 定义如下:

$$\begin{aligned} w^0 &= e, \\ w^{i+1} &= w^i \circ w, \quad \text{对每一个 } i \geq 0. \end{aligned}$$

例如, $w^1 = w$, $do^2 = dodo$ 。

这个定义是本书第一个归纳定义, 这是一种很常用的类型。我们已经看到归纳证明, 它们的基本思想是一样的。有一个定义的基础情况, 在这里是当 $i=0$ 时 w^i 的定义。然后, 当对于所有的 $j \leq i$ 都有定义时, 对于 $j=i+1$ 也有定义。在上述例子中, 用 w^i 定义 w^{i+1} 。为了确切地看到定义中的任何情况都能够如何回溯到基础情况, 考虑 do^2 。根据定义(取 $i=1$), $(do)^2 = (do)^1 \circ do$ 。再根据定义(取 $i=0$), $(do)^1 = (do)^0 \circ do$ 。现在运用基础情况 $(do)^0 = e$ 。因此, $(do)^2 = (e \circ do) \circ do = dodo$ 。

字符串 w 的反转是“反过来拼写的” w , 记作 w^R 。例如, $reverse^R = esrever$ 。它的形式定义可以用对字符串长度的归纳给出:

- (1) 如果 w 是长度为 0 的字符串, 则 $w^R = w = e$ 。
- (2) 如果 w 是长度为 $n+1 > 0$ 的字符串, 则对于某个 $a \in \Sigma$, $w = ua$, $w^R = au^R$ 。

我们用这个定义来说明如何根据归纳定义进行归纳证明。证明: 对于任意两个字符串 w 和 x , $(wx)^R = x^R w^R$ 。例如, $(dogcat)^R = (cat)^R (dog)^R = tacgod$ 。对 x 的长度作归纳。

基础步骤。 $|x| = 0$ 。于是, $x = e$, $(wx)^R = (we)^R = w^R = ew^R = e^R w^R = x^R w^R$ 。

归纳假设。 如果 $|x| \leq n$, 则 $(wx)^R = x^R w^R$ 。

归纳步骤。 设 $|x| = n+1$, 则 $x = ua$, 其中 $a \in \Sigma$, $u \in \Sigma^*$ 且 $|u| = n$ 。于是,

$$\begin{aligned} (wx)^R &= (w(ua))^R && \text{因为 } x = ua \\ &= ((wu)a)^R && \text{连接是可结合的} \\ &= a(wu)^R && \text{根据 } (wu)a \text{ 的反转的定义} \\ &= au^R w^R && \text{根据归纳假设} \\ &= (ua)^R w^R && \text{根据 } (ua) \text{ 的反转的定义} \\ &= x^R w^R && \text{因为 } x = ua \end{aligned}$$

现在我们从研究个别的字符串转移到研究有穷的和无穷的字符串集合。正则性刻画了即将要研究的简单的计算机模型,它们以这种方式处理许多不同的字符串,因此重要的是要首先了解描述字符串集合和把字符串集合结合起来的一般方式。

任一字母表 Σ 上的字符串集合(即, Σ^* 的任一子集)称作语言。例如, Σ^* , \emptyset 和 Σ 都是语言。由于语言只是一种特殊类型的集合,可以用列出它的所有字符串来说明一个有穷的语言。例如, $\{aba, czr, d, f\}$ 是 $\{a, b, \dots, z\}$ 上的语言。但是,大多数感兴趣的语言是无穷的,不可能列出它的所有字符串。要考虑的语言可能是 $\{0, 01, 011, 0111, \dots\}$, $\{w \in \{0, 1\}^* : w$ 有个数相同的 0 和 1 $\}$ 和 $\{w \in \Sigma^* : w = w^R\}$ 。于是,按照用来说明无穷集合的一般形式,将采用下述方案说明无穷语言:

$$L = \{w \in \Sigma^* : w \text{ 具有性质 } P\}.$$

如果 Σ 是一个有穷字母表,则 Σ^* 肯定是有穷的。但是,它是一个可数无穷集合吗?不难看到的确是如此。要构造一个双射 $f: \mathbb{N} \rightarrow \Sigma^*$ 。首先固定字母表中符号的顺序,比如 $\Sigma = \{a_1, \dots, a_n\}$, 这里 a_1, \dots, a_n 是不相同的。然后以下述方式枚举 Σ^* 的所有成员:

(1) 对于每一个 $k \geq 0$, 所有长度为 k 的字符串在所有长度为 $k+1$ 的字符串的前面枚举。

(2) 所有 n^k 个长度为 k 的字符串按照字典顺序枚举,即如果对于某个 m ($0 \leq m \leq k-1$), $i_l = j_l$ ($l=1, \dots, m$) 且 $i_{m+1} < j_{m+1}$, 则 $a_{i_1} \dots a_{i_k}$ 在 $a_{j_1} \dots a_{j_k}$ 的前面。

例如,如果 $\Sigma = \{0, 1\}$, 则排列如下:

$$\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \dots$$

如果 Σ 是罗马字母表并且 Σ 的排列是通常的排列 $\{a, \dots, z\}$, 则长度相同的字符串的字典顺序与字典中使用的顺序相同。但是, Σ^* 中所有的字符串按照(1)和(2)描述的排列不同于字典中的排列,它把较短的字符串排在较长的字符串的前面。

由于语言是集合,可以用集合运算并、交和差把它们结合在一起。当从上下文知道具体的字母表 Σ 时,用 A 的补 \bar{A} 代替差 $\Sigma^* - A$ 。

此外,某些运算仅对语言有意义。第一个这种运算是语言的连接。设 L_1 和 L_2 是 Σ 上的两个语言,它们的连接 $L = L_1 \circ L_2$ (简记作 $L = L_1 L_2$) 定义为

$$L = \{w \in \Sigma^* : w = x \circ y, \text{ 其中 } x \in L_1, \text{ 且 } y \in L_2\}$$

例如,如果 $\Sigma = \{0, 1\}$, $L_1 = \{w \in \Sigma^* : w \text{ 有偶数个 } 0\}$, $L_2 = \{w : w \text{ 以 } 0 \text{ 开始, 其余符号都是 } 1\}$, 则 $L_1 \circ L_2 = \{w \in \Sigma^* : w \text{ 有奇数个 } 0\}$ 。

另一个语言运算是语言 L 的 Kleene 星号,记作 L^* 。 L^* 是连接 L 中 0 个或多个字符串得到的所有字符串的集合。(0 个字符串的连接是 ϵ , 一个字符串的连接是它自己。) 于是,

$$L^* = \{w \in \Sigma^* : w = w_1 \circ \dots \circ w_k, \text{ 其中 } k \geq 0 \text{ 且 } w_1, \dots, w_k \in L\}.$$

例如,如果 $L = \{01, 1, 100\}$, 则 $110001110011 \in L^*$, 因为 $110001110011 = 1 \circ 100 \circ 01 \circ 1 \circ 100 \circ 1 \circ 1$, 而且每一个这种字符串都在 L 中。

注意,用 Σ^* 表示 Σ 上所有字符串的集合与 Σ Kleene 星号的表示法是一致的,只需把 Σ 看作一个有穷语言。也就是说,如果令 $L = \Sigma$, 并且运用上述定义,则 Σ^* 是所有能够写成 $w_1 \circ \dots \circ w_k$ 的字符串的集合,其中 $k \geq 0$ 且 $w_1, \dots, w_k \in \Sigma$ 。由于每一个 w_i 只是 Σ 中的单

个符号,从而正如原来定义的那样, Σ^* 是符号在 Σ 中的所有有穷字符串。

作为一个极端的例子,观察到 $\emptyset^* = \{e\}$ 。令上述定义中的 $L = \emptyset$ 。唯一可能的连接 $w_1 \circ \cdots \circ w_k (k \geq 0 \text{ 且 } w_1, \dots, w_k \in L)$ 是取 $k = 0$,即0个字符串的连接。因此,这时 L^* 的唯一成员是 e !

最后一个例子是证明:如果 L 是语言 $\{w \in \{0,1\}^* : w \text{ 中 } 0 \text{ 和 } 1 \text{ 的个数不相等}\}$,则 $L^* = \{0,1\}^*$ 。首先注意到对于任意两个语言 L_1 和 L_2 ,如果 $L_1 \subseteq L_2$,则 $L_1^* \subseteq L_2^*$ 。根据Kleene星号的定义,这是明显的。其次, $\{0,1\} \subseteq L$,因为把0和1看作字符串,它们的0和1的个数不相等。因而, $\{0,1\}^* \subseteq L^*$ 。根据定义,又有 $L^* \subseteq \{0,1\}^*$,所以 $L^* = \{0,1\}^*$ 。

~~用 L^+ 表示语言 LL^* 。等价地, L^+ 是语言~~

$$\{w \in \Sigma^* : w = w_1 \circ \cdots \circ w_k, \text{ 其中 } k \geq 1 \text{ 且 } w_1, \dots, w_k \in L\}$$

注意:可以把 L^+ 看作 L 在连接运算下的闭包。即, L^+ 是包括 L 且在连接运算下封闭的最小语言。

习 题

- 1.7.1 (a) 利用正文中给出的连接的定义,证明字符串连接是可结合的。
(b) 给出字符串连接的递归定义。
(c) 利用(b)中的递归定义,证明字符串连接是可结合的。
- 1.7.2 利用正文中给出的反转的递归定义,证明:
(a) 对于任意的字符串 w , $(w^R)^R = w$ 。
(b) 如果 v 是 w 的子串,则 v^R 是 w^R 的子串。
(c) 对于任意的字符串 w 和 $i \geq 0$, $(w^i)^R = (w^R)^i$ 。
- 1.7.3 设 $\Sigma = \{a_1, \dots, a_{26}\}$ 是罗马字母表,小心地定义 Σ^* 上的二元关系 $<$,使得 $x < y$ 当且仅当在标准的字典中 x 在 y 的前面。
- 1.7.4 证明:
(a) $\{e\}^* = \{e\}$ 。
(b) 对于任意的字母表 Σ 和 $L \subseteq \Sigma^*$, $(L^*)^* = L^*$ 。
(c) 设 a 和 b 是两个不同的符号,则 $\{a,b\}^* = \{a\}^* (\{b\} \{a\}^*)^*$ 。
(d) 设 Σ 是任一字母表, $e \in L_1 \subseteq \Sigma^*$ 且 $e \in L_2 \subseteq \Sigma^*$,则 $(L_1 \Sigma^* L_2)^* = \Sigma^*$ 。
(e) 对于任意的语言 L , $\emptyset L = L \emptyset = \emptyset$ 。
- 1.7.5 给出若干在和不在下述集合中的字符串,其中 $\Sigma = \{a,b\}$:
(a) $\{w : w = uu^R u, \text{ 其中 } u \in \Sigma^*\}$ 。
(b) $\{w : ww = w^R w\}$ 。
(c) $\{w : \text{对于某个 } u, v \in \Sigma^*, uvw = wvu\}$ 。
(d) $\{w : \text{对于某个 } u \in \Sigma^*, wuw = uu\}$ 。
- 1.7.6 在什么情况下, $L^+ = L^* - \{e\}$?
- 1.7.7 语言 L 的Kleene星号是 L 在什么关系下的闭包?

1.8 语言的有穷表示

计算理论中的一个核心问题是用有穷的规定说明表示语言。自然,任何有穷语言可以通过穷举该语言中的所有字符串给出它的有穷表示。只有当考虑无穷语言时这个问题才成为有争议的。

让我们更精确地讨论一下“语言的有穷表示”这个概念。要说的第一点是任何这样的表示本身必须是一个字符串,即某个字母表 Σ 上的一个有穷的符号序列。第二,肯定要求不同的语言有不同的表示,否则很难认为表示这个词是合适的。但是,这两条要求已经意味着有穷表示的可能性是极其有限的。由于字母表 Σ 上的字符串集合 Σ^* 是可数无穷的,所以可能的语言表示也是可数无穷的。(甚至不限制使用一个具体的字母表时,这仍然是对的,只要所有可供使用的符号是可数无穷的。)另一方面,由于 $2^{\mathbb{N}}$ 不是可数无穷的,从而任何可数无穷集合的幂集不是可数无穷的,因此在一个给定的字母表 Σ 上的所有可能的语言的集合 2^{Σ^*} 是不可数无穷的。只有可数个表示、但是有不可数个要表示的东西,我们不可能有穷地表示所有的语言。因此,我们顶多可以希望至少为某些我们更感兴趣的语言找到某种有穷表示。

这是我们在计算理论中的第一个结果:不论用来表示语言的方法怎样有力,只要表示的本身是有穷的,就只有可数多个语言能够被表示。总共有不可数多个语言,在任何有穷的表示方案下它们中的绝大多数将不可避免地被遗漏掉。

当然,这不是我们沿着这些线索要说的最后一件事情。我们将要叙述几个描述和表示语言的方式,在下述意义下每一个比前一个更有力,每一个能够描述前一个不能描述的语言。这个层次与所有这些有穷的表示方法由于刚才解释的原因不可避免地被限制在一角是不矛盾的。

我们还打算导出一些方法用来展示不能用我们所研究的各种表示方法表示的具体语言。我们知道在语言世界中有数量极大的这种不可表示的怪东西。但是,要抓住一个,把它拿出来展览并且为它提供证明是非常困难的。这大概是件奇怪的事情。对角化论证最终将在这里帮助我们。

下面开始研究有穷表示。首先考虑一种表达式,即一串符号,用它来描述怎样用上一节的运算构造出语言。

例 1.8.1 令 $L = \{w \in \{0,1\}^* : \text{在 } w \text{ 中 } 1 \text{ 出现两次或三次,并且第一次和第二次不相邻}\}$ 。这个语言可以只用单元集和符号 \cup 、 \cdot 及 $*$ 描述成

$$\{0\}^* \cdot \{1\} \cdot \{0\}^* \cdot \{0\} \cdot \{1\} \cdot \{0\}^* ((\{1\} \cdot \{0\}^*) \cup \emptyset^*)$$

不难看出上述表达式表示的语言正好是上面定义的语言 L 。要注意的重要事情是,在这个表达式中使用的符号只有花括号(和)、圆括号(和)、 \emptyset 、 0 、 1 、 $*$ 、 \cdot 和 \cup 。事实上,可以省略花括号和 \cdot ,简单地写成

$$L = 0^* 1 0^* 0 1 0^* (1 0^* \cup \emptyset^*) \quad \diamond$$

粗略地说,像例 1.8.1 中关于 L 那样的表达式叫做正则表达式。也就是说,正则表达式描述语言仅仅用单个符号和 \emptyset ,大概要结合使用符号 \cup 和 $*$ 、还可能有帮助。

为了保证我们谈论的表达式和用来讨论它们的“数学白话”不出问题,必须相当地小心。 \cup 、 $*$ 和 \emptyset 在本书中是某些运算和集合的名字,我们引入专门的符号 \cup 、 $*$ 和 \odot 分别代替它们,暂且认为这几个符号是完全没有含义的,就像在早先的例子中使用的符号 a 、 b 和 0 一样。我们用同样的方式引入专门的符号 $($ 和 $)$ 代替在数学中一直使用的圆括号 $($ 和 $)$ 。字母表 Σ 上的正则表达式是字母表 $\Sigma \cup \{ (,), \odot, \cup, * \}$ 上可以如下获得的所有字符串:

- (1) \odot 和 Σ 的每一个成员是正则表达式。
- (2) 如果 α 和 β 是正则表达式,则 $(\alpha\beta)$ 也是正则表达式。
- (3) 如果 α 和 β 是正则表达式,则 $(\alpha \cup \beta)$ 也是正则表达式。
- (4) 如果 α 是正则表达式,则 α^* 也是正则表达式。
- (5) 除由(1)到(4)得到的之外,没有任何别的正则表达式。

把符号 \cup 和 $*$ 分别解释成集合的并和 Kleene 星号,把表达式的并列解释成连接,每一个正则表达式表示一个语言。形式地、用函数 L 建立起正则表达式与它们表示的语言之间的关系,如果 α 是任意一个正则表达式,则 $L(\alpha)$ 是 α 表示的语言。也就是说, L 是从字符串到语言的函数。函数 L 的定义如下:

- (1) $L(\odot) = \emptyset$; 对每一个 $a \in \Sigma$, $L(a) = \{a\}$ 。
- (2) 如果 α 和 β 是正则表达式,则 $L((\alpha\beta)) = L(\alpha)L(\beta)$ 。
- (3) 如果 α 和 β 是正则表达式,则 $L((\alpha \cup \beta)) = L(\alpha) \cup L(\beta)$ 。
- (4) 如果 α 是正则表达式,则 $L(\alpha^*) = L(\alpha)^*$ 。

对于每一个由单个符号组成正则表达式 α ,陈述(1)定义了 $L(\alpha)$ 。当正则表达式 α 的长度大于1时,(2)到(4)用一个或两个长度短一些的正则表达式 α' 的 $L(\alpha')$ 定义 $L(\alpha)$ 。于是,每一个正则表达式以这种方式与某个语言相关联。

例 1.8.2 $L(((a \cup b)^*a))$ 是什么? 计算如下:

$$\begin{aligned}
 L(((a \cup b)^*a)) &= L((a \cup b)^*)L(a) && \text{由(2)} \\
 &= L((a \cup b)^*)\{a\} && \text{由(1)} \\
 &= L((a \cup b))^*\{a\} && \text{由(4)} \\
 &= (L(a) \cup L(b))^*\{a\} && \text{由(3)} \\
 &= (\{a\} \cup \{b\})^*\{a\} && \text{由(1)两次} \\
 &= \{a, b\}^*\{a\} \\
 &= \{w \in \{a, b\}^* : w \text{ 以 } a \text{ 结束}\}. && \diamond
 \end{aligned}$$

例 1.8.3 $(c^*(a \cup (bc^*))^*)$ 表示的语言是什么? 这个正则表达式表示 $\{a, b, c\}$ 上不含子串 ac 的所有字符串的集合。显然, $L((c^*(a \cup (bc^*))^*))$ 中的字符串不可能含有子串 ac ,因为在这些字符串中每一个 a 要么在字符串的结尾,要么后面跟着另一个 a ,或者跟着一个 b 。另一方面,令 w 是一个不含子串 ac 的字符串。那么, w 开头有0个或多个 c 。把这些 c 删去,得到的字符串不含子串 ac 并且不以 c 开头。从左到右读这个字符串,作为若干 a, b 和 c 的序列,任意一段 c 都紧跟在 b 的后面(不会跟在 a 的后面,也不会开头的地方),故这个字符串在 $L((a \cup (bc^*))^*)$ 中。因此, $w \in L((c^*(a \cup (bc^*))^*))$ 。 \diamond

例 1.8.4 $(0^* \cup (((0^*(1 \cup (11))))((00^*)(1 \cup (11))))^*0^*)$ 表示 $\{0, 1\}$ 上不含子串 111 的所有字符串的集合。 \diamond

每一个可以用正则表达式表示的语言可以用无穷多个正则表达式表示。例如, α 和 $(\alpha \cup \odot)$ 总是表示相同的语言; $((\alpha \cup \beta) \cup \gamma)$ 和 $(\alpha \cup (\beta \cup \gamma))$ 也表示相同的语言。由于集合的并和连接是可结合的运算, 即对于所有的 L_1, L_2 和 L_3 , 有 $(L_1 \cup L_2) \cup L_3 = L_1 \cup (L_2 \cup L_3)$ 和 $(L_1 \cdot L_2) \cdot L_3 = L_1 \cdot (L_2 \cdot L_3)$, 通常删去正则表达式中多余的符号(和)。例如, 把 $a \cup b \cup c$ 作为一个正则表达式来对待, 虽然“正式地说”它不是正则表达式。作为第二个例子, 例 1.8.4 中的正则表达式可以重新写成

$$0^* \cup 0^*(1 \cup 11)^*(00^*(1 \cup 11))^* 0^*$$

此外, 我们已经形式地、明确地定义了正则表达式和它们表示的语言。当不会产生混乱时, 可以不去严格地区分正则表达式和用来谈论语言的“数学白话”。于是, 我们可以在一个地方说 a^*b^* 是由若干个 a 后面跟着若干个 b 构成的所有字符串的集合。确切地讲, 应该写成 $\{a\}^* \cdot \{b\}^*$ 。在另一个地方我们又可以说 a^*b^* 是表示这个集合的正则表达式。确切地讲, 这时应该写成 (a^*b^*) 。

定义字母表 Σ 上的正则语言类由所有可写成 $L = L(\alpha)$ 的语言 L 组成, 其中 α 是 Σ 上的任一正则表达式。即, 正则语言是所有能够用正则表达式描述的语言。换一种说法, 可以用闭包来考虑正则语言。 Σ 上的正则语言类恰好是语言集合

$$\{\{\sigma\}; \sigma \in \Sigma\} \cup \{\emptyset\}$$

关于并、连接和 Kleene 星号函数的闭包。

我们已经看到正则表达式描述了某些非平凡的、有趣的语言。不幸的是, 不能用正则表达式描述某些可以用别的方法简单地描述的语言。例如, 在第 2 章将证明 $\{0^n 1^n; n \geq 0\}$ 不是正则的。任何语言的有穷表示理论确实应该至少能接纳这么简单的语言。因而, 正则表达式大体上是一种不充分的规定说明方法。

在寻找有关有穷地说明语言的一般方法时, 我们可能回到一般的方式

$$L = \{w \in \Sigma^*; w \text{ 具有性质 } P\}$$

但是, 应该需要什么样的性质 P ? 例如, 是什么使得前面的性质“ w 由若干个 0 后面跟着个数相同的 1”和“ w 中不出现 111”明显是可以接受的? 读者可以去考虑正确的答案, 但是我们现在接受并且只接受算法性质。也就是说, 为了使字符串的性质 P 可以接受为语言的规定说明, 必须有算法能够确定任给的字符串是否属于该语言。专门为某个语言 L 设计的、用来回答“字符串 w 是 L 的成员吗?”这种形式的问题的算法称作语言识别装置。例如, 识别语言

$$L = \{w \in \{0, 1\}^*; w \text{ 不含子串 } 111\}$$

的装置可以如下运算: 从左到右读字符串, 每次读一个。有一个计数器, 开始时为 0, 每当在输入串中遇到 0 时将它置回到 0, 每当在输入串中遇到 1 时加 1。如果计数器已经达到 3, 则停止计算并且回答“*No*”; 如果读完整个字符串并且计数器没有达到 3, 则停止计算并且回答“*Yes*”。

说明语言的另一种完全不同的方法是描述如何生成语言中的一般样品。例如, 可以把正则表达式 $(e \cup b \cup bb)(a \cup ab \cup abb)^*$ 看作生成语言的成员的一种方法: 为了生成 L 的一个成员, 第一步什么都不写、或者写 b 、或者写 bb ; 然后写 a 、或者 ab 、或 abb , 可以这样做任意次, 包括 0 次在内。 L 中的所有成员并且也只有 L 中的成员可以这样生成。

这种语言生成器不是算法,因为它们不是用来回答问题的,做什么也不是完全清楚的(怎样从 a, ab 和 abb 中选择一个要写的?)但它同样是一种重要的表示语言的方法。语言识别装置和语言生成器是语言有穷说明的两种类型,研究它们之间的关系是本书的另一个主要课题。

习 题

1.8.1 正则表达式 $((a^*a)b) \cup b$ 表示的语言是什么?

1.8.2 把下述正则表达式重写为表示同样集合的更简单的正则表达式:

(a) $\odot^* \cup a^* \cup b^* \cup (a \cup b)^*$

(b) $((a^*b^*)^*(b^*a^*)^*)^*$

(c) $(a^*b)^* \cup (b^*a)^*$

(d) $(a \cup b)^*a(a \cup b)^*$

1.8.3 令 $\Sigma = \{a, b\}$, 写出下述集合的正则表达式:

(a) Σ^* 中不含多于 3 个 a 的所有字符串。

(b) Σ^* 中 a 的个数可被 3 整除的所有字符串。

(c) Σ^* 中子串 aaa 恰好出现一次的所有字符串。

1.8.4 证明:如果 L 是正则的,则 $L' = \{w: \text{对于某个字符串 } u, uw \in L\}$ 也是正则的。(提示:说明怎样根据 L 的正则表达式构造 L' 的正则表达式。)

1.8.5 下述各条中哪些成立? 给予说明。

(a) $baa \in a^*b^*a^*b^*$

(b) $b^*a^* \cap a^*b^* = a^* \cup b^*$

(c) $a^*b^* \cap b^*c^* = \emptyset$

(d) $abcd \in (a(cd)^*b)^*$

1.8.6 正则表达式 α 的星号高度 $h(\alpha)$ 归纳定义如下:

$h(\odot) = 0$,

$h(a) = 0$, 对每一个 $a \in \Sigma$,

$h(\alpha \cup \beta) = \max(h(\alpha), h(\beta))$,

$h(\alpha^*) = h(\alpha) + 1$ 。

例如,如果 $\alpha = (((ab)^* \cup b^*)^* \cup a^*)$, 则 $h(\alpha) = 2$ 。对于下述各式,找到表示相同语言且星号高度尽可能小的正则表达式:

(a) $((abc)^*ab)^*$

(b) $(a(ab^*c)^*)^*$

(c) $(c(a^*b)^*)^*$

(d) $(a^* \cup b^* \cup ab)^*$

(e) $(abb^*a)^*$

1.8.7 如果一个正则表达式的形式为 $(\alpha_1 \cup \alpha_2 \cup \dots \cup \alpha_n)$, 其中 $n \geq 1$, 每一个 α_i 都不含有 \cup , 则称它是析取范式。证明:每一个正则表达式都可以表示成析取范式。

参 考 文 献

- P. Halmos. Naive Set Theory. Princeton, N. J. : D. Van Nostrand, 1960 是一本极好的提供有关非形式集合论的原始资料的书.
- G. Polya. Induction and Analogy in Mathematics. Princeton, N. J. : Princeton University Press, 1954 是一本关于数学归纳法的好书.
- C. L. Liu. Topics in Combinatorial Mathematics. Buffalo, N. Y. : Mathematical Association of America, 1972 的第 1 章中有一些应用鸽巢原理的例子.
Cantor 原始的对角化论证可以在下面的书中找到:
- G. Cantor. Contributions to the Foundations of the Theory of Transfinite Numbers. New York, Dover Publications, 1947.
O 记法及其几个变种是在下述文章里引入的:
- D. E. Knuth. Big omicron and big omega and big theta. ACM SIGACT News, 8(2), pp. 18—23, 1976.
自反传递闭包的 $O(n^3)$ 算法取自,
- S. Warshall. A theorem on Boolean matrices. Journal of the ACM, 9(1), pp. 11—12, 1962.
下面两本是关于算法及其分析的著作:
- T. H. Cormen, C. E. Leiserson, R. L. Rivest. Introduction to Algorithms. Cambridge, Mass. : The MIT Press. , 1990.
- G. Brassard, P. Bratley. Fundamentals of Algorithms, Englewood Cliffs, N. J. : Prentice Hall, 1996.
下面两本是关于语言理论的高级著作:
- A. Salomaa. Formal Languages. New York : Academic Press, 1973.
- M. A. Harrison. Introduction to Formal Language Theory. Reading, Mass. : Addison-Wesley, 1978.

第 2 章 有穷自动机

2.1 确定型有穷自动机

这是一本关于计算机与算法的数学模型的书。本章和下面两章将要定义能力越来越强的计算模型,即越来越复杂的接受语言和生成语言的设备。从全书的进展来看,本章是一个很初步的开头:在这里介绍一个受到严格限制的实际计算机的模型,称作**有穷自动机**^①或**有穷状态机器**。有穷自动机和现实的计算机都有一个固定的、能力有限的“中心处理装置”。它接收输入,输入是一个字符串并且被传送到输入带上。它没有输出,只给出是否接受这个输入的信号。换句话说,它是第 1 章结束时描述的语言识别装置。使得有穷自动机成为这种受限制的现实计算机的模型的关键是除固定的中心处理器之外它完全没有存储能力。

一开始可能会认为如此简单的计算模型不值得认真地研究,一台没有存储的计算机能有多少用处!但是,有穷自动机实际上不是没有存储,只是它的存储能力“在制造厂里”就被固定下来并且此后不可能扩大。有理由认为任何计算机的存储能力都是有限的——由于预算的限制,由于物理的极限,极端地说,不能超过整个宇宙的大小。最好的计算机数学模型是具有有限存储的模型还是具有无限存储的模型,这是一个有趣的哲学问题。我们将研究这两种类型的模型,从存储能力有限的模型开始,然后详细得多地研究存储能力无限的模型。

即使和我们一样认为建立计算机和算法模型的正确途径是使用无限存储能力,也会肯定应该首先很好地了解有穷自动机的理论。原来有穷自动机理论是丰富、漂亮的,而且当我们了解它的时候能够更好地准确理解增加辅助的存储对增加计算能力起什么作用。

研究有穷自动机的进一步理由是它们可以应用于设计几个通用类型的计算机算法和程序。例如,编译程序的词法分析阶段(在这个阶段识别出程序单元,如‘begin’和‘+’)通常是以模拟一台有穷自动机为基础。还有,在一个字符串中寻找另一个字符串也可以用源于有穷自动机理论的方法有效地解决。

现在我们详细地描述有穷自动机的运行。字符串被送入称作**输入带**的设备,带被划分成方格,每一个带方格中写一个符号(见图 2-1)。机器的主要部分是一个带有内部结构的“黑盒子”,在任一特定的时刻它处于有穷个不同的内部**状态**中的一个。这个黑盒子叫作**有穷控制器**,它通过可移动的**读头**能够了解在输入带的任何位置上写着什么符号。开始的时候,读头放在带的最左边的方格上,有穷控制器处于一个指定的**初始状态**。每隔一定时间有穷自动机从输入带上读一个符号,然后进入一个新的状态。这个新的状态只与当前状态

^① 自动机是用来响应编码指令的机器,一种自动控制装置。

和刚读到的符号有关,这正是我们把这种装置叫做确定型有穷自动机的原因,而下一节引入的非确定型形式不是这样的。在读一个符号之后,读头在输入带上向右移一格,这样在下一步它将读到下一个带方格中的符号。这个过程一次又一次的重复,读一个符号、读头向右移和改变有穷控制器的状态。最后,读头到达输入串的结尾。有穷自动机通过它最后所处的状态表明批准还是不批准它读到的字符串:如果它结束在一个终结状态,则认为输入串被接受。机器接受的语言是它接受的所有字符串的集合。



图 2-1

把上述非形式的描述归结为它的数学本质,得到下述形式定义。

定义 2.1.1 确定型有穷自动机是一个五元组 $M=(K, \Sigma, \delta, s, F)$, 其中

K 是有穷的状态集合,

Σ 是字母表,

$s \in K$ 是初始状态,

$F \subseteq K$ 是终结状态集合,

δ 是从 $K \times \Sigma$ 到 K 的函数,叫做转移函数。

有穷自动机 M 选取下一个状态的规则被编码成转移函数。于是,如果 M 处于状态 $q \in K$ 且从输入带读到的符号是 $a \in \Sigma$, 则 $\delta(q, a) \in K$ 是 M 要转移到的唯一确定的状态。

有了确定型有穷自动机的形式化的基本构造之后,下面应该用数学术语描述有穷自动机在一个输入串上的计算。粗略地说,它是一个格局序列,表示机器(有穷控制器、读头及输入带)在相继时刻的状况。由于不允许确定型有穷自动机把它的读头移回到输入串已经读过的部分,读头左边的字符串部分不再可能影响机器后面的运行。于是,格局被当前状态和被处理的字符串的尚未读过的部分决定。换句话说,确定型有穷自动机 $(K, \Sigma, \delta, s, F)$ 的格局是 $K \times \Sigma^*$ 的任一元素。例如,图 2-1 表示的格局为 $(q_2, ababab)$ 。

二元关系 \vdash_M 在 M 的两个格局之间成立当且仅当 M 能够一步从一个格局变成另一个格局。于是,如果 (q, w) 和 (q', w') 是 M 的两个格局,则 $(q, w) \vdash_M (q', w')$ 当且仅当对于某个符号 $a \in \Sigma, w = aw'$ 且 $\delta(q, a) = q'$ 。这时我们称 (q, w) 一步产生 (q', w') 。注意, \vdash_M 实际上是从 $K \times \Sigma^*$ 到 $K \times \Sigma^*$ 的函数,对除 (q, e) 形式以外的每一个格局,有唯一确定的下一个格局。 (q, e) 形式的格局表明 M 已消耗完它的全部输入,因此运行到此结束。

用 \vdash_M^* 表示 \vdash_M 的自反传递闭包。 $(q, w) \vdash_M^* (q', w')$ 读作 (q, w) 产生 (q', w') (在若干步,可能是 0 步,之后)。称字符串 $w \in \Sigma^*$ 被 M 接受当且仅当存在状态 $q \in F$ 使 $(s, w) \vdash_M^* (q, e)$ 。最后, M 接受的语言是 M 接受的所有字符串的集合,记作 $L(M)$ 。

例 2.1.1 设 M 是确定型有穷自动机 $(K, \Sigma, \delta, s, F)$, 其中

$$\begin{aligned} K &= \{q_0, q_1\}, \\ \Sigma &= \{a, b\}, \\ s &= q_0, \\ F &= \{q_0\}, \end{aligned}$$

而 δ 是表 2-1 给出的函数。

容易看到 $L(M)$ 是 $\{a, b\}^*$ 中所有有偶数个 b 的字符串。因为当读到一个 b 时 M 从状态 q_0 转到 q_1 或者从 q_1 回到 q_0 ；而 M 根本不理睬 a ，当读到一个 a 时总是停留在它的当前状态不变。于是， M 按模 2 数 b 的个数。由于初始状态 q_0 又是唯一的终结状态，故 M 接受字符串当且仅当 b 的个数为偶数。

表 2-1

q	σ	$\delta(q, \sigma)$
q_0	a	q_0
q_0	b	q_1
q_1	a	q_1
q_1	b	q_0

表 2-2

q	σ	$\delta(q, \sigma)$
q_0	a	q_0
q_0	b	q_1
q_1	a	q_0
q_1	b	q_2
q_2	a	q_0
q_2	b	q_3
q_3	a	q_3
q_3	b	q_3

如果给 M 输入 $aabba$ ，则它的初始格局为 $(q_0, aabba)$ 。于是

$$\begin{aligned}
 (q_0, aabba) &\vdash_M (q_0, abba) \\
 &\vdash_M (q_0, bba) \\
 &\vdash_M (q_1, ba) \\
 &\vdash_M (q_0, a) \\
 &\vdash_M (q_0, \epsilon)
 \end{aligned}$$

因此， $(q_0, aabba) \vdash_M^* (q_0, \epsilon)$ ，从而 $aabba$ 被 M 接受。 \diamond

在这个例子中用表格表示转移函数不是描述机器的最清晰的方法。通常使用更方便的图形表示，叫做状态图（见图 2-2）。状态图是一个在图中加入了某些附加信息的有向图。用顶点表示状态，当 $\delta(q, a) = q'$ 时有一个从 q 到 q' 标记 a 的箭头。用双圈表示终结状态，用 \triangleright 标明初始状态。图 2-2 给出例 2.1.1 中确定型有穷自动机的状态图。



图 2-2

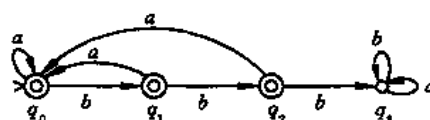


图 2-3

例 2.1.2 设计一台接受语言 $L(M) = \{w \in \{a, b\}^* : w \text{ 不含有 3 个连续的 } b\}$ 的确定型有穷自动机。令 $M = (K, \Sigma, \delta, s, F)$ ，其中

$$\begin{aligned}
 K &= \{q_0, q_1, q_2, q_3\}, \\
 \Sigma &= \{a, b\}, \\
 s &= q_0, \\
 F &= \{q_0, q_1, q_2\},
 \end{aligned}$$

而 δ 由表 2-2 给出。

状态图如图 2-3 所示。为了能看出 M 确实接受指定的语言，注意到只要还没有读到

连续的 3 个 b , M 在读到位于输入串开头或跟在 a 后面的 i 个连续的 b 之后处于状态 q_i (这里 i 等于 0, 1 或 2)。具体地说, 当读到 a 且 M 处于状态 q_0, q_1 或 q_2 时, M 返回到初始状态 q_0 。状态 q_0, q_1 和 q_2 都是终结状态, 故任何不含 3 个连续的 b 的输入串将被接受。然而, 连续的 3 个 b 将把 M 带到状态 q_3 , q_3 不是终结状态并且 M 将停留在这个状态而不管输入串中剩余的符号是什么。状态 q_3 称作停滞状态。如果 M 到达状态 q_3 , 则说它被俘获了, 因为后面的输入不可能使它逃离这个状态。

习 题

2.1.1 设 M 是一台确定型有穷自动机。恰好在什么情况下 $e \in L(M)$? 证明你的答案。

2.1.2 非形式地描述图 E2-1 中给出的确定型有穷自动机接受的语言。

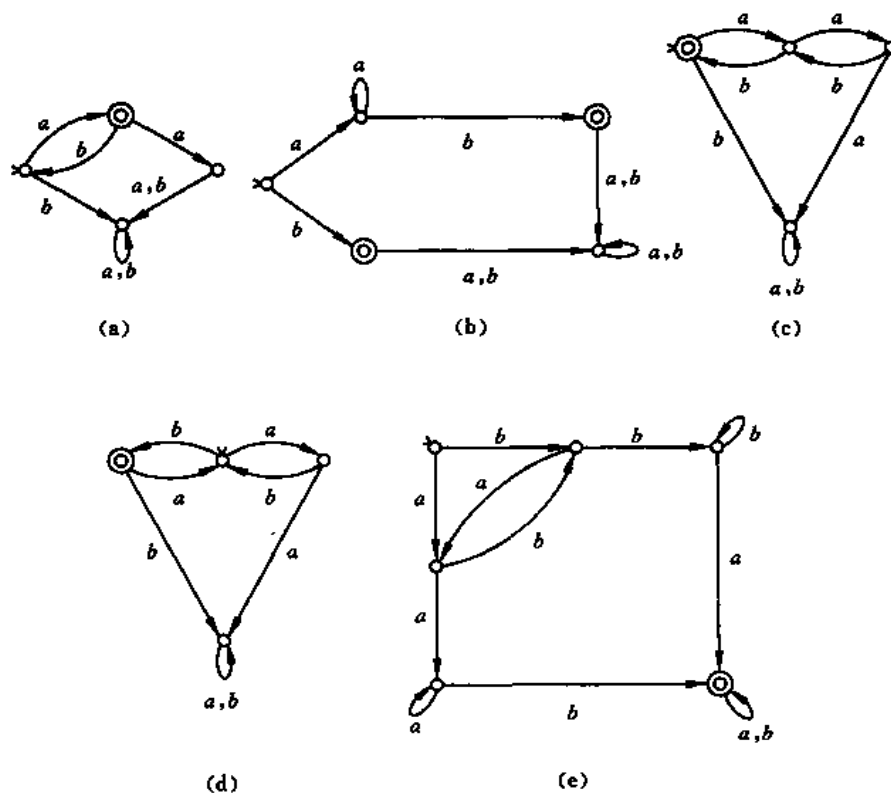


图 E2-1

2.1.3 构造接受下述语言的确定型有穷自动机:

- (a) $\{w \in \{a, b\}^* : w \text{ 中每一个 } a \text{ 的前面都是一个 } b\}$
- (b) $\{w \in \{a, b\}^* : w \text{ 含有子串 } abab\}$
- (c) $\{w \in \{a, b\}^* : w \text{ 不含子串 } aa \text{ 和 } bb\}$
- (d) $\{w \in \{a, b\}^* : w \text{ 有奇数个 } a \text{ 和偶数个 } b\}$
- (e) $\{w \in \{a, b\}^* : w \text{ 含有子串 } ab \text{ 和 } ba\}$

2.1.4 确定型有穷状态转换器是一种非常类似确定型有穷自动机的装置,不过它不是用来接受字符串或语言,而是把输入串转换成输出串。非形式地,和确定型有穷自动机一样,它从指定的初始状态开始,根据输入从一个状态转移到另一个状态。但是,在每一步它根据当前状态和输入符号发表(或写到输出带上)一个由0个、1个或多个符号组成的字符串。确定型有穷状态转换器的状态图看起来很像确定型有穷自动机的状态图,只是箭头上的标记是 a/w 样子的,它的意思是“如果输入符号是 a ,则跟着这个箭头走并且发表输出 w ”。例如,图 E2-2 描述的 $\{a,b\}$ 上的确定型有穷状态转换器传送输入串中所有的 b ,而 a 是每隔一个删去一个。

(a) 画出做下述事情的 $\{a,b\}$ 上的确定型有穷状态转换器的状态图:

- (I) 对于输入 w ,产生输出 a^n ,其中 n 是子串 ab 在 w 中出现的次数。
- (II) 对于输入 w ,产生输出 a^n ,其中 n 是子串 aba 在 w 中出现的次数。
- (III) 对于输入 w ,产生一个长度 $|w|$ 的字符串。如果 $i=1$ 或当 $i>1$ 时 w 中第 i 个符号与第 $i-1$ 个符号不同,则输出的第 i 个符号为 a ;否则输出的第 i 个符号为 b 。例如,对于输入 $aabba$,转换器应输出 $ababa$;而对于输入 $aaaab$,它应输出 $abbba$ 。

(b) 形式地定义

- (I) 确定型有穷状态转换器。
- (II) 这种自动机的格局。
- (III) 格局之间的一步产生关系 \vdash 。
- (IV) 这种自动机对于输入 w 产生输出 u 的概念。
- (V) 这种自动机计算一个函数的概念。

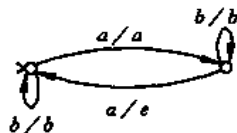


图 E2-2

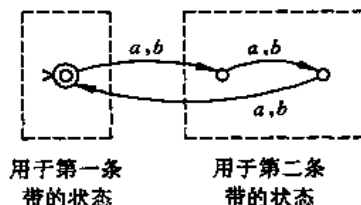


图 E2-3

2.1.5 确定型双带有穷自动机是一个类似确定型有穷自动机的装置,用来接受字符串对。每一个状态属于两个集合中的一个集合。根据状态在哪个集合中,转移函数看第一条带或第二条带。例如,图 E2-3 描述的自动机接受满足条件 $|w_2| = 2|w_1|$ 的所有字符串对 $(w_1, w_2) \in \{a,b\}^* \times \{a,b\}^*$ 。

(a) 画出接受下述字符串对的确定型双带有穷自动机的状态图:

- (I) $\{a,b\}^* \times \{a,b\}^*$ 中满足条件 $|w_1| = |w_2|$ 且对所有的 $i, w_1(i) \neq w_2(i)$ 的所有字符串对 (w_1, w_2) 。
- (II) $\{a,b\}^* \times \{a,b\}^*$ 中 w_2 的长度等于 w_1 中 a 的个数的两倍加 w_1 中 b 的

个数的三倍的所有字符串对 (w_1, w_2) 。

(Ⅱ) $\{(a^n b, a^n b^m); n, m \geq 0\}$ 。

(Ⅳ) $\{(a^n b, a^m b^n); n, m \geq 0\}$ 。

(b) 形式地定义

(Ⅰ) 确定型双带有穷自动机。

(Ⅱ) 这种自动机的格局。

(Ⅲ) 格局之间的一步产生关系 \vdash 。

(Ⅳ) 这种自动机接受字符串有序对的概念。

(Ⅴ) 这种自动机接受字符串有序对的集合的概念。

2.1.6 证明:如果 $f: \Sigma^* \rightarrow \Sigma^*$ 是一个能够用确定型有穷状态转换器计算的函数,则字符串有序对的集合 $\{(w, f(w)); w \in \Sigma^*\}$ 被某个确定型双带有穷自动机接受。参见习题 2.1.4 和 2.1.5。

2.1.7 设 $M = (K, \Sigma, \delta, q_0, F)$ 是一台确定型有穷自动机, $q \in K$ 。如果存在 $w \in \Sigma^*$ 使得 $(q_0, w) \xrightarrow{*}_M (q, \epsilon)$, 则称状态 q 是可达的。证明:删去 M 的任一不可达状态所得到的自动机接受相同的语言。给出计算确定型有穷自动机的所有可达状态的有效算法。

2.2 非确定型有穷自动机

本节给有穷自动机添加一个有力而有点奇怪的特性。它叫做非确定性,在本质上这是一种改变状态的能力,在这里当前状态和输入符号只能起部分的决定作用。也就是说,对于给定的当前状态和输入符号,允许几个可能的“下一个状态”。自动机在读它的输入串时,每一步可以选择转移到这些合法的下一个状态中的任一个。在这种模型中,选择不被任何东西决定,因而叫作非确定的。另一方面,选择也不是完全没有限制的,只能选择那些对于给定的状态和输入符号来说是合法的下一个状态。

这种非确定型装置不是现实的计算机模型。它们只是有穷自动机的表示方法的有力推广,可以极大地简化这些自动机的描述。此外,下面将要看到非确定性不是有穷自动机的本质特性,每一台非确定型有穷自动机等价于一台确定型有穷自动机。于是,我们将从非确定型有穷自动机这种有力的表示方法中获益。我们知道,如果需要的话,我们总可以返回去用实事求是的普通确定型有穷自动机的低水平语言重做每一件事情。

考虑语言 $L = (ab \cup aba)^*$, 图 2-4 给出的确定型有穷自动机接受这个语言。我们用这个例子说明用非确定型有穷自动机设计要比用确定型有穷自动机方便得多。甚至对于图 2-4 中的有向图也要花点时间查明它确实描述一台确定型有穷自动机,即必须核实每一个顶点恰好有两个箭头离开,一个标记 a , 另一个标记 b 。为了相信这台不算简单的设备正好接受语言 $(ab \cup aba)^*$, 也需要动动脑筋,人们可能希望找到一台更简单的确定型有穷自动机接受 L 。不幸的是,可以证明少于 5 个状态的确定型有穷自动机不可能接受这个语言(在本章早些时候将要开发最小化确定型有穷自动机状态数的方法)。

但是, L 被图 2-5 描述的简单的非确定型设备接受。当这台设备处于状态 q_1 且输入符

号为 b 时,有两个可能的下一次状态 q_0 和 q_2 。因此,图 2-5 不表示一台确定型有穷自动机。

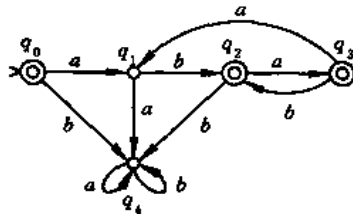


图 2-4

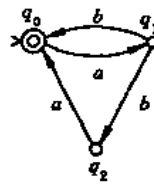


图 2-5

但是,可以自然地把这个图解释成一台接受 L 的设备。如果有某种办法按照给定字符串的符号所标记的箭头从初始状态(q_0)到达一个终结状态(在这里是 q_0),则接受这个字符串。例如,接受 ab ,可以从 q_0 到 q_1 ,再到 q_0 ;也接受 aba ,可以从 q_0 到 q_1 ,到 q_2 ,再到 q_0 。当然,这个设备可能猜错了。对于输入 $abab$,从 q_0 到 q_1 ,到 q_0 ,再到 q_1 ,结束在一个非终结状态。但是,这没有关系,因为对于这个输入有某种办法从初始状态到达一个终结状态。相反地,不接受输入 abb ,因为当读这个字符串的时候没有办法从 q_0 回到 q_0 。

的确如此,注意到当输入为 b 时从 q_0 出发不能进入任何状态。这是非确定型有穷自动机的另一个特点:正如对某些状态与输入符号的组合可能有若干个可能的下一个状态,对另外一些状态与输入符号的组合可能没有动作。

在非确定型有穷自动机的状态图中还允许标记空串 ϵ 的箭头。例如,图 2-6 中的设备也接受语言 L 。这台机器可以读一个 a 从 q_2 返回 q_0 ,也可以不消耗任何输入从 q_2 立即返回 q_0 。

图 2-5 和图 2-6 描绘的设备是下述一般类型的两个实例。

定义 2.2.1 非确定型有穷自动机是一个五元组 $M = (K, \Sigma, \Delta, s, F)$, 其中

K 是有穷的状态集合,

Σ 是字母表,

$s \in K$ 是初始状态,

$F \subseteq K$ 是终结状态集合,

$\Delta \subseteq K \times (\Sigma \cup \{\epsilon\}) \times K$ 是转移关系。

每一个三元组 $(q, u, p) \in \Delta$ 叫作 M 的一个转移,它在形式定义中相当于 M 的状态图中从 q 到 p 标记 u 的箭头。如果 M 处于状态 q 且下一个输入符号为 a ,则 M 下次可以按照任意一个 (q, a, p) 或 (q, ϵ, p) 形式的转移进行。如果按照转移 (q, ϵ, p) 进行,则不读任何输入符号。

非确定型有穷自动机的计算的形式定义与确定型有穷自动机的非常类似。 M 的格局也是 $K \times \Sigma^*$ 的一个元素。格局之间的二元关系 \vdash_M (一步产生)定义如下: $(q, w) \vdash_M (q', w')$ 当且仅当存在 $u \in \Sigma \cup \{\epsilon\}$ 使得 $w = uw'$ 且 $(q, u, q') \in \Delta$ 。注意 \vdash_M 不一定是函数。对于某些格局 (q, w) ,可能有若干个(包括零个)有序对 (q', w') 使得 $(q, w) \vdash_M (q', w')$ 。和前面一样, \vdash_M^* 是 \vdash_M 的自反传递闭包,并且字符串 $w \in \Sigma^*$ 被 M 接受当且仅当存在状态 $q \in F$ 使得 $(s, w) \vdash_M^* (q, \epsilon)$ 。最后, M 接受的语言 $L(M)$ 是所有 M 接受的字符串的集合。

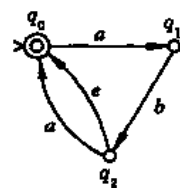


图 2-6

例2.2.1 图2-7描述一台非确定型有穷自动机,接受含有模式 bb 或 bab 出现的所有字符串的集合。还有几个非确定型有穷自动机也接受这个集合(见2.6节有关检测输入串中模式的自动机的系统研究)。形式地,这台机器是 $(K, \Sigma, \Delta, s, F)$, 其中

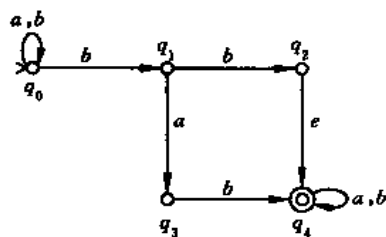


图 2-7

$$K = \{q_0, q_1, q_2, q_3, q_4\}$$

$$\Sigma = \{a, b\}$$

$$s = q_0$$

$$F = \{q_4\}$$

以及

$$\Delta = \{(q_0, a, q_0), (q_0, b, q_0), (q_0, b, q_1), (q_1, b, q_2), (q_1, a, q_3), (q_2, e, q_4), (q_3, b, q_4), (q_4, a, q_4), (q_4, b, q_4)\}$$

当给 M 输入字符串 $bababab$ 时,可能产生几个不同的动作序列。例如,只使用 (q_0, a, q_0) 和 (q_0, b, q_0) , M 能够结束在非终结状态 q_0 :

$$\begin{aligned} (q_0, bababab) &\vdash_M (q_0, ababab) \\ &\vdash_M (q_0, babab) \\ &\vdash_M (q_0, abab) \\ &\vdots \\ &\vdash_M (q_0, e) \end{aligned}$$

这个输入字符串能够把 M 从状态 q_0 转移到终结状态 q_4 ,并且能够用三种不同的方式做到。下面是其中的一个,

$$\begin{aligned} (q_0, bababab) &\vdash_M (q_1, ababab) \\ &\vdash_M (q_3, babab) \\ &\vdash_M (q_4, abab) \\ &\vdash_M (q_4, bab) \\ &\vdash_M (q_4, ab) \\ &\vdash_M (q_4, b) \\ &\vdash_M (q_4, e) \end{aligned}$$

因为一个字符串被非确定型有穷自动机接受当且仅当至少有一个引导到终结状态的计算序列,所以 $bababab \in L(M)$ 。◇

例2.2.2 设字母表 $\Sigma = \{a_1, \dots, a_n\}$, 其中 $n \geq 2$ 。考虑下述语言

$$L = \{w, \text{有一个符号 } a_i \in \Sigma \text{ 不出现在 } w \text{ 中}\}$$

即 L 包括 Σ^* 中 Σ 的符号不全出现的所有字符串。例如,如果 $n=3$, 则 $e, a_1, a_2, a_1a_1a_3a_1 \in L$, 而 $a_3a_1a_3a_1a_2 \notin L$ 。

设计一台接受这个相当复杂的语言的非确定型有穷自动机相对说来是容易的。这里

有 $n+1$ 个状态 $K = \{s, q_1, \dots, q_n\}$, 它们全都是终结状态 ($F = K$)。 Δ 有两种类型的转移 (见图 2-8 当 $n=3$ 时的说明)。初始转移是所有 $(s, e, q_i), 1 \leq i \leq n$; 主转移是所有 (q_i, a_j, q_i) , 其中 $i \neq j$ 。这是 Δ 中的全部转移。

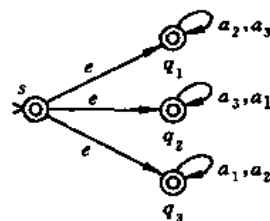


图 2-8

一开始 M 猜想输入串中缺少哪个符号并且转移到相应的状态。如果猜想是 a_i , 则访问状态 q_i 。在这个状态, 自动机核实猜想的符号确实不出现在这个字符串中。如果的确如此, 它以接受告终。这台自动机生动地说明了非确定型设备的非凡能力: 它们可以猜想并且总是对的, 因为接受只需要一次成功的计算。以后在 2.5 节将看到, 接受这个语言的任何确定型有穷自动机肯定要复杂得多。

确定型有穷自动机是非确定型有穷自动机的一种特殊类型: 在确定型有穷自动机中, 转移关系 Δ 实际上是从 $K \times \Sigma$ 到 K 的函数。也就是说, 非确定型有穷自动机 $(K, \Sigma, \Delta, s, F)$ 是确定型的, 当且仅当在 Δ 中没有 (q, e, p) 形式的转移并且对每一个 $q \in K$ 和 $a \in \Sigma$ 恰好有一个 $p \in K$ 使得 $(q, a, p) \in \Delta$ 。因此, 易见, 确定型有穷自动机接受的语言类是非确定型有穷自动机接受的语言类的子集。相当令人吃惊的是, 这两个语言类实际上是相等的。尽管在表面上非确定型有穷自动机更有能力和更一般性, 但是就接受语言而言它们不比确定型有穷自动机更强: 非确定型有穷自动机总可以转换成等价的确定型有穷自动机。

形式地, 称两台有穷自动机 M_1 和 M_2 (确定型或非确定型) 是等价的当且仅当 $L(M_1) = L(M_2)$ 。于是, 如果两台自动机接受相同的语言, 则认为它们是等价的, 虽然它们可能“用不同的方法”。例如, 图 2-4 到图 2-6 中的三台自动机都是等价的。

定理 2.2.1 对于每一台非确定型有穷自动机, 有一台等价的确定型有穷自动机。

证: 设 $M = (K, \Sigma, \Delta, s, F)$ 是一台非确定型有穷自动机。要构造一台等价于 M 的确定型有穷自动机 $M' = (K', \Sigma, \delta, s', F')$ 。关键的想法是认为非确定型有穷自动机在任一时刻不是处于一个状态而是处于一个状态集合, 即从初始状态出发通过到此为止消耗掉的输入能够达到的所有状态。于是, 如果 M 有 5 个状态 $\{q_0, \dots, q_4\}$, 在读了一段输入串之后它可能处于状态 q_0, q_2 或 q_3 , 而不可能处于 q_1 或 q_4 , 则可以认为它的状态是集合 $\{q_0, q_2, q_3\}$, 而不是这个集合的某个不确定的成员。如果下一个输入符号可以把 M 从 q_0 转移到 q_1 或 q_2 , 从 q_2 转移到 q_0 , 从 q_3 转移到 q_2 , 则 M 的下一个状态可以看作是集合 $\{q_0, q_1, q_2\}$ 。

构造 M' 就是将这个想法形式化。 M' 的状态集合 K' 是 M 的状态集合的幂集, 即 $K' = 2^K$ 。 M' 的终结状态集合由 K 中至少包含 M 的一个终结状态的所有子集组成。 M' 的转移函数的定义稍微复杂一些。基本想法是 M' 对读一个输入符号 $a \in \Sigma$ 的动作模仿 M 对输入符号 a 的动作, 后面可能跟着 M 的任意次 e 动作。为了形式化这个想法, 需要一个专门的定义。

对任一状态 $q \in K$, 令 $E(q)$ 等于 M 从状态 q 开始、不读任何输入能够到达的所有状态的集合。即,

$$E(q) = \{p \in K : (q, e) \vdash_M^* (p, e)\}$$

用另一种方式表述, $E(q)$ 是集合 $\{q\}$ 在关系

$$\{(p, r) : \text{有转移 } (p, e, r) \in \Delta\}$$

下的闭包。于是, $E(q)$ 可以用下述算法计算:

initially 令 $E(q) := \{q\}$;

while 存在转移 $(p, e, r) \in \Delta$ 且 $p \in E(q), r \notin E(q)$ do

$$E(q) := E(q) \cup \{r\}.$$

该算法是关于闭包计算的一般算法(见 1.6 节中的最后一个算法)关于这个问题的具体化。它在至多迭代 $|K|$ 次后结束,因为每执行一次 while 循环把一个状态加入 $E(q)$,而至多有 $|K|$ 个状态要加入。后面将看到这种闭包算法的许多实例。

例 2.2.3 在图 2-9 的自动机中, $E(q_0) = \{q_0, q_1, q_2, q_3\}$, $E(q_1) = \{q_1, q_2, q_3\}$, $E(q_2) = \{q_2\}$, $E(q_3) = \{q_3\}$ 及 $E(q_4) = \{q_3, q_4\}$ 。

下面形式地定义与 M 等价的确定型有穷自动机 $M' = (K', \Sigma, \delta', s', F')$, 其中

$$K' = 2^K$$

$$s' = E(s)$$

$$F' = \{Q \subseteq K : Q \cap F \neq \emptyset\}$$

以及对于每一个 $Q \subseteq K$ 和 $a \in \Sigma$, 定义

$$\delta'(Q, a) = \bigcup \{E(p) : p \in K \text{ 且对某个 } q \in Q, (q, a, p) \in \Delta\}.$$

即,取 $\delta'(Q, a)$ 为 M 从 Q 中的状态出发通过读输入符号 a (且后面可能跟着若干个 e 转移)可以到达的所有状态的集合。例如,如果 M 是图 2-9 中的自动机,则 $s' = \{q_0, q_1, q_2, q_3\}$ 。因为对输入 a 从 q_1 开始的转移只有 (q_1, a, q_0) 和 (q_1, a, q_4) , 故 $\delta'(\{q_1\}, a) = E(q_0) \cup E(q_4) = \{q_0, q_1, q_2, q_3, q_4\}$ 。

要证 M' 是确定型的和等价于 M 。证明 M' 是确定型的很简单,只要注意到 δ' 是单值的并且对所有的 $Q \in K'$ 和 $a \in \Sigma$ 有定义。(对于某个 $Q \in K'$ 和 $a \in \Sigma$, $\delta'(Q, a) = \emptyset$ 不表示 δ' 没有定义, \emptyset 是 K' 的一个成员。)

我们现在断言对任一字符串 $w \in \Sigma^*$ 和任意的状态 $p, q \in K$,

$$(q, w) \vdash_M^* (p, e) \text{ 当且仅当}$$

$$\text{对某个包含 } p \text{ 的集合 } P, (E(q), w) \vdash_{M'}^* (P, e).$$

根据这个断言,容易得到定理:为了证明 M 与 M' 等价,考虑任一字符串 $w \in \Sigma^*$ 。于是, $w \in L(M)$ 当且仅当对某个 $f \in F$, $(s, w) \vdash_M^* (f, e)$ (根据定义)当且仅当对某个包含 f 的 Q , $(E(s), w) \vdash_{M'}^* (Q, e)$ (根据上述断言),换句话说,当且仅当对某个 $Q \in F'$, $(s', w) \vdash_{M'}^* (Q, e)$ 。这最后一条是 $w \in L(M')$ 的定义。

对 $|w|$ 做归纳,证明上述断言。

基础步骤。当 $|w|=0$, 即 $w=e$ 时,要证 $(q, e) \vdash_M^* (p, e)$ 当且仅当对某个包含 p 的集合 P , $(E(q), e) \vdash_{M'}^* (P, e)$ 。第一个命题等价于 $p \in E(q)$ 。由于 M' 是确定型的,第二个命题等价于 $P=E(q)$ 且 P 包含 p , 即 $p \in E(q)$ 。这就完成了基础步骤的证明。

归纳假设。假设对于某个 $k \geq 0$, 当字符串 w 的长度小于等于 k 时断言成立。

归纳步骤。要证对任意长度为 $k+1$ 的字符串 w , 断言成立。令 $w=va$, 其中 $a \in \Sigma, v \in \Sigma^*$ 。

从左到右。设 $(q, w) \vdash_M^* (p, e)$ 。那么, 存在状态 r_1 和 r_2 使得

$$(q, w) \vdash_M^* (r_1, a) \vdash_M (r_2, e) \vdash_M^* (p, e)$$

即, M 从状态 q 出发经过若干步,在这中间读输入 v , 接着一步读输入 a , 再接着若干步不读输入的任何符号,最后到达状态 p 。而 $(q, va) \vdash_M^* (r_1, a)$ 相当于 $(q, v) \vdash_M^* (r_1, e)$ 。因为 $|v|$

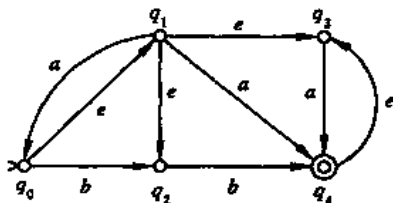


图 2-9

$=k$, 由归纳假设, 对某个包含 r_1 的集合 R_1 , $(E(q), v) \vdash_M^* (R_1, e)$ 。由于 $(r_1, a) \vdash_M (r_2, e)$, 有三元组 $(r_1, a, r_2) \in \Delta$, 从而根据 M' 的构造, 有 $E(r_2) \subseteq \delta'(R_1, a)$ 。又由 $(r_2, e) \vdash_M^* (p, e)$, 有 $p \in E(r_2)$, 因此 $p \in \delta'(R_1, a)$ 。所以, 对于某个包含 p 的 P , $(R_1, a) \vdash_M (P, e)$, 从而 $(E(q), va) \vdash_M^* (R_1, a) \vdash_M (P, e)$ 。

从右到左, 设 $(E(q), va) \vdash_M^* (R_1, a) \vdash_M (P, e)$, 其中 P 包含 p , $\delta'(R_1, a) = P$ 。根据 δ' 的定义, $\delta'(R_1, a)$ 是所有集合 $E(r_2)$ 的并, 这里对于某个状态 $r_1 \in R_1$ 且 (r_1, a, r_2) 是 M 的一个转移。由于 $p \in P = \delta'(R_1, a)$, 有一个 r_2 使得 $p \in E(r_2)$ 并且对于某个 $r_1 \in R_1$, (r_1, a, r_2) 是 M 的一个转移。于是, 根据 $E(r_2)$ 的定义, 有 $(r_2, e) \vdash_M^* (p, e)$ 。又由归纳假设, 有 $(q, v) \vdash_M^* (r_1, e)$ 。因而, $(q, va) \vdash_M^* (r_1, a) \vdash_M (r_2, e) \vdash_M^* (p, e)$ 。

这就完成了断言和定理的证明。 ■

例 2.2.4 定理 2.2.1 的证明是构造性的, 构造性是一种有益的性质, 它提供了从任一非确定型有穷自动机 M 出发, 构造一台等价的确定型有穷自动机 M' 的实际算法。

把这个算法运用于图 2-9 中的非确定型自动机。因为 M 有 5 个状态, 所以 M' 有 $2^5 = 32$ 个状态。但是, 这些状态中只有几个与 M' 的运行有关, 即它们能够从状态 s' 开始、通过读某个输入串达到。显然, K' 中从 s' 不可能达到的任何状态与 M' 的运行以及它接受的语言不相干。从 s' 开始, 当对于某个已引入的状态 $q \in K'$ 和某个 $a \in \Sigma$, $\delta'(q, a)$ 是一个新的状态时引入这个状态, 这样可以得到 M' 的这些可达到的状态。

对于 M 的每一个状态 q , 已经给出 $E(q)$ 。由于 $s' = E(q_0) = \{q_0, q_1, q_2, q_3\}$,

$$(q_1, a, q_0), (q_1, a, q_4) \text{ 和 } (q_3, a, q_4)$$

是全部形式为 (q, a, p) 且 $q \in s'$ 的转移, 从而,

$$\delta'(s', a) = E(q_0) \cup E(q_4) = \{q_0, q_1, q_2, q_3, q_4\}$$

类似地,

$$(q_0, b, q_2) \text{ 和 } (q_2, b, q_4)$$

是全部形式为 (q, b, p) 且 $q \in s'$ 的转移。从而,

$$\delta'(s', b) = E(q_2) \cup E(q_4) = \{q_2, q_3, q_4\}$$

对于新引入的状态重复这种计算, 得到

$$\delta'(\{q_0, q_1, q_2, q_3, q_4\}, a) = \{q_0, q_1, q_2, q_3, q_4\}$$

$$\delta'(\{q_0, q_1, q_2, q_3, q_4\}, b) = \{q_2, q_3, q_4\}$$

$$\delta'(\{q_2, q_3, q_4\}, a) = E(q_4) = \{q_3, q_4\}$$

$$\delta'(\{q_2, q_3, q_4\}, b) = E(q_4) = \{q_3, q_4\}$$

接下去是,

$$\delta'(\{q_3, q_4\}, a) = E(q_4) = \{q_3, q_4\}$$

$$\delta'(\{q_3, q_4\}, b) = \emptyset$$

最后,

$$\delta'(\emptyset, a) = \delta'(\emptyset, b) = \emptyset$$

M' 的有关部分如图 2-10 所示。由于 q_4 是 F 的唯一成员, M' 的终结状态集合 F' 包括每一个以 q_4 为成员的状态集合。在图中, M' 的三个状态 $\{q_0, q_1, q_2, q_3, q_4\}$, $\{q_2, q_3, q_4\}$ 和 $\{q_3, q_4\}$ 是终结状态。 ◇

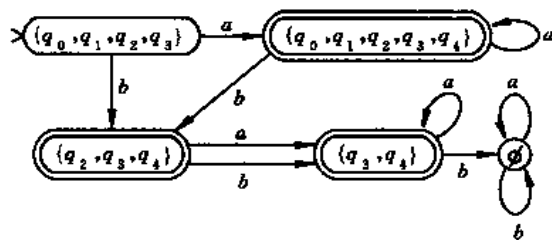


图 2-10

例 2.2.5 回忆一下例 2.2.2 中 $n+1$ 个状态的非确定型有穷自动机, 它的字母表 $\Sigma = \{a_1, \dots, a_n\}$, 接受语言 $L = \{w \in \Sigma^* : \text{有一个符号 } a_i \in \Sigma \text{ 不出现在 } w \text{ 中}\}$ 。直观上, 确定型有穷自动机没有办法用如此少的状态接受这个语言。

的确如此, 这次整个构造是指数的。等价的确定型有穷自动机 M' 用集合 $s' = E(s) = \{s, q_1, q_2, \dots, q_n\}$ 作为初始状态。即使是这样, M' 也有一些不相干的状态。实际上, 2^K 中的一半状态是不相干的。例如, 集合 $\{s\}$ 是不可能从 s' 达到的, 除 s' 之外任何含 s 的状态集合是不可能从 s' 达到的, $\{q_1, \dots, q_n\}$ 也是不可能从 s' 达到的。事实上, 这是 K' 中的全部不相干的状态。剩余的 2^n 个状态—— s' 和 $\{q_1, \dots, q_n\}$ 的任何真子集, 包括空集在内——是可以从 s' 达到的。

当然, 可能有人希望有其他的优化方法能够减少 M' 中的状态数。在 2.5 节中要对这种优化进行系统地研究。不幸的是, 根据那里的分析, M' 中的这个指数的状态数是绝对最小的可能值。◇

习 题

2.2.1 (a) 下述哪几个字符串被图 E2-4(a) 所示的非确定型有穷自动机接受?

- (I) a
- (II) aa
- (III) aab
- (IV) e

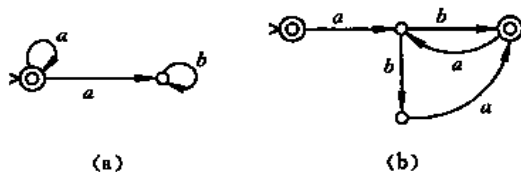


图 E2-4

(b) 下述哪几个字符串被图 E2-4(b) 所示的非确定型有穷自动机接受?

- (I) e
- (II) ab
- (III) $abab$

(N) aba

(V) $abaa$

2.2.2 写出习题 2.2.1 中非确定型有穷自动机接受的语言的正则表达式。

2.2.3 画出接受下述语言的非确定型有穷自动机的状态图：

(a) $(ab)^*(ba)^*\cup aa^*$

(b) $((ab\cup aab)^*a^*)^*$

(c) $((a^*b^*a^*)^*b^*)^*$

(d) $(ba\cup b)^*\cup(bb\cup a)^*$

2.2.4 某些作者把非确定型有穷自动机定义为五元组 $(K, \Sigma, \Delta, S, F)$, 其 K, Σ, Δ 及 F 和我们的一样。而 S 是有穷的初始状态集合, 这和 F 是有穷的终结状态集合一样。自动机可以非确定地从这些初始状态中的任意一个开始运行。

(a) 证明: 由 $\{a_1, \dots, a_n\}^*$ 中至少缺一个符号的所有字符串组成的语言 L (见例 2.2.2) 被有 n 个状态 q_1, \dots, q_n 的这种自动机接受, 这 n 个状态都既是终结状态又是初始状态, 而转移关系 $\Delta = \{(q_i, a_j, q_i); i \neq j\}$ 。

(b) 解释为什么这个定义在任何有意义的意义下不比我们的定义更具一般性。

2.2.5 图 2-7 中的非确定型有穷自动机可以用什么样不同于例 2.2.1 中的步骤接受输入 $bababab$?

2.2.6 (a) 找一台接受 $(ab\cup aab\cup aba)^*$ 的简单的非确定型有穷自动机。

(b) 用 2.2 节中的方法把 (a) 中的非确定型有穷自动机转换成确定型有穷自动机。

(c) 努力理解在 (b) 中构造的机器如何运行。你能找到一台状态数更少的等价的确定型有穷自动机吗?

2.2.7 对于语言 $(a\cup b)^*aabab$, 重复 2.2.6 题。

2.2.8 对于语言 $(a\cup b)^*a(a\cup b)(a\cup b)(a\cup b)(a\cup b)$, 重复 2.2.6 题。

2.2.9 构造等价于图 E2-5 中非确定型有穷自动机的确定型有穷自动机。

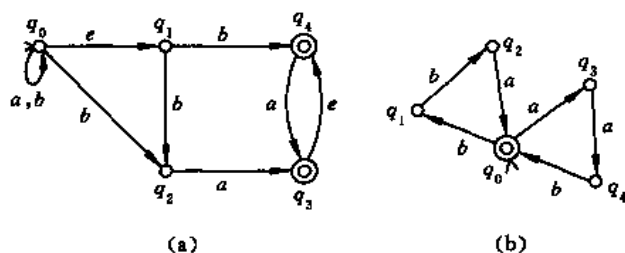


图 E2-5

2.2.10 准确地描述当把本节的构造方法运用于一台已经是确定型的有穷自动机时会发生什么情况?

2.3 有穷自动机与正则表达式

上一节的主要结果是即使允许具有非确定性这个新的看上去能力很强的特性,有穷自动机接受的语言类仍然是相同的。这表明有穷自动机接受的语言类具有稳定性的品质:两种不同的方式,其中一种表面上比另一种更强有力,结果定义相同的语言类。本节将证明这个语言类的另一个重要特征,进一步地表明它是多么异常的稳定:有穷自动机(确定型的和非确定型的)接受的语言类与正则表达式描述的语言类——正则语言类——相同,见 1.8 节中的讨论。

正则语言类是某些有穷的语言在并、连接及 Kleene 星号运算下的闭包。因此,我们应该从证明有穷自动机接受的语言类的类似封闭性质开始。

定理 2.3.1 有穷自动机接受的语言类在下述运算下是封闭的:

- (a) 并;
- (b) 连接;
- (c) Kleene 星号;
- (d) 补;
- (e) 交。

证: 给定两台自动机 M_1 和 M_2 (对于 (c) Kleene 星号和 (d) 补, 只有 M_1), 要说明如何构造接受所需语言的自动机 M 。

(a) 并。设 $M_1 = (K_1, \Sigma, \Delta_1, s_1, F_1)$ 和 $M_2 = (K_2, \Sigma, \Delta_2, s_2, F_2)$ 是两台非确定型有穷自动机。要构造一台非确定型有穷自动机 M 使得 $L(M) = L(M_1) \cup L(M_2)$ 。 M 的构造相当的简单, 直观上也是清楚的, 如图 2-11 所示。基本思想是, M 利用非确定性猜想输入在 $L(M_1)$ 中还是在 $L(M_2)$ 中, 然后和相应的自动机完全一样地处理这个输入, 从而 $L(M) = L(M_1) \cup L(M_2)$ 。下面给出形式细节和证明。

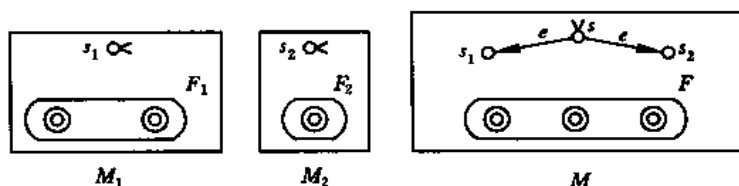


图 2-11

不失一般性, 可以假设 K_1 和 K_2 是两个不相交的集合。接受语言 $L(M_1) \cup L(M_2)$ 的有穷自动机定义如下 (见图 2-11): $M = (K, \Sigma, \Delta, s, F)$, 其中 s 是一个不在 K_1 和 K_2 中的新状态,

$$K = K_1 \cup K_2 \cup \{s\},$$

$$F = F_1 \cup F_2,$$

$$\Delta = \Delta_1 \cup \Delta_2 \cup \{(s, \epsilon, s_1), (s, \epsilon, s_2)\}.$$

即, M 进行任一计算的时候, 首先非确定地选择进入 M_1 的初始状态或者 M_2 的初始状态, 此后相应地模仿 M_1 或 M_2 。形式地, 对任意的 $w \in \Sigma^*$, 对于某个 $q \in F$, $(s, w) \vdash_M^* (q, \epsilon)$

当且仅当对于某个 $q \in F_1, (s_1, w) \vdash_{M_1}^* (q, e)$ 或者对于某个 $q \in F_2, (s_2, w) \vdash_{M_2}^* (q, e)$ 。因此, M 接受 w 当且仅当 M_1 接受 w 或者 M_2 接受 w , 进而 $L(M) = L(M_1) \cup L(M_2)$ 。

(b) 连接。再次设 M_1 和 M_2 是两台非确定型有穷自动机, 要构造一台非确定型有穷自动机 M 使得 $L(M) = L(M_1) \circ L(M_2)$ 。 M 的构造如图 2-12 所示, 它先模拟 M_1 一会儿, 然后非确定地从 M_1 的一个终结状态“跳到” M_2 的初始状态。此后, M 模拟 M_2 。这里略去形式定义和证明。

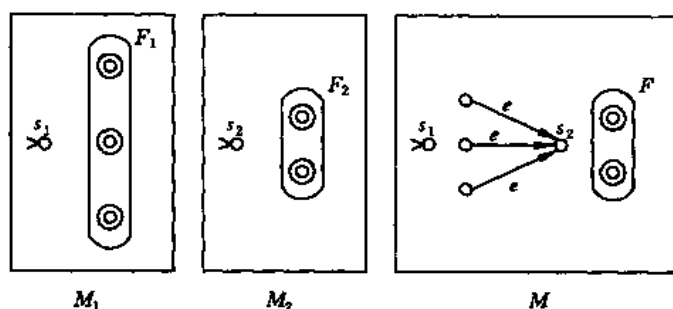


图 2-12

(c) Kleene 星号。设 M_1 是一台非确定型有穷自动机, 要构造一台非确定型有穷自动机 M 使得 $L(M) = L(M_1)^*$ 。 M 的构造类似接连中的构造, 如图 2-13 所示。 M 包括 M_1 的所有状态和所有转移, M_1 的终结状态也是 M 的终结状态。此外, M 有一个新的初始状态 s 。这个新初始状态也是终结状态, 从而 ϵ 被 M 接受。有一个从 s 到 M_1 的初始状态 s_1 的 ϵ 转移, 从而在 M 从状态 s 开始之后 M_1 可以从它的初始状态开始运行。最后, 添加从 M_1 的每一个终结状态到 s_1 的 ϵ 转移。这样一来只要读完 $L(M_1)$ 中的一个字符串, 计算就可以重新从 M_1 的初始状态开始继续进行。

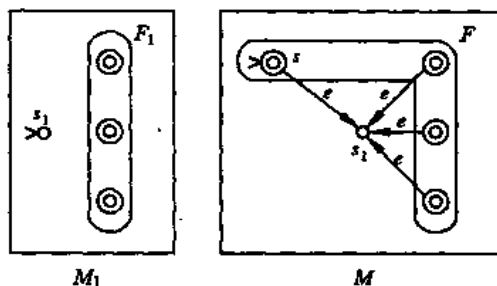


图 2-13

(d) 补。设 $M = (K, \Sigma, \delta, s, F)$ 是一台确定型有穷自动机。那么, 补语言 $\bar{L} = \Sigma^* - L(M)$ 被确定型有穷自动机 $\bar{M} = (K, \Sigma, \delta, s, K - F)$ 接受。也就是说, \bar{M} 和 M 完全一样, 只是恰好交换它们的终结状态与非终结状态。

(e) 交。因为

$$L_1 \cap L_2 = \Sigma^* - ((\Sigma^* - L_1) \cup (\Sigma^* - L_2)),$$

故由在并和补下的封闭性(上述(a)和(d))得到在交下的封闭性。 ■

下面是本节的主要结果, 语言有穷说明的两种重要技术的鉴别, 一个是语言产生器, 而另一个是语言接受器。

定理 2.3.2 一个语言是正则的当且仅当它被有穷自动机接受。

证: 仅当。回想一下, 正则语言类是包括空集 \emptyset 和单元集 $\{a\}$ 在并、连接和 Kleene 星

号下封闭的最小的语言类,其中 a 是一个符号。很明显,空集和所有的单元集确实被有穷自动机接受。又,根据定理 2.3.1,有穷自动机接受的语言在并、连接和 Kleene 星号下封闭。因此,每一个正则语言被一台有穷自动机接受。

例 2.3.1 考虑正则语言 $(ab \cup aab)^*$ 。可以用定理 2.3.1 各部分证明中的方法构造一台非确定型有穷自动机接受这个正则表达式所表示的语言,如图 2-14 所示。◇

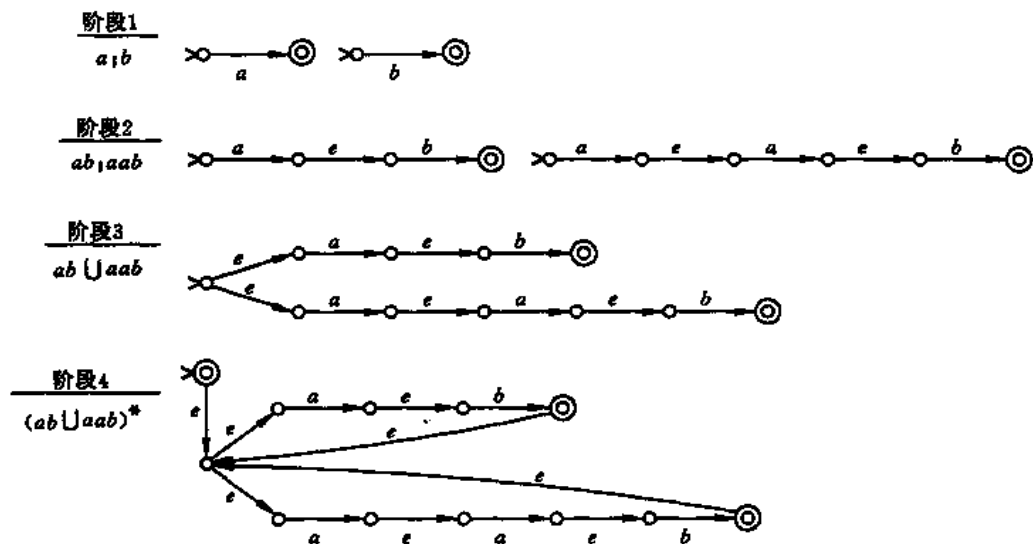


图 2-14

当 $M = (K, \Sigma, \Delta, s, F)$ 是一台有穷自动机(不必是确定型的),要构造一个正则表达式 R 使得 $L(R) = L(M)$ 。我们将把 $L(M)$ 表示成许多(但有穷个)简单语言的并。设 $K = \{q_1, \dots, q_n\}$ 且 $s = q_1$ 。对于 $i, j = 1, \dots, n$ 和 $k = 0, \dots, n$, 令 $R(i, j, k)$ 为 Σ^* 中可以使 M 从状态 q_i 到状态 q_j 且不经过编号大于 k 的中间状态的所有字符串,两端的状态 q_i 和 q_j 的编号 i 和 j 可以大于 k 。即, $R(i, j, k)$ 是由所有从 q_i 到 q_j 且秩小于等于 k 的通路拼成的字符串的集合(回忆 1.6 节中在计算一个关系的自反传递闭包时采用的类似策略,在那里也考虑中间顶点的编号逐渐变大的通路)。当 $k = n$ 时,

$$R(i, j, n) = \{w \in \Sigma^* : (q_i, w) \vdash_M^* (q_j, e)\}$$

因此,

$$L(M) = \bigcup \{R(i, j, n) : q_i \in F\}$$

关键是所有这些集合 $R(i, j, k)$ 是正则的,从而 $L(M)$ 是正则的。

对 k 作归纳,证明每一个 $R(i, j, k)$ 是正则的。对于 $k = 0, R(i, j, 0)$ 当 $i \neq j$ 时为 $\{a \in \Sigma \cup \{e\} : (q_i, a, q_j) \in \Delta\}$, 当 $i = j$ 时为 $\{e\} \cup \{a \in \Sigma : (q_i, a, q_i) \in \Delta\}$ 。这些集合都是有穷的,因而是正则的。

假设对于所有的 $i, j, R(i, j, k-1)$ 是正则语言,则每一个集合 $R(i, j, k)$ 可以用正则运算并、连接和 Kleene 星号及前面的正则语言表示如下:

$$R(i, j, k) = R(i, j, k-1) \cup R(i, k, k-1)R(k, k, k-1)^*R(k, j, k-1)$$

这个等式表示要从 q_i 到 q_j 且不经编号大于 k 的中间状态, M 有两种可能:

- (1) 从 q_i 到 q_j 且不经编号大于 $k-1$ 的中间状态;
- (2) (a) 从 q_i 到 q_k , 然后 (b) 从 q_k 到 q_k 零次或多次, 最后 (c) 从 q_k 到 q_j , 每一次都不经过编号大于 $k-1$ 的中间状态。

因此, 完成归纳证明, 对于所有的 i, j, k , 语言 $R(i, j, k)$ 是正则的。 ■

例 2.3.2 构造图 2-15 中确定型有穷自动机接受的语言的正则表达式。这台自动机接受语言

$$\{w \in \{a, b\}^* : w \text{ 有 } 3k+1 \text{ 个 } b, k \in \mathbb{N}\}$$

直接实现“当”证明中的构造可能是非常烦人的(对于这个简单的问题, 要构造 36 个正则表达式!)。如果假定非确定型有穷自动机具有下述两条简单的性质, 则事情大为简化:

- (a) 它有唯一的终结状态, $F = \{f\}$ 。
- (b) 如果 $(q, u, p) \in \Delta$, 则 $q \neq f$ 且 $p \neq s$ 。即, 没有进入初始状态的转移和离开终结状态的转移。

这种“特殊形式”不失一般性, 因为可以给任意一台自动机 M 添加一个新的初始状态 s 和一个新的终结状态 f 以及从 s 到 M 的初始状态和从 M 的每一个终结状态到 f 的 ϵ 转移(见图 2-16(a), 把图 2-15 中的自动机变成这种“特殊形式”)。现在按照构造的要求, 把自动机的状态编号为 q_1, q_2, \dots, q_n , 使得 $s = q_{n-1}, f = q_n$ 。显然, 所求的正则表达式是 $R(n-1, n, n)$ 。

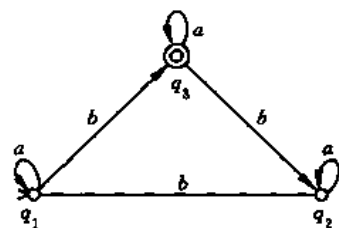


图 2-15

下面像证明中提出的那样, 首先计算 $R(i, j, 0)$, 然后用它们计算 $R(i, j, 1)$, 如此继续下去。在每一阶段, 把 $R(i, j, k)$ 作为标记绘在从状态 q_i 到状态 q_j 的箭头上。删去标记 \emptyset 的箭头和标记 $\{e\}$ 的环。按照这些约定, 最初的自动机正确地绘出了 $R(i, j, 0)$ 的值, 见图 2-16(a)。(是这样的, 因为在这台自动机中对于每一对状态 (q_i, q_j) , Δ 中至多有一个 (q_i, u, q_j) 形式的转移。在其他自动机中, 可以像证明中提出的那样用并运算把从 q_i 到 q_j 的所有箭头合并在一起。)

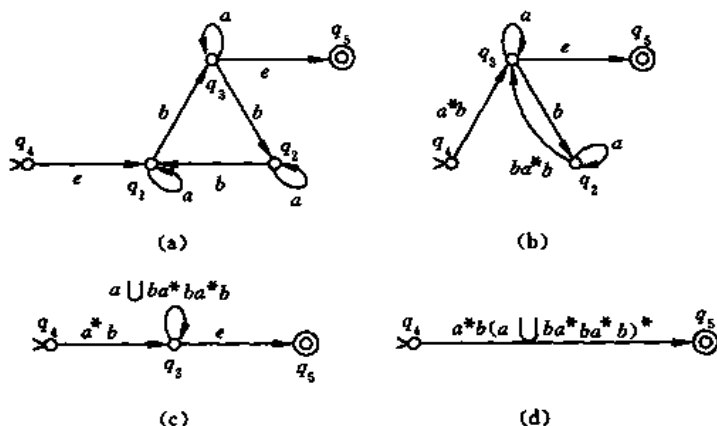


图 2-16

现在计算 $R(i, j, 1)$, 如图 2-16(b) 所示。马上注意到在后面的构造不再需要考虑状态 q_1 , 所有导致 M 经过状态 q_1 接受的字符串都已经考虑过了并且计入 $R(i, j, 1)$ 中。可以说状态 q_1 被消去。在某种意义上, 我们把图 2-16(a) 中的有穷自动机转变成一台等价的广义有穷自动机, 它的转移不仅可以标记 Σ 中的符号或 ϵ , 而且可以标记一个正则表达式。因为已删去 q_1 , 所以得到的广义有穷自动机比原来的少一个状态。

仔细地考察一下在消去一个状态 q 时要做些什么(见图 2-17)。对于每一对状态 q_i 和 q_j , 这里 $q_i, q_j \neq q$, 如果有从 q_i 到 q 标记 α 的箭头和从 q 到 q_j 标记 β 的箭头, 则加一条从 q_i 到 q_j 标记 $\alpha\gamma^*\beta$ 的箭头, 其中 γ 是从 q 到自身的箭头的标记(如果没有这样的箭头, 则 $\gamma = \emptyset$ 。于是, $\gamma^* = \{\epsilon\}$, 标记变成 $\alpha\beta$)。如果原来有一个从 q_i 到 q_j 标记 δ 的箭头, 则新的箭头标记 $\delta \cup \alpha\gamma^*\beta$ 。

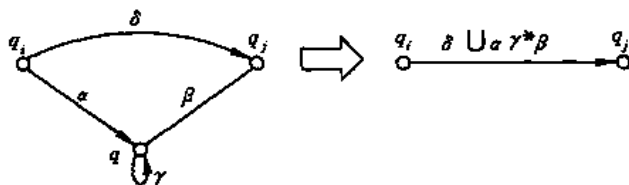


图 2-17

继续这样进行, 消去状态 q_2 得到图 2-16(c) 中的 $R(i, j, 2)$, 最后消去 q_3 。现在已消除初始状态和终结状态以外的所有状态, 把广义有穷自动机化简成一个从初始状态到终结状态的转移。这个转移的标记就是等价于 M 的正则表达式:

$$R = R(4, 5, 5) = R(4, 5, 3) = a^*b(ba^*ba^*b \cup a)^*$$

它确实是 $\{w \in \{a, b\}^*: w \text{ 有 } 3k+1 \text{ 个 } b, k \in \mathbb{N}\}$ 。

◇

习 题

- 2.3.1 在定理 2.3.1(d) 的证明中, 为什么一定要求 M 是确定型的? 如果交换一台非确定型有穷自动机的终结状态与非终结状态, 会怎么样?
- 2.3.2 在定理 2.3.1(c) 的证明中, 如果直接把 s_1 作为终结状态和添加从 F_1 的每一个成员回到 s_1 的箭头(不引入新的初始状态), 会出什么错误?
- 2.3.3 直接构造一台有穷自动机接受两台有穷自动机接受的语言的交。(提示: 用这两台有穷自动机的状态集合的笛卡儿积作为状态集合。) 当两个语言用非确定型有穷自动机给出时, 这两个构造方法(一个是正文中给出的, 一是你在这里给出的)哪一个更有效?
- 2.3.4 利用定理 2.3.1 证明中的构造方法, 构造接受下述语言的有穷自动机:
 - (a) $a^*(ab \cup ba \cup \epsilon)b^*$
 - (b) $((a \cup b)^*(a \cup c)^*)^*$
 - (c) $((ab)^* \cup (bc)^*)ab$
- 2.3.5 构造一台简单的非确定型有穷自动机接受语言 $(ab \cup aba)^*a$ 。然后对它运用定理

2.3.1(c) 的证明中的构造方法, 获得接受 $((ab \cup aba)^*a)^*$ 的非确定型有穷自动机。

2.3.6 设 $L, L' \subseteq \Sigma^*$ 。定义下述语言:

1. L 的前缀集合

$$\text{Pref}(L) = \{w \in \Sigma^* \mid \text{对于某个 } x \in L \text{ 和 } y \in \Sigma^*, x = wy\}.$$

2. L 的后缀集合

$$\text{Suf}(L) = \{w \in \Sigma^* \mid \text{对于某个 } x \in L \text{ 和 } y \in \Sigma^*, x = yw\}.$$

3. L 的子序列集合

$$\text{Subseq}(L) = \{w_1 w_2 \cdots w_k \mid k \in \mathbb{N}, w_i \in \Sigma^* (i = 1, \dots, k) \text{ 且有字符串 } x = x_0 w_1 x_1 w_2 x_2 \cdots w_k x_k \in L\}.$$

4. L 除以 L' 的右商

$$L/L' = \{w \in \Sigma^* \mid \text{对于某个 } x \in L', wx \in L\}.$$

5. $\text{Max}(L) = \{w \in L \mid \text{如果 } x \neq \epsilon \text{ 则 } wx \notin L\}.$

6. $L^R = \{w^R \mid w \in L\}.$

证明: 如果 L 被有穷自动机接受, 则下述语言也被有穷自动机接受:

(a) $\text{Pref}(L)$

(b) $\text{Suf}(L)$

(c) $\text{Subseq}(L)$

(d) L/L' , 其中 L' 可被有穷自动机接受。

(e) L/L' , 其中 L' 是任意的语言。

(f) $\text{Max}(L)$

(g) L^R

2.3.7 运用例 2.3.2 中的构造方法获得对应于图 E2-6 中每一台有穷自动机的正则表达式。尽量简化得到的正则表达式。

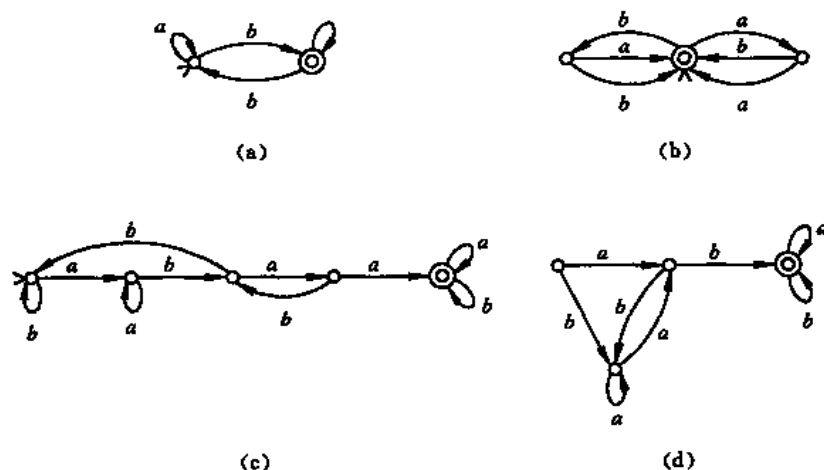


图 E2-6

2.3.8 对于任一自然数 $n \geq 1$, 定义非确定型有穷自动机 $M_n = (K_n, \Sigma_n, \Delta_n, s_n, F_n)$, 其中 $K_n = \{q_1, q_2, \dots, q_n\}$, $s_n = q_1$, $F_n = \{q_1\}$, $\Sigma_n = \{a_{ij}; i, j = 1, \dots, n\}$ 以及 $\Delta_n = \{(q_i, a_{ij}, q_j); i, j = 1, \dots, n\}$ 。

(a) 用文字描述 $L(M_n)$ 。

(b) 写出 $L(M_3)$ 的一个正则表达式。

(c) 写出 $L(M_5)$ 的一个正则表达式。

猜想对于每一个多项式 p , 存在一个 n 使得 $L(M_n)$ 没有长度小于 $p(n)$ 个符号的正则表达式。

2.3.9 (a) 类似习题 2.1.5 定义非确定型双带有穷自动机及其接受字符串有序对集合的概念。

(b) 证明 $\{(a^m b, a^n b^p); n, m, p \geq 0 \text{ 且 } n = m \text{ 或 } n = p\}$ 被一台非确定型双带有穷自动机接受。

(c) 将会看到(习题 2.4.9) 非确定型双带有穷自动机不是总可以转换成确定型双带有穷自动机。既然如此, 可以推广定理 2.3.1 证明中使用的哪些构造方法用来证明下述集合的封闭性质?

(i) 非确定型双带有穷自动机接受的字符串有序对集合。

(ii) 确定型双带有穷自动机接受的字符串有序对集合。

解释你的回答。

2.3.10 如果存在 k 使得对于任意的 w , 是否 $w \in L$ 仅依赖于 w 的最后 k 个符号, 则称 L 是一个定语言。

(a) 更形式地更新给出这个定义。

(b) 证明每一个定语言被有穷自动机接受。

(c) 证明定语言类在并和补运算下封闭。

(d) 给出一个定语言 L , 使得 L^* 不是定语言。

(e) 给出定语言 L_1 和 L_2 , 使得 $L_1 L_2$ 不是定语言。

2.3.11 设 Σ 和 Δ 是两个字母表, h 是从 Σ 到 Δ^* 的函数。把 h 如下扩张成从 Σ^* 到 Δ^* 的函数:

$$h(e) = e.$$

$$h(w\sigma) = h(w)h(\sigma), \text{ 对任意的 } w \in \Sigma^*, \sigma \in \Sigma.$$

例如, 如果 $\Sigma = \Delta = \{a, b\}$, $h(a) = ab$, $h(b) = aab$, 则

$$\begin{aligned} h(aab) &= h(aa)h(b) \\ &= h(a)h(a)h(b) \\ &= ababaab \end{aligned}$$

采取这种方式用函数 $h: \Sigma \rightarrow \Delta^*$ 定义的函数 $h: \Sigma^* \rightarrow \Delta^*$ 叫做同态。

设 h 是从 Σ^* 到 Δ^* 的同态。

(a) 证明: 如果 $L \subseteq \Sigma^*$ 被有穷自动机接受, 则 $h[L]$ 也被有穷自动机接受。

(b) 证明: 如果 $L \subseteq \Delta^*$ 被有穷自动机接受, 则 $\{w \in \Sigma^*; h(w) \in L\}$ 也被有穷自动机接受。(提示: 从一台接受 L 的确定型有穷自动机 M 出发, 构造一台

有穷自动机, 当它读一个输入符号 a 时试图模拟 M 对输入 $h(a)$ 的动作。))

2.3.12 在习题 2.1.4 中介绍了确定型有穷状态转换器。证明: 如果 L 被有穷自动机接受, f 可用确定型有穷状态转换器计算, 则下述各条成立:

- (a) $f[L]$ 被有穷自动机接受。
- (b) $f^{-1}[L]$ 被有穷自动机接受。

2.4 正则语言与非正则语言

前两节的结果证实正则语言在各种运算下封闭, 可以用正则表达式及确定型或非确定型有穷自动机说明, 这些事实(使用一个或同时使用几个) 提供了证明语言是正则的各种技术。

例 2.4.1 设 $\Sigma = \{0, 1, \dots, 9\}$, $L \subseteq \Sigma^*$ 是可以被 2 或 3 整除的非负整数的十进制表示的集合(前面没有多余的 0)。例如, $0, 3, 6, 244 \in L$, 而 $1, 03, 00 \notin L$ 。 L 是正则的, 我们把证明分成四部分。

令 L_1 是非负整数十进制表示的集合。容易看到

$$L_1 = 0 \cup \{1, 2, \dots, 9\} \Sigma^*$$

由于 L_1 是用正则表达式表示的, 故它是一个正则语言。

令 L_2 是可以被 2 整除的非负整数的十进制表示的集合。 L_2 正好是以 0, 2, 4, 6 或 8 结尾的 L_1 的成员组成的集合, 即

$$L_2 = L_1 \cap \Sigma^* \{0, 2, 4, 6, 8\}$$

根据定理 2.3.1(e), L_2 是正则的。

令 L_3 是可以被 3 整除的非负整数的十进制表示的集合。回忆一下, 一个数可以被 3 整除当且仅当它的数字之和可以被 3 整除。构造一台有穷自动机用它的有穷控制器保存输入数字的模 3 和。 L_3 是这台有穷自动机接受的语言与 L_1 的交。这台自动机在图 2-18 中画出。

最后, $L = L_2 \cup L_3$, 它一定是一个正则语言。◇

虽然我们现在有各种强有力的技术证明语言是正则的, 但是还没有办法证明语言不是正则的。根据基本原理, 我们知道存在非正则语言, 因为正则表达式(和有穷自动机) 的数目是可数的, 而语言的数目是不可数的。但是证明一个具体的语言不是正则的需要专门的工具。

为所有正则语言共有、而不为某些非正则语言具有的两条性质可以直观地叙述如下:

(1) 当从左到右扫描一个字符串时, 为了确定这个字符串最终是否在该语言中, 所需要的存储量必须是有界的、事先固定的和只与该语言有关而与具体的输入字符串无关。例如, 可以预料 $\{a^n b^n; n \geq 0\}$ 不是正则的, 因为很难想象如何构造一台有穷状态的设备能够正确地记住到达 a 和 b 之间的界线时它看见多少个 a , 从而可以用这个数与 b 的个数进行比较。

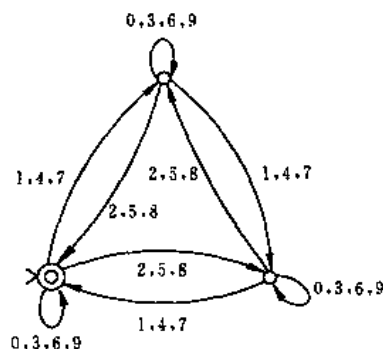


图 2-18

(2) 有无穷多个字符串的正则语言用带圈的有穷自动机和含 Kleene 星号的正则表达式表示。这样的语言一定有具有某种简单的重复构造的无穷子集, 这种构造是由对应的正则表达式中的 Kleene 星号或者有穷自动机的状态图中的圈产生的。这可能引导我们预料例如 $\{a^n; n \geq 1 \text{ 是一个素数}\}$ 不是正则的, 因为在素数集合中没有简单的周期性。

这些直观的想法是正确的, 但在形式证明中使用还不够精确。现在证明一个定理, 它体现了部分这种直观想法, 并且作为简单的推论, 给出某些语言的非正则性。

定理 2.4.1 设 L 是一个正则语言, 则存在正整数 $n \geq 1$ 使得任一字符串 $w \in L$ 只要 $|w| \geq n$ 就可以写成 $w = xyz$, 其中 $y \neq \epsilon$, $|xy| \leq n$ 且对每一个 $i \geq 0$, $xy^iz \in L$ 。

证: 由于 L 是正则的, 它被一台确定型有穷自动机 M 接受。设 M 有 n 个状态, $w \in L$ 是一个长度大于等于 n 的字符串。现在考虑 M 对 w 的前 n 步计算:

$$(q_0, w_1w_2\cdots w_n) \vdash_M (q_1, w_2\cdots w_n) \vdash_M \cdots \vdash_M (q_n, \epsilon)$$

其中 q_0 是 M 的初始状态, $w_1\cdots w_n$ 是 w 的前 n 个符号。因为 M 只有 n 个状态, 而在上面的计算中有 $n+1$ 个格局 $(q_i, w_{i+1}\cdots w_n)$, 所以根据鸽巢原理存在 i 和 j ($0 \leq i < j \leq n$) 使得 $q_i = q_j$ 。即, 字符串 $y = w_{i+1}\cdots w_j$ 把 M 从状态 q_i 带回到状态 q_i , 而且由于 $i < j$, y 不是空串。于是, 可以从 w 中删去 y 或者接在 w 的第 j 个符号后面重复任意次 y , 所得到的字符串仍被 M 接受。即, 对于每一个 $i \geq 0$, M 接受 xy^iz , 其中 $x = w_1\cdots w_i$ 和 $z = w_{j+1}\cdots w_m$ 。最后, 注意到 xy 的长度 j 根据定义不超过 n , 符合要求。 ■

这个定理是一系列泵定理中的一个, 它们都断言在某些字符串中存在某些点, 可以在这些地方重复插入一个子串而不影响字符串的接受性。就数学命题的结构来说, 这个泵定理是我们在本书到现在为止见到的最复杂的定理。虽然它的证明和应用都很简单, 但是它含有五个交错的量词。再来看看它说些什么:

对每一个正则语言 L ,

存在 $n \geq 1$, 使得

对 L 中每一个长度大于等于 n 的字符串 w ,

存在字符串 x, y, z , 使得 $w = xyz$, $y \neq \epsilon$, $|xy| \leq n$, 并且

对每一个 $i \geq 0$, $xy^iz \in L$ 。

正确地运用这个定理是很微妙的。把运用这个定理看作你与一名对手之间的游戏常常是有用的, 你作为证明者正在努力证明给定的语言 L 不是正则的, 而对手一直坚持 L 是正则的。定理称对于给定的 L , 一开始对手应该提供一个数 n , 然后你在 L 中找一个长度大于等于 n 的字符串 w , 接着对手必须把 w 适当地分解成 xyz 。最后, 你成功地指出一个 i , 对于这个 i , xy^iz 不在 L 中。如果不管对手多么高明, 你都有总能赢的策略, 则你证明 L 不是正则的。

根据这个定理, 本节早先叙述的两个语言不是正则的。

例 2.4.2 语言 $L = \{a^ib^j; i \geq 0\}$ 不是正则的。如果 L 是正则的, 则存在满足定理 2.4.1 中要求的整数 n 。考虑字符串 $w = a^n b^n \in L$ 。根据定理, 可以把它重写成 $w = xyz$ 使得 $|xy| \leq n$ 且 $y \neq \epsilon$, 即 $y = a^i$, 其中 $i > 0$ 。但是, $xz = a^{n-i} b^n \notin L$, 与定理矛盾。 ◇

例 2.4.3 语言 $L = \{a^n; n \text{ 是素数}\}$ 不是正则的。假设 L 是正则的, 令 x, y, z 和定理 2.4.1 中规定的一样。那么, $x = a^p, y = a^q, z = a^r$, 其中 $p, r \geq 0$ 且 $q > 0$ 。根据定理, 对每一个 $n \geq 0$, $xy^n z \in L$, 即 $p + nq + r$ 是素数。但这是不可能的。令 $n = p + 2q + r + 2$,

则 $p + nq + r = (q + 1)(p + 2q + r)$ 是两个大于 1 的自然数的乘积。 \diamond

例 2.4.4 有时使用封闭性证明一个语言不是正则的。例如

$$L = \{w \in \{a, b\}^* : w \text{ 中 } a \text{ 和 } b \text{ 的个数相同}\}$$

不是正则的。这是因为如果 L 是正则的, 则根据在交下的封闭性(定理 2.3.1(e)), $L \cap a^*b^*$ 也是封闭的。而后者正好是 $\{a^n b^n : n \geq 0\}$, 刚刚证明它不是正则的。 \diamond

事实上, 可以用 n 种方式对定理 2.4.1 作实质性的加强(习题 2.4.11 是其中的一个)。

习 题

2.4.1 对于任意固定的 $p, q \in \mathbb{N}$, 集合 $\{p + nq : n = 0, 1, 2, \dots\}$ 称作**算术级数**。

(a) (容易) 证明: 如果 $L \subseteq \{a\}^*$ 且 $\{n : a^n \in L\}$ 是一个算术级数, 则 L 是正则的。

(b) 证明: 如果 $L \subseteq \{a\}^*$ 且 $\{n : a^n \in L\}$ 是有穷个算术级数的并, 则 L 是正则的。

(c) (较难) 证明: 如果 $L \subseteq \{a\}^*$ 是正则的, 则 $\{n : a^n \in L\}$ 是有穷个算术级数的并。(这是(b)的逆命题。)

(d) 证明: 如果 Σ 是任一字母表, $L \subseteq \Sigma^*$ 是正则的, 则 $\{|w| : w \in L\}$ 是有穷个算术级数的并。(提示: 利用(c)。)

2.4.2 设 $D = \{0, 1\}$ 和 $T = D \times D \times D$ 。如果把 T 中的符号当作列向量, 可以把两个二进制数的加式形象地表成 T^* 中的字符串。例如,

$$\begin{array}{r} 0101 \\ + 0110 \\ \hline 1011 \end{array}$$

可以形象地表示成下述四个符号的字符串

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

证明: T^* 中所有表示正确的加式的字符串组成的集合是一个正则语言。

2.4.3 证明下述语言是或者不是正则语言。一个数的十进制表示是用普通方式写出来的这个数, 它是字母表 $\{0, 1, \dots, 9\}$ 上的一个字符串。例如, 13 的十进制表示是一个长度为 2 的字符串。在一进制表示中只使用一个符号 I , 5 的一进制表示是 $IIIII$ 。

(a) $\{w : w \text{ 是 } 7 \text{ 的倍数的一进制表示}\}$ 。

(b) $\{w : w \text{ 是 } 7 \text{ 的倍数的十进制表示}\}$ 。

(c) $\{w : w \text{ 是 } n \text{ 的一进制表示并且存在一对都大于 } n \text{ 的孪生素数 } p \text{ 和 } p + 2\}$ 。

(d) $\{w : w \text{ 是 } 10^n \text{ 的一进制表示, 其中 } n \geq 1\}$ 。

(e) $\{w : w \text{ 是 } 10^n \text{ 的十进制表示, 其中 } n \geq 1\}$ 。

(f) $\{w : w \text{ 是在 } 1/7 \text{ 的无穷小数展开中出现的十进制数字序列}\}$ 。(例如, 5714 是这样一个序列, 因为 $1/7 = 0.14285714285714\dots$)

- 2.4.4 证明: $\{a^n b a^m b a^{m+n}; n, m \geq 1\}$ 不是正则的。
- 2.4.5 使用泵定理和在交下的封闭性, 证明下述语言不是正则的:
- $\{ww^R; w \in \{a, b\}^*\}$
 - $\{ww; w \in \{a, b\}^*\}$
 - $\{w\bar{w}; w \in \{a, b\}^*\}$, 这里 \bar{w} 表示把 w 中的 a 都换成 b , 把 b 都换成 a 得到的字符串。
- 2.4.6 如果字母表 $\{(,)\}$ 上的字符串 x 满足下述条件则称作平衡的: (i) 在 x 的任一前缀中 $($ 的个数不少于 $)$ 的个数; (ii) x 中 $($ 的个数等于 $)$ 的个数。也就是说, 如果能从一个合法的算术表达式中删去所有的变量、数和运算符得到 x , 则 x 是平衡的。(见下一章稍微方便一些的定义。) 证明: $\{(,)\}^*$ 中所有平衡的字符串的集合不是正则的。
- 2.4.7 证明: 对于任一确定型有穷自动机 $M = (K, \Sigma, \delta, s, F)$, M 接受一个无穷语言当且仅当 M 接受一个长度大于等于 $|K|$ 且小于 $2|K|$ 的字符串。
- 2.4.8 下述各命题是真是假? 解释你的回答。(假设字母表 Σ 是固定的。)
- 正则语言的每一个子集都是正则的。
 - 每一个正则语言都有一个正则的真子集。
 - 如果 L 是正则的, 则 $\{xy; x \in L \text{ 且 } y \notin L\}$ 也是正则的。
 - $\{w; w = w^R\}$ 是正则的。
 - 如果 L 是正则的, 则 $\{w; w \in L \text{ 且 } w^R \in L\}$ 也是正则的。
 - 如果 C 是任一正则语言的集合, 则 $\bigcup C$ 是一个正则语言。
 - $\{xyx^R; x, y \in \Sigma^*\}$ 是正则的。
- 2.4.9 在习题 2.1.5 中定义了确定型双带有穷自动机。证明: $\{(a^n b, a^m b^p); n, m, p > 0, n = m \text{ 或 } n = p\}$ 不被任何确定型双带有穷自动机接受。(提示: 假设一台确定型双带有穷自动机 M 接受该集合, 则对于每一个 n , M 接受 $(a^n b, a^{n+1} b^n)$ 。用泵定理证明对于某个 $n \geq 0$ 和 $q > 0$, M 接受 $(a^n b, a^{n+1} b^{n+q})$ 。矛盾。) 于是, 由习题 2.3.9, 非确定型双带有穷自动机不一定可以转换成确定型双带有穷自动机。又由习题 2.1.5, 确定型双带有穷自动机接受的集合类在并运算下不封闭。
- 2.4.10 **双带头有穷自动机** 是一种有两个带头的有穷自动机。两个带头可以独立地在输入带上从左向右移动。与双带有穷自动机(习题 2.1.5)一样, 状态集合分成两部分, 每一部分对应于一个带头的读和移动。一个字符串被接受当且仅当两个带头同时停留在这个字符串的末尾时有穷控制器处于一个终结状态。双带头有穷自动机可以是确定型的或者非确定型的。用你自己设计的状态图, 证明双带头有穷自动机接受下述语言:
- $\{a^n b^n; n \geq 0\}$
 - $\{wcw; w \in \{a, b\}^*\}$
 - $\{a^1 b a^2 b a^3 b \cdots b a^k b; k \geq 1\}$
- 其中哪几个你可以用确定型的机器?
- 2.4.11 本题给出泵定理 2.4.1 的更强形式, 其目标是使“抽取”部分尽可能的长。设

$M = (K, \Sigma, \delta, s, F)$ 是一台确定型有穷自动机, w 是 $L(M)$ 中长度不小于 $|K|$ 的任一字符串。证明: 存在字符串 x, y 和 z 使得 $w = xyz$, $|y| \geq (|w| - |K| + 1)/|K|$, 并且对于每一个 $n \geq 0, xy^n z \in L(M)$ 。

- 2.4.12 令 $D = \{0, 1\}, T = D \times D \times D$ 。两个二进制数的乘式也可以表示成 T^* 中的一个字符串。例如, $10 \times 5 = 50$, 即

$$\begin{array}{r} 001010 \\ \times 000110 \\ \hline 110010 \end{array}$$

可以形象地表成下述六个符号的字符串

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

证明: T^* 中表示正确的乘式的所有字符串组成的集合不是正则的。(提示: 考虑 $(2^n + 1) \times (2^n + 1)$ 。)

- 2.4.13 设 $L \subseteq \Sigma^*$ 是一个语言, 定义 $L_n = \{x \in L: |x| \leq n\}$ 。 L 的密度是函数 $d_L(n) = |L_n|$ 。

(a) $(a \cup b)^*$ 的密度是什么?

(b) $ab^*ab^*ab^*a$ 的密度是什么?

(c) $(ab \cup aab)^*$ 的密度是什么?

(d) 证明: 任一正则语言的密度要么以一个多项式为上界, 要么以一个指数函数 (2^{cn} 形式的函数, 其中 c 是一个常数) 为下界。换句话说, 正则语言的密度函数不可能是像 $n^{\log n}$ 之类中间增长率的函数。(提示: 考虑接受 L 的确定型有穷自动机和它的状态图中的所有圈——顶点不重复的闭通路。如果没有两个圈共有一个顶点, 会怎么样? 如果有两个圈共有一个顶点, 又会怎么样?)

2.5 状态最小化

在上一节证实了我们的猜想: 确定型有穷自动机是一种贫乏的计算机模型。基于有穷自动机的计算连比较一个字符串中 a 和 b 的个数这样平常的计算任务都不能胜任。但是, 有穷自动机作为计算机和算法的基础部分是有用的。在这方面, 能够使给定的确定型有穷自动机的状态数最小化, 即给出一台状态数尽可能少的等价的确定型有穷自动机是重要的。下面给出一些必要的概念和结果, 进而得到这样一个状态最小化算法。

给定一台确定型有穷自动机, 可能有简单的办法删去若干状态。例如, 取图 2-19 中的确定型有穷自动机, 它接受语言 $L = (ab \cup ba)^*$ (不难验证)。考虑状态 q_7 。这个

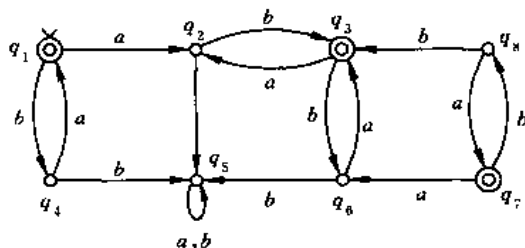


图 2-19

状态明显是不可达的,因为在自动机的状态图中没有从初始状态到它的通路。这是可以对任意确定型有穷自动机进行的最简单的优化:删去所有不可达的状态及所有进入或离开它们的转移。事实上,这种优化已隐含在把非确定型有穷自动机变成等价的确定型有穷自动机的转化中(回忆例 2.2.4)。在那里,不考虑从生成的自动机的初始状态不能达到的所有状态(原来的自动机的状态的集合)。

由于可达状态集合可以定义为 $\{s\}$ 在关系 $\{(p,q): \text{对某个 } a \in \Sigma, \delta(p,a) = q\}$ 下的闭包,容易在多项式时间内识别可达状态。所有可达状态的集合可以用下述简单的算法计算:

```

R := {s};
while 存在状态  $p \in R$  和  $a \in \Sigma$  使  $\delta(p,a) \notin R$  do
    把  $\delta(p,a)$  加入  $R$ 。

```

但是,删去不可达状态之后剩下的自动机(图 2-20)的状态仍然多于实际的需要,这次的理由更微妙一些。例如,状态 q_4 和 q_6 是等价的,因而可以把它们合并成一个状态。它的精确含义是什么?直观上,称这两个状态等价的理由是从其中任一状态开始,导致自动机接受的字符串完全一样。

下述定义体现了相对于一个语言具有“共同命运”的字符串之间的类似关系。

定义 2.5.1 设 $L \subseteq \Sigma^*$ 是一个语言, $x, y \in \Sigma^*$ 。如果对所有的 $z \in \Sigma^*$, $xz \in L$ 当且仅当 $yz \in L$,则称 x 和 y 是相对于 L 等价的,记作 $x \approx_L y$ 。注意 \approx_L 是一个等价关系。

也就是说,如果 x 与 y 都属于 L 或者都不属于 L ,并且把任一字符串附加在它们的后面所得到的两个字符串也同时属于 L 或者同时不属于 L ,则 $x \approx_L y$ 。

例 2.5.1 设 x 是一个字符串。当 L 根据上下文不言自明时,用 $[x]$ 表示这种 x 所在的相对于 L 的等价类。例如,对于图 2-20 中的自动机所接受的语言 $L = (ab \cup ba)^*$,不难看到 \approx_L 有四个等价类:

- (1) $[e] = L$
- (2) $[a] = La$
- (3) $[b] = Lb$
- (4) $[aa] = L(aa \cup bb)\Sigma^*$

在(1)中,对于任一字符串 $x \in L$,包括 $x = e$,使 $xz \in L$ 的 z 恰好都是 L 的成员。在(2)中,对于任一 $x \in La$,为了使 $xz \in L$ 需要 bL 形式的 z 。类似地,在(3)中, z 的形式为 aL 。最后,在(4)中,没有 z 能把具有 $L(aa \cup bb)$ 中前缀的字符串修补成 L 的成员。换句话说,集合(1)中的所有字符串相对于包含在 L 中具有同样的命运;(2)、(3)和(4)也一样。最后,容易看到这四个集合穷尽了 Σ^* 中的所有字符串。因此,它们是 \approx_L 的全部等价类。◇

注意: \approx_L 是根据语言而不是根据自动机给出字符串的关系。自动机提供另一种关系,它不如 \approx_L 基本,描述如下。

定义 2.5.2 设 $M = (K, \Sigma, \delta, s, F)$ 是一台确定型有穷自动机。直观上,如果两个字

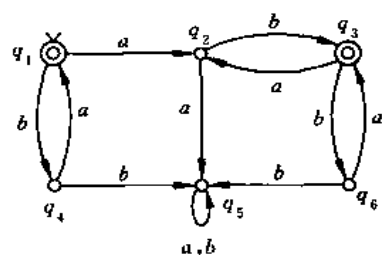


图 2-20

字符串 $x, y \in \Sigma^*$ 把 M 从 s 带到同一个状态, 则称 x 与 y 是相对于 M 等价的, 记作 $x \sim_M y$ 。形式地, $x \sim_M y$ 当且仅当存在状态 q 使得 $(s, x) \vdash_M^*(q, e)$ 和 $(s, y) \vdash_M^*(q, e)$ 。

\sim_M 也是一个等价关系。它的等价类可以用 M 的状态标识。更准确地说, 可以用从 s 能够到达的状态标识, 因而在对应的等价类中至少有一个字符串。对应于 M 的状态 q 的等价类记作 E_q 。

例 2.5.1(续) 例如, 对于图 2-20 中的自动机 M , \sim_M 的等价类为

- (1) $E_{q_1} = (ba)^*$
- (2) $E_{q_2} = La$
- (3) $E_{q_3} = (ba)^* abL$
- (4) $E_{q_4} = b(ab)^*$
- (5) $E_{q_5} = L(bb \cup aa)\Sigma^*$
- (6) $E_{q_6} = (ba)^* abLb$

其中 $L = (ab \cup ba)^*$ 是 M 接受的语言。它们也构成 Σ^* 的一个划分。 \diamond

这两个重要的等价关系, 一个与语言相关联, 另一个与自动机相关联, 两者之间的关系如下。

定理 2.5.1 对于任意的确定型有穷自动机 $M = (K, \Sigma, \delta, s, F)$ 和任意的字符串 $x, y \in \Sigma^*$, 如果 $x \sim_M y$, 则 $x \approx_{L(M)} y$ 。

证: 对于任意的字符串 $x \in \Sigma^*$, 令 $q(x) \in K$ 是使 $(s, x) \vdash_M^*(q(x), e)$ 的唯一状态。注意到, 对于任意的 $x, z \in \Sigma^*$, $xz \in L(M)$ 当且仅当对于某个 $f \in F$, $(q(x), z) \vdash_M^*(f, e)$ 。现在, 如果 $x \sim_M y$, 则根据 \sim_M 的定义, $q(x) = q(y)$ 。于是, $x \sim_M y$ 蕴涵对于所有的 $z \in \Sigma^*$,

$$xz \in L(M) \text{ 当且仅当 } yz \in L(M)$$

即 $x \approx_{L(M)} y$ 。 \blacksquare

定理 2.5.1 的一种很有启发的表述是说 \sim_M 是 $\approx_{L(M)}$ 的细化。一般地, 如果对于所有 x 和 y , $x \sim y$ 蕴涵 $x \approx y$, 则称等价关系 \sim 是 \approx 的细化。如果 \sim 是 \approx 的一个细化, 则 \sim 的每一个等价类包含在 \approx 的某个等价类中。也就是说, \approx 的每一个等价类是 \sim 的一个等价类或几个等价类的并。例如, 美国的任意两个城市在同一个县中的等价关系是在同一个州中的等价关系的细化。

例 2.5.1(续) 对于这个多次提到的例子, 根据定理 2.5.1, 关于图 2-20 中的自动机 M , \sim_M 的等价类在这个意义下“细化” $\approx_{L(M)}$ 的等价类。例如, E_{q_5} 与 $[aa]$ 重合, 而 $[e]$ 等于 E_{q_1} 与 E_{q_3} 的并。 \diamond

定理 2.5.1 蕴涵一件有关 M 以及接受相同语言 $L(M)$ 的任意一台其他的自动机的非常重要的事情: 它的状态数一定不小于 $\approx_{L(M)}$ 的等价类数。换句话说, $L(M)$ 的等价类数是任一等价于 M 的自动机的状态数的天然下界。能达到这个下界吗? 下面证明确实可以。

定理 2.5.2 (Myhill-Nerode 定理) 设 $L \subseteq \Sigma^*$ 是一个正则语言, 则有一台接受 L 的确定型有穷自动机, 它的状态恰好与 \approx_L 的等价类一样多。

证: 和前面一样, 把字符串 $x \in \Sigma^*$ 在等价关系 \approx_L 中的等价类记作 $[x]$ 。给定 L , 要构造一台确定型有穷自动机(关于 L 的标准自动机) $M = (K, \Sigma, \delta, s, F)$ 使得 $L = L(M)$ 。 M

定义如下:

$K = \{[x] : x \in \Sigma^*\}$, \approx_L 的等价类集合,

$s = [e]$, e 在 \approx_L 中的等价类,

$F = \{[x] : x \in L\}$,

对于每一个 $[x] \in K$ 和 $a \in \Sigma$, $\delta([x], a) = [xa]$.

怎么知道 K 是有穷的, 即 \approx_L 有有穷个等价类? 由于 L 是正则的, 它肯定被一台确定型有穷自动机 M' 接受。根据前面的定理, $\sim_{M'}$ 是 \approx_L 的一个细化, 因而 \approx_L 的等价类不多于 $\sim_{M'}$ 的等价类, 即不多于 M' 的状态。所以, K 是有穷的。还可以证明 δ 是良定义的, 即 $\delta([x], a) = [xa]$ 不依赖于字符串 $x \in [x]$ 。由于 $x \approx_L x'$ 当且仅当 $xa \approx_L x'a$, 这是显然的。

下面证明 $L = L(M)$ 。首先要证对于所有的 $x, y \in \Sigma^*$, 有

$$([x], y) \vdash_M^* ([xy], e) \quad (1)$$

对 $|y|$ 作归纳证明。当 $y = e$ 时, 这是显然的。假设对于所有长度不大于 n 的 y , (1) 成立。考虑长度为 $n+1$ 的 y , 记 $y = y'a$, 则由归纳假设, $([x], y'a) \vdash_M^* ([xy'], a) \vdash_M ([xy], e)$ 。

现在, 用 (1) 完成整个证明: 对于所有的 $x \in \Sigma^*$, $x \in L(M)$ 当且仅当对于某个 $q \in F$, $([e], x) \vdash_M^* (q, e)$ 。根据 (1), 后者等于说 $[x] \in F$ 。又根据 F 的定义, 这等于说 $x \in L$ 。■

例 2.5.1(续) 语言 $L = (ab \cup ba)^*$ 被图 2-20 中有六个状态的确定型有穷自动机接受。它的标准自动机如图 2-21 所示, 有四个状态。当然, 这是接受这个语言的最小的确定型有穷自动机。◇

顺便提一句, 定理 2.5.2 直接地蕴涵正则语言的下述特征。有时把它叫做 Myhill-Nerode 定理。

推论 语言 L 是正则的当且仅当 \approx_L 有有穷个等价类。

证: 如果 L 是正则的, 则 $L = L(M)$, 其中 M 是一台确定型有穷自动机。 M 的状态不少于 \approx_L 的等价类。因此, \approx_L 有有穷个等价类。

反之, 如果 \approx_L 有有穷个等价类, 则关于 L 的标准自动机 (见定理 2.5.2 的证明) 接受 L 。■

例 2.5.2 刚才证明的推论用另一种有趣的方式说明, 一个语言 L 是正则的意味什么。此外, 它提供了除泵定理之外的另一个证明一个语言不是正则的有用方法。

例如, 这里给出 $L = \{a^n b^n : n \geq 0\}$ 不是正则语言的另一个证明。如果 $i \neq j$, 由于把 b^i 附加在 a^i 后面得到 L 中的一个字符串, 而把 b^j 附加在 a^i 后面得到的字符串不在 L 中, 故在 \approx_L 下 a^i 与 a^j 不等价。因此, \approx_L 有无穷多个等价类 $[e], [a], [aa], [aaa], \dots$ 。根据推论, L 不是正则的。◇

对于任一正则语言 L , 定理 2.5.2 的证明中构造的自动机是接受 L 的状态最少的确定型有穷自动机, 这是具有明显实际重要性的目标。不幸的是, 这台自动机是用 \approx_L 的等价类定义的, 而对于任意给定的正则语言 L , 特别是用一台确定型有穷自动机 M 给定 L 时, 不清楚如何识别这些等价类。下面将要开发一个算法, 从任一接受语言 L 的确定型有穷自动机 M 出发, 构造这台最小的自动机。

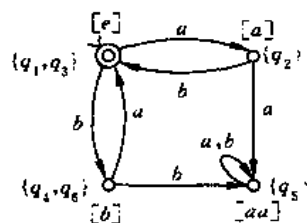


图 2-21

设 $M = (K, \Sigma, \delta, s, F)$ 是一台确定型有穷自动机。定义关系 $A_M \subseteq K \times \Sigma^*$ 如下: $(q, w) \in A_M$ 当且仅当对于某个 $f \in F, (q, w) \vdash_M^*(f, e)$ 。也就是说, $(q, w) \in A_M$ 的意思是 w 把 M 从 q 带到一个终结状态。如果对于所有的 $z \in \Sigma^*, (q, z) \in A_M$ 当且仅当 $(p, z) \in A_M$, 则称状态 q 与 p 是等价的, 记作 $q \equiv p$ 。于是, 如果两个状态是等价的, 则 \sim_M 的两个对应的等价类是 \approx_L 的同一个等价类的子集。换句话说, \equiv 的等价类恰好是为了获得 $L(M)$ 的标准自动机应该合并在一块的 M 的状态的集合^①。

我们将开发一个计算 \equiv 的等价类的算法。算法把 \equiv 作为下面定义的等价关系序列 $\equiv_0, \equiv_1, \equiv_2, \dots$ 的极限来计算。对于两个状态 $q, p \in K$, 如果对所有的长度不超过 n 的 $z, (q, z) \in A_M$ 当且仅当 $(p, z) \in A_M$, 则记作 $q \equiv_n p$ 。换句话说, \equiv_n 是比 \equiv 粗糙一些的等价关系, 只要求在长度不超过 n 的字符串的驱使下状态 q 与 p 的行为就是否接受而言是相同的。

显然, 在 $\equiv_0, \equiv_1, \equiv_2, \dots$ 中每一个等价关系是前一个的细化。又, $q \equiv_0 p$ 当且仅当 q 与 p 都是终结状态, 或者都是非终结状态。也就是说, \equiv_0 恰好有两个等价类: F 和 $K - F$ (假设它们都非空)。剩下还要给出 \equiv_{n+1} 与 \equiv_n 的关系。两者的关系如下。

引理 2.5.1 对于任意两个状态 $q, p \in K$ 和任意一个整数 $n \geq 1, q \equiv_n p$ 当且仅当 (a) $q \equiv_{n-1} p$ 且 (b) 对所有的 $a \in \Sigma, \delta(q, a) \equiv_{n-1} \delta(p, a)$ 。

证: 根据定义, $q \equiv_n p$ 当且仅当 $q \equiv_{n-1} p$, 而且任意长度恰好为 n 的字符串 $w = av$ 把 q 与 p 都带到接受或都带到不接受。而第二个条件等于说对任意的 $a \in \Sigma, \delta(q, a) \equiv_{n-1} \delta(p, a)$ 。 ■

引理 2.5.1 使我们想到可以用下述算法计算 \equiv , 进而给出 L 的标准自动机。

initially \equiv_0 的等价类是 F 与 $K - F$;

repeat for $n := 1, 2, \dots$

由 \equiv_{n-1} 的等价类计算 \equiv_n 的等价类

until \equiv_n 与 \equiv_{n-1} 相同。

每一次迭代可以应用引理 2.5.1 来实现: 对 M 的每一对状态检查引理中的条件是否成立, 如果成立则把这两个状态放入 \equiv_n 的同一个等价类。但是, 怎么知道这是一个算法, 迭代最终要终止呢? 回答很简单: 对于每一个不满足终止条件的迭代, $\equiv_n \neq \equiv_{n-1}$, \equiv_n 是 \equiv_{n-1} 的真细化, 至少比 \equiv_{n-1} 多一个等价类。由于等价类的数目不可能变得比 M 的状态数还多, 故算法至多迭代 $|K| - 1$ 次后终止。

当算法终止时, 比如说在第 n 次迭代时终止, 计算出 $\equiv_n = \equiv_{n-1}$, 则由引理有 $\equiv_n = \equiv_{n+1} = \equiv_{n+2} = \equiv_{n+3} = \dots$ 。因而, 计算出的关系正是 \equiv 。下一节给出这个重要算法的更仔细的复杂度分析, 证明它是多项式的。

例 2.5.3 把这个状态最小化算法应用于图 2-20 中的确定型有穷自动机 (当然, 根据前一个例子, 我们知道应该得到 $L(M)$ 的四个状态的标准自动机)。在各次迭代, 得到对应的 \equiv_i 的等价类:

① 可以把关系 \equiv 叫作 \sim_M 除以 \approx_L 的商关系。如果 \sim 是 \approx 的细化, 则 \sim 除以 \approx 的商关系是 \sim 的所有等价类上的一个等价关系, 记作 \sim / \approx 。如果 $x \approx y$, 则 \sim 的两个等价类 $[x]$ 与 $[y]$ 被 \sim / \approx 关联。容易看到这确实是一个等价关系。回到前面美国地理的例子, “同州” 关系除以 “同县” 关系的商关系关联任意两个位于同一个州的县 (每一个县中至少有一个城市)。

开始时, \equiv_0 的等价类是 $\{q_1, q_3\}$ 和 $\{q_2, q_4, q_5, q_6\}$ 。

第一次迭代后, \equiv_1 的等价类是 $\{q_1, q_3\}$, $\{q_2\}$, $\{q_4, q_6\}$ 和 $\{q_5\}$ 。由于 $\delta(q_2, b) \neq_0 \delta(q_4, b)$ 与 $\delta(q_5, b), \delta(q_4, a) \neq_0 \delta(q_5, a)$, 故发生分裂。

第二次迭代后, 没有进一步的等价类分裂。于是, 算法终止, 最少状态自动机如图 2-22 所示。正如所期望的, 它与图 2-21 所示的标准自动机同构。◇

例 2.5.4 回忆 $\{a_1, \dots, a_n\}^*$ 中 n 个符号不全出现的所有字符串组成的语言 L 及其对应的 2^n 个状态的确定型有穷自动机(见例 2.2.5)。现在可以证明这 2^n 个状态都是必需的。理由是 \approx_L 有 2^n 个等价类。即, 对于 Σ 的每一个子集 A , 令 L_A 是 A 的符号都出现、而 $\Sigma - A$ 中的符号都不出现的所有字符串组成的集合。例如, $L_\emptyset = \{e\}$ 。那么, 不难看到 2^n 个集合 L_A 正好是 \approx_L 的所有等价类。因为如果对于 Σ 的两个不同的子集 A 和 $B, x \in L_A, y \in L_B$, 则对于任意的 $z \in L_{\Sigma-B}, xz \in L$, 而 $yz \notin L$ (这里假设 B 不包含在 A 中, 否则交换 A 与 B 的角色)。

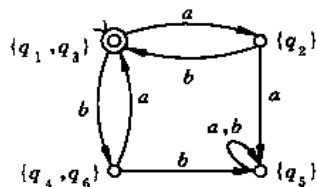


图 2-22

回想起有一台接受这个语言的有 $n+1$ 个状态的非确定型有穷自动机。虽然在原则上确定型有穷自动机与非确定型有穷自动机的能力相同, 但是在最坏的时候确定型要付出指数个状态的代价。这是考虑问题的两种不同方式。事实上, 这里提前讨论了在第 6 章和第 7 章中讨论的重要课题计算复杂性; 当把状态数考虑在内时, 在有穷自动机范围内非确定性的能力比确定性的指数地强。

习 题

2.5.1 (a) 对下述语言 L , 给出在 \approx_L 下的等价类:

- (i) $(aab \cup ab)^*$
- (ii) $\{x \in \{a, b\}^*; x \text{ 含有子串 } aababa\}$
- (iii) $\{xx^R; x \in \{a, b\}^*\}$
- (iv) $\{xx; x \in \{a, b\}^*\}$
- (v) $\{a, b\}^n a \{a, b\}^n$, 其中 $n > 0$ 是一固定整数
- (vi) 平衡的括号语言(见习题 2.4.6)

(b) 对(a)中答案为有穷的语言, 给出接受它的状态最少的确定型有穷自动机。

2.5.2 如果字符串 $x \in \Sigma^*$ 不能写成 $x = uvvw$ 的形式, 其中 $v \neq e$, 则称 x 是无平方的。例如, *lewis* 和 *christos* 是无平方的, 而 *hurry* 和 *papadimitriou* 不是无平方的。证明: 如果 $|\Sigma| > 1$, 则 Σ^* 中所有无平方的字符串组成的集合不是正则的。

2.5.3 对习题 2.1.2 和 2.2.9 中的每一台有穷自动机(确定型的或非确定型的), 求等价的状态最少的确定型有穷自动机。

2.5.4 **双向有穷自动机**类似一台确定型有穷自动机, 但是它的读头可以在输入带上前进或后退。如果它试图往后退出带的左端, 则停止运行并且不接受输入串。形式地, 双向有穷自动机 M 是一个五元组 $(K, \Sigma, \delta, s, F)$, 其中 K, Σ, s 和 F 与确定型有穷自动机中定义的一样, 而 δ 是从 $K \times \Sigma$ 到 $K \times \{\leftarrow, \rightarrow\}$ 的函数, 其中 \leftarrow 和 \rightarrow

表示读头运动的方向。格局是 $K \times \Sigma^* \times \Sigma^*$ 的一个成员, 格局 (p, u, v) 表示机器处于状态 p , 读头位于 v 的第一个符号, 而 u 是输入串直到读头左边的部分。如果 $v = \epsilon$, 则格局 (p, u, ϵ) 表示 M 已完成它对 u 的操作, 结束在状态 p 。

记 $(p_1, u_1, v_1) \vdash_M (p_2, u_2, v_2)$ 当且仅当对于某个 $\sigma \in \Sigma, v_1 = \sigma v, \delta(p_1, \sigma) = (p_2, \epsilon)$, 并且下述两条之一成立:

1. $\epsilon \Rightarrow, u_2 = u_1 \sigma, v_2 = v,$
2. $\epsilon \Leftarrow, u_1 = u_2 \sigma', v_2 = \sigma' v_1, \text{ 其中 } \sigma' \in \Sigma^*.$

照例, \vdash_M^* 是 \vdash_M 的自反传递闭包。 M 接受 w 当且仅当 $(s, \epsilon, w) \vdash_M^* (f, w, \epsilon)$, 其中 $f \in F$ 。在本题中, 你用 Myhill-Nerode 定理 (定理 2.5.2 及其推论) 证明被双向有穷自动机接受的语言可以用单向有穷自动机接受。因而, 允许读头向左移动不增加有穷自动机的能力。

设 M 是一台双向有穷自动机。

- (a) 设 $q \in K, w \in \Sigma^*$ 。证明: 至多有一个 $p \in K$ 使得 $(q, \epsilon, w) \vdash_M^* (p, w, \epsilon)$ 。
- (b) 设 t 是一个不在 K 中的固定元素。对任意的 $w \in \Sigma^*$, 定义函数 $\chi_w: K \rightarrow K \cup \{t\}$ 如下:

$$\chi_w(q) = \begin{cases} p & \text{如果 } (q, \epsilon, w) \vdash_M^* (p, w, \epsilon) \\ t & \text{否则} \end{cases}$$

根据 (a), χ_w 是良定义的。对任意的 $w \in \Sigma^*$, 又定义 $\theta_w: K \times \Sigma \rightarrow K \cup \{t\}$ 如下:

$$\theta_w(q, a) = \begin{cases} p & \text{如果 } (q, w, a) \vdash_M^+ (p, w, a) \text{ 且不存在 } r \neq p \\ & \text{使 } (q, w, a) \vdash_M^+ (r, w, a) \vdash_M^+ (p, w, a) \\ t & \text{如果不存在 } p \in K \text{ 使 } (q, w, a) \vdash_M^+ (p, w, a) \end{cases}$$

这里 \vdash_M^+ 表示 \vdash_M 的传递 (而非自反的) 闭包, 即在格局上的“一步或多步产生”关系。

假设 $w, v \in \Sigma^*, \chi_w = \chi_v$ 和 $\theta_w = \theta_v$ 。证明: 对任意的 $u \in \Sigma^*, M$ 接受 wu 当且仅当 M 接受 vu 。

- (c) 证明: 设 $L(M)$ 是被双向有穷自动机 M 接受的语言, 则 M 被一台普通的 (单向) 确定型有穷自动机接受。(提示: 利用上面的 (b) 证明 $\approx_{L(M)}$ 有有穷个等价类。)
- (d) 推出存在一个指数算法, 当给定一台双向有穷自动机 M 时, 可构造一台等价的确定型有穷自动机。(提示: 作为 $|K|$ 和 $|\Sigma|$ 的函数, 可能有多少个不同的函数 χ_w 和 θ_w ?)
- (e) 设计一台 $O(n)$ 个状态的双向有穷自动机接受语言 $L_n = \{a, b\}^* a \{a, b\}^*$ (见习题 2.5.1(a)(v))。与习题 2.5.1(b)(v) 比较, 推出在上面的 (d) 中指数增长率是必需的。
- (f) 能把本题中的论证和构造推广到非确定型双向有穷自动机吗?

2.6 关于有穷自动机的算法

本章的许多结果涉及正则语言的不同表示方式:正则语言是有穷自动机(确定型的或非确定型的)接受的语言,是正则表达式产生的语言。在前一节我们看到,给定任一确定型有穷自动机,能够找到状态尽可能少的等价的确定型有穷自动机。所有这些结果都是构造性的,根据它们的证明可以直接给出算法,给定一种类型的表示,产生出另一种类型的表示。在本节,将使这些算法更加清楚明确并且粗略地分析它们的复杂度。

先考虑把非确定型有穷自动机转换成确定型有穷自动机的算法,给出它的复杂度。算法的输入是一台非确定型有穷自动机 $M = (K, \Sigma, \Delta, s, F)$, 要计算算法的复杂度,把它表成 K, Σ 和 Δ 的基数的函数。基本需要是计算确定型有穷自动机的转移函数,即对每一个 $Q \subseteq K$ 和 $a \in \Sigma$, 计算

$$\delta'(Q, a) = \bigcup \{E(p) : p \in K \text{ 且对某个 } q \in Q, (q, a, p) \in \Delta\}$$

用在那个证明中给出的闭包算法预先一劳永逸地计算出所有的 $E(p)$ 更加方便一些。容易看到对于所有的 $E(p)$ 这可以在 $O(|K|^3)$ 时间内完成。一旦有了 $E(p)$, 要计算 $\delta'(Q, a)$ 可以先找到使得 $(q, a, p) \in \Delta$ 的所有状态 p , 然后求所有 $E(p)$ 的并, 一共有 $O(|\Delta| |K|^2)$ 个像把一个元素加入 K 的一个子集之类的基本操作。于是, 整个算法的复杂度为 $O(2^{|K|} |K|^3 |\Sigma| |\Delta| |K|^2)$ 。复杂度估计为输入规模的指数函数(正如因子 $2^{|K|}$ 所表明的)是不奇怪的, 我们已经看到在最坏的情况下算法的输出(等价的确定型有穷自动机)可能是指数的。

把正则表达式 R 转换成一台非确定型有穷自动机(定理 2.3.2)的效率比较高。很容易对 R 的长度做归纳, 证明得到的自动机的状态不多于 $2|R|$ 个, 从而转移不多于 $4|R|^2$ 个。这是多项式的。

把一台给定的有穷自动机 $M = (K, \Sigma, \Delta, s, F)$ (确定型的或非确定型的) 转换成一个产生相同语言的正则表达式(定理 2.3.2) 包括计算 $|K|^3$ 个正则表达式 $R(i, j, k)$ 。但是, 这些表达式的长度在最坏的情况是指数的。在对下标 k 的每一次迭代中, 每一个正则表达式的长度大约增长到三倍, 因为它是上一次迭代的三个正则表达式的连接。最后得到的正则表达式可能长到 $3^{|K|}$, 这是自动机的规模的指数函数。

上一节中的最小化算法(给定任意一台确定型有穷自动机 $M = (K, \Sigma, \delta, s, F)$, 计算状态最少的等价的确定型有穷自动机)是多项式的。算法至多进行 $|K| - 1$ 次迭代, 每一次迭代对每一对状态检查它们是否被 \equiv 关联。这样的检查只要做 $O(|\Sigma|)$ 次检查两个状态是否被上一次计算的等价关系 \equiv_{i-1} 关联的基本操作。于是, 这个最小化算法的整个复杂度为 $O(|\Sigma| |K|^3)$, 是多项式的。

给定两个语言生成器或两个语言接受器, 一个感兴趣的问题自然是问它们是否等价, 即它们是否生成或接受相同的语言。如果两个接受器是两台确定型有穷自动机, 则状态最小化算法也能解决这个等价问题; 两台确定型有穷自动机等价当且仅当它们的标准自动机是相同的。这是因为标准自动机仅与所接受的语言有关, 因而对于检查等价性这是一个有用的标准化。检查两台确定型有穷自动机是否相同是一个不难的同构问题, 可以从初始

状态开始,借助转移上的标记给出状态之间的对应。

可是,现在知道的解决两台非确定型有穷自动机或两个正则表达式是否等价的唯一方法是把它们转换成两台确定型有穷自动机,然后检查它们的等价性。这种算法当然是指数的。

现将有关正则语言及其表示的算法问题的讨论总结如下:

定理 2.6.1 (a) 有一个指数算法,给定一台非确定型有穷自动机,构造一台等价的确定型有穷自动机。

(b) 有一个多项式算法,给定一个正则语言,构造一台等价的非确定型有穷自动机。

(c) 有一个指数算法,给定一台非确定型有穷自动机,构造一个等价的正则表达式。

(d) 有一个多项式算法,给定一台确定型有穷自动机,构造一台等价的状态最少的确定型有穷自动机。

(e) 有一个多项式算法,给定两台确定型有穷自动机,判断它们是否等价。

(f) 有一个指数算法,给定两台非确定型有穷自动机,判断它们是否等价。关于两个正则表达式的等价性与此类似。

在上面的(a)和(c)中,指数复杂度是必需的,因为正如例 2.2.5 和习题 2.3.8 所表明的,算法的输出(在(a)中是一台确定型有穷自动机,在(c)中是一个等价的正则表达式)可能必须是指数的。但是,在定理 2.6.1 中有三个尚未解决的重要问题:

(1) 存在判断两台任意给定的非确定型有穷自动机是否等价的多项式算法,还是在(f)中指数复杂度是固有的?

(2) 能在多项式时间内找到与给定的非确定型有穷自动机等价且状态更少的非确定型有穷自动机吗?在指数时间内肯定可以做到;对每一台状态比给定自动机少的非确定型有穷自动机,用(f)中的指数算法检查等价性。

(3) 更使人感兴趣的是,假设给定一台非确定型有穷自动机,希望找到等价的状态最少的确定型有穷自动机。这可以结合使用上面(a)和(d)中的算法来实现。但是,尽管最后的自动机可能是小的,步数仍可能是所给的非确定型有穷自动机规模的指数,因为中间用子集构造法生成的非最优的确定型有穷自动机的状态可能比必需的状态指数地多。是否有直接地生成最少状态的等价的确定型有穷自动机的算法,其时间是输入与最后输出的多项式界限的?

正如在关于 NP 完全性的第 7 章将要看到的那样,我们强烈地认为这三个问题的答案都是否定的,虽然现在没有人知道怎么证明。

2.6.1 有穷自动机作为算法

关于确定型有穷自动机与算法的联系有一件事情可以说是非常基本的:确定型有穷自动机 M 是判断一个给定的字符串是否在 $L(M)$ 中的有效算法。例如,图 2-23 中的确定型有穷自动机可以重新表述成下述算法:

q_1 : Let $\sigma :=$ 下一个符号;
if $\sigma =$ 文件结束 then 拒绝;

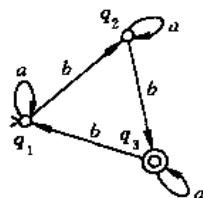


图 2-23


```

    else if  $\sigma = a$  then goto  $q_1$ ;
    else if  $\sigma = b$  then goto  $q_2$ ;
 $q_2$ : Let  $\sigma :=$  下一个符号;
    if  $\sigma =$  文件结束 then 拒绝;
    else if  $\sigma = a$  then goto  $q_2$ ;
    else if  $\sigma = b$  then goto  $q_3$ ;
 $q_3$ : Let  $\sigma :=$  下一个符号;
    if  $\sigma =$  文件结束 then 接受;
    else if  $\sigma = a$  then goto  $q_3$ ;
    else if  $\sigma = b$  then goto  $q_1$ ;

```

要把一台确定型有穷自动机 $M = (K, \Sigma, \delta, s, F)$ 转变成一个算法, 对于 K 中的每一个状态有 $|\Sigma| + 2$ 条指令, 其中第一条指令获取下一个输入符号, 其余的每一条负责执行对于一个输入符号的动作或者在到达输入串末端时的动作。我们把到达输入串末端叫做“文件结束”。上述讨论可以形式地表述如下:

定理 2.6.2 如果 L 是一个正则语言, 则有一个算法, 任给 $w \in \Sigma^*$, 它在 $O(|w|)$ 时间内检查 w 是否在 L 中。

但是, 关于非确定型有穷自动机怎么样呢? 它们确实有力地简化了表示方法, 并且是把正则表达式转换成自动机的最自然、最直接的方法(回忆定理 2.3.1 证明中的构造法)。但是, 它们不明显地对应于算法。当然, 总可以利用定理 2.2.1 证明中子集构造法把给定的非确定型有穷自动机转换成等价的确定型有穷自动机, 但是这个构造法本身(以及生成的自动机)可能是指数的。于是, 产生下述问题: 能够用运行确定型有穷自动机的方式直接地“运行”非确定型有穷自动机吗? 下面将要指出, 这是可以的, 其代价是要损失一定的速度。

回想隐藏在子集构造法后面的想法, 在读了部分输入以后, 非确定型自动机可能处于一个状态集合中的任一状态。子集构造法计算所有这些可能的状态集合。但是, 当我们只对自动机在一个字符串上的运行感兴趣时, 下述想法可能是比较好的: 可以计算这个输入串需要和提出的“飞行中”的状态集合。

具体地说, 设 $M = (K, \Sigma, \Delta, s, F)$ 是一台非确定型有穷自动机, 考虑下述算法:

$S_0 := E(s), n := 0$;

repeat

 let $n := n + 1, \sigma :=$ 第 n 个输入符号;

 if $\sigma \neq$ 文件结束 then

$S_n := \bigcup \{E(q) : \text{对某个 } p \in S_{n-1}, (p, \sigma, q) \in \Delta\}$

until $\sigma =$ 文件结束;

if $S_{n-1} \cap F \neq \emptyset$ then 接受 else 拒绝

这里和子集构造法中一样, $E(q)$ 表示 $\{p : (q, e) \vdash_M^* (p, e)\}$ 。

例 2.6.1 让我们用这个算法对输入串 $aaaba$ “运行”图 2-24 中的非确定型有穷自动

机。集合 S_n 的值如下所示：

$$S_0 = \{q_0, q_1\}$$

$$S_1 = \{q_0, q_1, q_2\}$$

$$S_2 = \{q_0, q_1, q_2\}$$

$$S_3 = \{q_0, q_1, q_2\}$$

$$S_4 = \{q_2, q_3, q_4\}$$

$$S_5 = \{q_2, q_3, q_4\}$$

因为 S_5 包含终结状态，所以机器停机时接受输入 $aaaba$ 。 \diamond

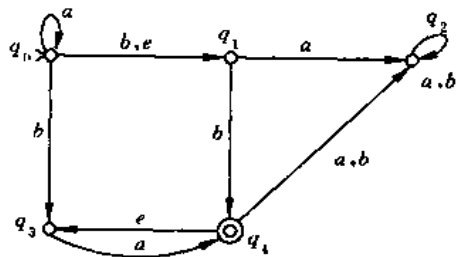


图 2-24

容易通过对输入串长度做归纳，证明该算法正确地给出 M 在读输入串的前 n 个符号后可能到达的所有状态的集合 S_n 。换句话说，算法模拟等价的确定型有穷自动机，而不构造这台确定型有穷自动机。正如下述定理所述，算法的时间需求是适中的（关于时间界限的证明及其改进见习题 2.6.2）。

定理 2.6.3 如果 $M = (K, \Sigma, \Delta, s, F)$ 是一台非确定型有穷自动机，则有一个算法，任给 $w \in \Sigma^*$ ，它在 $O(|K|^2|w|)$ 时间内检查 w 是否在 $L(M)$ 中。

下面假设给定一个字母表 Σ 上的正则表达式 α ，要判断任给的字符串 $w \in \Sigma^*$ 是否在 α 生成的语言 $L(\alpha)$ 中。由于 α 可以容易地转化成一台等价的非确定型有穷自动机 M ，其规模与 α 的相当（回忆定理 2.3.1 中的构造法），上述算法也可以用来回答这样的问题①。

一个同样可以用基于有穷自动机的算法解决的计算问题是字符串匹配，这是计算机系统及其应用中的一个最核心的问题。固定字符串 $x \in \Sigma^*$ ，称作模式。要设计一个有效的算法，任给字符串 w ， w 称作正文（通常 w 比 x 长得多），判断 x 是否作为 w 的子串出现。注意我们对以 x 和 w 作输入并且回答 x 是否出现在 w 中的算法不感兴趣，而是要求算法专门在所有可能的字符串中发现模式 x 。把这个算法叫作 A_x 。自然地，我们的策略是设计一台接受语言 $L_x = \{w \in \Sigma^* : x \text{ 是 } w \text{ 的子串}\}$ 的有穷自动机。

事实上，设计一台接受 L_x 的非确定型有穷自动机是很简单的。例如，设 $\Sigma = \{a, b\}$ ， $x = ababaab$ ，对应的非确定型有穷自动机如图 2-25(a) 所示。为了把它变成一个有用的算法，必须采用定理 2.6.3 的直接模拟，其运行时间为 $O(|\Sigma|^2|w|)$ ，这是多项式的，但对于运用还是太慢了；或者把它转换成一台等价的确定型有穷自动机，我们知道这个构造法可能是指数的。幸运的是，对于在字符串匹配中产生的非确定型有穷自动机，子集构造法总是有效的，所得到的确定型有穷自动机 M_x 的状态数恰好与原来的非确定型有穷自动机的相同（见图 2-25(b)）。它明显地是最小的等价自动机。因此，这台自动机 M_x 是对任一字符串 $w \in \Sigma^*$ 在 $O(|w|)$ 时间内检查是否 $w \in L_x$ 的算法。

但是，这个算法有一个缺点使它不适合字符串匹配的许多实际应用。在实际应用中，基本字母表 Σ 有数十个甚至常有数百个符号。作为算法提出的确定型有穷自动机必须对每一个输入符号执行一个很长的 if 语句序列，对于字母表中的每一个符号有一个 if 语句

① 对于熟悉 Unix 操作系统的读者，该算法以命令 `grep` 和 `egrep` 为基础。

(回忆本小节中的第一个算法)。换句话说,在算法的运行时间 $\mathcal{O}(|w|)$ 中的 \mathcal{O} 记号“隐藏着”一个潜在的大常数:运行时间实际上是 $\mathcal{O}(|\Sigma||w|)$ 。对此有一个巧妙的补救方法,见习题 2.6.3。

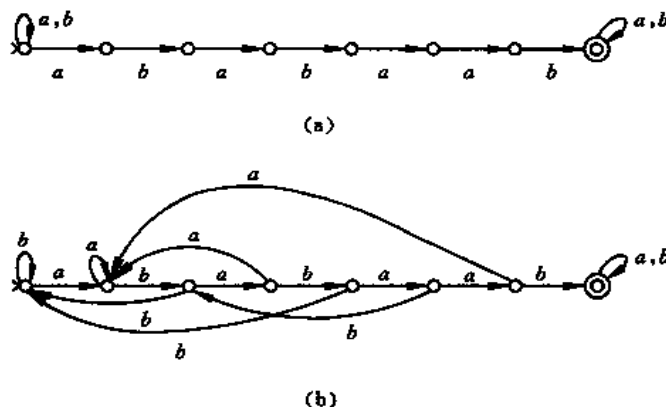


图 2-25

习 题

- 2.6.1 证明这两个正则表达式不表示相同的语言: $aa(a \cup b)^* \cup (bb)^*a^*$ 和 $(ab \cup ba \cup a)^*$ 。
- 对它们运用一个通用算法。
 - 找一个字符串,它能用一个正则表达式生成而不能用另一个生成。
- 2.6.2 (a) 如果给例 2.6.1 中的非确定型有穷自动机提供输入 $bbabbabba$, 产生的 S_i 序列是什么?
- (b) 证明这个算法在长度为 n 的输入上运行 m 个状态的非确定型有穷自动机的时间为 $\mathcal{O}(m^2n)$ 。
- (c) 假设给出的非确定型有穷自动机的每一个状态至多有 p 个离开它的转移。证明有一个 $\mathcal{O}(mnp)$ 算法。
- 2.6.3 设 Σ 是一个字母表, $x = a_1 \cdots a_n \in \Sigma^*$, 考虑非确定型有穷自动机 $M_x = (K, \Sigma, \Delta, s, F)$, 其中 $K = \{q_0, q_1, \dots, q_n\}$, $\Delta = \{(q_{i-1}, a_i, q_i) : i = 0, 1, \dots, n-1\} \cup \{(q_i, a, q_i) : a \in \Sigma, i = 0, n\}$ (回忆图 2-25)。
- 证明, $L(M_x) = \{w \in \Sigma^* : x \text{ 是 } w \text{ 的子串}\}$ 。
 - 证明, 与 M_x 等价的状态最少的确定型有穷自动机 M'_x 也有 $n+1$ 个状态。构造 M'_x 需要的最坏情况的时间是多少? 把它表成 n 的函数。
 - 证明, 存在与 M_x 等价的非确定型有穷自动机 M''_x , 它也有 $n+1$ 个状态 $\{q_0, q_1, \dots, q_n\}$ 且具有下述重要性质: 除 q_0 和 q_n 外, 每一个状态恰好有两个离开它的转移, 其中一个是 ϵ 转移。(提示: 在图 2-25 的确定型有穷自动机中, 用适当的 ϵ 转移代替每一个向后的转移, 将其一般化。)
 - 证明, M''_x 解决了在正文的最后一段中讨论的隐藏的常数 $|\Sigma|$ 问题。

- (e) 给出由 x 构造 M''_x 的算法, 这个算法的复杂度是什么? 把它表成 n 的函数。
- (f) 给上面(e)中的问题设计一个 $O(n)$ 算法。(提示: 假设 M''_x 的 ϵ 转移是 $(q_i, \epsilon, q_{f(i)}), i = 1, \dots, n-1$ 。说明如何根据 $f(j)$ 的值 ($j < i$) 计算 $f(i)$ 。对这个计算的巧妙的“分期偿还的”分析给出 $O(n)$ 上界。)
- (g) 设 $\Sigma = \{a, b\}$, $x = aabbaab$ 。构造 M_x, M'_x 和 M''_x 。对输入 $aababbbaabbaabbaabb$ 运行这三台自动机。

参 考 文 献

下面两篇是关于有穷自动机的早期文章:

- G. H. Mealy. A method for synthesizing sequential Circuits. Bell System Technical Journal, 34(5), pp. 1045-1079, 1955.
- E. F. Moore. Gedanken experiments on sequential machines. In: Automata Studies, C. E. Shannon and J. McCarthy (ed.), pp. 129-153, Princeton: Princeton University Press, 1956.

下面是关于有穷自动机的经典文章(包括定理 2.2.1),

- M. O. Rabin and D. Scott. Finite automata and their decision problems. IBM Journal of Research and Development, 3, pp. 114-125, 1959.

陈述有穷自动机接受正则语言的定理 2.3.2 属于:

- S. C. Kleene. Representation of events by nerve nets. In: Automata Studies, C. E. Shannon and J. McCarthy (ed.), pp. 3-42, Princeton: Princeton University Press, 1956.

定理 2.3.2 的证明取自:

- R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. IEEE Transactions on Electronic Computers, EC-9(1), pp. 39-47, 1960.

定理 2.4.1(“泵定理”)取自:

- V. Bar-Hillel, M. Perls and E. Shamir. On formal properties of simple phrase structure grammars. Zeitschrift für Phonetik, Sprachwissenschaft, und Kommunikationsforschung, 14, pp. 143-172, 1961.

有穷状态转换器(习题 2.1.4)是在下面文章里引入的:

- S. Ginsburg. Examples of abstract machines. IEEE Transactions on Electronic Computers, EC-11(2), pp. 132-135, 1962.

在下面文章中考察了双带有穷状态自动机(习题 2.1.5 和 2.4.7):

- M. Bird. The equivalence problem for deterministic two-tape automata. Journal of Computer and Systems Science, 7, pp. 218-236, 1973.

Myhill-Nerode 定理(定理 2.5.2)取自:

- A. Nerode. Linear automaton transformations. Proc. AMS, 9, pp. 541-544, 1958.

有穷自动机最小化算法取自上面引用的 Moore 的文章。下文给出一个更有效的算法:

- J. E. Hopcroft, An $n \log n$ algorithm for minimizing the states in a finite automaton. In: The Theory of Machines and Computations, Z. Kohavi (ed.), New York: Academic Press, 1971.

对非确定型有穷自动机的模拟(定理 2.6.3)基于:

- K. Thompson. Regular expression search algorithms. Communications ACM, 11(6), pp. 419-422, 1968.

习题 2.6.3 中的快速模式匹配算法取自：

- D. E. Knuth, J. H. Morris, Jr. V. R. Pratt. Fast pattern matching in strings. SIAM J. on Computing, 6(2), pp. 323—350, 1976.

下文给出单向和双向有穷自动机的等价性(习题 2.5.4)：

- J. C. Shepherdson. The reduction of two-way automata to one-way automata. IBM Journal of Research and Development, 3, pp. 198—200, 1959.

第3章 上下文无关语言

3.1 上下文无关文法

想象你自己是一台语言处理器。当听到一句英语时,你能够认出合法的句子。“The cat is in the hat”至少在语法上是正确的(不管它说的事情是否是真的),而“Hat the the in is cat”是莫名其妙的。然而,当读到这样的句子时你能够立即说出它们是否是按照普遍接受的规则造出来的。在这方面你正在担任**语言识别器**——一种接受有效字符串的设备——的角色。上一章的有穷自动机就是一种形式化的语言识别器。

你还能够造出合法的英语句子。我们不关心你为什么要造句子和怎样造句子,而实际情况是在需要的时候你会说或写一些句子,而且它们在语法上一般都是正确的(甚至当它们是谎言的时候)。在这方面你正在担任**语言生成器**的角色。在本节将要研究某些类型的形式语言生成器。当给出某种“开始”信号后,这样的设备开始构造一个字符串。它的操作从开始就不是完全确定的,不过要受到一组规则的限制。这个过程最终停止并且输出一个完成的字符串。该设备定义的语言是它能够生成的所有字符串的集合。

制造英语的识别器和生成器一点也不容易,几十年来设计关于自然语言的大子集的这种设备确实已经成为具有挑战性的研究前沿。尽管如此,语言生成器的思想在试图讨论人类语言时起到某种解释的作用。但是,对我们来说,“人工的”形式语言的生成器的理论更加重要。正则语言和下面将要介绍的“上下文无关的”语言都是人工的形式语言。这个理论很好地补充了对识别语言的自动机的研究,并且也具有规范和分析计算机语言的实用价值。

可以把正则表达式看作语言生成器。例如,考虑正则表达式 $a(a^* \cup b^*)b$ 。怎样按照这个表达式生成字符串可用言语描述如下:

首先输出一个 a 。然后输出若干个 a 或者输出若干个 b 。最后输出一个 b 。

与这个语言生成器相关联的语言——即,可以用刚才描述的过程产生的所有字符串组成的集合——当然恰好是正则表达式 $a(a^* \cup b^*)b$ 用早先描述的方式定义的正则语言。

本章将研究一种更复杂的语言生成器,称作上下文无关文法,它以对属于语言的所有字符串的结构更加完全的了解为基础。仍以 $a(a^* \cup b^*)b$ 生成的语言为例,注意到该语言中的任一字符串由一个前导 a 、跟着中间部分(由 $(a^* \cup b^*)$ 生成),再跟着一个尾巴 b 。如果令 S 是一个新符号,可解释为“语言中的一个字符串”,而 M 是一个代表“中间部分”的符号,则可以把它写成

$$S \rightarrow aMb$$

这里 \rightarrow 读作“可以是”。我们称这样的表达式是一个规则。中间部分 M 是什么呢?回答是:

一个若干 a 的字符串, 或一个若干 b 的字符串。用添加下述两个规则表达这件事

$$M \rightarrow A \quad \text{和} \quad M \rightarrow B$$

这里 A 和 B 是两个新符号, 分别代表若干 a 的字符串和若干 b 的字符串。接下去, 若干 a 的字符串是什么? 它可以是空串,

$$A \rightarrow e$$

或者由一个前导 a 跟着一个若干 a 的字符串组成,

$$A \rightarrow aA$$

类似地, 对于 B ,

$$B \rightarrow e \quad \text{和} \quad B \rightarrow bB$$

于是, 正则表达式 $a(a^* \cup b^*)b$ 表示的语言可以换一种方式用下述语言生成器定义:

从由一个符号 S 组成的字符串开始。在当前的字符串中找一个出现在上面的一条规则的 \rightarrow 左边的符号。把它替换成这条规则的 \rightarrow 右边的字符串。重复这个过程直到不能找到这样的符号。

例如, 要生成 $aaab$ 。像规定的那样, 从 S 开始。然后按照第一条规则 $S \rightarrow aMb$ 把 S 替换成 aMb 。对 aMb 运用规则 $M \rightarrow A$, 得到 aAb 。接下去两次运用规则 $A \rightarrow aA$, 得到 $aaaAb$ 。最后, 运用规则 $A \rightarrow e$ 。在所得到的字符串 $aaab$ 中, 找不到在某条规则的 \rightarrow 左边出现的符号。于是, 这个语言生成器的运算结束, 并且像许诺的那样, 生成 $aaab$ 。

上下文无关文法是像上面那样用一组规则进行运算的语言生成器。在这里让我们来解释为什么把这样一个语言生成器叫作上下文无关的。考虑字符串 $aaAb$, 它是生成 $aaab$ 的一个中间阶段。自然地, 把围绕符号 A 的字符串 aa 和 b 叫作 A 在这个字符串中的上下文。规则 $A \rightarrow aA$ 说可以用字符串 aA 替换 A , 而不管围绕 A 的字符串是什么, 换句话说, 与 A 的上下文无关。在第 4 章将要考察更一般的文法, 在那些文法中替换可能以有合适的上下文为条件。

在上下文无关文法中, 有些符号出现在规则中 \rightarrow 的左边 (如上例中的 S, M, A 和 B), 有些符号不出现在 \rightarrow 的左边 (如 a 和 b)。后一种符号叫作终结符, 这是因为生成一个仅由这种符号组成的字符串标志生成过程的终结。下述定义形式地叙述这些思想。

定义 3.1.1 上下文无关文法 G 是一个四元组 (V, Σ, R, S) , 其中

V 是一个字母表,

Σ 是终结符集合, 它是 V 的子集,

R 是规则集合, 它是 $(V - \Sigma) \times V^*$ 的有穷子集,

$S \in V - \Sigma$ 是起始符。

$V - \Sigma$ 的成员叫做非终结符。对任意的 $A \in V - \Sigma$ 和 $u \in V^*$, 当 $(A, u) \in R$ 时记作 $A \rightarrow_G u$ 。对任意的字符串 $u, v \in V^*$, 记 $u \Rightarrow_G v$ 当且仅当存在字符串 $x, y \in V^*$ 和 $A \in V - \Sigma$ 使得 $u = xAy, v = xv'y$ 和 $A \rightarrow_G v'$ 。关系 \Rightarrow_G 是 \rightarrow_G 的自反传递闭包。最后, G 生成的语言 $L(G)$ 为 $\{w \in \Sigma^*; S \Rightarrow_G w\}$, 也说 G 生成 $L(G)$ 中的每一个字符串。如果 $L = L(G)$, 其中 G 是一个上下文无关文法, 则称 L 是一个上下文无关语言。

当涉及的文法不言自明时, 用 $A \rightarrow w$ 和 $u \Rightarrow v$ 分别代替 $A \rightarrow_G w$ 和 $u \Rightarrow_G v$ 。

形如

$$w_0 \Rightarrow_G w_1 \Rightarrow_G \cdots \Rightarrow_G w_n$$

的序列称作 w_n 在 G 中从 w_0 开始的推导, 这里 w_0, \dots, w_n 可以是 V^* 中的任何字符串, 推导的长度 n 是任何自然数, 包括 0 在内。也说推导有 n 步。

例 3.1.1 考虑上下文无关文法 $G = (V, \Sigma, R, S)$, 其中 $V = \{S, a, b\}$, $\Sigma = \{a, b\}$, 而 R 包括规则 $S \rightarrow aSb$ 和 $S \rightarrow \epsilon$ 。可以有推导

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

在这里, 头两步使用规则 $S \rightarrow aSb$, 最后一步使用规则 $S \rightarrow \epsilon$ 。事实上, 不难看出 $L(G) = \{a^n b^n, n \geq 0\}$ 。因此, 某些上下文无关语言不是正则的。◇

但是, 我们很快将看到所有的正则语言是上下文无关的。

例 3.1.2 设 $G = (W, \Sigma, R, S)$, 其中

$$\begin{aligned} W &= \{S, A, N, V, P\} \cup \Sigma \\ \Sigma &= \{\text{Jim}, \text{big}, \text{green}, \text{cheese}, \text{ate}\} \\ R &= \{P \rightarrow N, \\ &\quad P \rightarrow AP, \\ &\quad S \rightarrow PVP, \\ &\quad A \rightarrow \text{big}, \\ &\quad A \rightarrow \text{green}, \\ &\quad N \rightarrow \text{cheese}, \\ &\quad N \rightarrow \text{Jim}, \\ &\quad V \rightarrow \text{ate}\} \end{aligned}$$

在这里, 把 G 设计成一个表示部分英语的文法, S 代表句子, A 代表形容词, N 代表名词, V 代表动词, P 代表短语。下面是 $L(G)$ 中的几个字符串:

Jim ate cheese

big Jim ate green cheese

big cheese ate Jim

不幸的是, 下面也是 $L(G)$ 中的字符串

big cheese ate green green big green big cheese

green Jim ate green big Jim ◇

例 3.1.3 用任何程序设计语言写成的计算机程序必须符合某些严格的标准, 这样才能保证它在语法上是正确的和能够对它进行机械的解释。幸运的是, 和人类语言不同, 多数程序设计语言的语法是上下文无关文法。在 3.7 节中将看到当对一个程序进行语法分析, 即为了理解它的语法而分析它的时候, 是上下文无关的非常有帮助。在这里给出一个文法, 它生成许多普通程序设计语言的一个片断。这个语言由字母表 $\{(,), +, *, \text{id}\}$ 上所有表示包含 $+$ 和 $*$ 的语法正确的算术表达式的字符串组成。id 代表标识符, 即变量名^①。例如, id 和 $\text{id} * (\text{id} * \text{id} + \text{id})$ 是这样的字符串, 而 $* \text{id} +$ 和 $+ * \text{id}$ 不是。

① 顺便说一句, 在程序中寻找标识符(或语言的保留字、数值常数)是用基于正则表达式和有穷自动机的算法在更早的词法分析阶段完成的。

令 $G = (V, \Sigma, R, E)$, 这里 V, Σ 和 R 如下:

$$V = \{+, *, (,), id, T, F, E\}$$

$$\Sigma = \{+, *, (,), id\}$$

$$R = \{E \rightarrow E + T, \quad (R1)$$

$$E \rightarrow T, \quad (R2)$$

$$T \rightarrow T * F, \quad (R3)$$

$$T \rightarrow F, \quad (R4)$$

$$F \rightarrow (E), \quad (R5)$$

$$F \rightarrow id \quad (R6)$$

符号 E, T 和 F 分别代表表达式, 项和因子。

文法 G 用下述推导生成字符串 $(id * id + id) * (id + id)$:

$E \Rightarrow T$	规则 R2
$\Rightarrow T * F$	规则 R3
$\Rightarrow T * (E)$	规则 R5
$\Rightarrow T * (E + T)$	规则 R1
$\Rightarrow T * (T + T)$	规则 R2
$\Rightarrow T * (F + T)$	规则 R4
$\Rightarrow T * (id + T)$	规则 R6
$\Rightarrow T * (id + F)$	规则 R4
$\Rightarrow T * (id + id)$	规则 R6
$\Rightarrow F * (id + id)$	规则 R4
$\Rightarrow (E) * (id + id)$	规则 R5
$\Rightarrow (E + T) * (id + id)$	规则 R1
$\Rightarrow (E + F) * (id + id)$	规则 R4
$\Rightarrow (E + id) * (id + id)$	规则 R6
$\Rightarrow (T + id) * (id + id)$	规则 R2
$\Rightarrow (T * F + id) * (id + id)$	规则 R3
$\Rightarrow (F * F + id) * (id + id)$	规则 R4
$\Rightarrow (F * id + id) * (id + id)$	规则 R6
$\Rightarrow (id * id + id) * (id + id)$	规则 R6

关于生成程序设计语言更大子集的上下文无关文法, 见习题 3.1.8. \diamond

例 3.1.4 下述文法生成所有完全平衡的左右括号的字符串; 每一个左括号与后面的唯一一个右括号配对, 每一个右括号与前面的唯一一个左括号配对, 而且任意一对这样的括号之间的字符串也具有这样的性质。令 $G = (V, \Sigma, R, S)$, 其中

$$V = \{S, (,)\}$$

$$\Sigma = \{(,)\}$$

$$R = \{S \rightarrow e,$$

$$\begin{aligned} S &\rightarrow SS, \\ S &\rightarrow (S) \end{aligned}$$

下面是该文法中的两个推导：

$$\begin{aligned} S &\Rightarrow SS \Rightarrow S(S) \Rightarrow S((S)) \Rightarrow S(()) \Rightarrow (S)(()) \Rightarrow ()(()) \\ S &\Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()((S)) \Rightarrow ()(()) \end{aligned}$$

可见，在上下文无关文法中同一个字符串可以有几个推导。在下一节要讨论联系这种推导的复杂方式。

顺便说一句， $L(G)$ 是又一个非正则的上下文无关语言（习题 2.4.6 要求证明它不是正则的）。◇

例 3.1.5 很明显，存在非正则的上下文无关语言（我们已经看到两个例子）。可是，所有正则语言是上下文无关的。在本章有好几个它的证明。例如，在 3.3 节将看到上下文无关语言正好是下推自动机接受的语言。而下推自动机是有穷自动机的推广，任意一台有穷自动机都可以直接地看作一台下推自动机。因而，所有正则语言是上下文无关的。

关于另一个证明，在 3.5 节将看到上下文无关语言类在并、连接和 Kleene 星号下是封闭的（定理 3.5.1）；而且，平凡的语言 \emptyset 和 $\{a\}$ 的确都是上下文无关的（分别用不含规则和仅有一条规则 $S \rightarrow a$ 的上下文无关文法生成）。因此，上下文无关语言类一定包括所有的正则语言——平凡语言在这些运算下的闭包。

现在我们用直接构造法证明所有的正则语言是上下文无关的。考虑确定型有穷自动机 $M = (K, \Sigma, \delta, s, F)$ 接受的语言。文法 $G(M) = (V, \Sigma, R, S)$ 生成同样的语言，其中 $V = K \cup \Sigma$, $S = s$ ，而

$$R = \{q \rightarrow ap, \delta(q, a) = p\} \cup \{q \rightarrow e, q \in F\}$$

也就是说，非终结符是自动机的状态；至于规则，对于每一个从 q 到 p 关于输入 a 的转移，在 R 中有规则 $q \rightarrow ap$ 。例如，对于图 3-1 中的自动机，这个文法为：

$$S \rightarrow aS, S \rightarrow bA, A \rightarrow aB, A \rightarrow bA, B \rightarrow aS, B \rightarrow bA, B \rightarrow e$$

证明所得到的上下文无关文法正好生成这台自动机接受的语言留作习题（见习题 3.1.10 关于像上面 $G(M)$ 这种上下文无关文法的一般性处理，以及它们与有穷自动机的关系）。◇

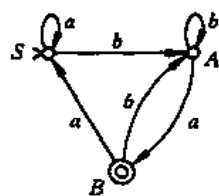


图 3-1

习 题

3.1.1 考虑文法 $G = (V, \Sigma, R, S)$ ，其中

$$\begin{aligned} V &= \{a, b, S, A\} \\ \Sigma &= \{a, b\} \\ R &= \{S \rightarrow AA, \\ &\quad A \rightarrow AAA, \\ &\quad A \rightarrow a, \\ &\quad A \rightarrow bA, \\ &\quad A \rightarrow Ab\} \end{aligned}$$

- (a) 用不超过四步的推导能够生成 $L(G)$ 的哪些字符串?
 (b) 至少给出关于字符串 $babbab$ 的四个不同的推导。
 (c) 对任意的 $m, n, p > 0$, 描述字符串 $b^m ab^n ab^p$ 在 G 中的一个推导。

3.1.2 考虑文法 $G = (V, \Sigma, R, S)$, 其中

$$\begin{aligned} V &= \{a, b, S, A\} \\ \Sigma &= \{a, b\} \\ R &= \{S \rightarrow aAa, \\ &\quad S \rightarrow bAb, \\ &\quad S \rightarrow e, \\ &\quad A \rightarrow SS\} \end{aligned}$$

给出字符串 $baabbbb$ 在 G 中的一个推导。(注意, 与我们至今看到的所有其他上下文无关语言不同, 这个语言很难用言语描述。)

3.1.3 构造生成下述语言的上下文无关文法:

- (a) $\{w cw^R; w \in \{a, b\}^*\}$
 (b) $\{ww^R; w \in \{a, b\}^*\}$
 (c) $\{w \in \{a, b\}^*; w = w^R\}$

3.1.4 考虑字母表 $\Sigma = \{a, b, (,), \cup, *, \odot\}$ 。构造一个上下文无关文法, 它生成 Σ^* 中所有是 $\{a, b\}$ 上的正则表达式的字符串。

3.1.5 考虑上下文无关文法 $G = (V, \Sigma, R, S)$, 其中

$$\begin{aligned} V &= \{a, b, S, A, B\}, \\ \Sigma &= \{a, b\} \\ R &= \{S \rightarrow aB, \\ &\quad S \rightarrow bA, \\ &\quad A \rightarrow a, \\ &\quad A \rightarrow aS, \\ &\quad A \rightarrow BAA, \\ &\quad B \rightarrow b, \\ &\quad B \rightarrow bS, \\ &\quad B \rightarrow ABB\} \end{aligned}$$

- (a) 证明: $ababba \in L(G)$ 。
 (b) 证明: $L(G)$ 是 $\{a, b\}^*$ 中所有 a 和 b 个数相同的字符串的集合。

3.1.6 设 G 是一个上下文无关文法, $k > 0$ 。令 $L_k(G) \subseteq L(G)$ 是在 G 中有 k 步或小于 k 步推导的所有字符串的集合。

- (a) $L_5(G)$ 是什么? 这里 G 是例 3.1.4 中的文法。
 (b) 证明: 对于所有的上下文无关文法 G 和所有的 $k > 0$, $L_k(G)$ 是有穷的。

3.1.7 设 $G = (V, \Sigma, R, S)$, 其中

$$\begin{aligned} V &= \{a, b, S\} \\ \Sigma &= \{a, b\} \end{aligned}$$

$$R = \{S \rightarrow aSb, \\ S \rightarrow aSa, \\ S \rightarrow bSa, \\ S \rightarrow bSb, \\ S \rightarrow e\}$$

证明: $L(G)$ 是正则的。

3.1.8 在现实的程序设计语言(如 C 或 Pascal) 中, 一个程序由若干个语句组成, 语句有下述几种类型:

- (1) 赋值语句, 形式为 $id := E$, 其中 E 是任一算术表达式(用例 3.1.3 中的文法生成)。
- (2) 条件语句, 形式为 $\text{if } E < E \text{ then 语句}$, 或 while 语句 , 形式为 $\text{while } E < E \text{ do 语句}$ 。
- (3) goto 语句。此外, 每一个语句的前面可以有一个标号。
- (4) 复合语句, 即若干个语句的前面是一个 begin, 后面跟着一个 end, 语句之间用一个“;”分开。

给出一个上下文无关文法, 它生成上面描述的简化程序设计语言中所有的语句。

3.1.9 给出生成下述语言的上下文无关文法, 从而证明这些语言是上下文无关的:

- (a) $\{a^m b^n; m \geq n\}$
- (b) $\{a^m b^n c^p d^q; m + n = p + q\}$
- (c) $\{w \in \{a, b\}^*; w \text{ 中 } b \text{ 的个数是 } a \text{ 的个数的两倍}\}$
- (d) $\{uawb; u, w \in \{a, b\}^* \text{ 且 } |u| = |w|\}$
- (e) $\{w_1 c w_2 c \cdots c w_k c w_k^R; k \geq 1, 1 \leq j \leq k, w_i \in \{a, b\}^+, i = 1, \dots, k\}$
- (f) $\{a^m b^n; m \leq 2n\}$

3.1.10 如果上下文无关文法 $G = (V, \Sigma, R, S)$ 的 $R \subseteq (V - \Sigma) \times \Sigma^* ((V - \Sigma) \cup \{e\})$, 即每一个规则的右边是一个终结符串至多再跟着一个非终结符, 则称 G 是正则的, 或右线性的。

(a) 考虑正则文法 $G = (V, \Sigma, R, S)$, 其中

$$V = \{a, b, A, B, S\} \\ \Sigma = \{a, b\} \\ R = \{S \rightarrow abA, S \rightarrow B, S \rightarrow baB, S \rightarrow e, \\ A \rightarrow bS, B \rightarrow aS, A \rightarrow b\}$$

构造一台非确定型有穷自动机 M 使得 $L(M) = L(G)$ 。跟踪导致 M 接受字符串 $abba$ 的转移, 并且与这个字符串在 G 中的推导进行比较。

- (b) 证明: 一个语言是正则的当且仅当它被一个正则文法生成。(提示: 回忆例 3.1.5.)
- (c) 称上下文无关文法 $G = (V, \Sigma, R, S)$ 是左线性的当且仅当 $R \subseteq (V - \Sigma) \times ((V - \Sigma) \cup \{e\})\Sigma^*$ 。证明: 一个语言是正则的当且仅当它被一个左线性文法生成。
- (d) 假设 $G = (V, \Sigma, R, S)$ 是一个上下文无关文法, 它的每一个规则形如 $A \rightarrow wB$, 或者形如 $A \rightarrow Bw$, 或者形如 $A \rightarrow w$, 这里 $A, B \in V - \Sigma, w \in \Sigma^*$ 。 $L(G)$ 一定是正则的吗? 证明或举出反例。

3.2 语法分析树

设 G 是一个上下文无关文法。字符串 $w \in L(G)$ 在 G 中可能有很多推导。例如, 如果 G 是生成平衡括号语言的上下文无关文法(回忆例 3.1.4), 则字符串 $()()$ 至少可以用两个不同的推导从 S 得到:

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()()$$

和

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow (S)() \Rightarrow ()()$$

可是, 这两个推导在某种意义下是“相同的”。使用的规则相同, 在中间字符串使用它们的地点也相同。唯一的区别是使用规则的次序。直观上, 这两个推导都可以用图 3-2 表示。

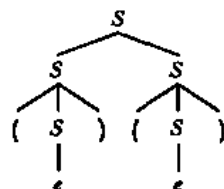


图 3-2

这样的图叫做语法分析树。图中的点叫做顶点。每一个顶点有一个标记, 标记是 V 中的一个符号。最上面的顶点叫做根, 最底层的顶点叫做树叶。树叶都标记终结符, 也可能标记空串 ϵ 。从左到右连接树叶的标记, 得到推导出来的终结字符串, 称作语法分析树的结果。

更形式地, 对于任一上下文无关文法 $G = (V, \Sigma, R, S)$, 定义它的语法分析树及其根、树叶和结果如下:

1. $\bullet a$

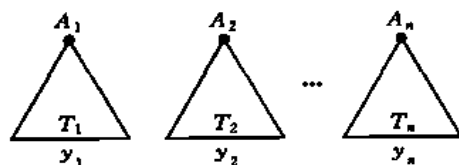
对于每一个 $a \in \Sigma$, 这是一棵语法分析树。它的唯一的一个顶点既是根也是树叶, 它的结果是 a 。

2. 如果 $A \rightarrow \epsilon$ 是 R 中的一个规则, 则

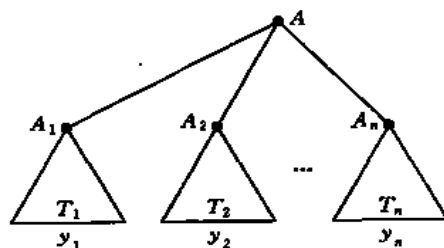


是一棵语法分析树。它的根是标记 A 的顶点, 它的唯一一片树叶是标记 ϵ 的顶点, 它的结果是 ϵ 。

3. 如果 $A \rightarrow A_1 \dots A_n$ 是 R 中的一个规则,



是 n 棵语法树, 它们的根分别标记 A_1, \dots, A_n , 结果分别为 y_1, \dots, y_n , 这里 $n \geq 1$, 则



等价类包含推导 D_1, D_2, D_3 , 还包含下面的七个:

$$D_4 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$$

$$D_5 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow ((S))(S) \Rightarrow ((S))() \Rightarrow (())()$$

$$D_6 = S \Rightarrow SS \Rightarrow (S)S \Rightarrow (S)(S) \Rightarrow (S)() \Rightarrow ((S))() \Rightarrow (())()$$

$$D_7 = S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ((S))(S) \Rightarrow (())(S) \Rightarrow (())()$$

$$D_8 = S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow ((S))(S) \Rightarrow ((S))() \Rightarrow (())()$$

$$D_9 = S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow (S)() \Rightarrow ((S))() \Rightarrow (())()$$

$$D_{10} = S \Rightarrow SS \Rightarrow S(S) \Rightarrow S() \Rightarrow (S)() \Rightarrow ((S))() \Rightarrow (())()$$

这十个推导之间的 $<$ 关系如图 3-5 所示。

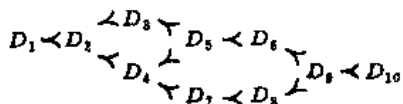


图 3-5

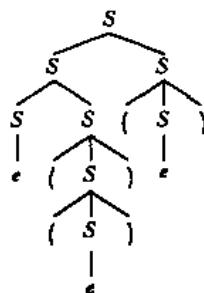


图 3-6

这十个推导全都是相似的。非形式地, 它们表示在字符串的同样位置应用同样的规则, 唯一的区别是这些应用的相对次序, 等价地, 重复使用 $<$ 或 $<$ 的逆可以把它们中的任一个关联到另一个。再没有别的推导与它们相似。

但是, $(())()$ 有另外的推导不与上面的推导相似, 因而不能用图 3-4 中的语法分析树体现, 下面是一个这样的推导:

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow S(S)S \Rightarrow S((S))S \Rightarrow S(())S \Rightarrow S(())(S) \Rightarrow S(())() \Rightarrow (())()$$

它的语法分析树如图 3-6 所示(与图 3-4 比较)。

每一个在相似性下的等价类, 即每一棵语法分析树, 有一个推导在 $<$ 下是极大的, 即没有先于它的其他推导。这个推导叫做最左推导。每一棵语法分析树中都有一个最左推导, 它可以如下得到: 从根的标记 A 开始, 按照这棵语法分析树提供的规则反复替换当前字符串中最左边的非终结符。类似地, 最右推导是不先于任何其他推导的推导, 它是从语法分析树经过总是替换当前字符串中的最右边的非终结符得到的。每一棵语法分析树恰好有一个最左推导和一个最右推导。这是因为一棵语法分析树的最左推导是唯一确定的, 在每一步只有一个要替换的非终结符, 即最左边的非终结符。最右推导也与此类似。在上面的例子中, D_1 是最左推导, D_{10} 是最右推导。

容易指出推导中的一步什么时候可以成为最左推导中的一部分: 应该替换最左边的非终结符。记 $x \xrightarrow{L} y$ 当且仅当 $x = wA\beta, y = w\alpha\beta$, 其中 $w \in \Sigma^*, \alpha, \beta \in V^*, A \in V - \Sigma$, 且 $A \rightarrow \alpha$ 是 G 的一条规则。于是, 如果 $x_1 \Rightarrow x_2 \Rightarrow \dots \Rightarrow x_n$ 是最左推导, 则事实上, $x_1 \xrightarrow{L} x_2 \xrightarrow{L} \dots \xrightarrow{L} x_n$ 。最右推导也与此类似(记号是 $x \xrightarrow{R} y$)。

综合本节对语法分析树和推导的深入观察,给出下述定理而不再予以形式的证明。

定理 3.2.1 设 $G = (V, \Sigma, R, S)$ 是一个上下文无关文法, $A \in V - \Sigma$ 及 $w \in \Sigma^*$, 则下述命题是等价的:

- (a) $A \xRightarrow{*} w$ 。
- (b) 有一棵根为 A 、结果为 w 的语法分析树。
- (c) 有最左推导 $A \xRightarrow{L} w$ 。
- (d) 有最右推导 $A \xRightarrow{R} w$ 。

歧义性

我们在例 3.2.2 中看到,在上下文无关文法生成的语言中可能有这样的字符串,它有两个不相似的推导,也就是说,它有两棵不同的语法分析树,或等价地,有两个不同的最右推导(和两个不同的最左推导)。举一个更有实际意义的例子,回忆例 3.1.3 中生成所有 id 上算术表达式的文法 G 。再考虑生成这个语言的另一个文法 G' , 它的规则为:

$$E \rightarrow E + E, \quad E \rightarrow E * E, \quad E \rightarrow (E), \quad E \rightarrow id$$

不难看出 $L(G') = L(G)$ 。可是, G 和 G' 之间有一个重要的区别。直观上, G' 由于“混淆了”因子(F)和项(T)之间的区别,有违反乘法优先于加法的危险。实际上,在 G' 中有两棵关于表达式 $id + id * id$ 的语法分析树,如图 3-7 所示。其中的一棵(图 3-7(a))对应于这个表达式的“正常”意思(先 * 后 +), 而另一棵是“错误的”。

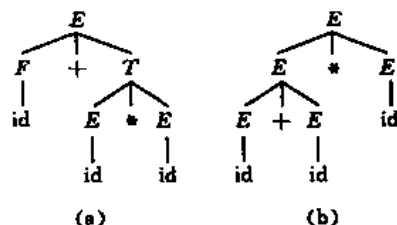


图 3-7

像 G' 这种能够生成有两棵或两棵以上不同语法分析树的字符串的文法叫做歧义的。正如将在 3.7 节看到的那样,把一棵语法分析树赋给某语言中的一个给定的字符串,即对该字符串做语法分析,是理解它的结构、理解它属于这个语言的原因,并且最终理解它的“意思”的重要的第一步。当然,在文法生成程序设计语言片断的情况下(上面的 G 和 G' 就是这样),这是特别重要的。像 G' 这样的歧义文法无助于语法分析,它们把两棵或两棵以上的语法分析树,从而把两个或两个以上的“意思”赋给语言中的某些字符串。

幸运的是,通过引入新的非终结符 T 和 F , 有办法“消去” G' 的歧义性(回想例 3.1.3 中的文法 G)。也就是说,有生成相同语言的非歧义的文法(那就是例 3.1.3 中定义的文法 G 。关于文法 G 确实是非歧义的证明,见习题 3.2.1)。类似地,例 3.1.4 中给出的生成平衡的括号字符串的文法也是歧义的(见在例 3.2.2 结尾处的讨论),能够容易地使它变成非歧义的(见习题 3.2.2)。事实上,存在具有下述性质的上下文无关语言,生成它的所有上下文无关文法一定是歧义的。这样的语言叫做固有歧义的。幸运的是,程序设计语言不会是固有歧义的。

习 题

- 3.2.1 证明:在例 3.1.3 中给出的生成所有 id 上算术表达式的上下文无关文法 G 是非歧义的。
- 3.2.2 证明:在例 3.1.4 中给出的生成所有平衡的括号字符串的上下文无关文法是歧义的。给出一个等价的非歧义的上下文无关文法。
- 3.2.3 考虑例 3.1.3 中的文法。给出字符串 $id * id + id$ 的两个推导,一个是最左推导,另

一个是最右推导。

3.2.4 画出下述字符串在指定文法中的语法分析树：

(a) 例 3.1.2 中的文法和字符串 big Jim ate green cheese。

(b) 例 3.1.3 中的文法和字符串 $id + (id + id) * id$ 与 $(id * id + id * id)$ 。

3.3 下推自动机

我们已经看到某些上下文无关语言不是正则的,所以不是每一个上下文无关语言都能够用有穷自动机识别。什么样更有力的设备能够用来识别任意的上下文无关语言?或者更具体一点,为了使有穷自动机接受任意的上下文无关语言必须给它增加什么额外的特性?

取一个具体的例子,考虑 $\{ww^R; w \in \{a,b\}^*\}$ 。它是上下文无关的,可由规则为 $S \rightarrow aSa, S \rightarrow bSb$ 及 $S \rightarrow \epsilon$ 的文法生成。似乎任何从左到右读一遍就能识别这个语言中的所有字符串的设备必须“记住”输入串的前一半,以便能够对照输入串的后一半(按照相反的顺序)核对它。有穷自动机不能实现这个功能是不奇怪的。但是,如果机器在读输入时能够把它存起来,每次在存储的字符串上添加一个符号(见图 3-8),则它可以非确定地猜想什么时候已经到达这个输入的中心,并且在这以后核对它存储的符号,每次核对一个。这个存储装置不必是通用的。“栈”或“下推存储器”就可以了,这种存储器只允许读写访问顶符号。

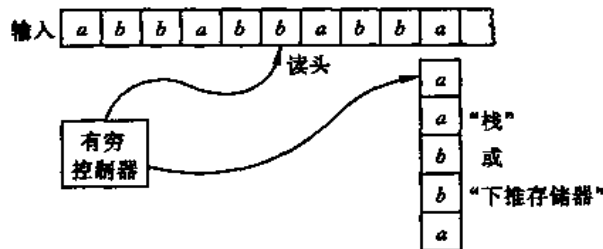


图 3-8

再举一个例子,平衡的括号字符串集合(例 3.1.4)也是非正则的。可是,计算机程序员都熟悉一个识别该语言的简单算法:从零开始记数,遇到每一个左括号加 1,遇到每一个右括号减 1。如果在某一步计数变成负的,或者结束时不等于零,则字符串不是平衡的,应该拒绝它;否则应该接受它。然而,可以把计数器看作栈的特殊情况,只能把一种符号写入它。

还可以从另一个角度考虑这个问题,像 $A \rightarrow aB$ 这样的规则容易用有穷自动机模拟:如果在状态 A 读到 a ,则转移到状态 B 。可是,如果一个规则的右边不是一个终结符跟着一个非终结符,比如规则 $A \rightarrow aBb$,怎么办?机器肯定还是必须读 a 并且从状态 A 转移到状态 B ,但是 b 怎么办?为了记住在这个规则中存在 b ,还应该做什么?在这里栈是有用的:把 b 推到栈顶上,这就解决了记住它的问题并且当它再次上升到栈顶时再对它起作用——可能是与输入中的一个 b 进行核对。

用栈作为辅助存储器的自动机的思想可以形式化如下:

定义 3.3.1 下推自动机是一个六元组 $M = (K, \Sigma, \Gamma, \Delta, s, F)$, 其中

K 是有穷的状态集合,

Σ 是一个字母表(所有输入符号),

Γ 是一个字母表(所有栈符号),

$s \in K$ 是初始状态,

$F \subseteq K$ 是终结状态集合,

Δ 是转移关系,它是 $(K \times (\Sigma \cup \{e\}) \times \Gamma^*) \times (K \times \Gamma^*)$ 的有穷子集。

直观上,如果 $((p, a, \beta), (q, \gamma)) \in \Delta$, 则当 M 处于状态 p 、栈顶为 β 时,它可以从输入带读 a (如果 $a = e$, 则不访问输入带)、在栈顶用 γ 代替 β 、然后进入状态 q 。这样的有序对 $((p, a, \beta), (q, \gamma))$ 叫做 M 的转移。由于在每一步 M 有若干个转移可以利用,机器的运行是非确定的。(后面将要修改这个定义给出一类受限制的下推自动机——确定型下推自动机。)

推入一个符号是把这个符号加到栈顶上,托出一个符号是把这个符号从栈顶移去。例如,转移 $((p, u, e), (q, a))$ 推入 a ,而 $((p, u, a), (q, e))$ 托出 a 。

与有穷自动机一样,在计算中已读过的输入部分对机器以后的运行不再起作用。因此,下推自动机的格局定义为 $K \times \Sigma^* \times \Gamma^*$ 的成员;第一个分量是机器的状态,第二个分量是尚未读过的输入部分,第三个分量是下推存储器的内容,从顶向下读。例如,如果格局是 (q, w, abc) ,则 a 在栈顶和 c 在栈底。设 (p, x, α) 和 (q, y, ζ) 是两个格局,如果存在转移 $((p, a, \beta), (q, \gamma)) \in \Delta$ 使得 $x = ay, \alpha = \beta\eta$ 和 $\zeta = \gamma\eta$, 其中 $\eta \in \Gamma^*$, 则称 (p, x, α) 一步生成 (q, y, ζ) , 记作 $(p, x, \alpha) \vdash_M (q, y, \zeta)$ 。把 \vdash_M 的自反传递闭包记作 \vdash_M^* 。称 M 接受字符串 $w \in \Sigma^*$ 当且仅当对于某个 $p \in F, (s, w, e) \vdash_M^* (p, e, e)$ 。换句话说, M 接受字符串 w 当且仅当存在格局序列 $C_0, C_1, \dots, C_n (n \geq 0)$ 使得 $C_0 \vdash_M C_1 \vdash_M \dots \vdash_M C_n$, 其中 $C_0 = (s, w, e), C_n = (p, e, e)$ 且 $p \in F$ 。如果格局序列 C_0, C_1, \dots, C_n 满足条件 $C_i \vdash_M C_{i+1}, i = 0, \dots, n-1$, 则称作 M 的计算,又称它的长度为 n 或有 n 步。 M 接受的语言是 M 接受的所有字符串的集合,记作 $L(M)$ 。

当不会混淆时,用 \vdash 和 \vdash^* 分别代替 \vdash_M 和 \vdash_M^* 。

例 3.3.1 设计一台下推自动机 M 接受语言 $L = \{w c w^R, w \in \{a, b\}^*\}$ 。例如, $ab a b c b a b a \in L$, 而 $a b c a b \notin L, c b c \notin L$ 。令 $M = (K, \Sigma, \Gamma, \Delta, s, F)$, 其中 $K = \{s, f\}, \Sigma = \{a, b, c\}, \Gamma = \{a, b\}, F = \{f\}$, 而 Δ 包含下述五个转移:

- (1) $((s, a, e), (s, a))$
- (2) $((s, b, e), (s, b))$
- (3) $((s, c, e), (f, e))$
- (4) $((f, a, a), (f, e))$
- (5) $((f, b, b), (f, e))$

这台机器按下述方式运行。当它读输入的前一半时,它保持在初始状态 s 并且用转移 1 和 2 把符号从输入串转移到下推存储器中。注意,不管下推存储器当前的内容是什么都可以应用这两个转移,因为在下推存储器顶部对应的“字符串”是空串。当机器在输入串中看到一个 c 时,它从状态 s 转换到状态 f 且不对栈进行任何操作。此后,只有转移 4 和 5 起作用。当栈顶的符号与下一个输入符号相同时,它们可以把栈顶的符号删去。如果下一个输入符号与栈顶的符号不相同,则不可能继续运行。如果机器用这种方式到达格局 (f, e, e) ——终结状态、输入的结尾和空栈,则输入必定形如 $w c w^R$, 机器接受。另一方面,如果机器发现输入与栈符号不匹配或者在栈被排空前已耗尽了输入,则它不接受。

为了说明 M 的运行,表 3-1 描述了关于输入串 $abbcbba$ 的格局序列。

表 3-1

状态	未读的输入	栈	使用的转移
s	$abbcbba$	e	—
s	$bcbba$	a	1
s	$cbba$	ba	2
s	bba	bba	2
f	ba	bba	3
f	a	ba	5
f	e	a	5
f		e	4

例 3.3.2 构造一台下推自动机接受 $L = \{ww^R; w \in \{a,b\}^*\}$ 。也就是说,这台机器接受的字符串和前一个例子的机器接受的字符串一样,但是删去了标示字符串中心的符号 c 。因此,这台机器必须以非确定的方式“猜想”它什么时候已经到达输入串的中点并且从状态 s 改变成状态 f 。于是, $M = (K, \Sigma, \Gamma, \Delta, s, F)$, 其中 $K = \{s, f\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b\}$, $F = \{f\}$, 而 Δ 是下述五个转移的集合:

- (1) $((s, a, e), (s, a))$
- (2) $((s, b, e), (s, b))$
- (3) $((s, e, e), (f, e))$
- (4) $((f, a, a), (f, e))$
- (5) $((f, b, b), (f, e))$

这台机器和前一个例子中的机器除转移 3 之外完全一样。只要它处于状态 s 就能够非确定地选择把下一个输入符号推入栈,或者转换到状态 f 而不消耗任何输入。因此,即使从形式为 ww^R 的字符串开始, M 也有一些计算不把 M 引导到接受格局 (f, e, e) 。但是,存在把 M 引导到这个接受格局的计算当且仅当输入串是这个形式。 ◇

例 3.3.3 这台下推自动机接受语言 $\{w \in \{a,b\}^*; w \text{ 中 } a \text{ 与 } b \text{ 的个数相同}\}$ 。 M 在栈中保存一串 a 或者保存一串 b 。栈中是一串 a 表示至此读过的 a 多于 b , 栈中是一串 b 表示至此读过的 b 多于 a 。不管是哪一种情况, M 都在栈底存放着一个特殊的符号 c 作为记号。令 $M = (K, \Sigma, \Gamma, \Delta, s, F)$, 其中 $K = \{s, q, f\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b, c\}$, $F = \{f\}$, 而 Δ 列表如下:

- (1) $((s, e, e), (q, c))$
- (2) $((q, a, c), (q, ac))$
- (3) $((q, a, a), (q, aa))$
- (4) $((q, a, b), (q, e))$
- (5) $((q, b, c), (q, bc))$
- (6) $((q, b, b), (q, bb))$
- (7) $((q, b, a), (q, e))$

(8) $((q, e, c), (f, e))$

转移 1 把 M 置于状态 q , 同时把 c 放入栈底。在状态 q , 当 M 读到一个 a 时, 若栈中只有栈底记号则把 a 推入栈, 开始在栈中堆放 a , 栈底记号保留不动(转移 2); 若栈中是一串 a 则把这个 a 推入栈(转移 3); 若栈中是一串 b 则托出一个 b (转移 4)。当从输入中读到一个 b 时, 动作类似, 把一个 b 推入有一串 b 的栈或仅有一个栈底记号的栈, 或者从有一串 a 的栈中托出一个 a (转移 5、6 和 7)。最后, 当 c 在栈顶(因而是栈中的唯一符号) 时, 机器可以把它移去并且转到终结状态(转移 8)。如果这时所有输入都已被读过, 则到达格局 (f, e, e) , 接受输入串。

表 3-2 说明了 M 的运行。

表 3-2

状态	未读的输入	栈	转移	注 释
s	$abbbabaa$	e	—	初始格局
q	$abbbabaa$	c	1	栈底记号
q	$bbbabaa$	ac	2	开始一串 a
q	$bbabaa$	c	7	移去一个 a
q	$babaa$	bc	5	开始一串 b
q	$abaa$	bbc	6	
q	baa	bc	4	
q	aa	bbc	6	
q	a	bc	4	
q	e	c	4	
f	e	e	8	接 受

例 3.3.4 每一台有穷自动机都可以看作一台不对栈做任何操作的下推自动机。精确地说, 设 $M = (K, \Sigma, \Delta, s, F)$ 是一台非确定型有穷自动机, 而 $M' = (K, \Sigma, \emptyset, \Delta', s, F)$ 是一台下推自动机, 其中 $\Delta' = \{((p, u, e), (q, e)), (p, u, q) \in \Delta\}$ 。换句话说, M' 总是把一个空串推入或托出它的栈, 并且除此之外都是模拟 M 的转移。显然, M 和 M' 接受相同的语言。

习 题

3.3.1 考虑下推自动机 $M = (K, \Sigma, \Gamma, \Delta, s, F)$, 其中

$$K = \{s, f\}$$

$$F = \{f\}$$

$$\Sigma = \{a, b\}$$

$$\Gamma = \{a\}$$

$$\Delta = \{((s, a, e), (s, a)), ((s, b, e), (s, a)), ((s, a, e), (f, e)),$$

$$((f, a, a), (f, e)), ((f, b, a), (f, e))\}$$

(a) 给出 M 关于输入 aba 的所有格局序列及使用的转移。

(b) 证明: $aba, aa, abb \in L(M)$, 而 $baa, bab, baaaa \notin L(M)$ 。

(c) 用言语描述 $L(M)$ 。

3.3.2 构造接受下述语言的下推自动机:

(a) 文法 $G = (V, \Sigma, R, S)$ 生成的语言, 其中

$$V = \{S, (,), [,]\}$$

$$\Sigma = \{ (,), [,] \}$$

$$R = \{ S \rightarrow e, \\ S \rightarrow SS, \\ S \rightarrow [S], \\ S \rightarrow (S) \}$$

(b) $\{a^m b^n; m \leq n \leq 2m\}$

(c) $\{w \in \{a, b\}^*; w = w^R\}$

(d) $\{w \in \{a, b\}^*; w \text{ 中的 } b \text{ 是 } a \text{ 的两倍}\}$

3.3.3 设 $M = (K, \Sigma, \Gamma, \Delta, s, F)$ 是一台下推自动机。 M 以终结状态方式接受的语言定义如下:

$$L_f(M) = \{w \in \Sigma^*; \text{对某个 } f \in F \text{ 和 } a \in \Gamma^*,$$

$$(s, w, e) \vdash_M^* (f, e, a)\}$$

(a) 证明: 存在下推自动机 M' 使得 $L(M') = L_f(M)$ 。

(b) 证明: 存在下推自动机 M'' 使得 $L_f(M'') = L(M)$ 。

3.3.4 设 $M = (K, \Sigma, \Gamma, \Delta, s, F)$ 是一台下推自动机。 M 以空栈方式接受的语言定义如下:

$$L_e(M) = \{w \in \Sigma^*; \text{对某个 } q \in K, (s, w, e) \vdash_M^* (q, e, e)\}$$

(a) 证明: 存在下推自动机 M' 使得 $L_e(M') = L(M)$ 。

(b) 证明: 存在下推自动机 M'' 使得 $L(M'') = L_e(M)$ 。

(c) 举出反例说明不一定有 $L_e(M) = L(M) \cup \{e\}$ 。

3.4 下推自动机与上下文无关文法

在这一节证明下推自动机正好是接受任意上下文无关语言所需要的自动机。这一事实在数学上和实践上都具有重要意义, 在数学上, 它把对同一语言类的两种不同的形式看法紧密地结合在一起; 在实践上, 它奠定了研究程序设计语言之类“实际的”上下文无关语言的语法分析的基础(在 3.7 节里有进一步的阐述)。

定理 3.4.1 下推自动机接受的语言正好是上下文无关语言类。

证: 这个证明分两部分。

引理 3.4.1 每一个上下文无关语言都被一台下推自动机接受。

证: 设 $G = (V, \Sigma, R, S)$ 是一个上下文无关文法, 要构造一台下推自动机 M 使得 $L(M) = L(G)$ 。我们构造的 M 只有两个状态 p 和 q , 并且在第一步之后永远保持在状态 q 。 M 还用终结符与非终结符集合 V 作为它的栈字母表。令 $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, 其中 Δ 包含下述转移:

(1) $((p, e, e), (q, S))$

(2) 对于 R 中的每一条规则 $A \rightarrow x, ((q, e, A), (q, x))$

(3) 对于每一个 $a \in \Sigma, ((q, a, a), (q, e))$

下推自动机 M 首先把 G 的起始符 S 推入栈(开始时栈是空的)并且进入状态 q (转移 1)。在接下去的每一步有两种可能:如果栈顶是一个非终结符 A ,则把 A 替换成 R 中某条规则 $A \rightarrow x$ 右边的 x (类型 2 的转移);如果栈顶是一个终结符且与下一个输入符号相同,则把这个符号托出栈(类型 3 的转移)。 M 的栈在接受计算中模仿输入串的最左推导,在栈中断断续续地实现推导中的步骤并且在这样的两步之间从栈顶删除终结符且把它们与输入串中的符号匹配。把终结符托出栈又有使最左边的非终结符暴露出来的作用,使得这个过程能够继续。

例 3.4.1 考虑文法 $G = (V, \Sigma, R, S)$, 其中 $V = \{S, a, b, c\}, \Sigma = \{a, b, c\}, R = \{S \rightarrow aSa, S \rightarrow bSb, S \rightarrow c\}$ 。它生成语言 $\{wcw^R; w \in \{a, b\}^*\}$ 。按照上面的构造方法,对应的下推自动机 $M = (\{p, q\}, \Sigma, V, \Delta, p, \{q\})$, 其中

$$\Delta = \{((p, e, e), (q, S)), \quad (T1)$$

$$((q, e, S), (q, aSa)), \quad (T2)$$

$$((q, e, S), (q, bSb)), \quad (T3)$$

$$((q, e, S), (q, c)), \quad (T4)$$

$$((q, a, a), (q, e)), \quad (T5)$$

$$((q, b, b), (q, e)), \quad (T6)$$

$$((q, c, c), (q, e))\} \quad (T7)$$

M 经过表 3-3 中的动作序列接受字符串 $abbcbba$ 。

表 3-3

状态	未读的输入	栈	使用的转移
p	$abbcbba$	e	—
q	$abbcbba$	S	$T1$
q	$abbcbba$	aSa	$T2$
q	$bcbba$	Sa	$T5$
q	$bcbba$	$bSba$	$T3$
q	$cbba$	Sba	$T6$
q	$cbba$	$bSbba$	$T3$
q	$cbba$	$Sbba$	$T6$
q	$cbba$	$cbba$	$T4$
q	bba	bba	$T7$
q	ba	ba	$T6$
q	a	a	$T6$
q	e	e	$T5$

把它与例 3.3.1 中的下推自动机对这个字符串的操作进行比较。◇

继续证明引理。为了证明 $L(M) = L(G)$, 我们证明下述断言。

断言 设 $w \in \Sigma^*$ 和 $a \in (V - \Sigma)V^* \cup \{e\}$, 则 $S \xrightarrow{L} wa$ 当且仅当 $(q, w, S) \vdash_M^* (q, e, a)$ 。

这个断言足以证明引理 3.4.1, 因为由它可以得到(取 $\alpha = e$); $S \xrightarrow{*} w$ 当且仅当 $(q, w, S) \vdash_M^* (q, e, e)$, 换句话说, $w \in L(G)$ 当且仅当 $w \in L(M)$ 。

(仅当) 假设 $S \xrightarrow{*} wa$, 其中 $w \in \Sigma^*, \alpha \in (V - \Sigma)V^* \cup \{e\}$ 。对 w 从 S 开始的最左推导的长度做归纳证明 $(q, w, S) \vdash_M^* (q, e, \alpha)$ 。

基本步骤 如果推导的长度为 0, 则 $w = e$ 且 $\alpha = S$, 因而 $(q, w, S) \vdash_M^* (q, e, \alpha)$ 。

归纳假设 假设 $S \xrightarrow{*} wa$ 的推导长度不超过 $n, n \geq 0$, 则 $(q, w, S) \vdash_M^* (q, e, \alpha)$ 。

归纳步骤 设

$$S = u_0 \xrightarrow{*} u_1 \xrightarrow{*} \dots \xrightarrow{*} u_n \xrightarrow{*} u_{n+1} = wa \quad (1)$$

是从 S 到 wa 的最左推导。设 A 是 u_n 的最左边的非终结符。于是, $u_n = xA\beta, u_{n+1} = x\gamma\beta$, 其中 $x \in \Sigma^*, \beta, \gamma \in V^*$ 且 $A \rightarrow \gamma$ 是 R 中的一条规则。因为有从 S 到 $u_n = xA\beta$ 长度为 n 的推导, 根据归纳假设, 有

$$(q, x, S) \vdash_M^* (q, e, A\beta) \quad (2)$$

因为 $A \rightarrow \gamma$ 是 R 中的一条规则, 用类型 2 的转移得到

$$(q, e, A\beta) \vdash_M (q, e, \gamma\beta) \quad (3)$$

注意到 u_{n+1} 等于 wa , 而它又等于 $x\gamma\beta$ 。因此, 存在字符串 $y \in \Sigma^*$ 使得 $w = xy$ 和 $ya = \gamma\beta$ 。于是, 可以把上面的(2)和(3)合写成

$$(q, w, S) \vdash_M^* (q, y, \gamma\beta) \quad (4)$$

又, 因为 $ya = \gamma\beta$, 用 $|y|$ 个类型 3 的转移, 得

$$(q, y, \gamma\beta) \vdash_M^* (q, e, \alpha) \quad (5)$$

(4)与(5)结合在一起完成归纳步骤。

(当) 假设 $(q, w, S) \vdash_M^* (q, e, \alpha)$, 其中 $w \in \Sigma^*, \alpha \in (V - \Sigma)V^* \cup \{e\}$, 要证明 $S \xrightarrow{*} wa$ 。再一次使用归纳法证明, 但是这次是对 M 的计算中使用类型 2 的转移的次数作归纳。

基础步骤 因为在任何计算中的第一步是使用类型 2 的转移, 故如果 $(q, w, S) \vdash_M^* (q, e, \alpha)$ 不使用任何类型 2 的转移, 则 $w = e$ 且 $\alpha = S$ 。结论成立。

归纳假设 如果 $(q, w, S) \vdash_M^* (q, e, \alpha)$ 使用不超过 n 次类型 2 的转移, $n \geq 0$, 则 $S \xrightarrow{*} wa$ 。

归纳步骤 假设 $(q, w, S) \vdash_M^* (q, e, \alpha)$ 使用 $n+1$ 次类型 2 的转移。考虑最后一次使用这样的转移

$$(q, w, S) \vdash_M^* (q, y, A\beta) \vdash_M (q, y, \gamma\beta) \vdash_M^* (q, e, \alpha)$$

这里 $w = xy, x, y \in \Sigma^*, A \rightarrow \gamma$ 是文法的一条规则。根据归纳假设, 有 $S \xrightarrow{*} xA\beta$ 。从而, $S \xrightarrow{*} x\gamma\beta$ 。而 $(q, y, \gamma\beta) \vdash_M^* (q, e, \alpha)$ 只使用类型 3 的转移, 得到 $ya = \gamma\beta$ 。从而, $S \xrightarrow{*} xya = wa$ 。这就完成了引理 3.4.1 的证明, 也就完成了定理 3.4.1 证明的一半。■

下面证明定理 3.4.1 的另一半。

引理 3.4.2 如果一个语言被一台下推自动机接受, 则它是上下文无关语言。

证: 对要考虑的下推自动机加以限制是有帮助的。如果下推自动机的所有转移 $((q, a,$

$\beta), (p, \gamma))$, 只要 q 不是初始状态就有 $\beta \in \Gamma$ 且 $|\gamma| \leq 2$, 则称这台下推自动机是简单的。换句话说, 机器总是考虑它的栈顶符号(而不管它下面的符号), 并且把它替换成 e , 或者一个栈符号, 或者两个栈符号。容易看出, 任何有意义的下推自动机不能只有这种转移, 因为当空栈时它不能做任何操作(例如, 因为开始时栈是空的, 所以它不能开始计算)。因此, 不限制从初始状态出发的转移。

如果一个语言被一台不受限制的下推自动机接受, 则它被一台简单的下推自动机接受。为了看到这一点, 设 $M = (K, \Sigma, \Gamma, \Delta, s, F)$ 是任意一台下推自动机, 要构造一台简单的下推自动机 $M' = (K', \Sigma, \Gamma \cup \{Z\}, \Delta', s', \{f'\})$ 也接受 $L(M)$, 其中 s' 和 f' 是不在 K 中的新状态, Z 是一个不在 Γ 中的新的栈符号, 叫做栈底符。首先把转移 $((s', e, e), (s, Z))$ 加入 Δ' , 它在计算开始时把栈底符放在栈底, 在整个计算过程中栈底符一直呆在那里。除把 Z 放置在栈底的这个转移之外, 没有转移把 Z 推入栈。再对每一个 $f \in F$, 把转移 $((f, e, Z), (f', e))$ 加入 Δ' 。这些转移把 Z 移出栈底后结束计算, 并且接受至此所看到的字符串。

开始时, Δ' 包括上面描述的开始和结束的转移以及 Δ 的所有转移。然后, 把 Δ' 中所有不符合简单性条件的转移替换成等价的满足简单性条件的转移。分三个阶段进行: 首先, 替换 $|\beta| \geq 2$ 的转移。然后, 去掉 $|\gamma| \geq 2$ 的转移, 而不引入任何 $|\beta| \geq 2$ 的转移。最后, 去掉 $\beta = e$ 的转移, 而不引入 $|\beta| \geq 2$ 或 $|\gamma| > 2$ 的转移。

考虑任意一个转移 $((q, a, \beta), (p, \gamma)) \in \Delta'$, 这里 $\beta = B_1 \cdots B_n$ 且 $n > 1$ 。把它替换成若干个新转移, 这些新转移一个一个地托出 $B_1 \cdots B_n$ 中的符号而不是一步把这个字符串移走。具体地说, 把这个转移替换成下述转移:

$$\begin{aligned} &((q, e, B_1), (q_{B_1}, e)) \\ &((q_{B_1}, e, B_2), (q_{B_1 B_2}, e)) \\ &\vdots \\ &((q_{B_1 B_2 \cdots B_{n-2}}, e, B_{n-1}), (q_{B_1 B_2 \cdots B_{n-1}}, e)) \\ &((q_{B_1 B_2 \cdots B_{n-1}}, a, B_n), (p, \gamma)) \end{aligned}$$

这里 $q_{B_1 B_2 \cdots B_i}$ 是新的状态, 它的直观意思是“符号 $B_1 B_2 \cdots B_i$ 已被托出后的状态 q ”。对 Δ' 中所有 $|\beta| > 1$ 的转移 $((q, a, \beta), (p, \gamma))$ 重复进行。显然, 所得到的下推自动机与原来的等价。

类似地, 把转移 $((q, a, \beta), (p, \gamma))$, 其中 $\gamma = C_1 \cdots C_m$ 且 $m \geq 2$, 替换成下述转移:

$$\begin{aligned} &((q, a, \beta), (r_1, C_m)) \\ &((r_1, e, e), (r_2, C_{m-1})) \\ &\vdots \\ &((r_{m-2}, e, e), (r_{m-1}, C_2)) \\ &((r_{m-1}, e, e), (p, C_1)) \end{aligned}$$

这里 r_1, \dots, r_{m-1} 是新状态。注意, 现在所有的转移 $((q, a, \beta), (p, \gamma)) \in \Delta'$ 有 $|\gamma| \leq 1$ 。这比简单性的要求还严格(实际上不能这样要求)。在下一阶段要回到 $|\gamma| \leq 2$ 。此外, 没有加入 $|\beta| > 1$ 的转移 $((q, a, \beta), (p, \gamma))$ 。

最后, 考虑任一转移 $((q, a, e), (p, \gamma))$, 其中 $q \neq s'$ 。把每一个这样的转移替换成所有

形式为 $((q, a, A), (p, \gamma A))$ 的转移, 其中 $A \in \Gamma \cup \{Z\}$ 。即, 如果机器动作不用查看栈, 则它也可以先查看栈顶符号(不管它是什么), 再接着又把这个符号放回栈中。我们知道, 在栈中至少有一个符号; 除开始和结束外, 在整个计算过程中栈是非空的。还注意到, 在这个阶段可能引入长度为 2 的 γ 。这不违反简单性条件, 但对于获得一般的下推自动机是必需的。

容易看出这样构造出一台简单的下推自动机 M' 并且 $L(M') = L(M)$ 。继续证明引理, 要给出一个上下文无关文法 G 使得 $L(G) = L(M')$ 。这将完成引理的证明, 从而也完成定理 3.4.1 的证明。

令 $G = (V, \Sigma, R, S)$ 。除一个新符号 S 和 Σ 中的所有符号外, 对每一个 $q, p \in K' - \{s'\}$ 和 $A \in \Gamma \cup \{e, Z\}$, V 包含一个新符号 $\langle q, A, p \rangle$ 。要理解非终结符 $\langle q, A, p \rangle$ 的作用, 记住 G 应生成 M' 接受的所有字符串。因此, G 的非终结符代表 M' 接受的所有字符串的不同部分。具体地说, 如果 $A \in \Gamma$, 则非终结符 $\langle q, A, p \rangle$ 表示 M' 处于状态 q 且栈顶为 A 与 M' 从栈中移去这个 A 且进入状态 p 这两点之间可能读的输入串的任何部分。如果 $A = e$, 则 $\langle q, e, p \rangle$ 表示 M' 处于状态 q 与处于状态 p 且栈不变这两点之间可能读的输入串的任何部分, 在这中间不改变、不查看栈。

R 中的规则有四种类型:

- (1) 规则 $S \rightarrow \langle s, Z, f' \rangle$, 其中 s 是原来下推自动机 M 的初始状态, f' 是新的终结状态。
- (2) 对每一个转移 $((q, a, B), (r, C))$, 其中 $q, r \in K' - \{s'\}$, $a \in \Sigma \cup \{e\}$, $B \in \Gamma \cup \{Z\}$ 及 $C \in \Gamma \cup \{Z, e\}$, 和每一个 $p \in K' - \{s'\}$, 有一条规则 $\langle q, B, p \rangle \rightarrow a \langle r, C, p \rangle$ 。
- (3) 对每一个转移 $((q, a, B), (r, C_1 C_2))$, 其中 $q, r \in K' - \{s'\}$, $a \in \Sigma \cup \{e\}$ 及 $B, C_1, C_2 \in \Gamma \cup \{Z\}$, 和每一个 $p, p' \in K' - \{s'\}$, 有一条规则 $\langle q, B, p \rangle \rightarrow a \langle r, C_1, p' \rangle \langle p', C_2, p \rangle$ 。
- (4) 对每一个 $q \in K' - \{s'\}$, 有一条规则 $\langle q, e, q \rangle \rightarrow e$ 。

注意, 根据 M' 的构造, (2)和(3)包含了 M' 中所有 $q \neq s'$ 的转移。

类型 1 的规则在本质上是说 M' 在从状态 s 到它的终结状态 f' 且对栈的净作用是托出栈底符的情况下能够读到的任何输入串是语言 $L(M)$ 中的字符串。类型 4 的规则说从一个状态到它自身且不改变栈不需要任何计算。最后, 类型 2 和 3 的规则说, 如果 $((q, a, B), (r, \gamma)) \in \Delta'$, 则从状态 q 到状态 r 且从栈顶消耗 B 的计算从读输入 a 开始, 把 B 替换成 γ , 经过状态 p , 然后继续消耗 γ 且最终停在状态 r (在这段计算中读什么输入都是可以的)。如果 $\gamma = C_1 C_2$, 这后一段计算大体上在中间托出 C_1 后经过某个状态 p' (这是类型 3 的规则)。

下述断言形式地给出这些直观的解释。

断言 对任意的 $q, p \in K' - \{s'\}$, $A \in \Gamma \cup \{Z, e\}$ 和 $x \in \Sigma^*$, $\langle q, A, p \rangle \xrightarrow{*}_G x$ 当且仅当 $\langle q, x, A \rangle \vdash_M (p, e, e)$ 。

因为 $\langle s, Z, f' \rangle \xrightarrow{*}_G x$ 当且仅当 $\langle s, x, Z \rangle \vdash_M (f', e, e)$, 即 $x \in L(G)$ 当且仅当 $x \in L(M')$, 故由断言很容易得到引理 3.4.2, 进而得到定理 3.4.1。

断言的两个方向可以用对 G 的推导或 M' 的计算的长度作归纳证明, 把它们留作习题(习题 3.4.5)。

习 题

- 3.4.1 对例 3.1.4 中的文法运用引理 3.4.1 的构造法。给出你构造出的自动机关于输入串 $(())$ 的运算。
- 3.4.2 对例 3.3.2 中的下推自动机运用引理 3.4.2 的构造法,并且令 G 是所得到的文法。集合 $\{w \in \{a,b\}^* : \langle q,a,p \rangle \xrightarrow{G} w\}$ 等于什么?与引理 3.4.2 的证明进行比较。
- 3.4.3 对例 3.3.3 中的下推自动机运用引理 3.4.2 的构造法,得到的文法将有 25 条规则,但许多条由于无用可以删去。给出字符串 $aababbbba$ 在这个文法中的推导。(为了便于叙述清楚,可改变非终结符的名字。)
- 3.4.4 证明:如果 $M = (K, \Sigma, \Gamma, \Delta, s, F)$ 是一台下推自动机,则存在另一台下推自动机 $M' = (K', \Sigma, \Gamma, \Delta', s, F)$ 使得 $L(M') = L(M)$ 且对所有的 $((q, u, \beta), (p, \gamma)) \in \Delta', |\beta| + |\gamma| \leq 1$ 。
- 3.4.5 完成引理 3.4.2 的证明。
- 3.4.6 上下文无关文法是线性的当且仅当它的每一条规则右边的字符串至多有一个非终结符。下推自动机 $(K, \Sigma, \Gamma, \Delta, s, F)$ 称作单轮的当且仅当只要它开始从栈中托出符号,以后就不再会把任何符号推入栈。(即,栈的高度一旦开始降低,就不会再上升。)证明:一个语言由一个线性上下文无关文法生成当且仅当它被一台单轮下推自动机接受。

3.5 上下文无关语言与非上下文无关语言

3.5.1 封闭性质

在上一节证明对上下文无关语言的两种看法——上下文无关文法生成的语言和下推自动机接受的语言——是等价的。这两种描述提供了两种认识上下文无关语言的不同方法,从而丰富了对上下文无关语言的理解。例如,对于像例 3.1.3 中语言那样的程序设计语言片断,文法表示更自然、更令人感兴趣;而对于语言 $\{w \in \{a,b\}^* : w \text{ 中 } a \text{ 和 } b \text{ 的数目相等}\}$ (见例 3.3.3),用下推自动机表示更容易理解。这一小节将提供另一种确认上下文无关语言的工具:证明上下文无关语言在语言运算下的封闭性质,这和正则语言的情况很像。在下一小节要证明一个更有力的泵定理,它使我们能够证明某些语言不是上下文无关的。

定理 3.5.1 上下文无关语言在并、连接和 Kleene 星号下是封闭的。

证: 设 $G_1 = (V_1, \Sigma_1, R_1, S_1)$ 和 $G_2 = (V_2, \Sigma_2, R_2, S_2)$ 是两个上下文无关文法。不失一般性,可以假设它们的非终结符集合不相交,即 $V_1 - \Sigma_1$ 与 $V_2 - \Sigma_2$ 不相交。

并 令 S 是一个新符号,令 $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R, S)$, 其中 $R = R_1 \cup R_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}$ 。则, $L(G) = L(G_1) \cup L(G_2)$ 。事实上,因为只有 $S \rightarrow S_1$ 和 $S \rightarrow S_2$ 与 S 有关,所以 $S \xrightarrow{G} w$ 当且仅当 $S_1 \xrightarrow{G} w$ 或 $S_2 \xrightarrow{G} w$ 。又因为 G_1 和 G_2 的非终结符集合不相同,所以第二个条件等价于 $w \in L(G_1) \cup L(G_2)$ 。

连接 构造方法类似, $L(G_1)L(G_2)$ 由下述文法生成:

$$G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2\}, S)$$

Kleene 星号 $L(G_1)^*$ 由下述文法生成:

$$G = (V_1 \cup \{S\}, \Sigma_1, R_1 \cup \{S \rightarrow e, S \rightarrow SS_1\}, S)$$

很快将会看到, 上下文无关语言类在交和补下不是封闭的。这并不奇怪。回忆一下, 正则语言在交下封闭的证明依赖于在补下的封闭性, 而证明在补下封闭的构造方法要求自动机是确定型的。但是, 不是所有的上下文无关语言都被确定型下推自动机接受(见定理 3.7.1 的推论)。

正则语言在交下的封闭性有一个有趣的直接证明。它不依赖在补下的封闭性, 而是直接构造一台有穷自动机。这台有穷自动机的状态集合是原有两台有穷自动机的状态集合的笛卡儿积(回想习题 2.3.3)。该构造法当然不能推广到下推自动机上, 这样构造出来的自动机有两个栈。但是, 当两台自动机中有一台是有穷自动机时, 可以使用这个方法。

定理 3.5.2 一个上下文无关语言与一个正则语言的交是一个上下文无关语言。

证: 如果 L 是一个上下文无关语言, R 是一个正则语言, 则 $L = L(M_1)$, $R = L(M_2)$, 其中 $M_1 = (K_1, \Sigma, \Gamma_1, \Delta_1, s_1, F_1)$ 是一台下推自动机, $M_2 = (K_2, \Sigma, \delta, s_2, F_2)$ 是一台确定型有穷自动机。想法是把这两台机器联合成一台下推自动机 M , 并行地实现 M_1 和 M_2 的计算, 并且仅当两个都接受时才接受。具体地说, 令 $M = (K, \Sigma, \Gamma, \Delta, s, F)$, 其中

$$K = K_1 \times K_2, M_1 \text{ 和 } M_2 \text{ 的状态集合的笛卡儿积}$$

$$\Gamma = \Gamma_1$$

$$s = (s_1, s_2)$$

$$F = F_1 \times F_2$$

转移关系 Δ 规定如下。对 M_1 的每一个转移 $((q_1, a, \beta), (p_1, \gamma)) \in \Delta_1$ 和 M_2 的每一个状态 $q_2 \in K_2$, 把转移 $((q_1, q_2), a, \beta), ((p_1, \delta(q_2, a)), \gamma)$ 加入 Δ ; 对每一个转移 $((q_1, e, \beta), (p_1, \gamma)) \in \Delta_1$ 和每一个状态 $q_2 \in K_2$, 把转移 $((q_1, q_2), e, \beta), ((p_1, q_2), \gamma)$ 加入 Δ 。也就是说, M 从状态 (q_1, q_2) 到状态 (p_1, p_2) 的方式与 M_1 从状态 q_1 到状态 p_1 的方式相同, 除此之外 M 还要保存 M_2 由于读这个输入符号引起的状态变化。

不难看出, 确有 $w \in L(M)$ 当且仅当 $w \in L(M_1) \cap L(M_2)$ 。

例 3.5.1 设 L 由所有 a 和 b 个数相同且不含子串 $abaa$ 和 $babb$ 的 a, b 字符串组成。因为它是例 3.3.3 中的下推自动机接受的语言与正则语言 $\{a, b\}^* - \{a, b\}^*(abaa \cup babb)\{a, b\}^*$ 的交, 所以 L 是上下文无关的。

3.5.2 泵定理

无穷的上下文无关语言表现出形式比正则语言更加微妙的周期性。为了搞清楚上下文无关语言的这方面情况, 我们先熟悉一下有关语法分析树的数量关系。 G 的扁出是 G 的规则右边的最大符号数, 记作 $\phi(G)$ 。语法分析树中的一条通路是一个不同顶点的序列, 每一个顶点有一条线段与前一个顶点连接, 第一个顶点是根, 而最后一个顶点是一片树叶。通路的长度是它里面的线段数。语法分析树的高度是树中最长通路的长度。

引理 3.5.1 G 的任意一棵高度为 h 的语法分析树的结果的长度不超过 $\phi(G)^h$ 。

证:对 h 做归纳证明。当 $h = 0$ 时,语法分析树只有一个顶点(这是语法分析树定义中的情况 1),因此它的结果最多有 $\phi(G)^1 = 1$ 个符号。

假设对于高度不超过 h ($h \geq 0$) 的语法分析树结论成立。考虑归纳步骤,任意一棵高度为 $h + 1$ 的语法分析树由一个根最多连接 $\phi(G)$ 棵高度不超过 h 的子语法分析树组成(这是语法分析树定义中的情况 3)。根据归纳假设,每一棵子语法分析树的结果的长度都不超过 $\phi(G)^h$ 。因此,整个语法分析树的结果的长度不超过 $\phi(G)^{h+1}$ 。这就完成了归纳证明。 ■

换一种说法, $L(G)$ 中任意一个长度大于 $\phi(G)^h$ 的字符串的语法分析树中必有一条长度大于 h 的通路。在证明下述关于上下文无关语言的泵定理中这是很关键的。

定理 3.5.3 设 $G = (V, \Sigma, R, S)$ 是一个上下文无关文法。那么, $L(G)$ 中任一长度大于 $\phi(G)^{|V-\Sigma|}$ 的字符串 w 可以写成 $w = uvxyz$ 使得 v 或 y 不是空串,并且对每一个 $n \geq 0$, $uv^nxy^n z \in L(G)$ 。

证:令 w 是这样一个字符串, T 是根标记 S ,结果为 w 的所有语法分析树中叶子数最少的语法分析树。由于 T 的结果的长度大于 $\phi(G)^{|V-\Sigma|}$,因而 T 有一条长度至少为 $|V-\Sigma| + 1$ 的通路,即有一条至少有 $|V-\Sigma| + 2$ 个顶点的通路。这些顶点中只有一个标记终结符,其余的都标记非终结符。由于这条通路上的顶点数多于非终结符数,故这条通路有两个顶点标记同一个非终结符 A 。让我们更详细地观察这条通路(见图 3-9)。

把 T 的结果分成五部分,分别叫做 u, v, x, y 和 z ,如图 3-9 所示。 x 是子树 T'' 的结果,子树 T'' 的根是标记 A 的两个顶点中位置较低的顶点; v 是子树 T' 的结果中到 x 开始为止的部分, T' 的根是标记 A 的位置较高的顶点; u 是 T 的结果中到 v 开始为止的部分; y 是 T' 的结果删去 v 和 x 之后的剩余部分; z 是 T 的结果删去前面四个字符串之后的部分。

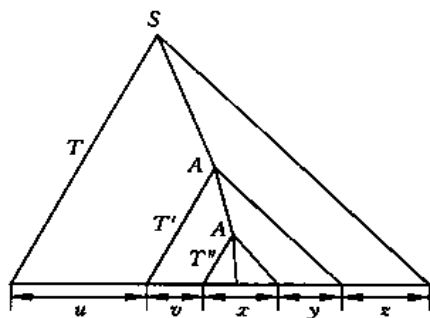


图 3-9

显然,重复任意次(包括 0 次) T' 删去 T'' 之后的部分能够产生 G 的另一棵语法分析树,它的结果是 $uv^nxy^n z$ 形式的字符串,其中 $n \geq 0$ 。为了完成

证明,还要证明 $vy \neq \epsilon$ 。如果 $vy = \epsilon$,则从 T 中删去 T' 不包括 T'' 在内的部分得到一棵根为 S 、结果为 w 且叶子数少于 T 的树。这与假设 T 是这类树中叶子数最少的树矛盾。 ■

例 3.5.2 与关于正则语言的泵定理(定理 2.4.1)一样,也可以用这个定理证明某些语言不是上下文无关的。例如, $L = \{a^n b^n c^n; n \geq 0\}$ 不是上下文无关的。假设存在某个上下文无关文法 $G = (V, \Sigma, R, S)$ 使得 $L = L(G)$ 。令 $n > \frac{1}{3}\phi(G)^{|V-\Sigma|}$ 。那么, $w = a^n b^n c^n \in L(G)$,可以表成 $w = uvxyz$ 使得 v 或 y 不是空串并且 $uv^i xy^i z \in L(G), i = 0, 1, 2, \dots$ 。有两种可能,它们都导致矛盾。如果 vy 中 a, b, c 三个符号都出现,则 v 和 y 中必有一个至少含有 a, b, c 中的两个符号。于是, $uv^2 xy^2 z$ 中 a, b, c 的排列顺序不对——有 b 在 a 的前面,或者有 c 在 a 或 b 的前面。如果 vy 中只出现 a, b, c 中的一个或两个符号,则 $uv^2 xy^2 z$ 中 a, b, c 的个数不相等。 ◇

例 3.5.3 $L = \{a^n; n \text{ 是一个素数}\}$ 不是上下文无关的。假设 $G = (V, \Sigma, R, S)$ 是生成 L 的上下文无关文法, 取一个大于 $\phi(G)^{|V|-2|}$ 的素数 p 。那么, $w = a^p$ 可以写成定理规定的形式 $w = uvxyz$, 这里 u, v 等都是 a 的字符串且 $vy \neq e$ 。设 $vy = a^r, uxz = a^q$, 这里 r 和 q 是自然数且 $q > 0$ 。定理说对所有的 $n \geq 0, r + nq$ 是素数。在例 2.4.3 中已经发现这是不可能的。

上述证明与例 2.4.3 中证明这个语言不是正则的非常相似, 这不是偶然的。原来任何单符号字母表上的上下文无关语言都是正则的。于是, 根据这个事实和例 2.4.3 立即可以得到本例的结果。◇

例 3.5.4 现在证明语言 $L = \{w \in \{a, b, c\}^*; w \text{ 中 } a, b \text{ 和 } c \text{ 的个数相等}\}$ 不是上下文无关的。这次需要定理 3.5.3 和 3.5.2: 如果 L 是上下文无关的, 则它与正则语言 $a^*b^*c^*$ 的交也是上下文无关的。而在上面的例 3.5.2 中已经证明这个语言 $\{a^n b^n c^n; n \geq 0\}$ 不是上下文无关的。◇

这些否定的事实还揭示了上下文无关语言不具备某些封闭性。

定理 3.5.4 上下文无关语言在交和补下不是封闭的。

证: $\{a^m b^n c^m; m, n \geq 0\}$ 和 $\{a^m b^n c^n; m, n \geq 0\}$ 都明显地是上下文无关的。它们的交是语言 $\{a^n b^n c^n; n \geq 0\}$, 刚才已经证明不是上下文无关的。又, 由于

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

如果上下文无关语言在补下封闭, 则它们在交下也封闭(已经知道它们在并下是封闭的, 定理 3.5.1)。

习 题

3.5.1 利用在并下的封闭性, 证明下述语言是上下文无关的:

- (a) $\{a^m b^n; m \neq n\}$
- (b) $\{a, b\}^* - \{a^n b^n; n \geq 0\}$
- (c) $\{a^m b^n c^p d^q; n = q, \text{ 或 } m \leq p, \text{ 或 } m + n = p + q\}$
- (d) $\{a, b\}^* - L$, 其中

$$L = \{babaabaaab \cdots ba^{n-1}ba^nb; n \geq 1\}$$

- (e) $\{w \in \{a, b\}^*; w = w^R\}$

3.5.2 利用定理 3.5.2 和 3.5.3, 证明下述语言不是上下文无关的:

- (a) $\{a^p; p \text{ 是素数}\}$ 。
- (b) $\{a^{n^2}; n \geq 0\}$ 。
- (c) $\{www; w \in \{a, b\}^*\}$ 。
- (d) $\{w \in \{a, b, c\}^*; w \text{ 中 } a, b \text{ 和 } c \text{ 的个数相同}\}$ 。

3.5.3 回忆一下, 同态是字符串到字符串的函数 h 使得对任意两个字符串 v 和 $w, h(vw) = h(v)h(w)$ 。于是, 一个同态被它在单个符号上的值所决定: 如果 $w = a_1 \cdots a_n$, 其中每一个 a_i 是一个符号, 则 $h(w) = h(a_1) \cdots h(a_n)$ 。注意同态能够起“删除作用”: 即使 w 不是 $e, h(w)$ 也可能是 e 。证明: 如果 L 是上下文无关语言, h 是一个同态, 则

- (a) $h[L]$ 是上下文无关的。

(b) $h^{-1}[L]$ (即, $\{w \in \Sigma^* : h(w) \in L\}$) 是上下文无关的。(提示: 从一台接受 L 的下推自动机 M 出发。另外构造一台类似于 M 的下推自动机, 但是它不是从输入带读它的输入, 而是从一个有穷的缓冲器读它的输入, 以某种方式不时地补充缓冲器。给出直观想法的其余部分和整个的形式细节。)

3.5.4 在定理 3.5.2 的证明中, 为什么要假设 M_2 是确定型的?

3.5.5 证明: 语言 $L = \{babaabaaab \cdots ba^{n-1}ba^nb; n \geq 1\}$ 不是上下文无关的。

(a) 使用泵定理(定理 3.5.3)。

(b) 使用习题 3.5.3 的结果。(提示: 当 $h(a) = aa$ 和 $h(b) = a$ 时, $h[L]$ 是什么?)

3.5.6 设 $L_1, L_2 \subseteq \Sigma^*$, L_1 除以 L_2 的右商定义为

$$L_1/L_2 = \{w \in \Sigma^* : \text{存在 } u \in L_2 \text{ 使得 } wu \in L_1\}$$

(a) 证明: 如果 L 是上下文无关的, R 是正则的, 则 L/R 是上下文无关的。

(b) 证明: $\{a^nb^n; p \text{ 是一个素数且 } n > p\}$ 不是上下文无关的。

3.5.7 证明下述更强的泵定理: 设 G 是一个上下文无关文法, 则存在数 K 和 k 使得任意的 $w \in L(G)$ 只要 $|w| \geq K$ 就能写成 $w = uvxyz$ 且满足下述条件: $|vxy| \leq k$, v 或 y 不是空串, 以及对每一个 $n \geq 0$, $uv^nxy^n z \in L(G)$ 。

3.5.8 利用习题 3.5.7 证明语言 $\{ww, w \in \{a, b\}^*\}$ 不是上下文无关的。

3.5.9 设 $G = (V, \Sigma, R, S)$ 是一个上下文无关文法, G 的非终结符 A 称作自嵌入的当且仅当对于某个 $u, v \in V^*$, $A \xrightarrow{G} uAv$ 。

(a) 给出一个算法检查给定上下文无关文法的一个指定的非终结符是否是自嵌入的。

(b) 证明: 如果 G 没有自嵌入的非终结符, 则 $L(G)$ 是一个正则语言。

3.5.10 如果上下文无关文法 $G = (V, \Sigma, R, S)$ 的每一条规则形如 $A \rightarrow w$, 其中 $w \in \Sigma(V - \Sigma)^*$, 则称 G 为 Greibach 范式。

(a) 证明: 对于每一个上下文无关文法 G , 存在 Greibach 范式的上下文无关文法 G' 使得 $L(G') = L(G) - \{e\}$ 。

(b) 证明: 像引理 3.4.1 的证明中那样从一个 Greibach 范式的文法构造 M , 则 M 关于输入 w 的任一计算中的步数不超过 w 长度的某个函数。

3.5.11 在习题 2.1.4 中引入了确定型有穷状态转换器。证明: 如果 L 是上下文无关的, f 可用确定型有穷状态转换器计算, 则

(a) $f[L]$ 是上下文无关的。

(b) $f^{-1}[L]$ 是上下文无关的。

3.5.12 给出关于上下文无关语言的泵定理的一种形式, 使其“被抽取部分”尽可能地长。

3.5.13 设 M_1 和 M_2 是两台下推自动机。说明如何构造接受 $L(M_1) \cup L(M_2)$ 、 $L(M_1)L(M_2)$ 和 $L(M_1)^*$ 的下推自动机, 从而给出定理 3.5.1 的另一种证明。

3.5.14 下述语言是上下文无关的吗? 给出扼要的说明。

(a) $\{a^mb^nc^p; m = n \text{ 或 } n = p \text{ 或 } m = p\}$

(b) $\{a^mb^nc^p; m \neq n \text{ 或 } n \neq p \text{ 或 } m \neq p\}$

- (c) $\{a^m b^n c^p; m = n \text{ 且 } n = p \text{ 且 } m = p\}$
 (d) $\{w \in \{a, b, c\}^*; w \text{ 中 } a, b \text{ 和 } c \text{ 的个数不全相同}\}$
 (e) $\{w \in \{a, b\}^*; w = w_1 w_2 \cdots w_m, \text{ 其中 } m \geq 2 \text{ 且 } |w_1| = |w_2| = \cdots = |w_m| \geq 2\}$

3.5.15 设 L 是上下文无关的, R 是正则的。 $L - R$ 肯定是上下文无关的吗? $R - L$ 呢? 证实你的回答。

3.6 关于上下文无关文法的算法

在这一节考虑与上下文无关文法有关的计算问题, 开发这些问题的算法并且分析它们的复杂性。总之, 给出了下述结果。

定理 3.6.1 (a) 有多项式算法, 任给一个上下文无关文法, 构造一台等价的下推自动机。

(b) 有多项式算法, 任给一台下推自动机, 构造一个等价的上下文无关文法。

(c) 有多项式算法, 任给一个上下文无关文法 G 和一个字符串 x , 判断是否 $x \in L(G)$ 。

将定理 3.6.1 与总结关于有穷自动机算法的对应定理(定理 2.6.1) 作比较, 可以从中学到启发。诚然, 它们有某些相似的地方: 都有把语言接受器转换成语言生成器和把语言生成器转换成语言接受器的算法。以前是把有穷自动机转换成正则表达式和反过来, 现在是把下推自动机转换成上下文无关文法和反过来。但是, 两者的区别可能更引人注意。首先, 由于正则语言可以用确定型有穷自动机描述, 而确定型有穷自动机正是确定上面(c) 中成员资格问题的有效算法, 故在定理 2.6.1 中不需要类似上面(c) 的部分。相反地, 对于上下文无关语言我们至今只引入了非确定型语言接受器——下推自动机, 它是非算法的。为了证明(c), 下面将要证明对于任意上下文无关语言, 能够构造一台确定型接受器, 这个构造方法不是直接的并且相当复杂, 所得到的算法虽然是多项式的, 但不再是输入长度线性界限的。

定理 2.6.1 与定理 3.6.1 之间的第二个主要区别是这次没有叙述检验任给的两个上下文无关文法(或两台下推自动机) 等价的算法, 也没有说存在下推自动机状态数的最小化算法。在第 5 章将看到, 关于上下文无关文法和下推自动机的这种问题不能用任何算法解决, 不管效率怎么低也不行!

3.6.1 动态规划算法

现在返回去证明定理 3.6.1 的(c)((a) 和(b) 是引理 3.4.1 和 3.4.2 证明中的构造方法的直接结果)。确定上下文无关语言成员资格问题的算法以一个“标准化”上下文无关文法的有用方法为基础。

定义 3.6.1 如果 $R \subseteq (V - \Sigma) \times V^2$, 则称上下文无关文法 $G = (V, \Sigma, R, S)$ 为 Chomsky 范式^①。

^① 多数著作和文献采用下述定义: 如果上下文无关文法 $G = (V, \Sigma, R, S)$ 的规则形如 $A \rightarrow BC$ 或 $A \rightarrow a$, 其中 $A, B, C \in V - \Sigma, a \in \Sigma$, 即 $R \subseteq (V - \Sigma) \times ((V - \Sigma)^2 \cup \Sigma)$, 则称 G 为 Chomsky 范式。——译者注

换句话说,在 Chomsky 范式上下文无关文法中,每一条规则的右边必须长度为 2。注意,Chomsky 范式文法不可能生成长度小于 2 的字符串,如不可能生成 a 、 b 和 ϵ 。因此,含有长度小于 2 的字符串的上下文无关语言不可能用 Chomsky 范式文法生成。不过,下面的结果表明这是 Chomsky 范式带来的唯一损失。

定理 3.6.2 对于任意的上下文无关文法 G ,存在 Chomsky 范式的上下文无关文法 G' 使得 $L(G') = L(G) - (\Sigma \cup \{\epsilon\})$,并且 G' 的构造可以在 G 的规模的多项式时间内完成。

换句话说,除长度小于 2 的字符串外, G' 与 G 生成的字符串完全相同。我们知道,由于 G' 是 Chomsky 范式的,它不可能生成长度小于 2 的字符串。

证:下面将要说明如何把任给的一个上下文无关文法 $G = (V, \Sigma, R, S)$ 转换成 Chomsky 范式。一条规则 $A \rightarrow x$ 的右边可能有三种方式违反 Chomsky 范式的规定:长规则(它们的右边有三个或三个以上的符号), ϵ 规则($A \rightarrow \epsilon$ 的形式)以及短规则($A \rightarrow a$ 或 $A \rightarrow B$ 的形式)。

首先处理 G 的长规则。设 $A \rightarrow B_1 B_2 \cdots B_n \in R$,其中 $B_1, B_2, \dots, B_n \in V$ 且 $n \geq 3$ 。把这条规则替换成 $n-1$ 条新规则:

$$\begin{aligned} A &\rightarrow B_1 A_1, \\ A_1 &\rightarrow B_2 A_2, \\ &\vdots \\ A_{n-2} &\rightarrow B_{n-1} B_n, \end{aligned}$$

其中 A_1, \dots, A_{n-2} 是新的非终结符并且不在文法中的其他地方使用。因为可以用这些新规则模拟 $A \rightarrow B_1 B_2 \cdots B_n$,并且这也是使用这些新规则的唯一方式,所以得到的上下文无关文法等价于原来的文法。对文法中的每一条长规则重复上述操作。所得到的文法等价于原来的文法,并且每一条规则的右边长度等于或小于 2。

例 3.6.1 取生成平衡括号串集合的文法,其规则为 $S \rightarrow SS, S \rightarrow (S), S \rightarrow \epsilon$ 。这里只有一条长规则 $S \rightarrow (S)$ 。把它替换成两条规则 $S \rightarrow (S_1$ 和 $S_1 \rightarrow S)$ 。◇

接下去应该考虑 ϵ 规则。为此,先确定可删除的非终结符集合

$$\mathcal{E} = \{A \in V - \Sigma : A \Rightarrow^+ \epsilon\}$$

即,能够推导出空串的所有非终结符的集合。用下述简单的闭包算法可以得到 \mathcal{E} :

$\mathcal{E} := \emptyset$

while 有一条规则 $A \rightarrow \alpha$, 其中 $\alpha \in \mathcal{E}^*$ 且 $A \notin \mathcal{E}$ do 把 A 加入 \mathcal{E} 。

得到集合 \mathcal{E} 之后,从 G 中删去所有的 ϵ 规则,重复下述操作:对每一条 $A \rightarrow BC$ 或 $A \rightarrow CB$ 形式的规则,其中 $B \in \mathcal{E}, C \in V$,在文法中加入规则 $A \rightarrow C$ 。在原文法中的任何推导都能够用新文法模拟,反之亦然。这里仅有一个例外:不再能够推导出 ϵ ,因为在这一步可能删去规则 $S \rightarrow \epsilon$ 。幸运的是,定理允许这个例外。

例 3.6.1(续) 继续进行,现在文法的规则为:

$$S \rightarrow SS, S \rightarrow (S_1, S_1 \rightarrow S), S \rightarrow \epsilon$$

先计算可删除的非终结符集合 \mathcal{E} :开始时 $\mathcal{E} = \emptyset$;然后,由规则 $S \rightarrow \epsilon$,得 $\mathcal{E} = \{S\}$,而且这是 \mathcal{E} 的最终的值。从文法中删去所有的 ϵ 规则(这里只有一条 ϵ 规则 $S \rightarrow \epsilon$),并且对每一条

右边出现 S 的规则,删去这个 S 得到一条新的规则。新的规则集合为

$$S \rightarrow SS, S \rightarrow (S_1, S_1 \rightarrow S), S \rightarrow S, S_1 \rightarrow)$$

由规则 $S \rightarrow SS$ 及 $S \in \mathcal{E}$, 添加规则 $S \rightarrow S$ 。当然,这一条规则是无用的,可以删去。由规则 $S_1 \rightarrow S$ 及 $S \in \mathcal{E}$, 添加规则 $S_1 \rightarrow)$ 。

例如,在原文法中的推导

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow S() \Rightarrow ()$$

现在可以模拟如下:

$$S \Rightarrow (S_1 \Rightarrow ()$$

由于第一个 S 最终要被去掉,故删去 $S \Rightarrow SS$ 这一步;使用规则 $S_1 \Rightarrow)$ 提前删去了规则 $S_1 \Rightarrow S$ 中的 S 。◇

现在文法中只有右边长度为 1 或 2 的规则。下面应该去掉短规则,这些规则的右边的长度为 1。做法如下:对每一个 $A \in V$,再次用简单的闭包算法计算用文法从 A 能够推导出的符号的集合 $\mathcal{D}(A) = \{B \in V : A \Rightarrow^+ B\}$ 如下:

$$\mathcal{D}(A) := \{A\}$$

while 有一个规则 $B \rightarrow C$, 其中 $B \in \mathcal{D}(A)$ 且 $C \notin \mathcal{D}(A)$ do 把 C 加入 $\mathcal{D}(A)$ 。

注意,对所有的符号 $A, A \in \mathcal{D}(A)$ 。如果 a 是一个终结符,则 $\mathcal{D}(a) = \{a\}$ 。

在转换的最后一步,从文法中删去所有的短规则,并且把每一条 $A \rightarrow BC$ 形式的规则替换成所有 $A \rightarrow B'C'$ 形式的规则,其中 $B' \in \mathcal{D}(B)$ 和 $C' \in \mathcal{D}(C)$ 。这样一条规则模拟原来的规则 $A \rightarrow BC$ 以及从 B 产生 B' 和从 C 产生 C' 的一系列短规则的效果。最后,对每一条规则 $A \rightarrow BC$,如果 $A \in \mathcal{D}(S) - \{S\}$,则添加规则 $S \rightarrow BC$ 。

因为当一条短规则的左边第一次出现在推导中时“提前”使用这条短规则就模拟了它的作用,故所得到的文法等价于删除所有短规则之前的文法(如果短规则的左边是 S ,则从它开始推导,在构造的最后添加的规则 $S \rightarrow BC$ 足以保证等价性)。这里也只有一个例外:可能删去规则 $S \rightarrow a$,从而从 G 生成的语言中删去字符串 a 。也同样幸运的是,这是定理所允许的。

例 3.6.1(续) 在修改后的文法中,规则为

$$S \rightarrow SS, S \rightarrow (S_1, S_1 \rightarrow S), S_1 \rightarrow)$$

有 $\mathcal{D}(S_1) = \{S_1,)\}$ 和 $\mathcal{D}(A) = \{A\}, A \in V - \{S_1\}$ 。删去所有长度为 1 的规则,这里只有一条 $S_1 \rightarrow)$ 。只有一个非终结符(即, S_1) 的集合 \mathcal{D} 是非平凡的,它仅出现在第二条规则的右边。因此,把这条规则替换成两条规则 $S \rightarrow (S_1$ 和 $S \rightarrow ($, 分别对应 $\mathcal{D}(S_1)$ 中的两个元素。最后得到的 Chomsky 范式文法是

$$S \rightarrow SS, S \rightarrow (S_1, S_1 \rightarrow S), S \rightarrow ($$

◇

经过这三步之后,文法成为 Chomsky 范式的,并且除可能丢失长度小于 2 的字符串外,它与原来的文法生成相同的语言。

为了完成定理的证明,还应该证明整个构造能够在原来的文法 G 的规模的多项式时间内完成。我们用“ G 的规模”表示足以完整地描述 G 的字符串的长度,即 G 的所有规则的长度之和。令 n 为这个量。变换的第一部分(消去长规则)用 $\mathcal{O}(n)$ 时间产生一个规模为 $\mathcal{O}(n)$ 的文法。第二部分消去 ϵ 规则,闭包计算用 $\mathcal{O}(n^2)$ 时间($\mathcal{O}(n)$ 次迭代,每次可在 $\mathcal{O}(n)$)

时间内完成),加上用来添加新规则的时间 $O(n)$ 。最后,第三部分(处理短规则)也可以在多项式时间内实现(计算 $O(n)$ 个闭包,每个用 $O(n^2)$ 时间)。证毕。 ■

Chomsky 范式的优点是有一个简单的多项式算法能够判断一个字符串是否可以用这个文法生成。假设给定一个 Chomsky 范式的上下文无关文法 $G = (V, \Sigma, R, S)$, 问字符串 $x = x_1x_2\cdots x_n (n \geq 2)$ 是否在 $L(G)$ 中。下面给出算法。它通过分析 x 的所有子串确定是否 $x \in L(G)$ 。对于每一个 i 和 $s (1 \leq i \leq i+s \leq n)$, 定义 $N[i, i+s]$ 为在 G 中能够推导出字符串 $x_i \cdots x_{i+s}$ 的符号的集合。算法计算这些集合 $N[i, i+s]$, 从短的字符串(小的 s)到越来越长的字符串。从很小的子问题开始, 构造越来越大的问题直至整个问题的解, 这种求解问题的一般原理称作动态规划。

```
for  $i := 1$  to  $n$  do  $N[i, i] := \{x_i\}$ ; 所有其他  $N[i, j]$  开始时为空集
for  $s := 1$  to  $n - 1$  do
  for  $i := 1$  to  $n - s$  do
    for  $k := i$  to  $i + s - 1$  do
      if 有  $A \rightarrow BC \in R$  且  $B \in N[i, k]$  和  $C \in N[k + 1, i + s]$ 
      then 把  $A$  加入  $N[i, i + s]$ ;
if  $S \in N[1, n]$  then 接受  $x$ 
```

为了证明上述算法正确地确定是否 $x \in L(G)$, 我们将证明下述断言。

断言 对于每一个自然数 $s (0 \leq s \leq n)$, 在第 s 次迭代以后, 对所有 $i = 1, \dots, n - s$,

$$N[i, i + s] = \{A \in V : A \xRightarrow{*} x_i \cdots x_{i+s}\}$$

证: 对 s 作归纳证明。

基础步骤 当 $s = 0$ 时, 我们把“第 0 次迭代”理解为第一(初始化)行, 断言为真: 因为 G 是 Chomsky 范式的, 故能够生成终结符 x_i 的唯一符号是它自己。

归纳步骤 假设对于所有小于 $s > 0$ 的整数, 断言为真。考虑子串 $x_i \cdots x_{i+s}$ 的一个推导。比如说, 它从非终结符 A 开始, 由于 G 为 Chomsky 范式, 推导从使用一条规则 $A \rightarrow BC$ 开始, 即

$$A \Rightarrow BC \xRightarrow{*} x_i \cdots x_{i+s}$$

其中 $B, C \in V$ 。因此, 对某个 $k (i \leq k < i + s)$,

$$B \xRightarrow{*} x_i \cdots x_k, \quad C \xRightarrow{*} x_{k+1} \cdots x_{i+s}$$

由此得到 $A \in \{A \in V : A \xRightarrow{*} x_i \cdots x_{i+s}\}$ 当且仅当存在整数 $k (i \leq k < i + s)$ 和两个符号 $B \in \{A \in V : A \xRightarrow{*} x_i \cdots x_k\}$ 与 $C \in \{A \in V : A \xRightarrow{*} x_{k+1} \cdots x_{i+s}\}$ 使得 $A \rightarrow BC \in R$ 。可以把字符串 $x_i \cdots x_k$ 写成 $x_i \cdots x_{i+s'}$, 其中 $s' = k - i$; 把字符串 $x_{k+1} \cdots x_{i+s}$ 写成 $x_{k+1} \cdots x_{k+1+s''}$, 其中 $s'' = i + s - k - 1$ 。注意到, 由于 $i \leq k < i + s$, 有 $s', s'' < s$ 。因而, 可以利用归纳假设!

根据归纳假设, $\{A \in V : A \xRightarrow{*} x_i \cdots x_k\} = N[i, k]$, $\{A \in V : A \xRightarrow{*} x_{k+1} \cdots x_{i+s}\} = N[k + 1, i + s]$ 。因此, $A \in \{A \in V : A \xRightarrow{*} x_i \cdots x_{i+s}\}$ 当且仅当存在整数 $k (i \leq k < i + s)$ 和两个符号 $B \in N[i, k]$ 与 $C \in N[k + 1, i + s]$ 使得 $A \rightarrow BC \in R$ 。而这些正好是算法把 A 加入 $N[i, i + s]$ 的条件。所以, 断言对 s 也为真。断言的证明结束。 ■

由断言可立即推出上述算法正确地确定是否 $x \in L(G)$; 在结束时, 集合 $N[1, n]$ 将包

含能够推导出字符串 $x_1 \cdots x_n = x$ 的所有符号。因此, $x \in L(G)$ 当且仅当 $S \in N[1, n]$ 。

下面分析算法的时间性能。注意到算法由三层嵌套的循环组成, 每个循环的范围不超过 $|x| = n$ 。在最里层的循环, 需要对每一条规则 $A \rightarrow BC$ 检查是否 $B \in N[i, j]$ 和 $C \in N[j+1, i+s]$, 这可以在正比于文法 G 的规模(即, 它的规则的总长度)的时间内完成。因此, 总的运算量为 $O(|x|^3|G|)$, 这是 x 的长度与 G 的规模的二元多项式。对于任意固定的文法 G (即, 当把 $|G|$ 看作常数时), 算法的运行时间为 $O(n^3)$ 。■

例 3.6.1(续) 让我们把这个动态规划算法应用于关于平衡括号的文法, 它已被重新写成 Chomsky 范式, 其规则为

$$S \rightarrow SS, S \rightarrow (S_1, S_1 \rightarrow S), S \rightarrow ()$$

假设要回答字符串 $((()))$ 是否能够被 G 生成。在图 3-10 中列出算法迭代过程中得到的所有 $N[i, j]$, 其中 $1 \leq i \leq j \leq n = 8$ 。计算沿表中对角线进行。主对角线对应 $s = 0$, 它的内容是要分析的字符串。为了填写一个格子, 比如 $N[2, 7]$, 我们看格子的有序对 $N[2, k]$ 与 $N[k+1, 7]$, $2 \leq k < 7$ 。这些格子位于要填写的格子的左面或上面。当 $k = 3$ 时, $S \in N[2, 3]$, $S \in N[4, 7]$ 和 $S \rightarrow SS$ 是一条规则, 应把这条规则的左边 S 加入格子 $N[2, 7]$, 等等。右下角是 $N[1, n]$, 它包含 S , 因此这个字符串确实在 $L(G)$ 中。实际上, 通过检查这张表容易找到字符串 $((()))$ 在 G 中的一个推导。可以很容易地修改这个动态规划算法使得它给出这个推导, 见习题 3.6.2。◇

							8)
						7)	∅
					6)	∅	∅
				5	(S	S ₁	∅
			4	(∅	∅	S	S ₁
		3)	∅	∅	∅	∅	∅
	2	(S	∅	∅	∅	S	S ₁
1	(∅	∅	∅	∅	∅	∅	S
	1	2	3	4	5	6	7	8

图 3-10

由定理 3.6.2 和上面的断言可以得到定理 3.6.1 的(c)。任意给定一个上下文无关文法 G 和一个字符串 x , 如下确定是否 $x \in L(G)$: 首先, 采用定理 3.6.2 证明中的构造方法在多项式时间内把 G 变换成等价的 Chomsky 范式文法 G' 。当 $|x| \leq 1$ 时, $x \in L(G)$ 当且仅当在变换过程中删去规则 $S \rightarrow x$, 因而能够确定是否 $x \in L(G)$ 。否则, 对文法 G' 和字符串 x 运行上面描述的动态规划算法。算法使用的总运算量不超过原来的文法 G 的规模和字符串 x 的长度的一个多项式。■

习 题

- 3.6.1 把例 3.1.3 中生成算术表达式的上下文无关文法 G 转换成等价的 Chomsky 范式上下文无关文法。运用动态规划算法确定字符串 $x = (\text{id} + \text{id}) * (\text{id})$ 是否在 $L(G)$ 中。
- 3.6.2 如何修改正文中的动态规划算法, 使得当输入 x 在 G 生成的语言中时, 算法给出 x 在 G 中的一个推导?
- 3.6.3 (a) 设 $G = (V, \Sigma, R, S)$ 是一个上下文无关文法, $A \in V - \Sigma$ 。如果对于某个 $x \in \Sigma^*$, $A \Rightarrow_G x$, 则称 A 是生成的。给出求 G 的所有生成的非终结符的多项式算法。(提示, 这是一个闭包算法。)
(b) 给出一个多项式算法, 任给一个上下文无关文法 G , 判断是否 $L(G) = \emptyset$ 。

3.6.4 描述一个算法,任给一个上下文无关文法 G ,判断 $L(G)$ 是否是无穷的。(提示:一种方法是使用泵定理。)你给出的算法的复杂度是多少?你能够找到一个多项式算法吗?

3.7 确定性与语法分析

正如例 3.1.3 所表明的,上下文无关文法广泛地用于建立程序设计语言的语法模型。这种程序设计语言的编译程序必须包含一个语法分析器。语法分析器是一个算法,任给一个字符串和一个上下文无关文法,它能够判断这个字符串是否在这个文法生成的语言中,并且如果在的话,构造出这个字符串的语法分析树。(然后,编译程序继续把这棵语法分析树翻译成更基本的语言的程序,如汇编语言的程序。)在上一节开发的通用上下文无关的语法分析器(即,那个动态规划算法)虽然完全是多项式的,但是用来处理几万条指令的程序显得太慢了(记得它三次地依赖于这个字符串的长度)。在过去的 40 年,编译程序设计人员开发出许多研究语法分析问题的方法。有趣的是,它们中大多数成功的方法植根于下推自动机的思想。归根结底是下推自动机与上下文无关文法的等价性在起作用,这种等价性在 3.4 节中已被证明。但是,由于下推自动机是一种非确定型设备,它不能直接地实际应用于语法分析。于是,产生一个问题,总能够确定性地运行下推自动机(像我们对有穷自动机能够做到的那样)吗?

本节的第一个目的是研究确定型下推自动机问题。我们将看到某些上下文无关语言不可能被确定型下推自动机接受。这是相当令人失望的,它表明在 3.4 节中把文法转换成自动机的方法不可能作为任何实际方法的基础。虽然如此,实际上并没失去什么。原来对于大多数程序设计语言,人们能够构造出确定型下推自动机接受所有语法正确的程序。本节在晚些时候要给出某些启发式规则,即经验规则。从合适的上下文无关文法构造确定型下推自动机时,这些规则是有用的。从任意的上下文无关文法,这些规则不总是产生一个有用的下推自动机,我们已经说过这是不可能的。但是,它们是在构造程序设计语言的编译程序时实际使用的典型方法。

3.7.1 确定型上下文无关语言

如果在下推自动机 M 的计算中每一个格局至多能够有一个格局接在它的后面,则称 M 是确定型的。这个条件可以用另一种等价的方式描述。如果两个字符串中的一个另一个的前缀,则称这两个字符串是相容的。设两个转移 $((p, a, \beta), (q, \gamma))$ 和 $((p, a', \beta'), (q', \gamma'))$ 。如果 a 和 a' 相容、 β 和 β' 也相容,换句话说,在某种情况下这两个转移都可以利用,则称这两个转移是兼容的。如果 M 没有两个不同的转移是兼容的,则称 M 是确定型的。

例如,在例 3.3.1 中构造的接受语言 $\{w^R; w \in \{a, b\}^*\}$ 的机器是确定型的;对于每一个状态和输入符号,只可能有一个转移。相反地,在例 3.3.2 中构造的接受 $\{ww^R; w \in \{a, b\}^*\}$ 的机器不是确定型的:转移 3 与转移 1 和 2 都兼容。注意到这几个都是“猜想”字符串中点的转移,这个动作在直观上就是非确定性的。

确定型上下文无关语言在本质上是确定型下推自动机接受的语言。然而,由于很快就会清楚的原因,必须对接受的规定作一点修改。如果一个语言被一台具有额外的觉察输入串结束的能力的确定型下推自动机接受,则称这个语言是确定型上下文无关的。形式地,如果 $L\$ = L(M)$, 其中 M 是一台确定型下推自动机, 则称 $L \subseteq \Sigma^*$ 是确定型上下文无关语言。这里 $\$$ 是一个不在 Σ 中的新符号, 把它附在每一个输入串的后面用来标记这个输入串的结束。

相反地, 如果不采用这种特殊的接受规定, 则许多直观上是确定型的上下文无关语言按照我们的定义不是确定型的。 $L = a^* \cup \{a^n b^n; n \geq 1\}$ 就是一个例子。一台确定型下推自动机不可能既记住已经看见多少个 a , 以便核查可能跟在后面的一串 b , 同时又准备以空栈接受后面不跟着 b 的情况。可是, 很容易设计一台确定型下推自动机接受 $L\$$; 如果当机器还在积累 a 的时候碰到 $\$$, 则输入是 a^* 中的一个字符串。如果出现这种情况, 则将栈排空并且接受这个输入。

这里有一个自然的问题: 是否像每一个正则语言都被一台确定型有穷自动机接受一样, 每一个上下文无关语言都是确定型的? 如果这是真的, 那将是惊人的。例如, 考虑上下文无关语言

$$L = \{a^m b^n c^p; m, n, p \geq 0 \text{ 且 } m \neq n \text{ 或 } m \neq p\}$$

看来一台下推自动机接受这个语言一定要猜想比较哪两段符号: a 段与 b 段, 还是 b 段与 c 段。不这样使用非确定性, 似乎机器不可能在比较 b 段与 a 段的同时, 又在准备比较 b 段与 c 段。但是, 证明 L 不是确定型的需要一个间接的论据: L 的补不是上下文无关的。

定理 3.7.1 确定型上下文无关语言类在补运算下是封闭的。

证: 设 $L \subseteq \Sigma^*$ 是一个语言, $L\$$ 被确定型下推自动机 $M = (K, \Sigma, \Gamma, \Delta, s, F)$ 接受。像引理 3.4.2 证明中那样, 为方便起见可以假设 M 是简单的。即, M 的转移至多把一个符号托出栈, 同时初始转移把栈底符 Z 放入栈直到在计算的结束前把它移出去。容易看出在引理 3.4.2 的证明中用于这种结束的构造方法不影响 M 的确定型性质。

因为 M 是确定型的, 所以看来为了获得一台接受 $(\Sigma^* - L)\$$ 的设备只需要调换接受状态和非接受状态, 这与在定理 2.3.1(d) 的证明中对确定型有穷自动机所做过的一样, 而且在下一章还要对更复杂的确定型设备这样办。但是, 现在不能这样简单地调换, 因为一台确定型下推自动机拒绝它的输入不仅可以是读它的输入并且最后进入一个非接受状态, 也可以是根本不读完它的输入。这种迷惑人的可能性在下述两种情况下出现: 第一种, 在 M 进入的格局中没有转移可供利用。第二种, M 进入一个格局, 从这个格局开始 M 执行一个永不结束的 e 动作 (形如 $((q, e, a), (p, \beta))$ 的转移) 序列, 后一种可能更迷人。

设 $C = (q, w, \alpha)$ 是 M 的一个格局。如果对于任一不同的格局 $C' = (q', w', \alpha')$, $C \vdash_M C'$ 蕴涵 $w' = w$ 且 $|\alpha'| \geq |\alpha|$, 则称 C 是一个死结束。也就是说, 如果从一个格局开始不可能再读输入和减少栈的高度, 则称这个格局是死结束。显然, 如果 M 处于一个死结束格局, 则它不可能读完它的输入。相反地, 不难看出如果 M 没有死结束格局, 则它一定读完全部输入。这是因为在没有死结束格局的情况下, 在每一步, 后面总有一步读下一个输入符号或者减少栈的高度。而第二种可能只能出现有穷次, 因为栈的高度不能减少无穷多次。

下面要说明如何把任一台简单的确定型下推自动机 M 变换成一台等价的没有死结束格局的确定型下推自动机。关键的一点是,由于假设 M 是简单的,一个格局是不是死结束仅与当前的状态,下一个输入符号以及栈顶符号有关。具体地说,设 $q \in K$ 是一个状态、 $a \in \Sigma$ 是一个输入符号以及 $A \in \Gamma$ 是一个栈符号。如果不存在状态 p 和栈符号串 α 使得格局 (q, a, A) 产生 (p, e, a) 或 (p, a, e) , 则称三元组 (q, a, A) 是死结束。也就是说,如果把一个三元组看成格局时它是一个死结束,则称这个三元组是一个死结束。令 $D \subseteq K \times \Sigma \times \Gamma$ 表示所有死结束三元组的集合。注意我们没有说能够有效地回答一个三元组是不是在 D 中(尽管这是能够做到的),我们只说集合 D 是有定义的,是一个有穷的三元组集。

如下修改 M : 对于每一个三元组 $(q, a, A) \in D$, 从 Δ 中删去所有与 (q, a, A) 兼容的转移,并且把转移 $((q, a, A), (r, e))$ 加入 Δ , 其中 r 是一个新的非接受状态。最后,在 Δ 中加入下述转移: 对每一个 $a \in \Sigma$, $((r, a, e), (r, e))$; $((r, \$, e), (r', e))$; 以及对每一个 $A \in \Gamma \cup \{Z\}$, $((r', e, A), (r', e))$, 其中 r' 是另一个新的非接受状态。这些转移保证自动机当处于状态 r 时,能够读完整个输入(不查阅栈),再读一个 $\$$, 然后排空栈并且拒绝这个输入。把所得到的这个下推自动机叫做 M' 。

容易验证 M' 是确定型的,与 M 接受相同的语言(一旦 M 不能再读输入剩余的部分,从而肯定要拒绝的时候, M' 就直截了当地拒绝)。此外,根据构造的方法, M' 没有死结束格局,因而它总是在读完整个输入后结束。现在调换 M' 的接受状态和非接受状态产生一台接受 $(\Sigma^* - L) \$$ 的确定型下推自动机。证毕。 ■

定理 3.7.1 说明上面的上下文无关语言 $L = \{a^m b^n c^p; m \neq n \text{ 或 } m \neq p\}$ 不是确定型的; 如果 L 是确定型的,则它的补 \bar{L} 也是确定型上下文无关的,当然是上下文无关的。因而,根据定理 3.5.2 它与正则语言 $a^* b^* c^*$ 的交是上下文无关的。但是,容易看出 $\bar{L} \cap a^* b^* c^*$ 正好是 $\{a^n b^n c^n; n \geq 0\}$, 我们知道它不是上下文无关的。因此,上下文无关语言 L 不是确定型上下文无关的。

推论 确定型上下文无关语言类是上下文无关语言类的真子集。

换句话说,就下推自动机而言,非确定性比确定性更有力。相反地,在上一章我们看到非确定性没有增加有穷自动机的能力,除非把状态数考虑在内。如果考虑状态数,非确定性使能力指数地强。在各种计算模型中非确定性的能力这个吸引人的题目大概是唯一贯穿全书的最重要的线索。

3.7.2 自顶向下的语法分析

我们已经证明,不是每一个上下文无关语言都能够用一台确定型下推自动机接受。现在考虑那些能够用确定型下推自动机接受的上下文无关语言。本章的剩余部分的总目标是研究上下文无关文法能够转换成可实际用于“产业级”语言识别的确定型下推自动机的情况。不过,这儿的风格与本书其余部分的不同,很少证明,并且不打算把我们介绍的各种目的的想法联系在一起。我们提出一些准则,叫做“启发式规则”。它们不是在所有的情况下都有用,而我们甚至不打算准确地划定什么时候它们是有用的。也就是说,我们的目标是介绍本章前面讲过的理论的某些有启发的应用,而且也只能把它看作一个介绍。

从一个例子开始,语言 $L = \{a^n b^n\}$ 由上下文无关文法 $G = (\{a, b, S\}, \{a, b\}, R, S)$ 生

成,其中 R 包含两条规则 $S \rightarrow aSb$ 和 $S \rightarrow e$ 。我们知道如何构造一台接受 L 的下推自动机:正好是对这个文法 G 实现引理 3.4.1 中的构造方法。结果是

$$M_1 = (\{p, q\}, \{a, b\}, \{a, b, S\}, \Delta_1, p, \{q\})$$

其中

$$\Delta_1 = \{((p, e, e), (q, S)), ((q, e, S), (q, aSb)), \\ ((q, e, S), (q, e)), ((q, a, a), (q, e)), ((q, b, b), (q, e))\}$$

因为 M_1 有两个不同的转移具有相同的第一个分量(它们对应于 G 的两条左边相同的规则),所以它不是确定型的。

可是, L 是一个确定型上下文无关语言,能够把 M_1 修改成一台接受 $L\$$ 的确定型下推自动机 M_2 。直观上, M_1 在每一步为了决定使用这两条规则中的哪一条只需要知道下一个输入符号。如果下一个输入符号是 a , M_1 应在它的栈中把 S 替换成 aSb ,这样才有希望得到接受计算。相反地,如果下一个输入符号是 b ,则机器应托出 S 。 M_2 通过提前消耗一个输入符号并且把这个信息并入它的状态达到需要预先知道下一个输入符号的目的。形式地,

$$M_2 = (\{p, q, q_a, q_b, q_\$ \}, \{a, b\}, \{a, b, S\}, \Delta_2, p, \{q_\$ \})$$

其中 Δ_2 包含下述转移:

- (1) $((p, e, e), (q, S))$
- (2) $((q, a, e), (q_a, e))$
- (3) $((q_a, e, a), (q, e))$
- (4) $((q, b, e), (q_b, e))$
- (5) $((q_b, e, b), (q, e))$
- (6) $((q, \$, e), (q_\$, e))$
- (7) $((q_a, e, S), (q_a, aSb))$
- (8) $((q_b, e, S), (q_b, e))$

M_2 在状态 q 读一个输入符号并且进入三个新状态 q_a 、 q_b 和 $q_\$$ 中的一个,而不改变栈。然后,它使用区分两个兼容的转移 $((q, e, S), (q, aSb))$ 和 $((q, e, S), (q, e))$ 的那个信息:第一个转移只能在状态 q_a 使用,第二个转移只能在状态 q_b 使用。所以, M_2 是确定型的。它接受输入 $ab\$$ 如表 3-4 所示。

因此, M_2 可以用作一台识别 a^*b^* 形式的字符串的确定型设备。此外,记住 M_2 的哪一个转移是由文法的哪一条规则得到的(这是表 3-4 的最后一列),可以用 M_2 的运行轨迹重新构造出这个输入串的最左推导。具体地说,在计算中替换栈顶非终结符的步骤(例中的步骤 3 和 6)对应从根向树叶构造一棵语法分析树,见图 3-11(a)。

像 M_2 这种能够正确地决定一个字符串是否属于一个上下文无关语言,并且当属于时生成对应的语法分析树的设备叫做**语法分析器**。特别地,因为 M_2 用自顶向下和从左到右的方式跟踪它在栈中替换非终结符的步骤的操作,重新构造一棵语法分析树,所以 M_2 是一个**自顶向下的语法分析器**(见图 3-11(b),一个有启发性的表示在自顶向下的语法分析器中如何进行的方法)。我们很快将看到一个更有价值的例子。

自然,不是所有的上下文无关语言都能用这种提前看的思想从标准的非确定型的接

表 3-4

步 骤	状 态	未读的输入	栈	使用的转移	G 的规则
0	p	$ab\$$	e	—	
1	q	$ab\$$	S	1	
2	q_a	$b\$$	S	2	
3	q_a	$b\$$	aSb	7	$S \rightarrow aSb$
4	q	$b\$$	Sb	3	
5	q_b	$\$$	Sb	4	
6	q_b	$\$$	b	8	$S \rightarrow e$
7	q	$\$$	e	5	
8	$q\$$	e	e	6	

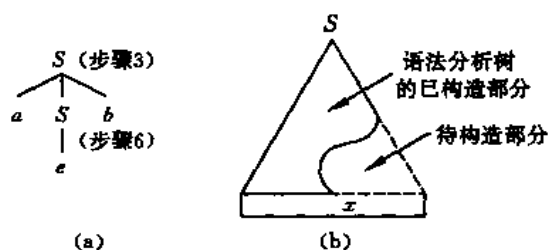


图 3-11

受器得到确定型的接受器。例如，我们在前一小节看到某些上下文无关语言一开始就不是确定性的。甚至对于某些确定型上下文无关语言，只提前看一个符号可能不足以解决所有的不确定性。可是，某些语言不能直接用提前看的办法是由于一些表面上的原因，稍微修改一下文法就能够消除掉。下面着重考虑这些问题。

回想生成带 $+$ 和 $*$ 运算的算术表达式的文法 G (例 3.1.3)。给这个文法增加一条规则

$$F \rightarrow \text{id}(E) \quad (\text{R7})$$

使得在算术表达式中可以出现函数调用，如 $\text{sqrt}(x * x + 1)$ 和 $f(x)$ 。

下面构造这个文法的自顶向下的语法分析器，用 3.4 节中的构造方法给出下述下推自动机

$$M_3 = (\{p, q\}, \Sigma, \Gamma, \Delta, p, \{q\})$$

其中

$$\Sigma = \{ (,), +, *, \text{id} \}$$

$$\Gamma = \Sigma \cup \{ E, T, F \}$$

以及 Δ 如下：

$$(0) ((p, e, e), (q, E))$$

$$(1) ((q, e, E), (q, E + T))$$

- (2) $((q, e, E), (q, T))$
- (3) $((q, e, T), (q, T * F))$
- (4) $((q, e, T), (q, F))$
- (5) $((q, e, F), (q, (E)))$
- (6) $((q, e, F), (q, id))$
- (7) $((q, e, F), (q, id(E)))$

以及对每一个 $a \in \Sigma, ((q, a, a), (q, e))$ 。

M_3 的非确定性表现在转移 1 与 2, 3 与 4, 以及 5、6、与 7 三组, 它们有相同的第一个分量。更糟的是, 不能根据下一个输入符号决定使用哪一个转移。让我们更仔细地考察为什么会这样。

转移 6 与 7。假设 M_3 的格局是 (q, id, F) 。这时 M_3 可以按照转移 5、6 和 7 中的任意一个动作。看见下一个符号 id , M_3 可以把转移 5 排除在外, 因为转移 5 要求下一个符号是 $($ 。 M_3 仍然不能决定使用转移 6 还是使用转移 7, 因为它们都在栈顶生成 id , 能够与下一个符号一致。出现这个问题的原因是 G 的规则 $F \rightarrow id$ 和 $F \rightarrow id(E)$ 不仅左边相同, 而且右边的第一个符号也相同。

这里有一个很简单的绕过这个问题的方法: 把 G 中的规则 $F \rightarrow id$ 和 $F \rightarrow id(E)$ 替换成 $F \rightarrow idA, A \rightarrow e$ 和 $A \rightarrow (E)$, 其中 A 是一个新的非终结符 (A 代表变量)。这样做的效果是把规则 $F \rightarrow id$ 和 $F \rightarrow id(E)$ 之间作出决定的时间“拖延”到掌握所有必需的信息为止。现在从修改后的文法得到一个修改了的下推自动机, 转移 6 和 7 被替换成:

- (6') $((q, e, F), (q, idA))$
- (7') $((q, e, A), (q, e))$
- (8') $((q, e, A), (q, (E)))$

现在提前看见一个符号足以作出做正确动作的决定。例如, 格局 $(q, id(id), F)$ 将会产生 $(q, id(id), idA), (q, (id), A), (q, (id), (E))$ 等等。

这种避免非确定性的技术叫做左因子分解。可以将它总结如下。

启发式规则 1 如果有规则 $A \rightarrow \alpha\beta_1, A \rightarrow \alpha\beta_2, \dots, A \rightarrow \alpha\beta_n$, 其中 $\alpha \neq e$ 且 $n \geq 2$, 则把它们替换成规则 $A \rightarrow \alpha A'$ 和 $A' \rightarrow \beta_i, i = 1, 2, \dots, n$, 其中 A' 是一个新的非终结符。

不难看出使用启发式规则 1 不改变文法生成的语言。

下面考查妨碍我们把 M_3 转换成确定型语法分析器的第二种异常情况。

转移 1 与 2。这两个转移提出更严重的问题。如果自动机的下一个输入符号是 id , 栈的内容正好是 E , 则它可以做好几种动作。它可以执行转移 2, 把 E 替换成 T (比如当输入是 id 时, 可以证明这样做是正当的)。它也可以把 E 替换成 $E + T$ (转移 1), 然后用 T 替换栈顶的 E (如果输入是 $id + id$, 则应该这样做)。它还可以执行转移 2 两次和转移 1 一次 (当输入是 $id + id + id$ 时) 等等。为了选择正确的动作, 自动机必须提前看多远似乎无论如何都是无界的。这里的罪魁祸首是规则 $E \rightarrow E + T$, 它左边的非终结符又是右边的第一个符号。这种现象叫作左递归, 对文法进一步做外科手术能够消除掉。

为了消除规则 $E \rightarrow E + T$ 产生的左递归, 只需把它替换成 $E \rightarrow TE', E' \rightarrow + TE'$ 和 $E' \rightarrow e$, 其中 E' 是一个新的非终结符。可以证明这样替换不改变文法生成的语言。对 G 的

另一个左递归规则 $T \rightarrow T * F$ 也要运用相同的方法。于是,得到文法 $G' = (V', \Sigma, R', E)$, 其中 $V' = \Sigma \cup \{E, E', T, T', F, A\}$, 规则如下:

- (1) $E \rightarrow TE'$
- (2) $E' \rightarrow + TE'$
- (3) $E' \rightarrow e$
- (4) $T \rightarrow FT'$
- (5) $T' \rightarrow * FT'$
- (6) $T' \rightarrow e$
- (7) $F \rightarrow (E)$
- (8) $F \rightarrow \text{id}A$
- (9) $A \rightarrow e$
- (10) $A \rightarrow (E)$

上面消除上下文无关文法中左递归的技术可以表达如下^①。

启发式规则 2 设 $A \rightarrow A\alpha_1, \dots, A \rightarrow A\alpha_n$ 和 $A \rightarrow \beta_1, \dots, A \rightarrow \beta_m$ 是左边为 A 的所有规则, 其中 β_i 不以 A 开始, $n, m > 0$ (即, 至少有一个左递归规则)。那么, 把这些规则替换成 $A \rightarrow \beta_1 A', \dots, A \rightarrow \beta_m A', A \rightarrow \alpha_1 A', \dots, A \rightarrow \alpha_n A'$ 和 $A' \rightarrow e$, 其中 A' 是一个新的非终结符。

例中文法 G' 仍有左边相同的规则, 只是现在所有的不确定性能够用提前看下一个输入符号解决。于是, 可以构造出下述接受 $L(G) \$$ 的确定型下推自动机。

$$M_4 = (K, \Sigma \cup \{\$, \}, V', \Delta, p, \{q_\$ \})$$

其中

$$K = \{p, q, q_\$, q_+, q_*, q_(), q_\$ \}$$

以及 Δ 列表如下:

$$\begin{aligned} & ((p, e, e), (q, E)) \\ & ((q, a, e), (q_a, e)), \text{对每一个 } a \in \Sigma \cup \{\$ \} \\ & ((q_a, e, a), (q, e)), \text{对每一个 } a \in \Sigma \\ & ((q_a, e, E), (q_a, TE')), \text{对每一个 } a \in \Sigma \cup \{\$ \} \\ & ((q_+, e, E'), (q_+, + TE')) \\ & ((q_a, e, E'), (q_a, e)), \text{对每一个 } a \in \{(), \$ \} \\ & ((q_a, e, T), (q_a, FT')), \text{对每一个 } a \in \Sigma \cup \{\$ \} \\ & ((q_*, e, T'), (q_*, * FT')) \\ & ((q_a, e, T'), (q_a, e)), \text{对每一个 } a \in \{+, (), \$ \} \\ & ((q_(), e, F), (q_(), (E))) \\ & ((q_\$, e, F), (q_\$, \text{id}A)) \\ & ((q_(), e, A), (q_(), (E))) \end{aligned}$$

^① 这里假设没有 $A \rightarrow A$ 形式的规则。

$((q_a, e, A), (q_a, e))$, 对每一个 $a \in \{+, *,), \$\}$

M_1 是 G' 的一个语法分析器。例如, 接受输入串 $\text{id} * (\text{id}) \$$ 如表 3-5 所示。

表 3-5

步骤	状态	未读的输入	栈	G' 的规则
0	p	$\text{id} * (\text{id}) \$$	e	
1	q	$\text{id} * (\text{id}) \$$	E	
2	q_{id}	$* (\text{id}) \$$	E	
3	q_{id}	$* (\text{id}) \$$	TE'	1
4	q_{id}	$* (\text{id}) \$$	$FT'E'$	4
5	q_{id}	$* (\text{id}) \$$	$\text{id} AT'E'$	8
6	q	$* (\text{id}) \$$	$AT'E'$	
7	$q_.$	$(\text{id}) \$$	$AT'E'$	
8	$q_.$	$(\text{id}) \$$	$T'E'$	9
9	$q_.$	$(\text{id}) \$$	$* FT'E'$	5
10	q	$(\text{id}) \$$	$FT'E'$	
11	$q_(($	$\text{id}) \$$	$FT'E'$	
12	$q_(($	$\text{id}) \$$	$(E)T'E'$	7
13	q	$\text{id}) \$$	$E)T'E'$	
14	q_{id}	$) \$$	$E)T'E'$	
15	q_{id}	$) \$$	$TE')T'E'$	1
16	q_{id}	$) \$$	$FT'E')T'E'$	4
17	q_{id}	$) \$$	$\text{id}AT'E')T'E'$	8
18	q	$) \$$	$AT'E')T'E'$	
19	$q_)$	$\$$	$AT'E')T'E'$	
20	$q_)$	$\$$	$T'E')T'E'$	10
21	$q_)$	$\$$	$E')T'E'$	6
22	$q_)$	$\$$	$)T'E'$	3
23	q	$\$$	$T'E'$	6
24	$q\$$	e	$T'E'$	
25	$q\$$	e	E'	6
26	$q\$$	e	e	3

在表 3-5 中标示出按照 G' 的一条规则替换栈中非终结符的计算步骤。顺序运用表的最后一列中 G' 的这些规则, 得到这个输入串的最左推导:

$$\begin{aligned}
 E &\Rightarrow TE' \Rightarrow FT'E' \Rightarrow \text{id}T'E' \Rightarrow \text{id} * FT'E' \Rightarrow \text{id} * (E)T'E' \\
 &\Rightarrow \text{id} * (TE')T'E' \Rightarrow \text{id} * (FT'E')T'E' \Rightarrow \text{id} * (\text{id}T'E')T'E' \\
 &\Rightarrow \text{id} * (\text{id}E')T'E' \Rightarrow \text{id} * (\text{id})T'E' \Rightarrow \text{id} * (\text{id})E' \Rightarrow \text{id} * (\text{id})
 \end{aligned}$$

事实上, 可以重新构造这个输入的语法分析树(见图 3-12, 在每个顶点的旁边还表明对应

这个顶点展开的下推自动机步骤)。注意,这个语法分析器以自顶向下、从左到右的方式,从 E 开始,反复对最左边的非终结符运用合适的规则构造这个输入的语法分析树。

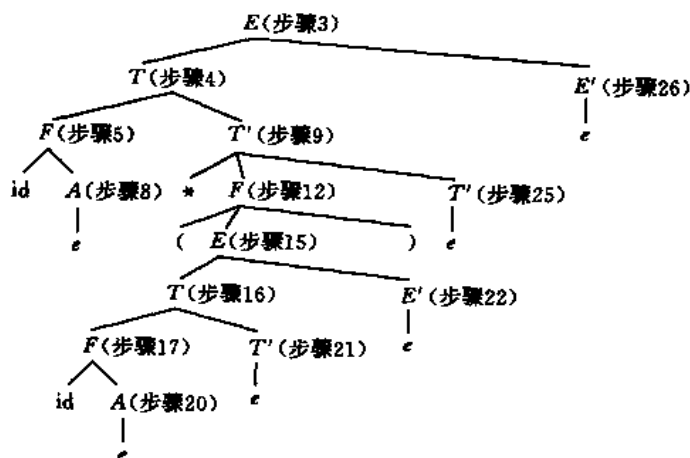


图 3-12

一般地,给定文法 G ,可以试图构造 G 的自顶向下的语法分析器如下:对 G 的所有左递归的非终结符 A 运用启发式规则 2,消去 G 中的左递归。只要需要,就运用启发式规则 1 左因子分解 G 。然后检查得到的文法是否有下述性质:能够用看下一个输入符号的办法在具有相同的左边的规则之间作出正确的选择。具有这种性质的文法叫做 $LL(1)$ 。虽然我们并没有精确地说明如何确定一个文法是否是 $LL(1)$,也没有精确说明当它是 $LL(1)$ 时如何构造对应的确定型语法分析器,但是有系统的方法做这些事情。总之,检查文法和做一些试验常常就行了。

3.7.3 自底向上的语法分析

不存在分析上下文无关语言的最好方法,对于不同的文法常常要采用不同的方法。本章最后简单地考虑一个与自顶向下的语法分析很不同的方法。然而同样可以在下推自动机的构造方法中找到它们的来源。

除引理 3.4.1 中的构造方法外,有一种完全相反的方法构造下推自动机接受给定上下文无关文法生成的语言。前一种方法构造的自动机(从它们得到上一小节研究的自顶向下的语法分析器)运行时在栈中实现最左推导,当生成终结符时把它们与输入串作比较。在下面给出的构造方法中,自动机试图先读整个输入,并且根据实际读到的输入推演它应实现的推导。我们将会看到,一般的结果不是从根到树叶,而是反过来从树叶到根重新构造一棵语法分析树,因此这类方法叫做自底向上的。

自底向上的下推自动机构造如下。设 $G = (V, \Sigma, R, S)$ 是任一上下文无关文法,则令 $M = (K, \Sigma, \Gamma, \Delta, p, F)$,其中 $K = \{p, q\}$, $\Gamma = V$, $F = \{q\}$,而 Δ 包含下述转移:

- (1) 对每一个 $a \in \Sigma, ((p, a, e), (p, a))$,
- (2) 对 R 中的每一条规则 $A \rightarrow \alpha, ((p, e, \alpha^R), (p, A))$,
- (3) $((p, e, S), (q, e))$,

在着手证明 M 与 G 的等价性之前,先比较这些转移与引理 3.4.1 证明中构造的自动机的转移。这里类型 1 的转移把输入符号移入栈,而引理 3.4.1 中类型 3 的转移当栈顶的终结符与输入符号一致时把它们托出栈。这里类型 2 的转移在栈中把一条规则的右边替换成对应的左边,规则的右边在栈中是反转的,而引理 3.4.1 中类型 2 的转移在栈中把一条规则的左边替换成对应的右边。这里类型 3 的转移当栈中只剩一个起始符时进入终结状态结束计算,而引理 3.4.1 中类型 1 的转移把起始符放入初始的空栈中开始计算。因此,在某种意义上这样构造的机器与引理 3.4.1 中的正好相反。

引理 3.7.1 设 G 和 M 如上所述,则 $L(M) = L(G)$ 。

证: $L(G)$ 中的任何字符串有从起始符开始的最右推导,因此只需证明下述断言。

断言 对于任意的 $x \in \Sigma^*$ 和 $\gamma \in \Gamma^*$,

$$(p, x, \gamma) \vdash_M^* (p, e, S) \text{ 当且仅当 } S \xRightarrow{R}_G \gamma^R x$$

如果令 x 是 M 的一个输入和 $\gamma = e$,则由于 q 是唯一的终结状态且只有用转移 3 才能进入,故断言蕴涵 M 接受 x 当且仅当 G 生成 x 。断言的“仅当”方向可以用对 M 的计算步数作归纳来证明,而“当”方向可以用对从 S 开始的 x 的最右推导的步数作归纳来证明。

■

让我们再一次考虑关于算术表达式的文法(例 3.1.3,不包括前一小节的规则 $F \rightarrow \text{id}(E)$)。文法的规则如下:

$$E \rightarrow E + T \quad (R1)$$

$$E \rightarrow T \quad (R2)$$

$$T \rightarrow T * F \quad (R3)$$

$$T \rightarrow F \quad (R4)$$

$$F \rightarrow (E) \quad (R5)$$

$$F \rightarrow \text{id} \quad (R6)$$

对这个文法运用新的构造方法,得到下述转移:

$$((p, a, e), (p, a)), \text{对每一个 } a \in \Sigma \quad (\Delta 0)$$

$$((p, e, T + E), (p, E)) \quad (\Delta 1)$$

$$((p, e, T), (p, E)) \quad (\Delta 2)$$

$$((p, e, F * T), (p, T)) \quad (\Delta 3)$$

$$((p, e, F), (p, T)) \quad (\Delta 4)$$

$$((p, e,)E(), (p, F)) \quad (\Delta 5)$$

$$((p, e, \text{id}), (p, F)) \quad (\Delta 6)$$

$$((p, e, E), (q, e)) \quad (\Delta 7)$$

把这台下推自动机叫作 M 。 M 接受输入 $\text{id} * (\text{id})$ 如表 3-6 所示。

M 不是确定型的: $\Delta 0$ 中的转移与 $\Delta 1$ 到 $\Delta 7$ 的所有转移都是兼容的。但是它的总体操作“哲学”还是有启发的。在任一时刻, M 可以把一个终结符从它的输入中移动到栈顶($\Delta 0$ 中的转移,在表 3-6 的步骤 1、4、5、6 和 10 中使用)。另一方面,它可以不时地认出栈顶的几个符号正好是 G 的一条规则的右边(的反转),并且把这几个符号化简成对应的左边的符

表 3-6

步骤	状态	未读的输入	栈	使用的转移	G 的规则
0	p	id * (id)	ϵ		
1	p	* (id)	id	$\Delta 0$	
2	p	* (id)	F	$\Delta 6$	$R6$
3	p	* (id)	T	$\Delta 4$	$R4$
4	p	(id)	* T	$\Delta 0$	
5	p	id)	(* T	$\Delta 0$	
6	p)	id (* T	$\Delta 0$	
7	p)	F (* T	$\Delta 6$	$R6$
8	p)	T (* T	$\Delta 4$	$R4$
9	p)	E (* T	$\Delta 2$	$R2$
10	p	ϵ) E (* T	$\Delta 0$	
11	p	ϵ	F * T	$\Delta 5$	$R5$
12	p	ϵ	T	$\Delta 3$	$R3$
13	p	ϵ	E	$\Delta 2$	$R2$
14	q	ϵ	ϵ	$\Delta 7$	

号($\Delta 1$ 到 $\Delta 6$ 的转移,表 3-6 中最右一列标有“G 的规则”的计算中使用)。对应这些化简步骤的规则序列原来恰好以相反的次序反映输入串的最右推导。在这个例子中,隐含的最右推导如下:

$$\begin{aligned}
 E &\Rightarrow T \\
 &\Rightarrow T * F \\
 &\Rightarrow T * (E) \\
 &\Rightarrow T * (T) \\
 &\Rightarrow T * (F) \\
 &\Rightarrow T * (\text{id}) \\
 &\Rightarrow F * (\text{id}) \\
 &\Rightarrow \text{id} * (\text{id})
 \end{aligned}$$

把表 3-6 最右边一列中列出的规则自下而上地运用于最右边的非终结符,可以从计算中得到这个推导。这个过程可以等价地看作自底向上、从右向左的构造一棵语法分析树(即,恰好与图 3.12(b) 相反)。

要想构造 $L(G)$ 的一个实际有用的语法分析器,必须把 M 转变成一台接受 $L(G)$ 的确定型设备。和处理自顶向下的语法分析器时一样,这里也不给出系统的处理过程。而是把这个 G 做完,通过这个例子指出指导这种构造方法的基本的启发式规则。

首先,需要有办法在两种基本动作之间作出选择。这两个基本动作是把下一个输入符号移动到栈中和按照文法的一条规则把栈顶的几个符号化简成一个非终结符。一种可能的办法是看两个信息:下一个输入符号(记作 b)和栈顶符号(记作 a),符号 a 可能是非终结符。通过一个关系 $P \subseteq V \times (\Sigma \cup \{\$ \})$ 选择移动或化简。关系 P 叫作优先关系。如果 $(a, b) \in P$ 则化简,否则移动 b 。表 3-7 给出文法 G 的优先关系。直观上, $(a, b) \in P$ 意味着

存在下述形式的最右推导

$$S \xRightarrow{R} \beta A b x \xRightarrow{R} \beta \gamma a b x$$

表 3-7

	()	id	+	*	\$
(
)		✓		✓	✓	✓
id		✓		✓	✓	✓
+						
*						
E						
T		✓		✓		✓
F		✓		✓	✓	✓

因为在倒着重新构造最右推导,所以当发现 a 正好在 b 的前面时脱去规则 $A \rightarrow \gamma a$ 是有意义的。有系统的方法计算优先关系以及发现优先关系是否足以在移动与化简之间作出决定。然而,在许多情况下检查和经验就能解决问题。

我们还必须面对非确定性的另一种来源:当决定化简时,怎么选择栈字符串的一个前缀把它替换成一个非终结符?例如,如果栈的内容是字符串 $F * T + E$ 并且应该化简,则可以选择把 F 化简成 T (规则 R_4) 或把 $F * T$ 化简成 T (规则 3)。对于文法 G ,正确的作法是永远选择栈字符串的与一条规则的右边的反转一致的最长的前缀,并且把它化简成这条规则左边的非终结符。于是,在上述情况下应该取第二种可能,并且把 $F * T$ 化简成 T 。

用这两条规则(当栈顶符号与下一个输入符号由 P 关联时化简,否则移动;当化简时,化简栈顶部尽可能长的字符串),下推自动机 M 的运行完全变成确定型的。事实上,能够设计一台确定型下推自动机“执行”这两条规则(见习题 3.7.9)。

和前面一样,也要记住这两条启发式规则——(1)用优先关系决定移动还是化简,(2)当有多种可能的化简时,化简尽可能长的字符串——不是在所有情况下都是行得通的。这两条规则能行得通的文法叫做弱优先文法。在实践中,许多与程序设计语言有关的文法是弱优先文法,或者能够容易地转变成弱优先文法。有许多构造自底向上的语法分析器的更复杂的方法,它们能适用于更大的文法类。

习 题

3.7.1 证明下述语言是确定型上下文无关的:

- (a) $\{a^m b^n, m \neq n\}$
- (b) $\{w c w^R, w \in \{a, b\}^*\}$
- (c) $\{c a^m b^n, m \geq 0\} \cup \{d a^m b^{2m}, m \geq 0\}$
- (d) $\{a^m c b^n, m \geq 0\} \cup \{a^m d b^{2m}, m \geq 0\}$

3.7.2 证明:确定型上下文无关语言类在同态映射下不是封闭的。

- 3.7.3 证明:如果 L 是确定型上下文无关语言,则 L 不是固有歧义的。
- 3.7.4 证明:假设下推自动机 M 接受 $L\$$,则在 3.7.1 小节中构造的 M' 接受语言 L 。
- 3.7.5 考虑下述上下文无关文法 $G = (V, \Sigma, R, S)$,其中 $V = \{ (,), \cdot, S, A \}$, $\Sigma = \{ (,), \cdot, \}$ 及 $R = \{ S \rightarrow (, S \rightarrow a, S \rightarrow (A), A \rightarrow S, A \rightarrow A.S \}$ 。(对于熟悉程序设计语言 LISP 的读者, $L(G)$ 包含所有的原子和表,这里符号 a 代表任一非空原子。)
- (a) 将启发式规则 1 和 2 应用于 G 。设 G' 是得到的文法。证明 G' 是 $LL(1)$ 的。构造一台确定型下推自动机 M 接受 $L(G)\$$ 。研究 M 关于字符串 $((()) \cdot a \cdot)$ 的计算。
- (b) 将 R 的第一条规则替换成 $A \rightarrow e$,重复(a)。
- (c) 将 R 的最后一条规则替换成 $A \rightarrow S.A$,重复(a)。
- 3.7.6 再次考虑习题 3.7.5 中的文法 G 。证明 G 是一个弱优先文法,其优先关系如表 3-8 所示。试构造一台接受 $L(G)\$$ 的确定型下推自动机。

表 3-8

	a	()	\cdot	$\$$
a		✓		✓	✓
(
)		✓		✓	✓
\cdot					
A					
S		✓		✓	

- 3.7.7 设文法 $G' = (V, \Sigma, R', S)$ 的规则为 $S \rightarrow (A), S \rightarrow a, A \rightarrow S.A, A \rightarrow e$ 。 G' 是弱优先的吗?如果是,请给一个适当的优先关系;如果不是,请说明为什么不是。
- 3.7.8 在习题 3.3.3 中定义了以终结方式接受。证明: L 是确定型上下文无关的当且仅当它被一台确定型下推自动机以终结方式接受。
- 3.7.9 根据 3.7.3 小节中的非确定型下推自动机 M 和优先关系 P ,给出接受算术表达式语言的确定型下推自动机。与 3.7.2 小节中的下推自动机 M_4 非常相似,这台自动机必须提前看下一个输入符号。
- 3.7.10 考虑下述语言类:
- 正则语言,
 - 上下文无关语言,
 - 上下文无关语言的补,
 - 确定型上下文无关语言。

试画出这些语言类的文氏图,即把每一个语言类表示成一个圆,准确地反映出它们之间的包含、相交等关系。你能给图中每一个非空区域提供一个语言吗?

参 考 文 献

上下文无关文法是 Noam Chomsky 提出的。见：

- N. Chomsky. Three models for the description of languages. *IEEE Transactions on Information Theory*, 2(3), pp. 113—124, 1956.
- N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2), pp. 137—167, 1959.

在第二篇文章中又引入了 Chomsky 范式。在 20 世纪 50 年代后期还发明了一个与程序设计语言的语法紧密相关的概念 BNF (Backus 范式或 Backus-Naur 形式), 见：

- P. Naur (ed.). Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1), pp. 1—17, 1963. 重新发表于 S. Rosen (ed.). *Programming Systems and Languages*. New York, McGraw-Hill, pp. 79—118, 1967.

习题 3.1.9 关于正则文法与有穷自动机的等价性取自：

- N. Chomsky, G. A. Miller. Finite-state languages. *Information and Control*, 1(2), pp. 91—112, 1958.

下文引入了下推自动机：

- A. G. Oettinger. Automatic Syntactic analysis and the pushdown store. *Proceedings of Symposia on Applied Mathematics*, Vol. 12, Providence, R. I., American Mathematical Society, 1961.

Schutzenberger, Chomsky 和 Evey 三人独立地证明了上下文无关文法与下推自动机的等价性 (定理 3.4.1)：

- M. P. Schutzenberger. On context-free languages and pushdown automata. *Information and Control*, 6(3), pp. 246—264, 1963.
- N. Chomsky. Context-free grammar and pushdown storage. *Quarterly Progress Report*, 65, pp. 187—194, M. I. T. Research Laboratory in Electronics, Cambridge, Mass., 1962.
- J. Evey. Application of pushdown store machines. *Proceedings of the 1963 Fall Joint Computer Conference*, pp. 215—217, Montreal, AFIPS Press, 1963.

下文指出 3.5.1 小节中给出的封闭性质以及许多其他性质：

- V. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14, pp. 143—172, 1961.

上文发现定理 3.5.3 的一个加强形式 (关于上下文无关文法的泵定理, 也见习题 3.5.7)。下文给出这个定理的更强的形式：

- W. G. Ogden. A helpful result for proving inherent ambiguity. *Mathematical Systems Theory*, 2, pp. 191—194, 1968.

下面两篇文章独立地讨论了关于上下文无关语言识别的动态规划算法：

- T. Kasami. An efficient recognition and syntax algorithm for context-free languages. Report AFCRL-65-758 (1965), Air Force Cambridge Research Laboratory, Cambridge, Mass.
- D. H. Younger. Recognition and parsing of context-free languages in time n^3 . *Information and Control*, 10(2), pp. 189—208, 1967.

当基本文法非歧义时, 这个算法的一种变形更快：

- J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13, pp. 94—

102, 1970.

最有效的通用上下文无关识别算法被认为属于 Valiant. 它的运行时间正比于两个 $n \times n$ 矩阵相乘所需要的时间, 目前这个时间是 $O(n^{2.376})$ 。

- L. G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and Systems Sciences*, 10(2), pp. 308—315, 1975.

下面两篇引入 LL(1) 语法分析器:

- P. M. Lewis II, R. E. Stearns. Syntax-directed transduction. *Journal of the ACM*, 15(3), pp. 465—488, 1968.
- D. E. Knuth. Top-down syntax analysis. *Acta Informatica*, 1(2), pp. 79—110, 1971.

下面是关于编译程序的权威著作:

- A. V. Aho, R. Sethi, J. D. Ullman. *Principles of Compiler Design*. Reading, Mass.: Addison-Wesley, 1985.

下面两篇文章分别首次研究歧义性和固有歧义性:

- N. Chomsky and M. P. Schutzenberger. The algebraic theory of context-free languages. In: *Computer Programming and Formal Systems* (pp. 118—161), P. Braffort, D. Hirschberg (ed.). Amsterdam North Holland, 1963.
- S. Ginsburg and J. S. Ullian. Preservation of unambiguity and inherent ambiguity in context-free languages. *Journal of the ACM*, 13(1), pp. 62—88, 1966.

Greibach 范式(习题 3.5.10)出自:

- S. Greibach. A new normal form theorem for context-free phrase structure grammars. *Journal of the ACM*, 12(1), pp. 42—52, 1965.

下面是两本关于上下文无关语言的高级著作:

- S. Ginsburg. *The Mathematical Theory of Context-free Languages*. New York: McGraw-Hill, 1966.
- M. A. Harrison. *Introduction to Formal Language Theory*. Reading Mass: Addison-Wesley, 1978.

弱优先语法分析器出自:

- J. D. Ichbiah and S. P. Morse. A technique for generating almost optimal Floyd-Evans productions for precedence grammars. *Communications of the ACM*, 13(8), pp. 501—508, 1970.

第 4 章 Turing 机

4.1 Turing 机的定义

我们在前两章看到,不能认为有穷自动机和下推自动机是真正一般的计算机模型,因为它们不能识别甚至像 $\{a^n b^n c^n : n \geq 0\}$ 这样简单的语言。在本章我们继续研究能识别这个语言和许多更复杂语言的装置。这些装置以它们的发明者 Alan Turing(1912—1954) 的名字命名为 **Turing 机**,虽然它们比前面研究过的自动机更一般,但是它们的基本外观与那些自动机相似。Turing 机包括有穷控制器、带和带头等,带头用来在带上读或写。Turing 机和它的操作的形式化定义在数学风格上与有穷自动机和下推自动机所用过的相同。因此为获得 Turing 机所具有的附加计算能力和功能一般性,我们不需要转向完全新型的计算机模型。

尽管这样,Turing 机也不仅仅是会被其他更强类型所取代的又一类自动机。我们在本章将看到 Turing 机虽然简陋,但是强化它们的尝试都没有获得任何效果。例如我们也研究有多条带的 Turing 机,或者有奇特存储装置的机器,这些存储装置用类似于真实计算机的随机存取方式来读或写;颇有意义的是,就计算能力而言这些装置不比基本 Turing 机更强。我们用模拟方法证明这种结果:任何“强化”机器都可转换成有类似功能的标准 Turing 机。因此在奇特型机器上完成的任何计算实际上在标准型 Turing 机上也可完成。另外在本章快结束时,我们定义对上下文无关文法做大胆推广的某种语言产生器并且证明它等价于 Turing 机。从完全不同的角度出发,我们还研究何时认为数值函数(比如 $2^x + x^2$) 是可计算的这样的问题,最终得出了再次等价于 Turing 机的概念!

因此,就可完成的计算而言,Turing 机似乎构成稳定的和极大的计算装置类。事实上在下一章我们将提出公认的观点:把“算法”的想法形式化的任何合理方式最终必然等价于 Turing 机的想法。

但是现在我们不能扯得太远。在介绍过程中要记住的要点是 Turing 机被设计成同时满足以下三条标准:

- (a) 它们应当是自动机,即它们的构造和功能在一般特点上应当与前面研究过的装置相同。
- (b) 它们应当尽量简单,这是对描述、形式化定义和讨论来说的。
- (c) 它们应当尽量一般,这是就可完成的计算而言。

现在让我们更仔细地观察这些机器。本质上 Turing 机包括有穷状态控制器和带(如图4-1所示)。两者之间的通讯由一只带头提供,它从带上读符号并在带上改变符

号。控制器一步一步地操作,根据当前状态和读写头当前扫描的带符号,控制器在每步完成两种功能:

- (1) 让控制器进入新状态。
- (2) (a) 在当前扫描的带方格里写一个符号替换那里的符号;或者,
- (b) 把读写头向左或向右移动一个带方格。

带虽然有左端,但它向右无限延伸。为防止机器把带头移出带左端,我们假定带的最左端总是用表示成 \triangleright 的特殊符号做标记;还假定所有 Turing 机都设计成每当带头读到 \triangleright 时它就立即向右移动。另外我们用不同的符号 \leftarrow 和 \rightarrow 表示带头向左或向右移动;我们假定这两个符号不属于我们考虑的任何字母表。

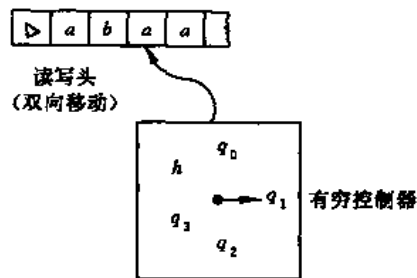


图 4-1

向 Turing 机提供输入的方法是把输入字符串写在带左端的带方格里,与 \triangleright 符号的右方相邻。其余的带方格起初都包含表示成 \sqcup 的空格符。机器被允许以它认为合适的任何方式改变输入并且在带右方无限的空格里写。因为机器一次只把带头移动一格,所以在任何有穷的计算之后只有有穷多个带方格被访问过。

现在我们给出 Turing 机的形式化定义。

定义 4.1.1 Turing 机是五元组 $(K, \Sigma, \delta, s, H)$, 其中

K 是状态的有穷集;

Σ 是字母表,包含空格符 \sqcup 和左端符 \triangleright ,但不包含符号 \leftarrow 和 \rightarrow ;

$s \in K$ 是初始状态;

$H \subseteq K$ 是停机状态的集合;

δ 是转移函数,它是从 $(K - H) \times \Sigma$ 到 $K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$ 的函数,使得

(a) 对所有 $q \in K - H$,若 $\delta(q, \triangleright) = (p, b)$ 则 $b = \rightarrow$,

(b) 对所有 $q \in K - H$ 和 $a \in \Sigma$,若 $\delta(q, a) = (p, b)$ 则 $b \neq \triangleright$ 。

如果 $q \in K - H, a \in \Sigma$, 并且 $\delta(q, a) = (p, b)$, 那么当 M 在状态 q 扫描符号 a 时它进入状态 p , 并且 (1) 若 b 是属于 Σ 的符号则 M 把当前扫描的符号 a 改写成 b , 或者 (2) 若 b 是 \leftarrow 或 \rightarrow 则 M 按照方向 b 移动带头。因为 δ 是函数,所以 M 的操作是确定性的,并且只有当 M 进入停机状态时才会停止。注意关于 δ 的要求 (a); 当 M 看到带左端 \triangleright 时,它必须向右移动。通过这种方式,最左边的 \triangleright 永不被删除, M 永不移出带的左端。根据 (b), M 永不写出 \triangleright , 因此 \triangleright 是清楚无误的带左端标志。换句话说,我们认为 \triangleright 只是防止 M 的带头不小心移出左端的“保护性屏障”,它不用任何其他方式影响 M 的计算。另外注意 δ 在 H 里的状态上无定义,当机器到达停机状态时操作停止。

例 4.1.1 考虑 Turing 机 $M = (K, \Sigma, \delta, s, \{h\})$, 其中

$$K = \{q_0, q_1, h\},$$

$$\Sigma = \{a, \sqcup, \triangleright\},$$

$$s = q_0,$$

并且 δ 由表 4-1 给定。

当 M 由初始状态 q_0 启动时,带头向右扫描,一边移动一边把所有 a 改成 \sqcup ,直到发现包含 \sqcup 的带方格为止;然后停机。(把非空格符改成空格符,称为删除非空格符。)具体地说,假设 M 启动时带头扫描四个 a 中的第一个,在最后一个 a 后面跟着 \sqcup 。于是 M 在状态 q_0 和 q_1 之间来回变化四次,交替地把 a 改成 \sqcup 并且向右移动带头;在这个移动序列里与 δ 有关的是 δ 表的第一和第五行。到这个时候 M 发现自己在状态 q_0 扫描 \sqcup ,因此根据 δ 表的第二行 M 停机。注意表的第四行 $\delta(q_1, a)$ 的值无关紧要,因为若 M 在状态 q_0 启动则 M 永不在状态 q_1 扫描 a 。尽管这样,还是必须有某个值与 $\delta(q_1, a)$ 关联,因为要求 δ 是有定义域 $(K - H) \times \Sigma$ 的函数。◇

表 4-1

q	σ	$\delta(q, \sigma)$
q_0	a	(q_1, \sqcup)
q_0	\sqcup	(h, \sqcup)
q_0	\triangleright	(q_0, \rightarrow)
q_1	a	(q_0, a)
q_1	\sqcup	(q_0, \rightarrow)
q_1	\triangleleft	(q_1, \rightarrow)

例 4.1.2 考虑 Turing 机 $M = (K, \Sigma, \delta, s, H)$, 其中

$$\begin{aligned} K &= \{q_0, h\}, \\ \Sigma &= \{a, \sqcup, \triangleright\}, \\ s &= q_0, \\ H &= \{h\}, \end{aligned}$$

并且 δ 由表 4-2 给定。

表 4-2

q	σ	$\delta(q, \sigma)$
q_0	a	(q_0, \leftarrow)
q_0	\sqcup	(h, \sqcup)
q_0	\triangleright	(q_0, \rightarrow)

这台机器向左扫描直到发现 \sqcup 为止,然后停机。如果从带头位置向左到带左端的每个带方格都包含 a ,当然带左端包含 \triangleright ,那么 M 将移动到带左端,并且从此就在带左端和它右方的方格之间无限地来回移动。与我们遇到的其他确定型装置不一样, Turing 机的操作可以永不停止。◇

现在我们形式化 Turing 机的操作。

为了说明 Turing 机的计算状态,我们需要说明机器状态、带内容和带头位置。因为除有穷的开始部分外所有带内容都是空格,所以带内容可用有穷字符串来说明。我们选择把这个字符串分成两段:被扫描方格左侧的部分,包括被扫描方格里的符号;被扫描方格右侧的部分,这部分可能为空。另外为了避免两对不同字符串与带头位置和带内容的相同组合相对应,我们坚持让第二个字符串不用空格结尾(假定在最后一个被明确表示的带方格右方的所有带方格都包含空格)。这些考虑使我们得出下列定义。

定义 4.1.2 Turing 机 $M = (K, \Sigma, \delta, s, H)$ 的格局是 $K \times \triangleright \Sigma^* \times (\Sigma^* (\Sigma - \{\sqcup\}) \cup \{e\})$ 的成员。

即假设,除非当前正在扫描空格符,否则所有格局都用左端符开始并且永远不用空格符结束。因此 $(q, \triangleright a, aba)$, $(h, \triangleright \sqcup \sqcup \sqcup, \sqcup a)$ 和 $(q, \triangleright \sqcup a \sqcup \sqcup, e)$ 是格局(如图 4-2 所示),但 $(q, \triangleright baa, abc \sqcup)$ 和 $(q, \sqcup aa, ba)$ 不是格局。状态分量属于 H 的格局称为停机格局。

我们用简化记号描述带内容(包括带头位置);我们写 $w \underline{au}$ 表示格局 (q, wa, u) 里的带内容,带下划线的符号指示带头位置。对于图 4-2 里说明的三种格局,带内容表示成 $\triangleright \underline{a}aba$, $\triangleright \sqcup \sqcup \sqcup \underline{\sqcup} a$ 和 $\triangleright \sqcup a \underline{\sqcup \sqcup}$ 。另外通过把状态包含在带和带头位置的记号里我们可写出格局,即把 (q, wa, u) 写成 $(q, w \underline{au})$ 。利用这个约定我们把图 4-2 里表示的三种格局写成

$(q, \triangleright \underline{a}aba), (h, \triangleright \square\square\square\square a)$ 和 $(q, \triangleright \square a\square\square)$ 。

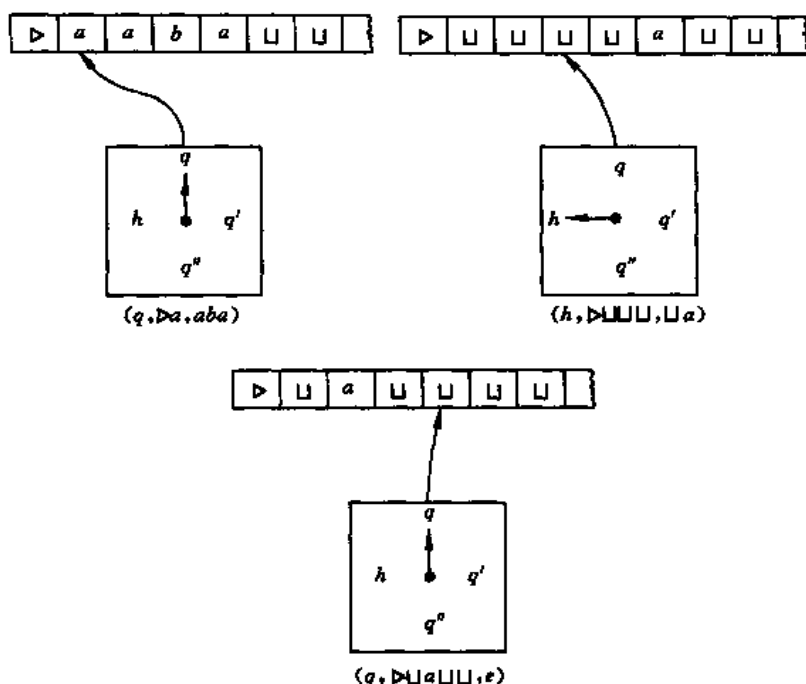


图 4-2

定义 4.1.3 设 $M = (K, \Sigma, \delta, s, H)$ 是 Turing 机, 考虑 M 的两个格局 $(q_1, \triangleright w_1 \underline{a_1} u_1)$ 和 $(q_2, \triangleright w_2 \underline{a_2} u_2)$, 其中 $a_1, a_2 \in \Sigma$, 那么

$$(q_1, \triangleright w_1 \underline{a_1} u_1) \vdash_M (q_2, \triangleright w_2 \underline{a_2} u_2)$$

当且仅当对某个 $b \in \Sigma \cup \{\leftarrow, \rightarrow\}$, $\delta(q_1, a_1) = (q_2, b)$,

1. 当 $b \in \Sigma$ 时, $w_1 = w_2, u_1 = u_2$, 并且 $a_2 = b$;
2. 当 $b = \leftarrow$ 时, $w_1 = w_2 a_2$, 而且
 - (a) 若 $a_1 \neq \square$ 或 $u_1 \neq e$ 则 $u_2 = a_1 u_1$,
 - (b) 若 $a_1 = \square$ 且 $u_1 = e$ 则 $u_2 = e$;
3. 当 $b = \rightarrow$ 时, $w_2 = w_1 a_1$, 而且
 - (a) $u_1 = a_2 u_2$, 或者
 - (b) $u_1 = u_2 = e$, 并且 $a_2 = \square$ 。

在情形 1 里 M 改写符号但不移动带头。在情形 2 里 M 把带头向左移动一格; 若正在从空格带向左移动则刚刚扫描的方格里的空格符从格局里消失。在情形 3 里 M 把带头向右移动一格; 若正在向右移动到空格带上, 则新的空格符出现在新的格局里作为新的被扫描符号。注意除停机格局外所有格局都恰恰产生一个格局。

例 4.1.3 为了说明这些情形, 设 $w, u \in \Sigma^*$, 其中 u 不用 \square 结尾, 并设 $a, b \in \Sigma$ 。

情形 1. $\delta(q_1, a) = (q_2, b)$

例子: $(q_1, w \underline{a} u) \vdash_M (q_2, w \underline{b} u)$

情形 2. $\delta(q_1, a) = (q_2, \leftarrow)$

(a) 的例子: $(q_1, wba\underline{u}) \vdash_M (q_2, wba\underline{u})$

(b) 的例子: $(q_1, wba\underline{\sqcup}) \vdash_M (q_2, wba\underline{u})$

情形 3. $\delta(q_1, a) = (q_2, \rightarrow)$

(a) 的例子: $(q_1, w\underline{a}bu) \vdash_M (q_2, w\underline{a}bu)$

(b) 的例子: $(q_1, w\underline{a}) \vdash_M (q_2, w\underline{a}\sqcup)$ ◇

定义 4.1.4 对任意 Turing 机 M , 设 \vdash_M^* 是 \vdash_M 的自反传递闭包, 若格局 $C_1 \vdash_M^* C_2$ 则称格局 C_1 产生格局 C_2 . M 的计算是格局序列 C_0, C_1, \dots, C_n , 其中 $n \geq 0$,

$$C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \dots \vdash_M C_n$$

称计算的长度为 n 或它有 n 步, 写成 $C_0 \vdash_M^n C_n$.

例 4.1.4 考虑例 4.1.1 里描述的 Turing 机 M . 若 M 在格局 $(q_1, \triangleright \sqcup aaaa)$ 启动, 则它的计算可形式化表示如下:

$$\begin{aligned} (q_1, \triangleright \sqcup aaaa) & \vdash_M (q_0, \triangleright \sqcup \underline{a}aaa) \\ & \vdash_M (q_1, \triangleright \sqcup \sqcup \underline{a}aa) \\ & \vdash_M (q_0, \triangleright \sqcup \sqcup \sqcup \underline{a}aa) \\ & \vdash_M (q_1, \triangleright \sqcup \sqcup \sqcup \sqcup \underline{a}a) \\ & \vdash_M (q_0, \triangleright \sqcup \sqcup \sqcup \sqcup \sqcup \underline{a}a) \\ & \vdash_M (q_1, \triangleright \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \underline{a}) \\ & \vdash_M (q_0, \triangleright \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \underline{a}) \\ & \vdash_M (q_1, \triangleright \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \underline{a}) \\ & \vdash_M (q_0, \triangleright \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \underline{a}) \\ & \vdash_M (h, \triangleright \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \underline{a}) \end{aligned}$$

这个计算有 10 步. ◇

4.1.1 Turing 机的记号

我们迄今为止见过的 Turing 机都是极其简单的——至少当与在本章宣布的目标相比时——然而它们的表格形式已经相当复杂和难以说明。显然我们需要更形象和更清楚的 Turing 机的记号。对于有穷自动机, 我们在第 2 章使用过包括状态和表示转移的箭头的记号。下一步我们对 Turing 机采取类似的记号。不过现在用箭头连接的东西它们本身就是 Turing 机。换句话说, 我们使用分层的记号, 其中越来越复杂的机器被从更简单的材料中构造出来。为了这个目的, 我们定义非常简单的基本机器的记号, 以及组合机器的规则。

基本机器。我们从非常低的起点开始: 写符号机和移带头机。让我们固定机器的字母表 Σ . 对每个 $a \in \Sigma \cup \{\leftarrow, \rightarrow\} - \{\triangleright\}$, 定义 Turing 机 $M_a = (\{s, h\}, \Sigma, \delta, s, \{h\})$, 其中对每个 $b \in \Sigma - \{\triangleright\}$, $\delta(s, b) = (h, a)$. 自然 $\delta(s, \triangleright)$ 仍然总是 (s, \rightarrow) . 即这台机器的唯一功能就是完成动作 a ——若 $a \in \Sigma$ 则写符号 a , 若 $a \in \{\leftarrow, \rightarrow\}$ 则按照 a 所示方向移动——然后立即停机。自然, 这样的行为有唯一例外: 若扫描符号是 \triangleright , 则机器尽职尽责地向右移动。

因为使用写符号机的时候太多了, 所以我们缩写它的名字, 仅仅写 a 来代替 M_a . 即若

$a \in \Sigma$ 则写 a 机仅仅表示成 a 。移带头机 M_L 和 M_R 缩写成 L (表示“左”) 和 R (表示“右”)。

组合机器的规则。有穷自动机的结构提示出组合 Turing 机的方式。单个机器就像有穷自动机的状态那样,用把有穷自动机的状态连接到一起来把这些机器彼此连接。不过直到前一台机器停机为止才应用从前一台机器到后一台机器的连接;后一台机器于是从初始状态和前一台机器留下的带和带头位置上启动。所以,若 M_1, M_2 和 M_3 都是 Turing 机,则图 4-3 里显示的机器操作如下:在 M_1 的初始状态启动,像 M_1 那样操作直到 M_1 停机为止;然后若当前扫描符号是 a 则启动 M_2 并且像 M_2 那样操作;否则若当前扫描符号是 b 则启动 M_3 并且像 M_3 那样操作。

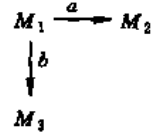


图 4-3

用它的各部分给出组合成的 Turing 机的明确定义是直截了当的。让我们用图 4-3 所示的机器做示范。假设三台 Turing 机 M_1, M_2 和 M_3 分别是 $M_1 = (K_1, \Sigma, \delta_1, s_1, H_1), M_2 = (K_2, \Sigma, \delta_2, s_2, H_2)$ 和 $M_3 = (K_3, \Sigma, \delta_3, s_3, H_3)$ 。我们假定所有这些机器的状态集都不相交,因为在组合机器的背景下这样假定是最方便的。于是,上面图 4-3 里所示的组合机器是 $M = (K, \Sigma, \delta, s, H)$, 其中

$$K = K_1 \cup K_2 \cup K_3,$$

$$s = s_1,$$

$$H = H_2 \cup H_3.$$

对每个 $\sigma \in \Sigma$ 和 $q \in K - H, \delta(q, \sigma)$ 如下定义:

(a) 若 $q \in K_1 - H_1$, 则 $\delta(q, \sigma) = \delta_1(q, \sigma)$ 。

(b) 若 $q \in K_2 - H_2$, 则 $\delta(q, \sigma) = \delta_2(q, \sigma)$ 。

(c) 若 $q \in K_3 - H_3$, 则 $\delta(q, \sigma) = \delta_3(q, \sigma)$ 。

(d) 最后,如果 $q \in H_1$ ——剩下的唯一情形——那么若 $\sigma = a$ 则 $\delta(q, \sigma) = s_2$, 若 $\sigma = b$ 则 $\delta(q, \sigma) = s_3$, 否则 $\delta(q, \sigma) \in H$ 。

现在记号的所有成分都准备就绪。我们将通过组合基本机器来构造机器,然后再组合组合机器去获得更复杂的机器,等等。我们知道假如我们希望的话,那么通过从基本机器的五元组开始并且完成上面示范的具体构造,就可获得这样描述的每台机器的五元组形式。

例 4.1.5 图 4-4(a) 说明由两台 R 的复制品组成的机器。这个示意图表示的机器把带头向右移动一格;然后若那个方格包含 a , 或 b , 或 \triangleright , 或 \sqcup , 则带头再向右移动一格。

像图 4-4(b) 那样表示这台机器是方便的,即标有多种符号的箭头等于是每只箭头标有一种符号的多只平行箭头。如果箭头标有机器的字母表 Σ 里的所有符号,那么就可以删除这些标记。因此,如果我们知道 $\Sigma = \{a, b, \triangleright, \sqcup\}$, 那么我们把上面的机器显示成

$$R \rightarrow R,$$

其中根据约定最左边的机器总是初始机器。有时通过把两台机器的表示并列来完全删除连接这两台机器的无标记箭头。利用这个约定上面的机器被简化成 RR , 甚至 R^2 。◇

例 4.1.6 若 $a \in \Sigma$ 是任意符号,则通过用 \bar{a} 表示“除 a 外的任意符号”,可以消除多重箭头和标记。因此图 4-5(a) 所示的机器向右扫描带直到发现空格为止。我们把这个最有用的机器表示成 R_{\sqcup} 。

图 4-5(b) 显示与图 4-5(a) 相同的机器的另一种缩写,在这里 $a \neq \sqcup$ 读作“除 \sqcup 外的

任意符号 a ”。这种记号的好处是 a 可在示意图的别处被用作机器的名字。作为说明，图 4-6 描绘一台机器，它向右扫描直到发现非空格的方格为止，然后把那个方格里的符号复制到其左方的相邻方格里。

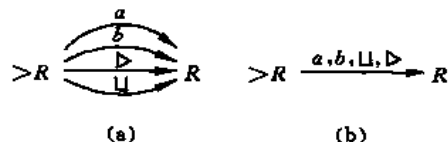


图 4-4

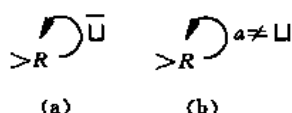


图 4-5

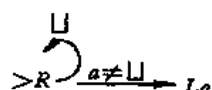


图 4-6

例 4.1.7 图 4-7 说明寻找有标记或无标记方格的机器。它们如下：

- (a) R_{\sqcup} ，它寻找在当前扫描方格右方的第一个空格方格。
- (b) L_{\sqcup} ，它寻找在当前扫描方格左方的第一个空格方格。
- (c) R_{\sqcup} ，它寻找在当前扫描方格右方的第一个非空格方格。
- (d) L_{\sqcup} ，它寻找在当前扫描方格左方的第一个非空格方格。

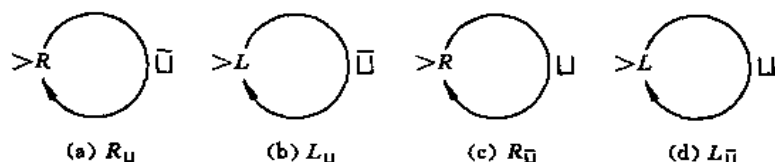


图 4-7

例 4.1.8 复制机 C 完成下列功能：如果 C 在输入 w 上启动，即如果把只包含非空格符（但可能为空）的字符串 w 写在带上，而带的其他部分都是空格，在 w 左方留出一个空格并且把带头放在 w 左方的这个空格上，那么这台机器最终停机并且在带上写有 $w\sqcup w$ ，带的其他部分都是空格。我们说 C 把 $\sqcup w$ 变换成 $\sqcup w\sqcup w$ 。

图 4-8 给出 C 的示意图。

例 4.1.9 左平移机 S_{\leftarrow} 把 $\sqcup w$ 变换成 $\sqcup w$ ，其中 w 不含空格。如图 4-9 所示。

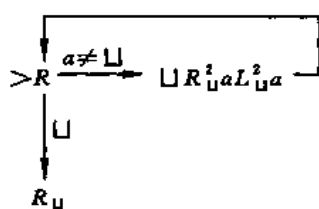


图 4-8

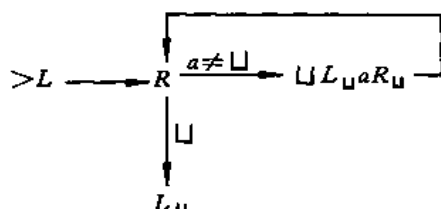


图 4-9

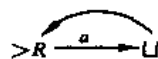


图 4-10

例 4.1.10 图 4-10 是例 4.1.1 里定义的机器，它删除带里的 a 。

事实上完整写出的这台机器的状态转移表与例 4.1.1 里给出的机器的状态转移表并不相同，不同之处是细微和没有多大影响的，将在习题 4.1.8 里探讨——即图 4-10 里的机器还包含某些多余状态，它们是子机器的终结状态。

习 题

4.1.1 设 $M = (K, \Sigma, \delta, s, \{h\})$, 其中

$$\begin{aligned} K &= \{q_0, q_1, h\}, \\ \Sigma &= \{a, b, \sqcup, \triangleright\}, \\ s &= q_0, \end{aligned}$$

并且 δ 由表 4-3 给定。

(a) 跟踪从格局 $(q_0, \triangleright aabbba)$ 启动的 M 的计算。

(b) 非形式化描述当在 q_0 且扫描任意带方格时启动, M 做什么动作。

4.1.2 对机器 $M = (K, \Sigma, \delta, s, \{h\})$ 重做习题 4.1.1, 其中

$$\begin{aligned} K &= \{q_0, q_1, q_2, h\}, \\ \Sigma &= \{a, b, \sqcup, \triangleright\}, \\ s &= q_0, \end{aligned}$$

并且 δ 由表 4-4 给定(在 \triangleright 上的转移是 $\delta(q, \triangleright) = (q, \triangleright)$, 已被省略)。

从格局 $(q_0, \triangleright aabb \sqcup bb \sqcup \sqcup \sqcup aba)$ 启动。

表 4-3

q	σ	$\delta(q, \sigma)$
q_0	a	(q_1, b)
q_0	b	(q_1, a)
q_0	\sqcup	(h, \sqcup)
q_0	\triangleright	(q_0, \rightarrow)
q_1	a	(q_0, \rightarrow)
q_1	b	(q_0, \rightarrow)
q_1	\sqcup	(q_0, \rightarrow)
q_1	\triangleright	(q_1, \rightarrow)

表 4-4

q	σ	$\delta(q, \sigma)$
q_0	a	(q_1, \leftarrow)
q_0	b	(q_0, \rightarrow)
q_0	\sqcup	(q_0, \rightarrow)
q_1	a	(q_1, \leftarrow)
q_1	b	(q_2, \rightarrow)
q_1	\sqcup	(q_1, \leftarrow)
q_2	a	(q_2, \rightarrow)
q_2	b	(q_2, \rightarrow)
q_2	\sqcup	(h, \sqcup)

表 4-5

q	σ	$\delta(q, \sigma)$	q	σ	$\delta(q, \sigma)$
q_0	a	(q_2, \rightarrow)	q_2	\sqcup	(h, \sqcup)
q_0	b	(q_3, a)	q_2	\triangleright	(q_2, \rightarrow)
q_0	\sqcup	(h, \sqcup)	q_3	a	(q_4, \rightarrow)
q_0	\triangleright	(q_0, \rightarrow)	q_3	b	(q_4, \rightarrow)
q_1	a	(q_2, \rightarrow)	q_3	\sqcup	(q_4, \rightarrow)
q_1	b	(q_2, \rightarrow)	q_3	\triangleright	(q_3, \rightarrow)
q_1	\sqcup	(q_2, \rightarrow)	q_4	a	(q_2, \rightarrow)
q_1	\triangleright	(q_1, \rightarrow)	q_4	b	(q_4, \rightarrow)
q_2	a	(q_1, b)	q_4	\sqcup	(h, \sqcup)
q_2	b	(q_3, a)	q_4	\triangleright	(q_4, \rightarrow)

4.1.3 对机器 $M = (K, \Sigma, \delta, s, \{h\})$ 重做习题 4.1.1, 其中

$$\begin{aligned} K &= \{q_0, q_1, q_2, q_3, q_4, h\}, \\ \Sigma &= \{a, b, \sqcup, \triangleright\}, \\ s &= q_0, \end{aligned}$$

并且 δ 由表 4-5 给定。

从格局 $(q_0, \triangleright aaabbbbaa)$ 启动。

4.1.4 设 M 是 Turing 机 $(K, \Sigma, \delta, s, \{h\})$, 其中

$$\begin{aligned} K &= \{q_0, q_1, q_2, h\}, \\ \Sigma &= \{a, \sqcup, \triangleright\}, \\ s &= q_0, \end{aligned}$$

并且 δ 由表 4-6 给定。

表 4-6

q	σ	$\delta(q, \sigma)$
q_0	a	(q_1, \leftarrow)
q_0	\sqcup	(q_0, \sqcup)
q_0	\triangleright	(q_0, \rightarrow)
q_1	a	(q_2, \sqcup)
q_1	\sqcup	(h, \sqcup)
q_1	\triangleright	(q_1, \rightarrow)
q_2	a	(q_2, a)
q_2	\sqcup	(q_0, \leftarrow)
q_2	\triangleright	(q_2, \rightarrow)

设 $n \geq 0$ 。仔细描述当在格局 $(q_0, \triangleright \sqcup a^n \sqcup)$ 里启动时, M 做什么动作。

4.1.5 在 Turing 机的定义里, 我们允许不移动带头但改写带方格, 以及不改写带方格但向左或向右移动带头。假如我们还允许不改写带方格且让带头保持静止, 那么可能发生什么事情?

4.1.6 (a) 下列哪些可能是格局?

(I) $(q, \triangleright a \sqcup a \sqcup \sqcup, a \sqcup)$

(II) $(q, abcb, abc)$

(III) $(p, \triangleright abca, e)$

(IV) $(h, \triangleright e, e)$

(V) $(q, \triangleright a \sqcup abb, \sqcup aa \sqcup)$

(VI) $(p, \triangleright aab, \sqcup a)$

(VII) $(q, \triangleright e, \sqcup aa)$

(VIII) $(h, \triangleright aa, \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup a)$

(b) 用缩写记号改写从部分 (I) 到 (VIII) 里那些是格局的部分。

(c) 把下列缩写格局改写成完整的格局。

(I) $(q, \triangleright abcd)$

(II) $(q, \triangleright a)$

(III) $(p, \triangleright aa \sqcup \sqcup)$

(IV) $(h, \triangleright \sqcup abc)$

4.1.7 设计并完整写出这样的 Turing 机, 它向右扫描直到发现两个连续的 a 为止, 然后停机, 这台 Turing 机的字母表是 $\{a, b, \sqcup, \triangleright\}$ 。

4.1.8 给出如图 E4-1 所示三台 Turing 机的完整细节。

4.1.9 机器 LR 和 RL 总是做相同的工作吗? 试解释之。

4.1.10 解释这台机器做什么工作。

$\triangleright R \xrightarrow{a \neq \sqcup} R \xrightarrow{b \neq \sqcup} R \sqcup a R \sqcup b$

4.1.11 跟踪例 4.1.8 里的 Turing 机当在 $\triangleright \sqcup aabb$ 上启动时的操作。

4.1.12 跟踪例 4.1.9 里的 Turing 机在 $\triangleright \sqcup aabb \sqcup$ 上的操作。

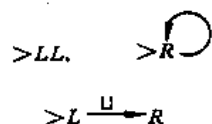


图 E4-1

4.2 用 Turing 机计算

我们在介绍 Turing 机时许诺说, 它们作为语言接受器在性能上超过我们在前几章里介绍的所有其他种类的自动机。不过迄今为止我们仅仅提供了 Turing 机的“机械部分”, 还没有说明如何用它们完成各种计算任务, 例如识别语言。这就像给你送来了没有键盘、没有磁盘驱动器、也没有屏幕的计算机, 即没有输入或输出信息的手段。因此现在有必要建立使用 Turing 机的某些约定。

首先我们采取下列策略向 Turing 机提供输入: 把不含空格符的输入字符串写在最左

端符号 \triangleright 的右方,在输入左方留下一个空格,输入右方都是空格;带头就在 \triangleright 与输入字符串之间的空格里,机器在初始状态里开始操作。如果 $M = (K, \Sigma, \delta, s, H)$ 是 Turing 机并且 $w \in (\Sigma - \{\sqcup, \triangleright\})^*$, 那么 M 在输入 w 上的初始格局是 $(s, \triangleright \sqcup w)$ 。有了这个约定之后,现在我们定义如何用 Turing 机来作为语言识别器。

定义 4.2.1 设 $M = (K, \Sigma, \delta, s, H)$ 是 Turing 机,使得 $H = \{y, n\}$ 包含两个不同的停机状态(y 和 n 分别表示“是”和“否”)。状态分量是 y 的任何停机格局都称为接受格局,而状态分量是 n 的停机格局称为拒绝格局。对输入 $w \in (\Sigma - \{\sqcup, \triangleright\})^*$,若 $(s, \triangleright \sqcup w)$ 产生接受格局则我们说 M 接受 w ;若 $(s, \triangleright \sqcup w)$ 产生拒绝格局则我们说 M 拒绝 w 。

设 $\Sigma_0 \subseteq (\Sigma - \{\sqcup, \triangleright\})$ 是字母表,称为 M 的输入字母表;通过固定 Σ_0 是 $\Sigma - \{\sqcup, \triangleright\}$ 的子集,我们允许 Turing 机在计算中使用除在输入里出现符号外的额外符号。如果 $L \subseteq \Sigma_0^*$ 是语言,并且对任何字符串 $w \in \Sigma_0^*$,下列关系为真:若 $w \in L$ 则 M 接受 w ;若 $w \notin L$ 则 M 拒绝 w ,那么我们就说 M 判定语言 L 。

最后,若存在 Turing 机判定语言 L 则 L 称为递归的。

即 Turing 机判定语言 L 的条件是,当在输入 w 上启动时它总是停机并且停机的状态是对输入的正确回答:若 $w \in L$ 则是 y ,若 $w \notin L$ 则是 n 。注意当机器的输入里包含空格或左端符时,定义里没有给出关于发生什么事情的保证。

例 4.2.1 考虑语言 $L = \{a^n b^n c^n; n \geq 0\}$,在此之前的所有类型的语言识别器都不能识别它。示意图显示在图 4-11 里的 Turing 机判定 L 。在示意图里我们还利用有助于判定语言的两台新的基本机器:机器 y 让新状态是接受状态 y ,而机器 n 把状态变成 n 。

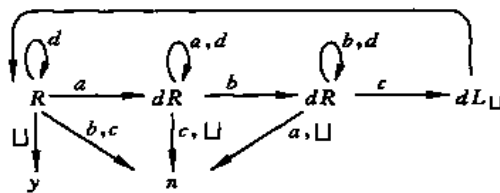


图 4-11

M 使用的策略是简单的:在输入 $a^n b^n c^n$ 上它分 n 个阶段操作。在每个阶段 M 从输入字符串的左端开始,向右移动搜索 a 。当发现 a 时把它改成 d ,然后继续向右寻找 b 。当发现 b 时把它改成 d ,然后寻找 c 。当发现 c 并把它改成 d 时,这个阶段结束,带头返回到输入左端。然后下一个阶段开始。即在每个阶段机器把一个 a ,一个 b 和一个 c 分别改成 d 。如果在任何时刻机器感觉到输入不属于 $a^n b^n c^n$ 或者有多余的某种符号(例如,如果当寻找 a 时它看到 b 或 c),那么它进入状态 n 并且立即拒绝。不过若在寻找 a 时它遇到输入的右端,则这意味着全部输入都已改成了 d ,因此对某个 $n \geq 0$,输入确实形如 $a^n b^n c^n$ 。于是这台机器接受。◇

判定语言的 Turing 机有些微妙之处;对于我们迄今为止在本书里见过的其他语言识别器(即使非确定型的),要么机器接受输入,要么机器拒绝输入,二者必取其一。在另一方面,即使 Turing 机只有两个停机状态 y 和 n ,通过不停机,它也总有机会不给出答案(“是”或“否”)。给定 Turing 机,它可能判定也可能不判定语言,没有明显办法去辨别它究竟做什么。这种缺陷,其影响深远的重要性以及必要性,将在本章和下一章变得明显起来。

4.2.1 递归函数

因为 Turing 机可在带上写,所以它们可提供比仅仅“是”或“否”更详细的输出。

定义 4.2.2 设 $M = (K, \Sigma, \delta, s, \{h\})$ 是 Turing 机, $\Sigma_0 \subseteq \Sigma - \{\sqcup, \triangleright\}$ 是字母表, 并设 $w \in \Sigma_0^*$ 。假设 M 在输入 w 上停机, 而且对某个 $y \in \Sigma_0^*$, $(s, \triangleright \sqcup w) \vdash_M^* (h, \triangleright \sqcup y)$, 则 y 称为 M 在输入 w 上的输出, 并表示成 $M(w)$ 。注意仅当 M 在输入 w 上停机时 $M(w)$ 才有定义, 而且事实上是在形如 $(h, \triangleright \sqcup y)$ 的格局里停机, 其中 $y \in \Sigma_0^*$ 。

现在设 f 是从 Σ_0^* 到 Σ_0^* 的任意函数。若对所有 $w \in \Sigma_0^*$, $M(w) = f(w)$, 则我们说 M 计算函数 f 。即对所有 $w \in \Sigma_0^*$, M 在输入 w 上最终停机, 并且当它确实停机时带上包含字符串 $\triangleright \sqcup f(w)$ 。若存在 Turing 机计算函数 f , 则 f 称为递归的。

例 4.2.2 定义成 $\kappa(w) = ww$ 的函数 $\kappa: \Sigma^* \mapsto \Sigma^*$ 可用机器 CS_{\perp} 计算, 即在复制机后面跟着左平移机(它们是在上一节快结束时定义的)。◇

$\{0, 1\}^*$ 里的字符串可用来把非负整数表示成熟悉的二进制记号。任何字符串 $w = a_1 a_2 \dots a_n \in \{0, 1\}^*$ 表示数字

$$\text{num}(w) = a_1 \cdot 2^{n-1} + a_2 \cdot 2^{n-2} + \dots + a_n$$

任何自然数可用唯一的方式表示成 $0 \sqcup 1(0 \sqcup 1)^*$ 里的字符串, 即开头没有多余的 0。

相应地可认为计算从 $\{0, 1\}^*$ 到 $\{0, 1\}^*$ 的函数的 Turing 机是计算从自然数到自然数的函数的 Turing 机。事实上, 多变量数值函数, 比如加法和乘法, 可用计算从 $\{0, 1, ;\}^*$ 到 $\{0, 1\}^*$ 的函数的 Turing 机计算, 其中“;”是用来分隔二进制变量的符号。

定义 4.2.3 设 $M = (K, \Sigma, \delta, s, \{h\})$ 是 Turing 机使得 $0, 1, ; \in \Sigma$, 并设对某个 $k \geq 1$, f 是从 N^k 到 N 的函数。若对所有 $w_1, \dots, w_k \in 0 \sqcup 1(0 \sqcup 1)^*$ (即对都是整数的二进制编码的任意 k 个字符串), $\text{num}(M(w_1; \dots; w_k)) = f(\text{num}(w_1), \dots, \text{num}(w_k))$, 则我们说 M 计算函数 f 。即若 M 在整数 n_1, \dots, n_k 的二进制表示的输入上启动则它最终停机, 并且当它确实停机时带上有表示数字 $f(n_1, \dots, n_k)$ 的字符串——函数值。若存在计算函数 $f: N^k \mapsto N$ 的 Turing 机 M , 则 f 称为递归的。

事实上, 用来描述 Turing 机计算的函数和语言的递归一词就起源于对这样的数值函数的研究。它预言本章快结束时证明的结果, 即 Turing 机计算的数值函数等于那些从某些基本函数出发递归地定义的函数。

例 4.2.3 我们设计计算后继函数 $\text{succ}(n) = n + 1$ 的机器(如图 4-12 所示, 其中 S_R 是右平移机, 它与例 4.1.9 里的机器类似但向右平移输入)。这台机器首先找到输入的右端, 然后只要它看到 1 就向左移动, 同时把所有看到的 1 变成 0。当它看到 0 时就把 0 变成 1 并且停机。假如当寻找 0 时看到 \sqcup , 这意味着输入数字的二进制表示全是 1 (它是 2 的幂减去 1), 因此这台机器在 \sqcup 的地方再写一个 1, 并且把整个字符串向右平移一格然后停机。严格地说, 显示出的机器不计算 $n + 1$,

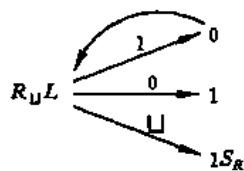


图 4-12

因为它不总是在停机时让带头在结果的左方, 但是通过添加 L_{\sqcup} (与图 4-5 类似), 就可纠正这个毛病。◇

前一小节最后关于我们不能辨别 Turing 机是否判定语言的议论也适用于函数计算。我们为得到 Turing 机计算的非常广泛的函数而必须付出的代价,是我们不能辨别给定的 Turing 机是否确实计算这样的函数——即它是否在所有输入上停机。

4.2.2 递归可枚举语言

若 Turing 机判定语言或计算函数,则可合理地认为它是正确而可靠地完成某项计算任务的算法。下一步我们介绍第三种更微妙的方式, Turing 机以这种方式定义语言。

定义 4.2.4 设 $M = (K, \Sigma, \delta, s, H)$ 是 Turing 机, $\Sigma_0 \subseteq \Sigma - \{\sqcup, \triangleright\}$ 是字母表, 并设 $L \subseteq \Sigma_0^*$ 是语言。若对任意字符串 $w \in \Sigma_0^*$, 下列关系为真: $w \in L$ 当且仅当 M 在输入 w 上停机, 则我们说 M 半判定 L 。语言 L 是递归可枚举的当且仅当存在 Turing 机 M 半判定 L 。

因此, 当向 M 提供输入 $w \in L$ 时, 要求它最终停机。只要它确实最终到达停机格局, 我们就不深究它到达哪种停机格局。不过, 若 $w \in \Sigma_0^* - L$, 则 M 必然永不进入停机状态。因为任何非停机格局都产生某种其他格局 (δ 是全函数), 所以在这种情形里机器必然无限地继续计算。

扩充我们在前一小节介绍的 Turing 机的“函数的”记号 (它允许我们写等式, 比如, $M(w) = v$), 若 M 在输入 w 上不停机, 则我们写成 $M(w) = \nearrow$ 。利用这种记号, 我们可重新陈述 Turing 机 M 半判定语言 $L \subseteq \Sigma_0^*$ 的定义如下: 对所有 $w \in \Sigma_0^*$, $M(w) = \nearrow$ 当且仅当 $w \notin L$ 。

例 4.2.4 设 $L = \{w \in \{a, b\}^* : w \text{ 至少包含一个 } a\}$ 。于是图 4-13 所示的 Turing 机半判定 L 。

当这台机器对某个 $w \in \{a, b\}^*$, 在格局 $(q_0, \triangleright \sqcup w)$ 里启动时, 它只是向右扫描直到遇到 a 为止, 然后停机。若没有找到 a , 这台机器就在输入后面跟着的空格里永远移动下去, 永不停机。所以 L 恰恰是使得 M 在输入 w 上停机的 $\{a, b\}^*$ 里的字符串 w 的集合。因此 M 半判定 L , 所以 L 是递归可枚举的。◇

“在空格里永远移动下去”只是 Turing 机不停机的方式之一。例如对满足 $\delta(q, a) = (q, a)$ 的任何机器, 若它在状态 q 扫描 a , 则它在原地“死循环”。自然, 可设计更复杂的循环行为, 让机器在有穷多个不同的格局里无限地运行下去。

Turing 机的半判定性的定义是对确定型有穷自动机的接受概念相当直截了当的扩充。不过还是有重大的区别。当有穷自动机读完了所有输入时它总是停机——问题是它究竟在终结状态还是在非终结状态停机。在这种意义下它是有用的计算装置, 是一个算法, 我们可靠地从它获得对输入是否属于被接受的语言的回答, 我们等待直到所有输入已被读完为止, 然后观察机器的状态。相反, 半判定语言 L 的 Turing 机不能被用来辨别字符串 w 是否属于 L , 因为如果 $w \notin L$, 那么我们永远不知道为了答案我们得等多久。①半判定语言的 Turing 机都不是算法。

① 在下推自动机上我们遇到过同样的困难 (回忆第 3.7 节)。下推自动机在原理上可通过永远操纵栈但再也不读输入来拒绝读输入。在 3.7 节为获得有关某些上下文无关语言的有计算用途的下推自动机, 我们不得不消除这样的行为。

我们从例 4.2.1 知道 $\{a^n b^n c^n : n \geq 0\}$ 是递归语言。但它是否递归可枚举? 回答是容易的: 任何递归语言也是递归可枚举的。为构造半判定而不是判定这个语言的另一台 Turing 机, 要做的全部工作就是把拒绝状态 n 变成非停机状态以保证机器永不停机。具体地说, 给定判定 L 的任意 Turing 机 $M = (K, \Sigma, \delta, s, \{y, n\})$, 我们可定义半判定 L 的 Turing 机如下, $M' = (K, \Sigma, \delta', s, \{y\})$, 其中 δ' 只是 δ 增加下列关于 n 的转移—— n 不再是停机状态: 对所有 $a \in \Sigma$, $\delta'(n, a) = \delta(n, a)$ 。很清楚, 若 M 确实判定 L 则 M' 半判定 L , 因为 M' 接受与 M 同样的输入; 另外若 M 拒绝输入 w , 则 M' 在 w 上不停机(它在状态 n “死循环”)。换句话说, 对所有输入 w , $M'(w) = \nearrow$ 当且仅当 $M(w) = n$ 。

我们证明了下列重要结果。

定理 4.2.1 若语言是递归的, 则它是递归可枚举的。

自然, 令人感兴趣的(也是困难的)问题是逆命题: 我们能否总是把每一台半判定语言的 Turing 机(对半判定性的单侧定义使得它作为计算装置其实是无用的)都变换成判定同一个语言的真正的算法? 我们在下一章将看到这里的回答是否定的: 存在非递归的递归可枚举语言。

递归语言类的重要性质是它在补运算下封闭。

定理 4.2.2 若 L 是递归语言则它的补 \bar{L} 也是递归的。

证明: 若 Turing 机 $M = (K, \Sigma, \delta, s, \{y, n\})$ 判定 L , 则 Turing 机 $M' = (K, \Sigma, \delta', s, \{y, n\})$ 判定 \bar{L} 。这里, 除了 M' 反转两种特殊停机状态 y 和 n 的作用以外, M' 与 M 是相同的, 即 δ' 定义如下

$$\delta'(q, a) = \begin{cases} n & \text{如果 } \delta(q, a) = y \\ y & \text{如果 } \delta(q, a) = n \\ \delta(q, a) & \text{否则} \end{cases}$$

$M'(w) = y$ 当且仅当 $M(w) = n$, 因此 M' 判定 \bar{L} 。 ■

递归可枚举语言是否也在补运算下封闭? 这次, 我们在下一章将看到回答是否定的。

习 题

- 4.2.1 给出计算下列函数的 Turing 机(用缩写记号), f 是从 $\{a, b\}^*$ 里的字符串到 $\{a, b\}^*$ 里的字符串的函数: $f(w) = ww^R$ 。
- 4.2.2 给出判定下列在 $\{a, b\}$ 上的语言的 Turing 机:
 - (a) \emptyset
 - (b) $\{e\}$
 - (c) $\{a\}$
 - (d) $\{a\}^*$
- 4.2.3 给出半判定语言 a^*ba^*b 的 Turing 机。
- 4.2.4 (a) 给出带有一种停机状态并且不计算从字符串到字符串的函数的 Turing 机的例子。
 (b) 给出带有两种停机状态 y 和 n 并且不判定任何语言的 Turing 机的例子。
 (c) 你能否给出带有一种停机状态并且不半判定任何语言的 Turing 机的例子?

4.3 Turing 机的扩充

前一节的例子说明尽管 Turing 机动作缓慢而笨拙,它们还是可完成相当强的计算。为了更好地理解它们惊人的能力,我们考虑在多种方向上扩充 Turing 机模型的效果。我们将看到在每种情形里附加的特性不增加可计算的函数或者可判定的语言:“新的改进型”的 Turing 机在每种情形里都可用标准模型模拟。这样的结果让我们更加相信 Turing 机确实是终极的计算装置,是向越来越强的自动机前进的终点。这些结果的额外好处是,今后当设计 Turing 机去解决具体问题时,我们允许使用附加特性,因为我们有足够的知识可在必要时消除对这样的特性的依赖。

4.3.1 多带 Turing 机

可以设想有多条带的 Turing 机(如图 4-14 所示)。每条带都通过读写头连接到有穷控制制上(在每条带上有有一只读写头)。机器可在一步里读出所有带头扫描的符号,然后根据这些符号和当前状态,改写某些扫描的方格,向左或向右移动某些带头,以及改变状态。对任意固定的整数 $k \geq 1$, k 带 Turing 机是像上面那样装备了 k 条带和相应带头的 Turing 机。因此迄今为止在本章里研究的“标准”Turing 机,仅仅是 $k = 1$ 的 k 带 Turing 机。

定义 4.3.1 设 $k \geq 1$ 是整数。 k 带 Turing 机是五元组 $(K, \Sigma, \delta, s, H)$, 其中 K, Σ, s 和 H 都是像在普通 Turing 机的定义里那样,而转移函数 δ 是从 $(K - H) \times \Sigma^k$ 到 $K \times (\Sigma \cup \{\leftarrow, \rightarrow\})^k$ 的函数。即,对每种状态 q 以及带符号的每个 k 元组 (a_1, \dots, a_k) , $\delta(q, (a_1, \dots, a_k)) = (p, (b_1, \dots, b_k))$, 其中 p 像前面那样是新状态, b_j 在直观上是 M 在带 j 上采取的动作。自然,我们还坚持,若对某个 $j \leq k, a_j = \triangleright$, 则 $b_j = \rightarrow$ 。

计算发生在 k 带 Turing 机的所有 k 条带上。相应地,这样一台机器的格局必须包括所有带的信息。

定义 4.3.2 设 $M = (K, \Sigma, \delta, s, H)$ 是 k 带 Turing 机。 M 的格局是

$$K \times (\triangleright \Sigma^* \times (\Sigma^* (\Sigma - \{\triangleright\}) \cup \{e\}))^k$$

的成员。即,格局说明当前状态以及 k 条带里每条的带内容和带头位置。

如果 $(q, (w_1 \underline{a}_1 u_1, \dots, w_k \underline{a}_k u_k))$ 是 k 带 Turing 机的格局(其中我们使用了 k 重的格局缩写记号),并且如果 $\delta(p, (a_1, \dots, a_k)) = (b_1, \dots, b_k)$, 那么经过一步移动机器可转移到格局 $(p, (w'_1 \underline{a}'_1 u'_1, \dots, w'_k \underline{a}'_k u'_k))$, 其中对 $i = 1, \dots, k$, 恰恰像在定义 4.1.3 里那样,动作 b_i 把 $w_i \underline{a}_i u_i$ 修改成 $w'_i \underline{a}'_i u'_i$ 。我们说格局 $(q, (w_1 \underline{a}_1 u_1, \dots, w_k \underline{a}_k u_k))$ 在一步里产生格局 $(p, (w'_1 \underline{a}'_1 u'_1, \dots, w'_k \underline{a}'_k u'_k))$ 。

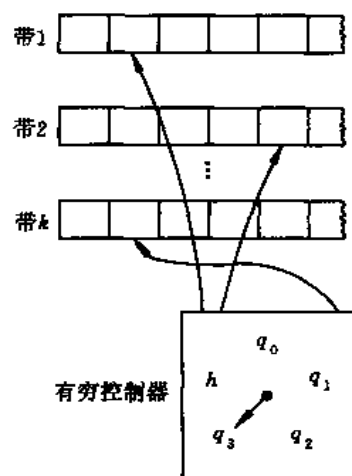


图 4-14

例 4.3.1 利用对于标准 Turing 机在前面讨论过的任何方式, k 带 Turing 机可计算函数, 判定或半判定语言。我们采取约定, 输入字符串像提供给标准 Turing 机那样, 用同样的方式放在第一条带上。其余的带起初都是空格, 带头都在每条带最左端的空格上。在计算结束时, k 带 Turing 机在第一条带上留下输出; 其余带的内容都被忽略。

多带常常有利于构造完成具体功能的 Turing 机。例如, 考虑例 4.1.8 里给出的复制机 C 的任务, 把 $\triangleright \sqcup w \sqcup$ 变换成 $\triangleright \sqcup w \sqcup w \sqcup$, 其中 $w \in \{a, b\}^*$ 。2 带 Turing 机可像下面那样完成这个任务。

(1) 把两条带的带头都向右移动, 把第一条带上的每个符号都复制到第二条带上, 直到在第一条带上发现空格为止。第二条带的第一个方格应当留作空格。

(2) 在第二条带上带头向左移动, 直到发现空格为止。

(3) 再把两条带的带头都向右移动, 这次从第二条带把符号复制到第一条带上。当在第二条带上发现空格时停机。

这个动作的序列可图示如下:

在开始时: 第一条带	$\triangleright \sqcup w$
第二条带	$\triangleright \sqcup$
在(1)之后: 第一条带	$\triangleright \sqcup w \sqcup$
第二条带	$\triangleright \sqcup w \sqcup$
在(2)之后: 第一条带	$\triangleright \sqcup w \sqcup$
第二条带	$\triangleright \sqcup w$
第(3)之后: 第一条带	$\triangleright \sqcup w \sqcup w \sqcup$
第二条带	$\triangleright \sqcup w \sqcup$

可用前几节里图示单带 Turing 机的同样方式去图示多带 Turing 机。我们只在表示每台机器的符号上附加它在上面对应的带的编号作为上标; 所有其余的带都不受影响。例如, \sqcup^2 在第二条带上写空格, L^1 在第一条带上向左寻找空格, $R^{1,2}$ 把第一和第二条带的带头都向右移动, 在箭头上的标记 a^1 表示当第一条带被扫描符号是 a 时动作发生, 等等。(当表示多带 Turing 机时, 我们避免使用缩写 M^2 去表示 MM 。)利用这个约定, 2 带型的复制机可被说明成图 4-15 那样。我们标明了完成上述功能(1)到(3)的子机器。◇

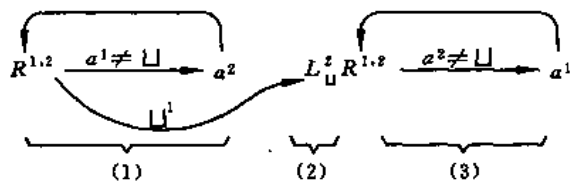


图 4-15

例 4.3.2 我们已看到(例 4.2.3) Turing 机可把 1 加到任意二进制整数上。不应让我们惊讶的是 Turing 机也可把任意两个二进制数字相加(回忆习题 2.4.3, 它提示在某种意义上这个任务甚至可用有穷自动机来完成)。利用两条带, 这个任务可被图 4-16 里描绘的机器来完成。在箭头上标记的成对比特都是缩写, 比如 01 表示 $a^1 = 0, a^2 = 1$ 。

首先这台机器把第一个二进制整数复制到第二条带上, 在第一条带里用 0 填充第一

个整数原来的位置(以及分隔这两个整数的“;”的位置);通过这种方式,第一条带包含第二个整数,前面加了一些0。然后机器用“小学方法”完成二进制加法,即从两个整数的最低位开始,把对应位相加,把结果写在第一条带上,并且用状态“记住进位”。

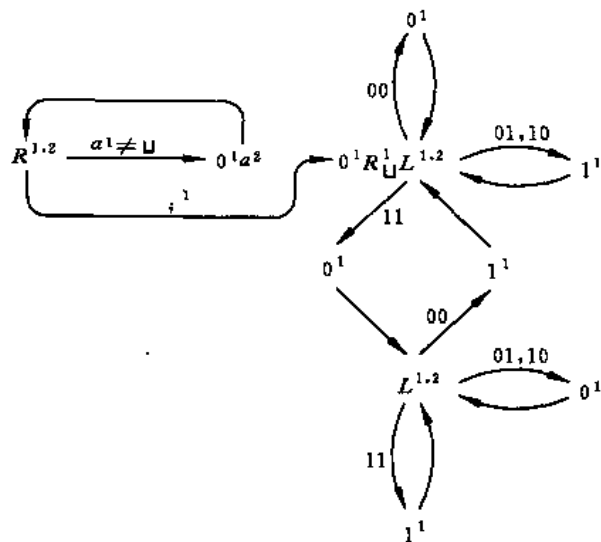


图 4-16

同理,我们可构造把两个数字相乘的3带 Turing 机;它的设计留作练习(习题 4.3.5)。

显然, k 带 Turing 机能够完成相当复杂的计算任务。下一步我们证明,任何 k 带 Turing 机可用单带 Turing 机模拟。我们这样说的意思是,给定任意 k 带 Turing 机,我们可设计标准 Turing 机,它表现出同样的输入输出行为——判定或半判定同样的语言,计算同样的函数。在本章和下面几章对计算装置能力的研究里,这样的模拟是我们一套方法中的一个重要方法。典型地,它们是这样的方法,用模拟机器的多步,去模仿被模拟机器的一步。我们的这种类型的第一个结果和它的证明,是对这条研究路线的相当好的说明。

定理 4.3.1 对某个 $k \geq 1$, 设 $M = (K, \Sigma, \delta, s, H)$ 是 k 带 Turing 机。那么存在标准 Turing 机 $M' = (K', \Sigma', \delta', s', H)$, 其中 $\Sigma \subseteq \Sigma'$, 使得下列关系成立: 对任意输入字符串 $x \in \Sigma^*$, M 在输入 x 上停机并且在第一条带上有输出 y 当且仅当 M' 在输入 x 上在同样的停机状态里停机, 并且在带上有同样的输出 y 。另外, 如果 M 在输入 x 上在 t 步之后停机, 那么 M' 在输入 x 上在 $O(t \cdot (|x| + t))$ 步之后停机。

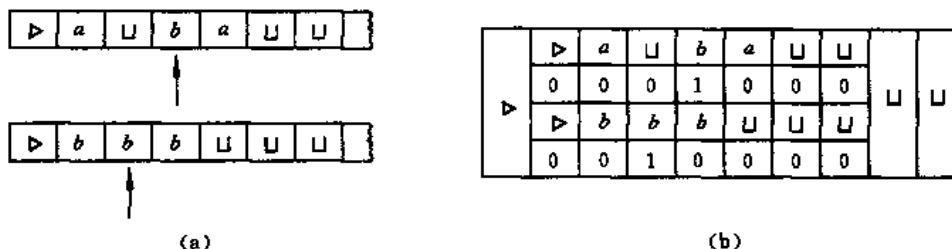


图 4-17

证明: M' 的带必须以某种方式包含 M 所有带里的全部信息,完成这个任务的简单方式是,设想把 M' 的带划分成多条轨道(如图 4-17(b) 所示),每条轨道专门模拟 M 的一条不同的带。具体地说,除了最左边方格,它像通常那样包含左端符号 \triangleright ,以及带右方的无穷空格部分以外, M' 的带被水平分割成 $2k$ 条轨道。 M' 的带的第一,第三, ..., 第 $(2k-1)$ 条轨道,分别对应于 M 的第一,第二, ..., 第 k 条带。 M' 的带的第二,第四, ..., 第 $2k$ 条轨道,用来记录 M 的第一,第二, ..., 第 k 条带的带头位置。用下列方式:如果在 M 的第 i 条带上,带头位于第 n 个带方格,那么在 M' 的第 $2i$ 条轨道上,第 $(n+1)$ 个带方格包含 1,除第 $(n+1)$ 个带方格外,所有其余带方格都包含 0。例如,如果 $k=2$,那么图 4-17(a) 所示的 M 的带和带头,对应于图 4-17(b) 所示的 M' 的带。

当然,把 M' 的带划分成轨道,这纯粹是想象的结果。在形式化上,达到这种效果的方法是设

$$\Sigma' = \Sigma \cup (\Sigma \times \{0,1\})^k$$

即, M' 的字母表包括 M 的字母表(这样使得 M' 收到与 M 同样的输入并且送出同样的输出),以及所有形如 $(a_1, b_1, \dots, a_k, b_k)$ 的 $2k$ 元组,其中 $a_1, \dots, a_k \in \Sigma$ 并且 $b_1, \dots, b_k \in \{0,1\}$ 。从这个字母表到解释成 $2k$ 条轨道,这样的变换是简单的:我们读到任何这样的 $2k$ 元组时,它说 M' 的第一条轨道包含 a_1 ,第二条轨道包含 b_1 ,等等,直到第 $2k$ 条轨道包含 b_k 。这又意味着 M 的第 i 条带的相应符号是 a_i ,并且这个符号被第 i 个带头扫描当且仅当 $b_i = 1$ (回忆图 4-17(b))。

当给定输入 $w \in \Sigma^*$ 时, M' 操作如下:

(1) 把输入向右平移一个带方格,返回 \triangleright 右方相邻的方格,在里面写符号 $(\triangleright, 0, \triangleright, 0, \dots, \triangleright, 0)$ ——这表示 k 条带的左端。向右移动一格并且写符号 $(\sqcup, 1, \sqcup, 1, \dots, \sqcup, 1)$ ——这表示所有 k 条带的第一个方格都包含 \sqcup ,并且都被带头扫描。继续向右前进。在每格上,若遇到符号 $a \neq \sqcup$,则在它的位置上写符号 $(a, 0, \sqcup, 0, \dots, \sqcup, 0)$ 。若遇到 \sqcup ,则第一阶段结束。 M' 的带内容忠实地表示 M 的初始格局。

(2) 模拟 M 的计算,直到 M 停机为止(假如它停机)。为了模拟 M 的一步计算, M' 不得不完成下列的操作序列(我们假设,当它开始每步模拟时,带头扫描第一个“真正空格”,即,在带上还没有被细分成轨道的第一个方格):

(a) 向左扫描整段的带,收集关于 M 的 k 个带头扫描符号的信息。在所有扫描符号被确认之后(通过相应偶数轨道里的 1),返回最右方的真正空格。虽然 M' 这个部分的操作不在带上写,但是当带头返回右端时,改变有穷控制器的状态使得状态反映出,在 k 条轨道里标记的带头位置上、由 Σ 里的符号组成的 k 元组。

(b) 先向左再向右扫描整段的带,根据 M 的被模拟动作更新各条轨道。在每对轨道上,这种更新包括:要么把带头位置标记向左或向右移动一格,要么改写带头位置上属于 Σ 的符号。

(3) 当 M 停机时, M' 首先把带从轨道转换成单个符号的格式,忽略除第一条轨道外的所有轨道,把带头置于 M 放第一只带头的地方,最后,停机在 M 停机时的状态。

在这个描述里省略掉了许多细节。尽管阶段(2)想象起来毫无困难,但是具体说明起来还是相当麻烦的,而且确实有多种选择去实际完成所描述的操作。有一个细节也许值得

描述。偶而,对于某个 $n > |w|$, M 可能不得不把一只带头首次移动到相应带的第 n 个方格。为了模拟这个动作, M' 不得不扩充划分成 $2k$ 条轨道的那部分带,并且把右方的第一个 \sqcup 改写成 $2k$ 元组 $(\sqcup, 0, \sqcup, 0, \dots, \sqcup, 0) \in \Sigma'$ 。

M' 像定理的陈述里说明的那样可模拟 M 的行为,这是清楚的。剩下要论证的是,为了模拟 M 在输入 x 上的 t 步, M' 需要的步数是 $\mathcal{O}(t \cdot (|x| + t))$ 。这个模拟的阶段(1)需要 M' 的 $\mathcal{O}(|x|)$ 步。然后,对于 M 的每步, M' 必须完成阶段(2)里的动作,即(a)和(b)。这个任务要求 M' 扫描两遍有 $2k$ 条轨道的那部分带,即,要求 M' 的步数与 M' 有 $2k$ 条轨道那部分带的长度成比例。问题是, M' 的这部分带可以有多长?起初它有 $|x| + 2$ 那么长,随后对于 M 被模拟的每步,它的长度增加不超过1。因此,若在输入 x 上模拟 M 的 t 步,则 M' 有 $2k$ 条轨道那部分带的长度最多是 $|x| + 2 + t$,所以 M 的每步可用 M' 的 $\mathcal{O}(|x| + t)$ 步模拟,像要证明的那样。 ■

利用对 k 带 Turing 机的输入和输出所描述的约定,下列结果从前一个定理轻易地推导出来。

推论 k 带 Turing 机计算的任何函数或者判定、半判定的任何语言,也分别可用标准 Turing 机计算或判定、半判定。

4.3.2 双向无穷带 Turing 机

现在假设机器的带在左右两个方向都是无穷的。除里面包含输入的那些带方格外,所有其余带方格起初都是空格。比方说,带头起初是在输入的左方。另外,对这样的机器,关于 \triangleright 符号的约定是不必要和无意义的。

不难看出,像多带一样,双向无穷带没有给 Turing 机添加重要的能力。双向无穷带可用 2 带机器轻易地模拟:一条带总是包含第一个输入符号所在方格右方的那部分带,另一条带用相反方向包含这个方格左方的那部分带。然后,这台 2 带机器可用标准 Turing 机模拟。事实上,模拟只需花费线性时间,而不是平方时间,因为在每步只有一条轨道是活动的。不言而喻,有多条双向无穷带的机器也可用同样方式模拟。

4.3.3 多带头 Turing 机

如果我们允许 Turing 机在仅有的一条带上有多只带头,那么发生什么事情?在一步里,带头都知道被扫描符号并且独立地移动或写。(必须采取某些约定,以规定当两只带头碰巧扫描同一个带方格,并且试图写不同的符号时,发生什么事情。也许让编号较低的带头有写的优先权。另外,让我们假设,带头感觉不到彼此在同一个带方格里的存在,例外的是,通过不成功的写,也许间接地能感觉到。)

不难看出,像我们用来模拟 k 带机器的那种模拟,可用来模拟在带上有多只带头的 Turing 机。基本想法又是把带划分成轨道,除一条轨道外,所有其余轨道都专门用来记录带头位置。为了模拟多带头机器的一步计算,必须把带扫描两遍:第一遍是为了发现所有带头位置上的符号,第二遍是为了改变这些符号或者适当地移动带头。像在定理 4.3.1 里那样,需要的步数又是平方的。

利用多带头,像利用多带一样,有时可极大地简化 Turing 机的构造。2 带头型的机复

制机 C (例 4.1.8), 可用比单带头型 (或者甚至 2 带型, 例 4.3.1) 自然得多的方式完成任务; 见习题 4.3.3。

4.3.4 二维带 Turing 机

Turing 机的另一种推广允许“带”是无穷的二维网格。(你甚至可允许更高维的空间。) 这样一台装置比标准 Turing 机更有利于解决问题, 比如“拼图玩具”(看下一章里的铺砖问题)。详细地定义这样的机器的操作, 我们把它留作练习 (习题 4.3.6)。不过, 又一次没有产生在能力上的根本性增加。有趣的是, 用普通 Turing 机模拟二维 Turing 机在输入 x 上的 t 步, 需要的步数又是 t 和 $|x|$ 的多项式。

在 Turing 机模型上的上述扩充可被组合起来; 可设想 Turing 机有多条带, 所有或部分的带都是双向无穷的并且上面有多只带头, 甚至都是多维的。尽管这样, 可相当直截了当地看出, Turing 机的根本能力还是保持原样。

我们总结迄今为止讨论过的 Turing 机的几个变种如下。

定理 4.3.2 有多带、多带头、双向无穷带或多维带的 Turing 机, 它们判定或半判定的任何语言以及计算的任何函数, 都分别可用标准 Turing 机判定、半判定或者计算。

习 题

4.3.1 形式化定义:

- (a) M 半判定 L , 其中 M 是双向无穷带 Turing 机;
- (b) M 计算 f , 其中 M 是 k 带 Turing 机, f 是从字符串到字符串的函数。

4.3.2 形式化定义:

- (a) k 带头 Turing 机 (只有一条单向无穷带);
- (b) 这样一台机器的格局;
- (c) 在这样一台机器的格局之间的一步产生关系。(存在多种正确的定义。)

4.3.3 描述 (用 k 带 Turing 机记号的扩充) 计算函数 $f(w) = ww$ 的 2 带头 Turing 机。

4.3.4 可认为下推自动机的栈是只能在右端写和删除的带; 在这种意义下 Turing 机是下推自动机的推广。在本题里我们考虑在另一个方向上的推广, 即双栈确定型下推自动机。

- (a) 非形式化但仔细地定义这样一台机器的操作。对这样一台机器, 定义它判定语言是什么意思。

- (b) 证明这样的机器判定的语言类恰恰是递归语言类。

4.3.5 给出 3 带 Turing 机, 当启动时, 第一条带上有用“,”分隔的两个二进制整数, 计算它们的乘积。(提示: 用例 4.3.2 的加法机作为“子程序”。)

4.3.6 形式化定义 2 维带 Turing 机和它的格局及它的计算。对这样一台机器, 定义它判定语言 L 是什么意思。证明在长度为 n 的输入上启动时, 这台机器的 t 步, 可用标准 Turing 机在 t 和 n 的多项式时间里模拟。

4.4 随机存取 Turing 机

我们迄今为止讨论过的 Turing 机的变种,尽管都有明显的能力和通用性,但也有产生相当局限性的共同特性:它们的内存是串行的,即为了访问保存在某个地址上的信息,机器必须首先逐个访问在当前地址和目标地址之间的所有地址。相反,真正的计算机有随机存取内存,只要适当地建立了地址,就可在一步里访问内存的任意一个单元。假如我们用这样一种随机存取能力装备机器,使它们能在一里访问需要的任意带方格,那么可能发生什么事情?为了获得这样一种能力,我们还必须给机器装备寄存器,让它们能保存并操纵带方格的地址。在本节里我们定义 Turing 机的这样一种扩充;颇有意义的是,我们看到,它也在能力上等价于标准的 Turing 机,在效率上只有多项式那么大的损失。

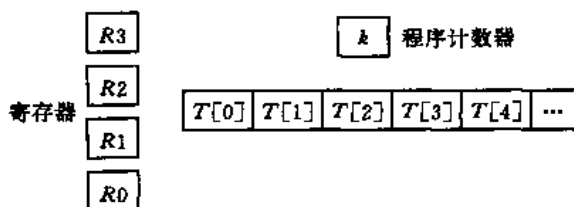


图 4-18

随机存取 Turing 机有固定多个寄存器和一条单向无穷带(如图 4-18 所示;尽管事实上我们将看到,它的行为更像是随机存取的内存芯片,但是为了保持与标准模型的兼容性和可比性,我们继续把机器的内存称为“带”)。每个寄存器和每个带方格可容纳任意的自然数。机器在带方格和寄存器上操作,像是由固定的程序——相当于普通 Turing 机的转移函数——所控制那样。随机存取 Turing 机的程序是指令的序列,有点像是真正计算机的指令集。允许的各种指令都列举在图 4-19 里。

起初,寄存器值都是 0,程序计数器是 1,带内容是输入字符串的编码,稍后将说明编码的简单方式。然后,机器执行程序的第一条指令。像图 4-19 里说明的那样,这将改变寄存器的内容或者带的内容;另外,程序计数器 κ 的值是一个整数,它指明下一步要执行的指令,它像图中说明的那样被计算出来。注意寄存器 0 的特殊作用:它是累加器,所有算术和逻辑计算发生在这里。下一步,执行程序的第 κ 条指令,等等,直到执行 halt 指令为止——在这个时刻随机存取 Turing 机的操作结束。

我们现在准备好了去形式化定义随机存取 Turing 机和它的格局及它的计算。

定义 4.4.1 随机存取 Turing 机 是二元组 $M = (k, \Pi)$, 其中 $k > 0$ 是寄存器的个数, $\Pi = (\pi_1, \pi_2, \dots, \pi_p)$, 即程序,是指令的有穷序列,其中每条指令 π_i 是图 4-19 所示类型之一。假设最后一条指令 π_p 总是 halt 指令(程序也可包含其他的 halt 指令)。

随机存取 Turing 机 (k, Π) 的格局是 $k + 2$ 元组 $(\kappa, R_0, R_1, \dots, R_{k-1}, T)$, 其中 $\kappa \in \mathbb{N}$ 是程序计数器,是在 0 与 p 之间的整数。若 κ 是零,则格局是停机格局。

对每个 $j, 0 \leq j < k, R_j \in \mathbb{N}$ 是寄存器 j 的当前值。

指令	操作数	语义
read	j	$R_0 := T[R_j]$
write	j	$T[R_j] := R_0$
store	j	$R_j := R_0$
load	j	$R_0 := R_j$
load	$= c$	$R_0 := c$
add	j	$R_0 := R_0 + R_j$
add	$= c$	$R_0 := R_0 + c$
sub	j	$R_0 := \max\{R_0 - R_j, 0\}$
sub	$= c$	$R_0 := \max\{R_0 - c, 0\}$
half		$R_0 := \lfloor \frac{R_0}{2} \rfloor$
jump	s	$\kappa := s$
jpos	s	if $R_0 > 0$ then $\kappa := s$
jzero	s	if $R_0 = 0$ then $\kappa := s$
halt		$\kappa := 0$

注意, j 表示寄存器编号, $0 \leq j < k$. $T[i]$ 表示带方格 i 的当前内容, R_j 表示寄存器 j 的当前内容, $s \leq p$ 表示程序里的任意指令编号. c 是任意自然数. 除非另有明确说明, 否则所有指令都把 κ 变成 $\kappa + 1$.

图 4-19

T , 即带内容, 是正整数有序对的有穷集, 即 $(\mathbb{N} - \{0\}) \times (\mathbb{N} - \{0\})$ 的有穷子集, 使得对所有 $i \geq 1$, 最多存在一个形如 $(i, m) \in T$ 的有序对。

在直观上, $(i, m) \in T$ 意味着第 i 个带方格当前包含整数 $m > 0$. 不在 T 里有序对第一分量上出现的所有带方格都被假定包含 0。

定义 4.4.1(续) 设 $M = (\kappa, \Pi)$ 是随机存取机器. 设 $C = (\kappa, R_0, R_1, \dots, R_{k-1}, T)$ 和 $C' = (\kappa', R'_0, R'_1, \dots, R'_{k-1}, T')$ 是 M 的两个格局. 若在直观上, κ' 与诸 R'_i 和 T' 的值, 正确地反映了把当前指令 π_κ 的“语义”(像图 4-19 里那样) 应用到 κ 与诸 R_j 和 T 上的效果, 则我们说格局 C 一步产生格局 C' , 表示成 $C \vdash C'$. 对图 4-19 里的十四种指令, 我们只详细说明其中几种指令的定义。

若 π_κ 形如 read j , 其中 $j < k$, 则这条指令的执行有下列效果: 寄存器 0 包含的值变成等于编号为 R_j 的带方格 —— 用寄存器 j “建立地址”的带方格 —— 保存的值. 即, $R'_0 = T[R_j]$, 其中若满足 $(R_j, m) \in T$ 的唯一值 m 存在, 则 $T[R_j]$ 是这样一个 m , 否则它是 0. 另外, $\kappa' = \kappa + 1$. 格局 C' 的所有其余分量都与 C 相同。

若 π_κ 形如 add $= c$, 其中 $c \geq 0$ 是固定的整数, 比如 5, 则 $R'_0 = R_0 + c$, 并且 $\kappa' = \kappa + 1$, 所有其余分量都保持原样。

若 π_κ 形如 write j , 其中 $j < k$, 则 $\kappa' = \kappa + 1$, T' 是经过如下变化的 T : 若存在形如 $(R_j, m) \in T$ 的有序对, 则删除任何这样的有序对, 并且, 若 $R_0 > 0$, 则添加有序对 (R_j, R_0) ; 所有其余分量都保持原样。

若 π_κ 形如 jpos s , 其中 $1 \leq s \leq p$, 则若 $R_0 > 0$ 则 $\kappa' = s$, 否则 $\kappa' = \kappa + 1$; 所有其余

分量都保持原样。

可类似地说明其他种类的指令。产生关系表示成 \vdash_M , 是 \vdash_M 的自反传递闭包。

例 4.4.1 随机存取 Turing 机的指令集(回忆图 4-19)没有乘法指令 mply。因为事实上, 假如我们允许这条指令作为基本指令, 那么随机存取 Turing 机尽管仍然等价于标准 Turing 机, 但模拟起来要费时得多(见习题 4.4.4)。

不过, 省略乘法指令并不是很大的损失, 因为这条指令可用图 4-20 所示的程序模拟。如果①寄存器 0 起初包含自然数 x , 并且寄存器 1 起初包含 y , 那么这台随机存取 Turing 机将停机, 并且寄存器 0 将包含乘积 $x \cdot y$ 。乘法是通过连续加法来完成的, 其中指令 half 用来处理 y 的二进制表示(实际上, 恰恰是为了这种用途, 我们的指令集才包含了这条不同寻常的指令)。

1. store 2	6. load 1	11. aad 2	16. load 3
2. load 1	7. sub 3	12. store 4	17. store 1
3. jzero 19	8. sub 3	13. load 2	18. jump 2
4. half	9. jzero 13	14. aad 2	19. load 4
5. store 3	10. load 4	15. store 2	20. halt

图 4-20

下面是典型的格局序列(因为这台机器不与带方格交互作用, 所以这些格局的 T 部分是空的; 存在 $\kappa = 5$ 个寄存器):

$(1; 5, 3, 0, 0, 0; \emptyset) \vdash (2; 5, 3, 5, 0, 0; \emptyset) \vdash (3; 3, 3, 5, 0, 0; \emptyset) \vdash (4; 3, 3, 5, 0, 0; \emptyset) \vdash$
 $(5; 1, 3, 5, 0, 0; \emptyset) \vdash (6; 1, 3, 5, 1, 0; \emptyset) \vdash (7; 3, 3, 5, 1, 0; \emptyset) \vdash (8; 2, 3, 5, 1, 0; \emptyset) \vdash$
 $(9; 1, 3, 5, 1, 0; \emptyset) \vdash (10; 1, 3, 5, 1, 0; \emptyset) \vdash (11; 0, 3, 5, 1, 0; \emptyset) \vdash$
 $(12; 5, 3, 5, 1, 0; \emptyset) \vdash (13; 5, 3, 5, 1, 5; \emptyset) \vdash (14; 5, 3, 5, 1, 5; \emptyset) \vdash$
 $(15; 10, 3, 5, 1, 5; \emptyset) \vdash (16; 10, 3, 10, 1, 5; \emptyset) \vdash (17; 1, 3, 10, 1, 5; \emptyset) \vdash$
 $(18; 1, 1, 10, 1, 5; \emptyset) \vdash (2; 1, 1, 10, 1, 5; \emptyset) \vdash^* (18; 0, 0, 20, 0, 15; \emptyset) \vdash$
 $(2; 0, 0, 20, 0, 15; \emptyset) \vdash (3; 0, 0, 20, 0, 15; \emptyset) \vdash (19; 0, 0, 20, 0, 15; \emptyset) \vdash$
 $(20; 15, 0, 20, 0, 15; \emptyset)$

设 x 和 y 分别是寄存器 0 和 1 在这个程序开始执行时保存的非负整数。我们断言, 机器最终停机, 并且乘积 $x \cdot y$ 保存在寄存器 0 里——就好像它执行指令“mply 1”。程序进行若干次迭代。一次迭代是执行从 π_2 到 π_{18} 的指令序列。在第 k 次($k \geq 1$)迭代时, 下列条件成立:

- (a) 寄存器 2 包含 $x2^k$,
- (b) 寄存器 3 包含 $\lfloor y/2^k \rfloor$,
- (c) 寄存器 1 包含 $\lfloor y/2^{k-1} \rfloor$,
- (d) 寄存器 4 包含“部分结果” $x \cdot y \pmod{2^k}$ 。

迭代竭力保持这些“不变条件”。因此, 假设在前一次迭代里不变条件(c)成立, 则从 π_2 到 π_5 的指令使得(b)成立。从 π_6 到 π_8 的指令计算 y 的倒数第 k 位, 并且, 若这一位不是 0, 则从 π_9 到 π_{12} 的指令, 像不变条件(d)所规定的那样, 把 $x2^{k-1}$ 加到寄存器 4 上。然后从

① 当随机存取 Turing 机的计算启动时, 所有寄存器都是 0。不过, 因为目前的程序打算被用作其他随机存取 Turing 机的一部分, 所以有理由去探讨假如在任意格局上启动它时, 可能发生什么事情。

π_{13} 到 π_{15} 的指令,把寄存器 2 加倍使得不变条件(a) 成立。最后,寄存器 3 的值被传递到寄存器 1,使得(c) 成立。然后迭代被重复。如果在某个时刻发现 $\lfloor y/2^{t-1} \rfloor = 0$,那么过程终止,并且从寄存器 4 把最终结果载入到累加器。

我们可把这段程序缩写成“mply 1”,即,具有语义 $R_0 := R_0 \cdot R_1$ 的指令。因此,我们允许在程序里使用指令“mply j ”或“mply $= c$ ”,因为我们知道可用上面的程序模拟它们。自然,指令编号可能不同,这取决于包含这条 mply 指令的程序。如果随机存取 Turing 机使用这条指令,那么隐含地假定,除明确指出的寄存器外,还必须要有另外三个寄存器,它们起到上面程序里寄存器 2,3 和 4 的作用。◇

事实上,通过采用某些有用的缩写,我们可避免随机存取 Turing 机程序那种笨头笨脑的样子。例如通过用 R_1 表示寄存器 1 保存的值,用 R_2 表示寄存器 2 的值,等等,我们可以用

$$R_1 := R_2 + R_1 - 1$$

来作为下面序列的缩写

1. load 1
2. add 2
3. sub = 1
4. store 1

一旦我们采取这种缩写,就可用更好看的名字表示寄存器 1 和 2 保存的量,并且把这个指令序列简单地表达成

$$x := x + y - 1$$

在这里 x 和 y 仅仅是寄存器 1 和 2 的内容的名字。我们甚至可用像

$$\text{while } x > 0 \text{ do } x := x - 3$$

(其中 x 表示寄存器 1 的值) 这样的缩写去代替序列

1. load 1
2. jzero 6
3. sub = 3
4. store 1
5. jump 1

例 4.4.1(续) 这里是对图 4-20 里 mply 程序的可读性更好的缩写,其中假设 x 和 y 是被乘数,乘积是 w :

```

w := 0
while y > 0 do
  begin
    z := half(y)
    if y - z - z ≠ 0 then w := w + x
    x := x + x
    y := z
  end
halt

```

在上述形式和图 4-20 的原始程序之间的对应关系是这样的: y 表示 R_1 , x 表示 R_2 , z 表示 R_3 , w 表示 R_4 。注意为了清楚起见,我们也省略了明确的指令编号;假如 goto 指令是必要的,则可用像 a 和 b 那样的符号化指令标号,在每个必要的地方建立指令标号。

自然,从缩写程序,比如上面的程序,得到等价的完整随机存取 Turing 机程序,比如原来的程序,是相当机械的过程。◇

虽然我们已解释了随机存取 Turing 机的机制,但还没有讲述如何接收输入和返回输出。为了便于与标准 Turing 机进行比较,我们假设随机存取 Turing 机的输入输出约定与普通 Turing 机的输入输出约定在特点上非常一致:输入表示成带里的符号序列。即,虽然随机存取 Turing 机的带可包含任意自然数,但我们假定起初这些数是某个输入字符串的符号的编码。

定义 4.4.2 让我们固定字母表 Σ ——随机存取 Turing 机将从这个字母表获得输入——满足 $\sqcup \in \Sigma$ 并且 $\triangleright \notin \Sigma$ (在这里不需要 \triangleright , 因为随机存取 Turing 机没有带头移出带左端的危险)。另外,设 E 是在 Σ 和 $\{0, 1, \dots, |\Sigma| - 1\}$ 之间的固定双射,这个双射是我们编码随机存取 Turing 机的输入和输出的方式。我们假设 $E(\sqcup) = 0$ 。随机存取 Turing 机 $M = (k, \Pi)$ 在输入 $w = a_1 a_2 \dots a_n$ 上的初始格局是 $(\kappa, R_0, \dots, R_{k-1}, T)$, 其中 $\kappa = 1$, 对所有 $j, R_j = 0$, 并且 $T = \{(1, E(a_1)), (2, E(a_2)), \dots, (n, E(a_n))\}$ 。

若在输入字符串 $x \in \Sigma^*$ 上的初始格局产生满足 $R_0 = 1$ 的停机格局,则我们说 M 接受 x 。若在输入 x 上的初始格局产生满足 $R_0 = 0$ 的停机格局,则我们说 M 拒绝 x 。换句话说,一旦 M 停机,我们就在寄存器 0 中读到判定结果:若这个值是 1,则机器接受;若值是 0,则它拒绝。

设 $\Sigma_0 \subseteq \Sigma - \{\sqcup\}$ 是字母表,并设 $L \subseteq \Sigma_0^*$ 是语言。若每当 $x \in L$, M 就接受 x ; 每当 $x \notin L$, M 就拒绝 x , 则称 M 判定 L 。若下列关系为真: $x \in L$ 当且仅当 M 在输入 x 上产生某个停机格局,则称 M 半判定 L 。

最后,设 $f: \Sigma_0^* \rightarrow \Sigma_0^*$ 是函数。若对所有 $x \in \Sigma_0^*$, 机器 M 在输入 x 上的初始格局产生停机格局,并且带的内容是 $\{(1, E(a_1)), (2, E(a_2)), \dots, (n, E(a_n))\}$, 其中 $f(x) = a_1 \dots a_n$, 则称 M 计算 f 。

例 4.4.2 下面用缩写形式描述随机存取 Turing 机程序,它判定语言 $\{a^n b^m c^n : n \geq 0\}$ 。

```

account := bcount := ccount := 0, n := 1
while T[n] = 1 do n := n + 1, account := account + 1
while T[n] = 2 do n := n + 1, bcount := bcount + 1
while T[n] = 3 do n := n + 1, ccount := ccount + 1
if account = bcount = ccount and T[n] = 0 then accept else reject

```

我们在这里假设 $E(a) = 1, E(b) = 2, E(c) = 3$, 分别用变量 *account*, *bcount* 和 *ccount* 表示迄今为止找到的 a, b 和 c 的个数。我们也用缩写 *accept* 表示“load = 1, halt”, *reject* 表示“load = 0, halt”。◇

例 4.4.3 为了说明更重要的例子,现在我们描述计算有穷二元关系自反传递闭包(回忆 1.6 节)的随机存取 Turing 机。给定有向图 $R \subseteq A \times A$, 其中 $A = \{a_0, \dots, a_{n-1}\}$, 我们希望计算 R^* 。

马上产生一个重要问题:如何把关系 $R \subseteq A \times A$ 表示成字符串? R 可表示成相邻矩阵

A_R , 即 0-1 项的 $n \times n$ 矩阵, 使得第 i, j 项是 1 当且仅当 $(a_i, a_j) \in R$ (见图 4-21 的例子)。然后, 通过首先安排矩阵的第一行, 然后第二行, 等等, 相邻矩阵可表示成 $\{0, 1\}^*$ 里长度为 n^2 的字符串。我们把关系 R 的相邻矩阵的字符串表示记作 x_R 。例如, 若 R 是图 4-21(a) 所示的二元关系, 则 A_R 和 x_R 分别如图 4-21(b) 和 (c) 所示。

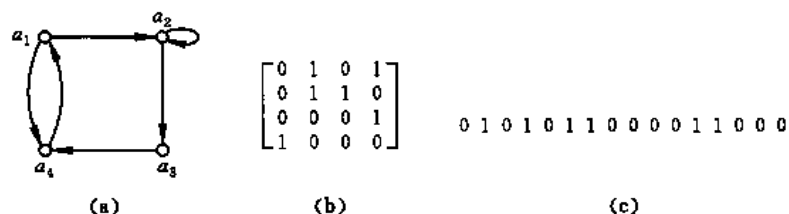


图 4-21 一个图, 它的相邻矩阵及其字符串表示

因此必须设计随机存取 Turing 机 M , 它计算如下定义的函数 f : 对在某个有穷集 $\{a_1, \dots, a_n\}$ 上的任意二元关系 R , $f(x_R) = x_R$ 。注意, 我们不关心 M 如何回答不属于 $\{0, 1\}^{n^2}$ 的输入, 即不关心不表示有向图相邻矩阵的输入。

程序 M 如下所示。假设 $E(0) = 1, E(1) = 2$, 以及像通常那样, $E(\sqcup) = 0$ 。

```

n := 1
while T[n · n] ≠ 0 do n := n + 1
n := n - 1
i := 0
while i < n do i := i + 1, T[i · n + i] := 2
i := j := k := 0
  while j < n do j := j + 1,
    while i < n do i := i + 1,
      while k < n do k := k + 1,
        if T[i · n + j] = 2 and T[j · n + k] = 2 then T[i · n + k] := 2
halt

```

前三条指令计算集合 A 的元素个数 n (它是输入带里空格的最小地址的平方根减一)。从那时起, 就可从 M 带上的第 $i \cdot n + j$ 个符号取出矩阵的第 (i, j) 项, 其中 $0 \leq i, j < n$ 。因为这个程序是 1.6 节里 $O(n^3)$ 算法的直截了当实现, 所以它确实计算输入表示的关系的自反传递闭包。自然, 从上述程序可机械地推导出非缩写的随机存取 Turing 机程序。

◇

显然, 随机存取 Turing 机是非同寻常地又强又快的模型。与标准 Turing 机相比它的能力如何? 容易看出, 随机存取 Turing 机至少像标准 Turing 机一样强。这毫不奇怪。设 $M = (K, \Sigma, \delta, s, H)$ 是 Turing 机, 可设计随机存取 Turing 机 M' 模拟 M 。 M' 有一个寄存器, 称为 n , 跟踪 M 的带头在带上的位置。起初 n 指向输入的开头。每个状态 $q \in K$ 被 M' 程序里的指令序列模拟。例如, 假设 $\Sigma = \{\sqcup, a, b\}$, $E(a) = 1, E(b) = 2$, 并设 q 是 M 的状态使得 $\delta(q, \sqcup) = (p, \rightarrow), \delta(q, a) = (p, \leftarrow), \delta(q, b) = (r, \sqcup)$ 以及 $\delta(q, \triangleright) = (s, \rightarrow)$ 。模拟状态 q 的指令序列如下所示

$q, \text{if } T[n] = 0 \text{ then } n := n + 1, \text{goto } p$
 $\text{if } T[n] = 1 \text{ then if } n > 0 \text{ then } n := n - 1, \text{goto } p$
 $\text{else goto } s$
 $\text{if } T[n] = 2 \text{ then } T[n] := 0, \text{goto } r$

第三行的 else 子句(它应当出现在模拟 \leftarrow 移动的任何行里)具有 \triangleright 的效果,它确保带头永不移出 M' 带的左端。我们已证明了:

定理 4.4.1 任何递归或递归可枚举语言,以及任何递归函数,分别可用随机存取 Turing 机判定、半可判定和计算。

值得注意的方向是逆命题:

定理 4.4.2 随机存取 Turing 机判定或半判定的任何语言,以及随机存取 Turing 机计算的任何函数,分别可用标准 Turing 机判定、半判定和计算。另外,如果随机存取机器在输入上停机,那么标准 Turing 机所花费的步数不超过随机存取 Turing 机在同一个输入上步数的多项式。

证明: 设 $M = (k, \Pi)$ 是随机存取 Turing 机,它判定或半判定语言 $L \subseteq \Sigma^*$,或者计算从 Σ^* 到 Σ^* 的函数。我们简述对模拟 M 的普通 Turing 机 M' 的设计。我们把 M' 描述成模拟 M 的 $(k+3)$ 带 Turing 机,其中 k 是 M 的寄存器个数。由定理 4.3.1 知, M' 可用基本模型模拟。

Turing 机 M' 跟踪随机存取 Turing 机 M 的当前格局,并反复地计算后继格局。第一条带只用来读 M 的输入,并且在 M 计算函数的情形里可能用来在结束时报告输出。第二条带用来跟踪格局的 T 部分—— M 的带内容。关系 T 始终是形如 $(111, 10)$ 的字符串的序列,即左圆括号后面依次跟着整数的二进制表示,逗号,另一个二进制整数,右圆括号等。上述字符串的预期含义是 M 的第七个带方格包含整数 2。在表示 T 的序列里的整数对,没有任何特殊的顺序,并且可用任意长的空格序列来间隔(这样一来,在 T 的表示的结尾就有必要用结束符,比如 \$,去帮助 M' 判定到何时为止它已看见了所有这样的整数对)。 M' 的下面 k 条带里每条都保存一个 M 寄存器的内容,也用二进制。 M 的程序计数器 κ 的当前值用下面解释的方法保存在 M' 的状态里。

模拟有三个阶段。在第一阶段里, M' 在第三条带上收到输入 $x = a_1 a_2 \cdots a_n \in \Sigma^*$,并在第二条带上把它转换成字符串 $(1, E(a_1)) \cdots (n, E(a_n))$ 。于是, M' 可从 M 在输入 x 上的初始格局开始第二阶段,对 M 进行模拟。

在第二阶段里, M' 反复地用它自己的若干步模拟 M 的一步。被模拟步的具体性质严重依赖于 M 的程序计数器 κ 。像我们在前面说过的那样, κ 被保存在 M' 的状态里。即,在这个阶段里使用的 M' 的状态的集合,被分成 p 个不相交的集合 $K_1 \cup K_2 \cup \cdots \cup K_p$,其中 p 是 M 的程序 Π 里的指令个数。状态集 K_j “专门”模拟 Π 的指令 π_j 。 M' 这个部分的具体性质当然依赖于指令 π_j 的类型。我们给出三个例子说明如何这样做。

首先假设 π_j 是 add 4,要求把寄存器 4 的内容加到寄存器 0 的内容上。于是 M' 在表示寄存器 4 和 0 的两条带之间完成二进制加法(回忆例 4.3.2),把结果留在表示寄存器 0 的带上,然后转移到 K_{j+1} 的第一种状态去启动对下一条指令的模拟。如果这条指令是,比方说, add = 33,那么 M' 首先把整数 33 用二进制写在第 $(k+3)$ 条带上(在此之前没有分配

这条带去表示 M 的任何部分); K_j 的状态“记住”固定整数 33 的二进制表示。然后, 它把 33 加到寄存器 0 的内容上。最后, 它删除最后这条带并转移到 K_{j+1} 。

其次假设 π_j 是 write 2, 要求把累加器的内容复制到寄存器 2 指示的带方格里。于是 M' 在第二条带的右端—— M 的带内容用 (x, y) 格式保存在这条带上——添加整数对 (x, y) , 其中 x 是寄存器 2 的内容而 y 是寄存器 0 的内容; x 和 y 都从 M' 的对应带上复制过来。然后 M' 在第二条带上扫描所有其他整数对 (x', y') , 逐位比较每个 x' 和寄存器 i 的内容。若发现匹配, 则删除匹配的整数对, 因此保持表的完整性。然后状态转移到 K_{j+1} , 并且执行下一条指令。

最后假设 π_j 是 jpos 19, 要求若寄存器 0 包含正整数, 则下一步执行指令 19。Turing 机 M' 仅仅扫描表示寄存器 0 的带; 若在里面的整数的二进制表示里发现 1, 则 M' 转移到 K_{19} ; 否则它转移到 K_{j+1} 。

用非常相似的方式模拟图 4-19 的表里所有其他种类的指令, 这是直截了当的。最终, M 可能到达 halt 指令。若这样的事情发生, 则 M' 进入第三阶段, 把 M 的输出翻译成 Turing 机的输出约定。若 M 判定语言, 则 M' 读寄存器 0 的内容。若内容是 1, 则 M' 在状态 y 停机; 若内容是 0, 则它在状态 n 停机。若 M 半判定语言, 则 M' 仅仅在状态 h 停机。最后, 若 M 计算函数, 则 M' 必须把 M 的带内容翻译成 Σ^* 里的字符串, 即逆转双射 E , 然后停机。

通过前面的讨论, 可设计 $k+3$ 带 Turing 机完成上述任务, 这是清楚的。因此, 根据定理 4.3.1, 标准 Turing 机也可完成上述任务。

为了证明定理的第二部分, 我们证明可在 $\mathcal{O}(t+n)^3$ 时间里模拟 M 在规模为 n 的输入上的 t 步。自然, 在 \mathcal{O} 记号里的常数像通常那样依赖于被模拟机器 M , 例如依赖于 M 程序里涉及的最大常数(像在 $\text{add} = 314159$ 里那样)。

$\mathcal{O}(t+n)^3$ 的界限基于下列三个观察事实:

(a) M 的每步(包括加法和减法, 见习题 4.4.3)可用 M' 的 $\mathcal{O}(m)$ 步模拟, 其中 m 是 M' 所有带的非空格部分的总长度, 即 M 当前格局里所有整数的二进制编码的总长度。

(b) 上面定义的参数 m 在每步上最多可增加 $\mathcal{O}(r)$, 其中 r 是 M 寄存器或带方格里保存的任何整数的最长二进制表示的长度。这是因为, 增加或者来自 add 指令, 或者来自 store 指令, 在这两种情形里, 容易看出增加只能是对 r 线性的。

(c) 最后, 容易看出 $r = \mathcal{O}(t)$, 即 M 表示的最大整数的长度每步只能增加常量。通过把这三个事实放在一起就得出所断言的界限。 ■

习 题

- 4.4.1 具体给出例 4.4.2 的随机存取 Turing 机程序的完整细节。给出这台机器在输入 $aabccc$ 上的格局序列。
- 4.4.2 给出(用缩写记号)判定语言 $\{wcw; w \in \{a, b\}^*\}$ 的随机存取 Turing 机程序。
- 4.4.3 证明在定理 4.4.2 的模拟里, 每步可用 M' 的 $\mathcal{O}(m)$ 步模拟, 其中 m 是 M' 带的总长度。(你必须证明例 4.3.2 里的 2 带加法 Turing 机在线性时间里操作。)你能否估计出 $\mathcal{O}(m)$ 里的常数?

4.4.4 假如随机存取 Turing 机有指令 mply, 那么在定理 4.4.2 的证明的第二部分里有什么地方出现错误?

4.5 非确定型 Turing 机

我们已向 Turing 机添加了许多似乎强大的特性——多带和多带头, 甚至随机存取——但是在能力上没有明显的增强。不过, 我们还没有尝试重要而熟悉的特性: 非确定性。

我们已看到, 虽然当允许有穷自动机非确定性地操作时, 没有产生计算能力的增强 (例外的是, 同样的任务所需要的状态数可能指数地减少), 但是非确定型下推自动机比确定型下推自动机更强。我们也可设想非确定性地操作的 Turing 机: 在状态和扫描符号的某种组合上, 这样的机器可能有多种动作选择。形式化地, 非确定型 Turing 机是五元组 $(K, \Sigma, \Delta, s, H)$, 其中 K, Σ, s 和 H 与标准 Turing 机相同, 并且 Δ 是 $((K - H) \times \Sigma) \times (K \times (\Sigma \cup \{\leftarrow, \rightarrow\}))$ 的子集, 而不是从 $(K - H) \times \Sigma$ 到 $K \times (\Sigma \cup \{\leftarrow, \rightarrow\})$ 的函数。用自然的方式定义格局以及关系 \vdash_M 和 \vdash_M^* 。但是现在 \vdash_M 不必是单值的: 一个格局可在一步里产生多个其他格局。

当允许 Turing 机非确定性地操作时, 在计算能力上是否有所增加呢? 我们必须首先定义非确定型 Turing 机计算某样东西意味着什么。因为非确定型机器从同样的输入上可能产生两个不同的输出或终止状态, 所以我们不得不仔细考虑什么东西是这样一台机器的最终计算结果。因为这个原因, 首先考虑半判定语言的非确定型 Turing 机, 这是最容易的事情。

定义 4.5.1 设 $M = (K, \Sigma, \Delta, s, H)$ 是非确定型 Turing 机。若对输入 $w \in (\Sigma - \{\triangleright, \sqcup\})^*$, 对某个 $h \in H$ 以及 $a \in \Sigma, u, v \in \Sigma^*, (s, \triangleright \sqcup w) \vdash_M^*(h, u \sqcup v)$, 则我们说 M 接受 w 。注意即使非确定型 Turing 机在输入上有许多非停机计算, 它也接受这个输入——只要存在至少一种停机格局。若对语言 $L \subseteq (\Sigma - \{\triangleright, \sqcup\})^*$, 对所有 $w \in (\Sigma - \{\triangleright, \sqcup\})^*$, 下列关系成立: $w \in L$ 当且仅当 M 接受 w , 则我们说 M 半判定 L 。

定义非确定型 Turing 机判定语言或计算函数的意思更微妙些。

定义 4.5.2 设 $M = (K, \Sigma, \Delta, s, (y, n))$ 是非确定型 Turing 机。设 $L \subseteq (\Sigma - \{\triangleright, \sqcup\})^*$ 是语言, 若对所有 $w \in (\Sigma - \{\triangleright, \sqcup\})^*$, 下列两个条件成立:

- (a) 存在自然数 N , 它依赖于 M 和 w , 使得不存在任何格局 C 满足 $(s, \triangleright \sqcup w) \vdash_M^N C$ 。
- (b) $w \in L$ 当且仅当对某个 $u, v \in \Sigma^*, a \in \Sigma, (s, \triangleright \sqcup w) \vdash_M^*(y, u \sqcup v)$ 。

则我们说 M 判定 L 。

最后, 设 $f: (\Sigma - \{\triangleright, \sqcup\})^* \rightarrow (\Sigma - \{\triangleright, \sqcup\})^*$ 是函数, 若对所有 $w \in (\Sigma - \{\triangleright, \sqcup\})^*$, 下列两个条件成立:

- (a) 存在 N , 依赖于 M 和 w , 使得不存在任何格局 C 满足 $(s, \triangleright \sqcup w) \vdash_M^N C$ 。
- (b) $(s, \triangleright \sqcup w) \vdash_M^*(h, u \sqcup v)$ 当且仅当 $ua = \triangleright \sqcup$, 并且 $v = f(w)$ 。

则我们说 M 计算 f 。

这些定义反映出用非确定型 Turing 机计算的困难。首先注意, 对判定语言和计算函

数的非确定型机器,我们要求它们所有的计算都停机;我们通过不让任何计算继续到 N 步之后来达到这个要求,其中 N 是依赖于机器和输入的“上界”(这是上述条件(a))。其次,对判定语言的 M ,我们只要求至少有一种可能的计算在结束时接受输入。其余的计算里的某些、大多数或者全部都可能在结束时拒绝——这个单独的接受计算就迫使机器接受。这是非常不寻常、非对称和反直觉的约定。例如为了构造判定补语言的机器,仅仅反转状态 y 和 n 的作用是不够的(因为对某些输入,机器既有接受计算又有拒绝计算)。像对非确定型有穷自动机那样,为了证明非确定型 Turing 机判定的语言类在补运算下封闭,我们必须求助于等价的确定型 Turing 机——本节的主要结果(定理 4.5.1)说明等价的确定型 Turing 机必定存在。

最后,对计算函数的非确定型 Turing 机,我们要求所有可能的计算都有一致的输出。假如不一致,我们就无法判定哪个输出是正确的函数值(在判定或半判定语言的情形里,我们通过规定肯定的答案优先来消除这种不确定性)。

在证明可从 Turing 机里消除非确定性之前,让我们考虑经典的例子,它说明非确定型 Turing 机作为想象中的装置的能力。

例 4.5.1 合数是两个大于 1 的自然数的乘积。例如 4, 6, 8, 9, 10 和 12 都是合数,而 1, 2, 3, 5, 7 和 11 都不是合数。换句话说,合数是既不等于 1 也不等于 0 的非素数。

设 $C = \{100, 110, 1000, 1001, 1010, \dots, 1011011, \dots\}$ 是全体合数的二进制表示的集合。设计判定 C 的“有效”算法是一个古老、重要而困难的问题。为了设计这样的算法,似乎有必要提出聪明的方法去发现数的因子(假如有因子),看起来这是一项非常复杂的任务。自然,通过穷举搜索所有小于给定数的数(事实上,小于它的平方根),我们可最终发现它的因子;难点在于,不清楚是否有更直接的方法。

不过,若可利用非确定性,则通过猜测因子(假如有因子的话),我们可设计出相当简单地半判定 C 的机器。当给定二进制表示的整数 n 作为输入时,机器操作如下:

(1) 非确定性地选择两个大于 1 的二进制数 p, q , 一位一位地给出,并且把它们的二进制表示写在输入后面。

(2) 利用习题 4.3.5 的乘法机(其实是模拟乘法机的单带机器),把 p 和 q 的二进制表示改成它们的乘积的二进制表示。

(3) 验证两个整数 n 和 $p \cdot q$ 相等。通过逐位比较它们,可轻易地完成这个验证。若这两个整数相等则停机;否则用某种方式永远继续下去(回忆一下,目前我们只关心半判定 C 的机器)。

在输入 1000010 上(66 的二进制表示),这台机器有许多拒绝(非停机)计算,它们分别对应于在上述阶段(1)里选择不能乘出 66 的二进制整数对,比如 101, 11101。要点在于,因为 66 是合数,所以 M 至少有一个计算最终接受——这是我们的全部要求。事实上,存在多种接受计算(分别对应于 $2 \cdot 33 = 6 \cdot 11 = 11 \cdot 6 = 33 \cdot 2 = 66$)。然而,假如输入是 1000011,就没有计算最终接受,因为 67 是素数。

这台机器可修改成判定语言 C 的非确定型机器。判定机器有相同的基本结构,差别在于,在阶段(1)里新机器永远不猜测比 n 自身更长的整数。显然,这样一个整数不是 n 的因子。在阶段(3)里,在比较输入和乘积之后,若它们相等则新机器在状态 y 停机,否则在状

态 n 停机。作为结果,在某个有穷步数之后,所有计算都最终停机。

定义 4.5.2 要求的上界 N 现在容易被具体地计算出来。假设给定的整数 n 有 l 位。设 N_1 是乘法机在不超过 $2l + 1$ 长度的输入上的最大计算步数;这个数是有穷数,它是有穷多个自然数中的最大值。设 N_2 是 M' 比较两个长度不超过 $3l$ 的字符串所花费的步数。于是 M' 的任何计算都在 $N_1 + N_2 + 3l + 6$ 步之后停机,它肯定是只依赖于机器和输入的有穷数。 \diamond

非确定性似乎是难以轻易被消除的非常强的特性。确实,似乎没有用确定型 Turing 机用步对步的方式模拟非确定型 Turing 机的容易方法,像对我们迄今为止检查过的所有其他增强型 Turing 机所做的那样。但是,事实上非确定型 Turing 机半判定或判定的语言分别与确定型 Turing 机半判定或判定的语言没有任何不同。

定理 4.5.1 如果非确定型 Turing 机 M 半判定或判定语言 L ,或者计算函数 f ,那么存在标准型 Turing 机 M' 半判定或判定语言 L ,或者计算函数 f 。

证明: 我们对 M 半判定语言 L 的情形描述 M 的构造,对判定语言或者计算函数的情形的构造是非常相似的。设 $M = (K, \Sigma, \Delta, s, H)$ 是半判定语言 L 的非确定型 Turing 机。给定输入 w , M' 试图系统地检查 M 所有可能的计算,去寻找一个停机计算。当它发现停机计算时,它也停机。所以, M' 停机当且仅当 M 停机,像所要求的那样。

但是从同样输入开始 M 有无穷多种不同的计算; M' 如何把它们全部探索一遍呢?它通过使用制作棒头过程(回忆图 1-8 里说明的方法)来这样做。关键的观察结果如下:虽然对 M 的任何格局 C ,存在多个格局 C' 使得 $C \vdash C'$,但是这样的格局 C' 的数目是固定和有界的,并且它只依赖于 M 而不依赖于 C 。具体地说,在任何时刻可用的四元组 $(q, a, p, b) \in \Delta$ 的数目是有穷的;事实上,它不超过 $|K| \cdot (|\Sigma| + 2)$,因为这个数是满足 $p \in K$ 和 $b \in \Sigma \cup \{\leftarrow, \rightarrow\}$ 的 (p, b) 可能组合的最大数目。把在任何时刻可用的四元组的最大数目称为 r ;通过检查 M 可确定数 r 。事实上我们假定对状态与符号的每种组合 (q, a) ,以及对每个整数 $i \in \{1, 2, \dots, r\}$,存在良定义的第 i 个可用的四元组 (q, a, p_i, b_i) 。如果对状态与符号的某种组合 (q, a) ,存在少于 r 个相关的 Δ 里的四元组,那么可重复某些四元组。

因为 M 是非确定型的,所以它没有确定的方式决定如何在每步从 r 种可用的“选择”里选择。假设我们帮助它决定。准确地说,设 M 的确定型形式 M_d ,是一个有与 M 同样的状态集但是却有两条带的装置。第一条带是 M 的带,起初包含输入 w ,第二条带起初包含一个范围 $1, \dots, r$ 里的 n 个整数的字符串,比方说 $i_1 i_2 \dots i_n$ 。于是 M_d 如下操作 n 步,在第一步,从对初始格局可用的 r 种可能的下一种状态与动作的组合 $(p_1, b_1), \dots, (p_r, b_r)$ 里, M_d 选择第 i_1 种,即 (p_{i_1}, b_{i_1}) ,这是第二条上当前被扫描符号 i_1 所提示的那种。 M_d 还向右移动第二只带头,以便下一步扫描 i_2 。在下一步 M_d 选择第 i_2 种组合,然后第 i_3 种组合,等等。当 M_d 在第二条带上看到意味着已用完了“提示”的空格时,它停机。

在模拟 M 的确定型 Turing 机 M' 的设计里, M_d 是重要组成部分。我们把 M' 描述成 3 带 Turing 机。根据定理 4.3.1 知道, M' 可被转换成等价的单带 Turing 机。 M' 的三条带使用如下:

(1) 第一条带永远不改变,它总是包含原始输入 w ,以便被模拟的 M 的每种计算可在相同输入上重新开始。

(2) 第二和第三条带用来对 $\{1, 2, \dots, r\}^*$ 里所有字符串, 模拟 M_d 的计算。在 M' 开始模拟每种新的计算之前, 把输入从第一条带复制到第二条带上。起初第三条带包含 e , 即空字符串(因此, 在最早的时候甚至没有开始对 M_d 的模拟)。

(3) 在对 M_d 的两次模拟之间, M' 利用 Turing 机 N 生成 $\{1, 2, \dots, r\}^*$ 在字典序下的后继字符串。即, N 从 e 开始生成字符串 $1, 2, \dots, r, 11, 12, \dots, rr, 111, \dots$ 。对 $r = 2$, N 恰恰是计算二进制后继函数(例 4.2.3)的 Turing 机; 对 $r > 2$ 的推广是相当直截了当的。

M' 是图 4-22 里给定的 Turing 机。我们用 $C^{1 \rightarrow 2}$ 表示删除第二条带并在第二条带上复制第一条带的简单 Turing 机。 B^3 是在第三条带上生成在字典序下后继字符串的机器。最后, $M_d^{2,3}$ 是在第二和第三条带上操作的 M_d 。这样就完成了对 M' 的描述。

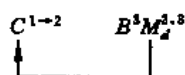


图 4-22

我们断言, M' 在输入 w 上停机当且仅当 M (的某种计算) 在输入 w 上停机。假设 M' 在输入 w 上确实停机, 通过检查图 4-22, 这意味着 M_d 停机时第三只带头不扫描空格。这蕴涵着对某个字符串 $i_1 i_2 \dots i_n \in \{1, 2, \dots, r\}^*$, 当 M_d 在第一条带上的 w 和第二条带上的 $i_1 i_2 \dots i_n$ 上启动时, 它在到达第二条带上的空格部分之前就停机。无论如何, 这意味着 M 在输入 w 上存在停机计算。相反地, 如果 M 在输入 w 上存在停机计算, 比方说有 n 步, 那么在最多 $r + r^2 + \dots + r^n$ 次失败尝试之后, B^3 生成对应于 M 停机计算的选择的 $\{1, 2, \dots, r\}^*$ 里的字符串, 并且 M_d 停机时扫描这个字符串的最后符号。因此 M' 停机, 证毕。 ■

如我们所预料的那样, 确定型 Turing 机对非确定型 Turing 机的模拟不是像我们在本章里见过的所有其他模拟那样步对步的模拟。它探讨非确定型 Turing 机所有可能的计算。作为结果, 它要求用 n 的指数多步去模拟非确定型机器的 n 步计算, 而本章描述的所有其他模拟事实上都是多项式的。究竟这个冗长而间接的模拟是非确定性的内在特性, 还是我们对非确定性缺乏理解所导致的人为结果, 这是本书第 6 和第 7 章所探讨的深刻、重要和悬而未决的问题。

习 题

4.5.1 给出(用缩写记号)接受下列语言的非确定型 Turing 机:

(a) $a^*abb^*baa^*$

(b) $\{ww^Ruu^R; w, u \in \{a, b\}^*\}$

4.5.2 设 $M = (K, \Sigma, \Delta, s, \{h\})$ 是下述非确定型 Turing 机:

$K = \{q_0, q_1, h\}$,

$\Sigma = \{a, \triangleright, \sqcup\}$,

$s = q_0$,

$\Delta = \{(q_0, \sqcup, q_1, a), (q_0, \sqcup, q_1, \sqcup), (q_1, \sqcup, q_1, \sqcup), (q_1, a, q_0, \rightarrow), (q_1, a, h, \rightarrow)\}$

描述 M 从格局 $\langle q_0, \triangleright \sqcup \rangle$ 开始所有可能的不超过五步的计算。用言语解释当 M 从这个格局开始时做什么事情。对于这台机器(在定理 4.5.1 的证明里)数 r 是多少?

4.5.3 尽管非确定型 Turing 机无助于证明递归语言在补运算下的封闭性, 但是它们对证明其他封闭性性质还是非常方便的。用非确定型 Turing 机证明递归语言类在并、连接和 Kleene 星号运算下封闭。对递归可枚举语言类重复这样的证明。

4.6 文 法

在本章里我们已介绍了多种计算装置,即 Turing 机和它的许多扩充,并且证明了在计算能力上它们都是等价的。所有这些不同种类的 Turing 机都可被合理地称为自动机,像在前几章里研究过的它们较弱的前身——有穷自动机和下推自动机那样。像那些自动机一样,Turing 机和它的扩充的基本作用是语言接受器,即收到输入,检查它,并且用不同方式表达接受或拒绝。结果产生了两族重要的语言,即递归和递归可枚举语言。

但是在前几章里我们也看到,存在着在特点上与语言接受器非常不同的另一类重要装置:语言生成器,比如正则表达式和上下文无关文法,它们可被用来定义有趣的语言类。事实上,我们已证明这两种形式化提供了对于用语言接受器定义的语言类的有价值的另一种刻画。假如没有这样的内容,本章就是不完整的;现在我们介绍新型的语言生成器,它是对上下文无关文法的推广,称为文法(或无限制文法,与上下文无关文法相对照),我们证明这样的文法所生成的语言类恰恰是递归可枚举语言类。

让我们回忆上下文无关文法的本质特性。它有被分成终结符集 Σ 和非终结符集 $V - \Sigma$ 两个部分的字母表 V 。它还有形如 $A \rightarrow u$ 的规则,其中 A 是非终结符并且 $u \in V^*$ 。操作上下文无关文法时,从起始符 S (非终结符)开始,重复地用规则的右方替换同一条规则的左方,直到不能继续进行这种替换为止。

在文法里适用所有同样的约定。差别在于,规则的左方不必是单个的非终结符。规则的左方可以是包含终结符和非终结符的任意字符串,其中至少有一个非终结符;单独一步推导要求删除规则左方的整个子串并且用这条规则的右方替换它。像在上下文无关文法里那样,最终的产物是只包含终结符的字符串。

定义 4.6.1 文法(或无限制文法,或改写系统)是四元组 $G = (V, \Sigma, R, S)$, 其中 V 是字母表;

$\Sigma \subseteq V$ 是终结符集, $V - \Sigma$ 称为非终结符集;

$S \in V - \Sigma$ 是起始符;并且

R , 即规则集, 是 $(V^* (V - \Sigma) V^*) \times V^*$ 的有穷子集。

若 $(u, v) \in R$ 则我们写 $u \rightarrow v$; 我们写 $u \Rightarrow_G v$ 当且仅当对某些 $w_1, w_2 \in V^*$ 以及某条规则 $u' \rightarrow v' \in R, u = w_1 u' w_2$ 并且 $v = w_1 v' w_2$ 。像通常那样, \Rightarrow_G^* 是 \Rightarrow_G 的自反传递闭包。字符串 $w \in \Sigma^*$ 是 G 生成的当且仅当 $S \Rightarrow_G^* w$; $L(G)$, 即 G 生成的语言, 是 G 生成的 Σ^* 里所有字符串的集合。

我们也使用原来介绍的上下文无关文法的其他术语。例如, 推导是形如 $w_0 \Rightarrow_G w_1 \Rightarrow_G \cdots \Rightarrow_G w_n$ 的序列。

例 4.6.1 任何上下文无关文法是文法。事实上, 上下文无关文法是这样的文法, 它的每条规则的左方是 $V - \Sigma$ 的成员, 而不是 $(V^* (V - \Sigma) V^*) \times V^*$ 的成员。因此, 在文法里, 规则可以具有形式 $uAv \rightarrow uvw$, 它可被读作“在 u 和 v 的上下文里用 w 替换 A ”。当然, 文法的规则能够具有比这种形式甚至更加一般的形式; 但是事实上文法所生成的任意语言, 可用所有规则都是这种“上下文有关替换”类型的文法来生成(习题 4.6.3)。 \diamond

例 4.6.2 下列文法 G 生成语言 $\{a^n b^n c^n; n \geq 1\}$, $G = (V, \Sigma, R, S)$, 其中

$$V = \{S, a, b, c, A, B, C, T_a, T_b, T_c\}$$

$$\Sigma = \{a, b, c\}$$

$$R = \{S \rightarrow ABCS, \\ S \rightarrow T_c, \\ CA \rightarrow AC, \\ BA \rightarrow AB, \\ CB \rightarrow BC, \\ CT_c \rightarrow T_c c, \\ CT_c \rightarrow T_c b, \\ BT_b \rightarrow T_b b, \\ BT_b \rightarrow T_a b, \\ AT_a \rightarrow T_a a, \\ T_a \rightarrow \epsilon\}$$

前两条规则生成形如 $(ABC)^n T_c$ 的字符串。然后, 接着的三条规则允许字符串里的 A, B 和 C 正确地“整理”它们自身, 使得字符串变成 $A^n B^n C^n T_c$ 。最后, 剩余的规则允许 T_c 向左移动, 把所有 C 变换成 c , 然后变成 T_b 。接着 T_b 向左移动, 把所有 B 变换成 b , 然后变成 T_a 。最后 T_a 把所有 A 变换成 a , 然后被删除。

显然, 形如 $a^n b^n c^n$ 的任何字符串都可用这种方式生成。当然, 包含非终结符的更多的字符串也可被产生出来; 不过, 不难看出删除所有非终结符的唯一方式是遵循上面简述的过程。因此, 用 G 生成的 $\{a, b, c\}$ 里的字符串仅仅是 $\{a^n b^n c^n; n \geq 1\}$ 里的那些。 \diamond

显然, 文法生成的语言类包含某些非上下文无关的语言类。但是究竟这个语言类有多大呢? 更重要地, 它与我们在本章见过的上下文无关语言的其他两种重要扩充(即递归和递归可枚举语言)的关系如何呢? 事实上, 文法与 Turing 机之间的关系恰恰像是上下文无关文法与下推自动机以及正则表达式与有穷自动机之间的关系。

定理 4.6.1 语言被文法生成当且仅当它是递归可枚举的。

证明: 仅当。设 $G = (V, \Sigma, R, S)$ 是文法。我们设计半判定 G 所生成的语言的 Turing 机 M 。事实上, M 是非确定型的, 定理 4.5.1 保证可把它转换成半判定同样语言的确定型机器。

M 有三条带。第一条带包含输入 w 并且永不被改变。在第二条带里, M 试图重新构造文法 G 里从 S 到 w 的推导, 因此 M 在开始时把 S 写在第二条带上。然后, 对应于所构造的推导的步, M 分步向前进行。每步开始于非确定性的状态转移, 从 $|R| + 1$ 种可能的状态里猜测一种。这些 $|R| + 1$ 种状态的前 $|R|$ 种里每种都是状态转移序列的开头, 这个序列把相应的规则应用到第二条带的当前内容上。假设被选择的规则是 $u \rightarrow v$ 。于是 M 从左向右扫描第二条带, 非确定性地停止在某个符号上。然后它验证后继的 $|u|$ 个符号与 u 匹配, 删除 u , 适当地平移剩下的字符串为 v 留出恰恰足够的空间, 并且写 v 代替 u 。如果验证失败, 则 M 进入不结束的计算——在生成 w 上的当前尝试已失败。

M 的第 $|R| + 1$ 种选择要求验证当前字符串是否等于输入 w 。若相等, 则 M 停机并接

受: w 确实可用 G 生成。若发现字符串不相等, 则 M 又进入死循环。

显然, 只有 M 的可能的停机计算对应于 G 里对 w 的推导。因此 M 接受 w 当且仅当 $w \in L(G)$, 所以仅当方向已被证明了。

当。现在假设 $M = (K, \Sigma, \delta, s, \{h\})$ 是 Turing 机。为方便起见, 假设 Σ 和 K 不相交并且都不包含新的结束符 \triangleleft 。我们还假设若 M 停机, 则它总是在格局 $(h, \triangleright \sqcup)$ 停机, 即在删除带之后停机。半判定语言的任何 Turing 机都可被变换成满足上述条件的等价 Turing 机。我们构造生成用 M 半判定的语言 $L \subseteq (\Sigma - \{\sqcup, \triangleright\})^*$ 的文法 $G = (V, \Sigma - \{\sqcup, \triangleright\}, R, S)$ 。

字母表 V 包括 Σ 里所有符号、 K 里所有状态以及初始符 S 和结束符 \triangleright 。起初, G 的推导模拟 M 的反向计算。我们用字符串 $\triangleright uaqw\triangleleft$ 模拟格局 $(q, \triangleright u\triangleleft w)$, 即用带的内容, 并把当前状态直接插入到当前被扫描符号之后, 把结束符 \triangleright 附加在字符串的结尾来模拟格局。 G 的规则模拟 M 的反向移动, 即对每个 $q \in K$ 和 $a \in \Sigma$, 根据 $\delta(q, a)$, G 有如下规则:

- (1) 如果对某个 $p \in K$ 和 $b \in \Sigma$, $\delta(q, a) = (p, b)$, 那么 G 有规则 $bp \rightarrow aq$ 。
- (2) 如果对某个 $p \in K$, $\delta(q, a) = (p, \rightarrow)$, 那么对所有 $b \in \Sigma$, G 有规则 $abp \rightarrow aqb$, 以及规则 $a \sqcup p \triangleleft \rightarrow aq\triangleleft$ (最后这条规则反转了用新的空格向右扩充带的动作)。
- (3) 如果对某个 $p \in K$ 和 $a \neq \sqcup$, $\delta(q, a) = (p, \leftarrow)$, 那么 G 有规则 $pa \rightarrow aq$ 。
- (4) 如果对某个 $p \in K$, $\delta(q, \sqcup) = (p, \leftarrow)$, 那么对所有 $b \in \Sigma$, G 有规则 $pab \rightarrow aqb$, 以及规则 $p\triangleleft \rightarrow \sqcup q\triangleleft$, 这条规则反转了删除多余空格的动作。

最后, G 包含计算开始(推导结束)和计算结束(推导开始)的某些状态转移。规则

$$S \rightarrow \triangleright \sqcup h \triangleleft$$

迫使推导恰恰从接受计算结束的地方开始, 其他规则是删除在输入左方的最后字符串的 $\triangleright \sqcup s \rightarrow e$, 以及删除结束符并且留下输入字符串的 $\triangleleft \rightarrow e$ 。

下列结果使得 G 模拟 M 的反向计算的概念精确化。

断言 对于 M 的任意两个格局 $(q_1, u_1 \triangleleft_1 w_1)$ 和 $(q_2, u_2 \triangleleft_2 w_2)$, $(q_1, u_1 \triangleleft_1 w_1) \vdash_M (q_2, u_2 \triangleleft_2 w_2)$ 当且仅当 $u_2 a_2 q_2 w_2 \triangleleft_2 \Rightarrow_G u_1 a_1 q_1 w_1 \triangleleft_1$ 。

这个断言的证明是对 M 的动作的直截了当的分情形讨论, 因此留作练习。

现在我们通过证明对所有 $w \in (\Sigma - \{\triangleright, \sqcup\})^*$, M 在 w 上停机当且仅当 $w \in L(G)$ 来完成定理的证明。 $w \in L(G)$ 当且仅当

$$S \Rightarrow_G \triangleright \sqcup h \triangleleft \Rightarrow_G^* \triangleright \sqcup sw \triangleleft \Rightarrow_G w \triangleleft \Rightarrow_G w$$

因为 $S \rightarrow \triangleright \sqcup h \triangleleft$ 是涉及 S 的唯一规则, 而规则 $\triangleright \sqcup s \rightarrow e$ 和 $\triangleleft \rightarrow e$ 则是允许最终删除状态和结束符 \triangleleft 的仅有规则。现在, 根据断言, $\triangleright \sqcup h \triangleleft \Rightarrow_G^* \triangleright \sqcup sw \triangleleft$ 当且仅当 $(s, \triangleright \triangleleft w) \vdash_M^* (h, \triangleright \sqcup)$, 它发生当且仅当 M 在 w 上停机, 这样就完成了定理的证明。 ■

定理 4.6.1 给出文法的半判定性, 半判定性是 Turing 机非现实的方面, 它来自它的单侧定义, 当输入不属于语言时, 这种单侧的定义不提供任何信息。这与我们对文法的了解是相容的: 若字符串可用文法生成, 则我们可耐心地搜索所有可能的推导, 从较短的推导开始并向较长的推导前进, 直到发现正确的推导为止。但是, 若不存在这种推导, 则这个过程无限地进行下去, 不能提供任何有用的信息。

事实上, 我们也可以认为文法是基于 Turing 机的更有用的计算方式。

定义 4.6.2 设 $G = (V, \Sigma, R, S)$ 是文法, 并设 $f: \Sigma^* \mapsto \Sigma^*$ 是函数。若对所有 $w, v \in$

Σ^* , 下列关系为真

$$SwS \Rightarrow_G^* v \text{ 当且仅当 } v = f(w)$$

即, 包括输入 w 和两侧的 G 的起始符的字符串恰好生成 Σ^* 里一个字符串: $f(w)$ 的正确值, 则我们说 G 计算 f 。

函数 $f: \Sigma^* \rightarrow \Sigma^*$ 称为是文法可计算的当且仅当存在计算它的文法 G 。

我们把下列结果的证明——定理 4.6.1 的证明的修改——留作练习(习题 4.6.4)。

定理 4.6.2 函数 $f: \Sigma^* \rightarrow \Sigma^*$ 是递归的当且仅当它是文法可计算的。

习 题

- 4.6.1 (a) 给出字符串 $aaabbbccc$ 在例 4.6.2 的文法里的推导。
(b) 仔细地证明例 4.6.2 里的文法生成语言 $L = \{a^n b^n c^n; n \geq 1\}$ 。
- 4.6.2 给出生成下列语言的文法:
(a) $\{ww; w \in \{a, b\}^*\}$
(b) $\{a^{2^n}; n \geq 0\}$
(c) $\{a^{n^2}; n \geq 0\}$
- 4.6.3 证明任何文法都可转换成规则形如 $uAv \rightarrow uuv$ 的等价的文法, 其中 $A \in V - \Sigma$, 并且 $u, v, w \in V^*$ 。
- 4.6.4 证明定理 4.6.2 (对于仅当方向, 给定文法 G , 证明如何构造 Turing 机, 它在输入 w 上输出字符串 $u \in \Sigma^*$, 使得 $SwS \Rightarrow_G^* u$, 假如这样的字符串 u 存在。对于当方向, 利用与定理 4.6.1 的证明类似的模拟, 差别在于用相反的(向前的)方向。)
- 4.6.5 利用文法计算函数的古怪之处是应用规则的次序不确定。在下列归功于 A. A. Markov (1903—1979) 的替代方案里避免了这种不确定性。**Markov 系统**是四元组 $G = (V, \Sigma, R, R_1)$, 其中 V 是字母表; $\Sigma \subseteq V$; R 是规则的有穷序列(不是集合) $(u_1 \rightarrow v_1, \dots, u_k \rightarrow v_k)$, 其中 $u_i, v_i \in V^*$; R_1 是 R 里的规则的集合。关系 $w \Rightarrow_G w'$ 定义如下: 如果存在 i 使得 u_i 是 w 的子串, 那么设 i 是最小的这样的数字, 并且设 w_1 是满足 $w = w_1 u_i w_2$ 的最短的字符串; 于是只要 $w' = w_1 v_i w_2$ 就有 $w \Rightarrow_G w'$ 。因此, 如果规则是可用的, 那么最多存在一条规则, 并且它恰恰在一个位置上可用。设 $f: \Sigma^* \rightarrow \Sigma^*$ 是函数, 若对所有 $u \in \Sigma^*$

$$u = u_0 \Rightarrow_G u_1 \Rightarrow_G \dots \Rightarrow_G u_{n-1} \Rightarrow_G u_n = f(u)$$

并且第一次使用 R_1 里的规则是最后一步 $u_{n-1} \Rightarrow_G u_n$, 则我们说 G 计算 f 。证明函数可用 Markov 系统计算当且仅当它是递归的。(证明类似于定理 4.6.2 的证明。)

4.7 数值函数

现在让我们采取完全不同的关于计算的观点, 它不立足于对计算或信息处理过程的任何具体形式化上, 比如 Turing 机或文法, 而是把注意力集中在被计算的对象上: 从数值到数值的函数。例如, 对给定的 m 和 n 的任意值, 函数

$$f(m, n) = m \cdot n^2 + 3 \cdot m^2 \cdot m + 17$$

的值是可计算的,因为它是加法、乘法、幂和常数等可计算函数的合成。我们如何知道幂函数可计算?因为它用更简单的函数(即乘法)和更小的自变量的函数值来递归地定义的。归根结底,若 $n=0$ 则 m^n 是1,否则它是 $m \cdot m^{n-1}$ 。乘法本身可用加法来递归地定义,等等。

原则上,我们可设法从非常简单、被公认为是可计算的自然数到自然数的函数出发(例如恒等函数和后继函数 $\text{succ}(n)=n+1$),然后用同样非常初等和明显可计算的组合方式——比如合成和递归定义——来缓慢而耐心地组合它们,最终得到相当一般的和非平凡的从自然数到自然数的函数类。在本节里我们完成这种练习。颇有意义的是,这样定义的计算概念随后被证明等同于在本章里用特点、视角和细节都如此不同的其他方法,比如 Turing 机、Turing 机的变种以及文法等,所得到的概念。

定义 4.7.1 我们首先对满足 $k \geq 0$ 的各种不同的 k 值,定义从 N^k 到 N 的某些最简单的函数(0元函数当然是常数,因为它不依赖于任何自变量)。基本函数如下:

(a) 对任何 $k \geq 0$, k 元零函数定义为对所有 $n_1, \dots, n_k \in N$, $\text{zero}_k(n_1, \dots, n_k) = 0$ 。

(b) 对任何 $k \geq j > 0$,第 j 个 k 元恒等函数定义为对所有 $n_1, \dots, n_k \in N$, $\text{id}_{k,j}(n_1, \dots, n_k) = n_j$ 。

(c) 后继函数定义为对所有 $n \in N$, $\text{succ}(n) = n+1$ 。

下一步我们介绍把函数组合成稍微复杂的函数的两种简单方法。

(1) 设 $k, l \geq 0$, $g: N^l \rightarrow N$ 是 k 元函数, h_1, \dots, h_k 都是 l 元函数,则 g 与 h_1, \dots, h_k 的合成是由

$$f(n_1, \dots, n_l) = g(h_1(n_1, \dots, n_l), \dots, h_k(n_1, \dots, n_l))$$

定义的 l 元函数。

(2) 设 $k \geq 0$, g 是 k 元函数, h 是 $k+2$ 元函数,则用 g 和 h 递归地定义的函数是由对所有 $n_1, \dots, n_k, m \in N$,

$$f(n_1, \dots, n_k, 0) = g(n_1, \dots, n_k)$$

$$f(n_1, \dots, n_k, m+1) = h(n_1, \dots, n_k, m, f(n_1, \dots, n_k, m))$$

定义的 $k+1$ 元函数。

原始递归函数是所有基本函数和用基本函数通过任意次相继应用合成和递归定义所获得的函数。

例 4.7.1 函数 $\text{plus2}(n) = n+2$ 是原始递归的,因为它可从基本函数 succ 出发通过 succ 自身的合成来获得。具体地说,在定义 4.7.1 的(1)里设 $k=l=1$,并设 $g=h_1=\text{succ}$ 。

同理,二元函数 $\text{plus}(m, n) = m+n$ 是原始递归的,因为它可通过组合恒等函数、零函数和后继函数,然后从所获得的函数出发来递归地定义。具体地说,在定义 4.7.1 的(2)里设 $k=1$,设 g 是 $\text{id}_{2,1}$ 函数,并设 h 是三元函数 $h(m, n, p) = \text{succ}(\text{id}_{3,3}(m, n, p))$,即 succ 与 $\text{id}_{3,3}$ 的合成。所得到的递归定义的函数恰恰是 plus 函数:

$$\text{plus}(m, 0) = m$$

$$\text{plus}(m, n+1) = \text{succ}(\text{plus}(m, n))$$

为什么要停下来? 乘法函数 $\text{mult}(m, n) = m \cdot n$ 递归地定义为

$$\text{mult}(m, 0) = \text{zero}(m)$$

$$\text{mult}(m, n+1) = \text{plus}(m, \text{mult}(m, n))$$

函数 $\exp(m, n) = m^n$ 定义为

$$\begin{aligned}\exp(m, 0) &= \text{succ}(\text{zero}(m)) \\ \exp(m, n+1) &= \text{mult}(m, \exp(m, n))\end{aligned}$$

因此所有这些函数都是原始递归的。

所有形如 $f(n_1, \dots, n_k) = 17$ 的常值函数都是原始递归的, 因为它们可通过合成适当的 zero 函数与 succ 函数来获得, 在本例里要合成 17 次。另外, 符号函数 $\text{sgn}(n)$ 也是原始递归的; $\text{sgn}(0) = 0, \text{sgn}(n+1) = 1$ 。

为了获得更好的可读性, 从现在起我们用 $m+n$ 代替 $\text{plus}(m, n)$, 用 $m \cdot n$ 代替 $\text{mult}(m, n)$, 并且用 $m \uparrow n$ 代替 $\exp(m, n)$ 。因此所有数值函数, 比如

$$m \cdot (n+m)^2 + 178^m$$

都是原始递归的, 因为它们都是通过相继合成上述函数所获得的。

因为我们局限于自然数之内, 所以没有真正的减法和除法。不过, 沿着这些路线, 我们可定义某些有用的函数, 比如 $m \sim n = \max\{m-n, 0\}$, 以及函数 $\text{div}(m, n)$ 和 $\text{rem}(m, n)$ (m 除以 n 的整数商和余数; 若 $n=0$ 则假设它们都是 0)。首先定义前驱函数:

$$\begin{aligned}\text{pred}(0) &= 0 \\ \text{pred}(n+1) &= n\end{aligned}$$

从它得到“非负减法”函数

$$\begin{aligned}m \sim 0 &= m \\ m \sim n+1 &= \text{pred}(m \sim n)\end{aligned}$$

商函数和剩余函数在后面的例子里定义。◇

相当清楚的是, 我们可对任意原始递归函数从给定的自变量值计算出函数值。同样, 可对关于自然数的断言, 比如

$$m \cdot n > m^2 + n + 7$$

对 m 和 n 的任意值计算断言的有效性。原始递归谓词是只取值 0 和 1 的原始递归函数。直观上, 原始递归谓词, 比如 $\text{greater-than}(m, n)$, 刻画在 m 和 n 的值之间既可能成立也可能不成立的关系。若关系成立, 则原始递归谓词的值为 1, 否则为 0。

例 4.7.2 若 $n=0$ 则它是 1, 若 $n>0$ 则它是 0 的函数 iszero 是原始递归谓词, 它递归地定义成

$$\begin{aligned}\text{iszero}(0) &= 1 \\ \text{iszero}(m+1) &= 0\end{aligned}$$

同理, $\text{isone}(0) = 0, \text{isone}(n+1) = \text{iszero}(n)$ 。谓词 $\text{positive}(n)$ 与已经定义过的 $\text{sgn}(n)$ 相同。另外, 写成 $m \geq n$ 的 $\text{greater-than-or-equal}(m, n)$ 可定义成 $\text{iszero}(n \sim m)$ 。它的否定, 即 $\text{less-than}(m, n)$, 当然是 $1 \sim \text{greater-than-or-equal}(m, n)$ 。一般地, 任何原始递归谓词的否定还是原始递归谓词。事实上, 两个原始递归谓词的析取和合取还是原始递归谓词: $p(m, n)$ or $q(m, n)$ 是 $1 \sim \text{iszero}(p(m, n) + q(m, n))$, $p(m, n)$ and $q(m, n)$ 是 $1 \sim \text{iszero}(p(m, n) \cdot q(m, n))$ 。例如 $\text{equals}(m, n)$ 可定义成 $\text{greater-than-or-equal}(m, n)$ 和 $\text{greater-than-or-equal}(n, m)$ 的合取。

另外, 如果 f, g 和 h 都是原始递归函数而 p 是原始递归谓词, 并且所有这 4 者都是 k

元的,那么分情形定义的函数

$$f(n_1, \dots, n_k) = \begin{cases} g(n_1, \dots, n_k) & \text{如果 } p(n_1, \dots, n_k) \\ h(n_1, \dots, n_k) & \text{否则} \end{cases}$$

也是原始递归的,因为它可写成

$$f(n_1, \dots, n_k) = p(n_1, \dots, n_k) \cdot g(n_1, \dots, n_k) + (1 \sim p(n_1, \dots, n_k)) \cdot h(n_1, \dots, n_k)$$

我们将看到,分情形定义是非常有用的缩写。◇

例 4.7.3 现在我们定义 div 和 rem 。

$$\text{rem}(0, n) = 0$$

$$\text{rem}(m+1, n) = \begin{cases} 0 & \text{如果 } \text{equal}(\text{rem}(m, n), \text{pred}(n)) \\ \text{rem}(m, n) + 1 & \text{否则} \end{cases}$$

以及

$$\text{div}(0, n) = 0$$

$$\text{div}(m+1, n) = \begin{cases} \text{div}(m, n) + 1 & \text{如果 } \text{equal}(\text{rem}(m, n), \text{pred}(n)) \\ \text{div}(m, n) & \text{否则} \end{cases}$$

另一种有趣的原始递归的函数是 $\text{digit}(m, n, p)$, 即 n 的 p 进制表示的倒数第 m 位数字。(作为对 digit 用途的说明,谓词 $\text{odd}(n)$ 简单地写成 $\text{digit}(1, n, 2)$ 。)容易验证 $\text{digit}(m, n, p)$ 可定义成 $\text{div}(\text{rem}(n, p \uparrow m), p \uparrow (m \sim 1))$ 。◇

例 4.7.4 如果 $f(n, m)$ 是原始递归函数,那么和函数

$$\text{sum}_f(n, m) = f(n, 0) + f(n, 1) + \dots + f(n, m)$$

也是原始递归的,因为它可定义成 $\text{sum}_f(n, 0) = 0$, 以及 $\text{sum}_f(n, m+1) = \text{sum}_f(n, m) + f(n, m+1)$ 。我们也可用这样的方法定义谓词的无界合取与无界析取。例如,设 $p(n, m)$ 是谓词,则析取式

$$p(n, 0) \text{ or } p(n, 1) \text{ or } p(n, 2) \text{ or } \dots \text{ or } p(n, m)$$

正好是 $\text{sgn}(\text{sum}_p(n, m))$ 。◇

显然,从定义 4.7.1 的最简单的材料出发,我们可证明许多相当复杂的函数是原始递归的。不过,原始递归函数并没有刻画出我们合理地认为可计算的所有函数。最好还是利用对角化论证来证明这个结论。

例 4.7.5 原始递归函数集是可枚举的。这是因为每个原始递归函数原则上都可用基本函数来定义,因此都可表示成有穷字母表里的字符串;字母表应当包括表示恒等函数、后继函数、零函数、原始递归、合成、括号、0 和 1 等的符号,0 和 1 用来给基本函数建立二进制下标,比如 $\text{id}_{17,11}$ (关于这样的下标的另一种用途看 5.2 节,这次是表示所有 Turing 机)。于是我们可枚举字母表里的所有字符串并且只保留是原始递归函数的合法定义的字符串,事实上,我们可选择只保留一元原始递归函数,即那些只有一个变量的原始递归函数。

然后假设把所有一元原始递归函数像字符串那样按照字典序排列

$$f_0, f_1, f_2, f_3, \dots$$

原则上,给定任何数字 $n \geq 0$,我们可在这个排列里找出第 n 个函数 f_n ,然后用它的定义去计算自然数 $f_n(n) + 1$ 。把这个数字称为 $g(n)$ 。显然, $g(n)$ 是可计算函数,我们刚刚简述过如何计算它。不过, g 不是原始递归函数。因为假如它是的话,比方说对某个 $m \geq 0, g = f_m$,

那么我们有 $f_m(m) = f_m(m) + 1$, 这是荒谬的。

这就是对角化论证。它依赖于我们对所有原始递归函数的顺序列表; 通过这种列表可以定义出与列表里所有函数都不同的函数。试比较这个论证和说 $2^{\mathbb{N}}$ 不可数的定理 1.5.2 的证明。在那里我们从所谓的对 $2^{\mathbb{N}}$ 所有成员的列表开始, 得到不在列表里的 $2^{\mathbb{N}}$ 的成员。◇

显然, 包含我们合理地称为“可计算的”每样东西的任何定义函数的方法, 不能仅仅立足于简单的操作, 比如合成和递归定义, 因为它们产生的函数总是可靠地被识别出来, 因此被枚举出来。因此我们发现了关于计算的形式化有趣的真理: 任何它的成员(计算装置)都是自明的这样的形式化(即, 给定字符串, 我们可轻易地判定它是否编码该形式化里的计算装置), 必然是要么太弱(像有穷状态自动机和原始递归函数那样), 要么太一般以致于在实际上没有用处(像在输入上既可能停机也可能不停机的 Turing 机那样)。任何给出且只给出所有可计算函数的形式化都必须包含非自明的函数(正如一台 Turing 机是否在所有输入上停机, 从而判定一个语言, 是非自明的。)确实, 下一步我们在函数上定义更微妙的操作, 它对应于熟悉的无限迭代的基本计算过程, 本质上这是 while 循环。我们将看到, 无限迭代的结果确实可能不是一个函数。

定义 4.7.2 设对某个 $k \geq 0$, g 是 $(k+1)$ 元函数。 g 的极小化是如下定义的 k 元函数

$$f(n_1, \dots, n_k) = \begin{cases} \text{使得 } g(n_1, \dots, n_k, m) = 1 \text{ 的最小的 } m, & \text{如果存在这样的 } m \\ 0, & \text{否则} \end{cases}$$

我们把 g 的极小化表示成 $\mu m [g(n_1, \dots, n_k, m) = 1]$ 。

尽管函数 g 的极小化总是良定义的, 也还是没有计算它的明显方法, 即使我们知道如何计算 g 。明显的方法是

```
m := 0;
while g(n1, ..., nk, m) ≠ 1 do m := m + 1;
output m
```

这不是一个算法, 因为它可能不终止。

若上述方法总是终止, 则把函数 g 称为可极小化的。即, 若 $(k+1)$ 元函数 g 具有下列性质: 对每个 $n_1, \dots, n_k \in \mathbb{N}$, 存在 $m \in \mathbb{N}$ 使得 $g(n_1, \dots, n_k, m) = 1$, 则 g 是可极小化的。

最后, 若函数可从基本函数出发, 通过合成、递归定义以及可极小化函数的极小化等操作来获得, 则它称为 μ 递归的。

注意现在我们不能重复例 4.7.5 的对角化论证去证明存在不是 μ 递归的可计算函数。难点在于, 给定一个自称 μ 递归函数的定义, 它是否确实定义一个 μ 递归函数, 这根本就是不清楚的, 即在这个定义里极小化的所有应用是否确实都作用在可极小化的函数上, 这是不清楚的!

例 4.7.6 我们已定义了加法的逆(\sim 函数)和乘法的逆(div 函数); 但是如何定义指数的逆——对数? 利用极小化^①, 我们可定义对数函数: $\log(m, n)$ 是为了得出不小于 $n+1$ 的整数而必须让 $m+2$ 自乘的最小幂次(即, $\log(m, n) = \lceil \log_{m+2}(n+1) \rceil$; 我们用 $m+2$ 和 $n+1$ 作为自变量去避免当 $m \leq 1$ 或 $n = 0$ 时, 在 $\log_m n$ 的定义里的数学陷阱)。函数 \log 定

^① 对数函数可不用极小化操作来定义, 见习题 4.7.2。我们在这里使用极小化仅仅是为了说明和方便。

义如下

$$\log(m, n) = \mu p [\text{greater-than-or-equal}((m+2) \uparrow p, n+1)]$$

注意这是一个 μ 递归函数的正常定义, 因为函数 $g(m, n, p) = \text{greater-than-or-equal}((m+2) \uparrow p, n+1)$ 是可极小化的: 确实, 对任何 $m, n \geq 0$, 存在 $p \geq 0$ 使得 $(m+2)^p \geq n$ 。因为通过把 ≥ 2 的整数自乘上越来越大的幂次, 我们可获得任意大的整数。◇

现在我们证明本节的主要结果。

定理 4.7.1 函数 $f: N^k \mapsto N$ 是 μ 递归的当且仅当它是递归的(即, 用 Turing 机可计算的)。

证明: 仅当。假设 f 是 μ 递归函数。于是它是从基本函数出发, 利用合成、递归定义和在可极小化函数上的极小化来定义的。我们将证明 f 是 Turing 可计算的。

首先, 容易看出基本函数是递归的; 我们已看到后继函数是这样的(例 4.2.3), 其余的函数只涉及删除部分或全部的输入。

其次, 假设 $f: N^k \mapsto N$ 是函数 $g: N^l \mapsto N$ 与 $h_1, \dots, h_l: N^k \mapsto N$ 的合成, 其中通过归纳我们知道如何计算 g 和诸 h_i 。于是我们可计算 f 如下(在这里和其他的情形里, 我们用随机存取 Turing 机程序的样式来给出计算这些函数的程序; 容易看出, 用标准 Turing 机可达到同样效果):

```
m1 := h1(n1, ..., nk);  
m2 := h2(n1, ..., nk);  
⋮  
ml := hl(n1, ..., nk);  
output g(m1, ..., ml).
```

同理, 如果 f 是用 g 和 h 递归地定义的(回忆定义), 那么 $f(n_1, \dots, n_k, m)$ 可用下列程序来计算:

```
v := g(n1, ..., nk);  
if m = 0 then output v  
else for i := 1, 2, ..., m do  
    v := h(n1, ..., nk, i-1, v);  
output v.
```

最后, 假设 f 定义成 $\mu m [g(n_1, \dots, n_k, m)]$, 其中 g 是可极小化的和可计算的。于是 f 可用下列程序来计算:

```
m := 0;  
while g(n1, ..., nk, m) ≠ 1 do m := m + 1;  
output m
```

因此我们已证明了所有基本函数是递归的, 递归函数的合成和递归定义以及可极小化递归函数的极小化都是递归的; 我们必然得出结论说所有 μ 递归函数都是递归的。这样就完成了对仅当方向的证明。

当。假设 Turing 机 $M = (K, \Sigma, \delta, s, \{h\})$ 计算函数 $f: N \mapsto N$ 。为了表示起来简单, 我们现在假设 f 是一元的; 一般情形是简单的推广(见习题 4.7.5)。我们将证明 f 是 μ 递归

的。我们将耐心地定义某些与 M 和 M 的操作有关的 μ 递归函数,直到积累起足够的材料去定义 f 本身为止。

不失一般性,假设 K 和 Σ 是不相交的。设 $b = |K| + |\Sigma|$, 并且让我们固定从 $K \cup \Sigma$ 到 $\{0, 1, \dots, b-1\}$ 的映射 E , 使得 $E(0) = 0$ 并且 $E(1) = 1$ 。回忆一下,因为 M 计算数值函数,所以它的字母表必须包含 0 和 1。利用这个映射,我们把 M 的格局表示成 b 进制整数。格局 $(q, a_1 a_2 \dots a_k \dots a_n)$ 表示成 b 进制整数 $(a_1 a_2 \dots a_k q a_{k+1} \dots a_n)$, 其中诸 a_i 都是 Σ 里的符号,即表示成整数

$$E(a_1)b^n + E(a_2)b^{n-1} + \dots + E(a_k)b^{n-k+1} + E(q)b^{n-k} + E(a_{k+1})b^{n-k-1} + \dots + E(a_n)$$

现在我们开始把 f 定义成 μ 递归函数。最终, f 将定义成

$$f(n) = \text{num}(\text{output}(\text{last}(\text{comp}(n))))$$

num 是一个函数,它取一个 b 进制表示为 0, 1 字符串的整数,输出这个字符串的二进制值。 output 取一个以 b 进制表示形如 $\triangleright \sqcup h w$ 的停机格局的整数,删除前三个符号 $\triangleright \sqcup h$ 。 $\text{comp}(n)$ 是这样的自然数,它的 b 进制表示是从 $\triangleright \sqcup s w$ 开始到 $\triangleright \sqcup h w'$ 结束的唯一格局序列的并置,其中 w 是 n 的二进制编码, w' 是 $f(n)$ 的二进制编码,这样一个序列存在,因为我们现在假设 M 计算函数 f 。 last 取一个表示格局并置的整数,从这个序列里取出最后一个格局(在最后一个 \triangleright 和结尾之间的部分)。

当然,我们不得不定义所有这些函数。下面给出大多数的定义,而把其余的定义留作练习(习题 4.7.4)。让我们从 last 开始。设 $\text{lastpos}(n)$ 是在编码成 b 进制数字 n 的字符串里 \triangleright 出现的最后(最右,最低)位置,可定义为

$$\text{lastpos}(n) = \mu [\text{equal}(\text{digit}(m, n, b), E(\triangleright)) \text{ or } \text{equal}(m, n)]$$

注意为了使得方括号里的函数是可极小化的,若在 n 里找不到 \triangleright , 则我们让 $\text{lastpos}(n)$ 是 n 。顺便提一句,这是另一次表面上的对极小化的应用,因为不用极小化也可轻易地重新定义这个函数。于是我们把 $\text{last}(n)$ 定义成 $\text{rem}(n, b \uparrow \text{lastpos}(n))$ 。我们也可用 $\text{div}(n, b \uparrow \text{lastpos}(n))$ 来定义 $\text{rest}(n)$, 即在删除 $\text{last}(n)$ 之后剩余的序列。

$\text{output}(n)$ 正好是 $\text{rem}(n, b \uparrow \log(b \sim 2, n \sim 1) \sim 2)$ 。(回忆我们对于 \log 的两个自变量必须分别减去 2 和 1 的约定。)

函数 $\text{num}(n)$ 可写成和式

$$\begin{aligned} & \text{digit}(1, n, b) \cdot 2 + \text{digit}(2, n, b) \cdot 2 \uparrow 2 + \dots \\ & + \text{digit}(\log(b \sim 2, n \sim 1), n, b) \cdot 2 \uparrow \log(b \sim 2, n \sim 1) \end{aligned}$$

这是 μ 递归函数,因为被加数和界限 $\log(b \sim 2, n \sim 1)$ 都是 μ 递归函数。它的逆函数 $\text{bin}(n)$ 把任意整数映射成该整数的二进制编码字符串,再编码成 b 进制整数。 $\text{bin}(n)$ 与 $\text{num}(n)$ 非常相似,其中交换了 2 和 b 的作用。

在上述的 $f(n)$ 的定义里,最有趣的(也最难定义的)函数是 $\text{comp}(n)$, 它把 n 映射成完成对 $f(n)$ 的计算的 M 格局序列——事实上,是编码这个序列的 b 进制整数。在最高的一级上,它正好是

$$\text{comp}(n) = \mu m [\text{iscomp}(m, n) \text{ and } \text{halted}(\text{last}(m))] \quad (1)$$

其中 $\text{iscomp}(m, n)$ 是一个谓词,它说 m 属于从 $\triangleright \sqcup \text{bin}(n)$ 开始的、不一定停机的计算的格局序列(顺便提一句,这是在本证明里唯一真正地和本证明里需要极小化的地方。)注意在式

(1)方括号里的函数确实可极小化,因为假设 M 计算 f ,所以对所有 n ,这样一个格局序列 m 存在。 $\text{halted}(n)$ 正好是 $\text{equal}(\text{digit}(\log(b \sim 2, n \sim 1) \sim 2, n, b), E(h))$ 。

我们把 iscomp 的精确定义留作练习(习题 4.7.4)。

由此得出 f 的确是 μ 递归函数,所以定理的证明就完成了。 ■

习 题

4.7.1 设 $f: \mathbb{N} \mapsto \mathbb{N}$ 是原始递归函数,定义 $F: \mathbb{N} \mapsto \mathbb{N}$ 是

$$F(n) = f(f(f(\cdots f(n) \cdots)))$$

其中有 n 次函数合成。证明 F 是原始递归的。

4.7.2 证明下列函数都是原始递归的:

(a) $\text{factorial}(n) = n!$

(b) $\text{gcd}(m, n)$, m 和 n 的最大公因子

(c) $\text{prime}(n)$, 若 n 是素数则值为 1 的谓词

(d) $p(n)$, 即第 n 个素数,其中 $p(0) = 2, p(1) = 3$, 等等

(e) 正文里定义的函数 \log

4.7.3 假设 f 是 μ 递归的从 \mathbb{N} 到 \mathbb{N} 的双射。证明它的逆 f^{-1} 也是 μ 递归的。

4.7.4 证明:在定理 4.7.1 的证明里描述的函数 iscomp 是原始递归的。

4.7.5 在定理 4.7.1 的当方向的证明里,如果 M 计算 $k > 1$ 的函数 $f: \mathbb{N}^k \mapsto \mathbb{N}$,那么必须对证明里的构造作哪些修改?

4.7.6 构造把原始递归函数表示成在你选择的字母表里的字符串的表示法。(对 Turing 机,这样一种表示法见下一章。)把例 4.7.5 里的论证形式化:不是所有可计算函数都是原始递归的。

参 考 文 献

第一个想到 Turing 机的是 Alan M. Turing:

- A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. Proceedings, London Mathematical Society, 2(42), pp. 230—265, and 2(43), pp. 544—546, 1936.

Turing 介绍这个模型是为了论证,用人工计算器可实现的所有详细的指令集,也可用适当定义的简单机器来实现。为了记录信息, Turing 原来的机器有一条双向无穷带和一只带头(见 4.5 节)。Post 独立地想到类似的模型,见:

- E. L. Post. Finite Combinatory Processes. Formulation I. Journal of Symbolic Logic, 1, pp. 103—105, 1936.

下列书籍包含对 Turing 机的有趣介绍:

- M. L. Minsky. Computation; Finite and Infinite Machines. Englewood Cliffs, N. J.: Prentice-Hall, 1967.
- F. C. Hennie. Introduction to Computability. Reading, Mass.: Addison-Wesley, 1977.

下列是关于本章和后续三章介绍的 Turing 机和相关概念的其他深入性书籍:

- M. Davis (ed.) The Undecidable. Hewlett, N. Y.: Raven Press, 1965 (本书包含关于本主题的多方面

的许多原始文章,包括上面引用过的 Turing 和 Post 的文章。)

- M. Davis (ed.) *Computability and Unsolvability*. New York, McGraw-Hill, 1958.
- S. C. Kleene. *Introduction to Metamathematics*. Princeton, N. J. ; D. Van Nostrand, 1952.
- W. S. Brainerd and L. H. Landweber. *Theory of Computation*. New York; John Wiley, 1974.
- M. Machtey and P. R. Young. *An Introduction to the General Theory of Algorithms*. New York: Elsevier North-Holland, 1978.
- H. Rogers, Jr. *The Theory of Recursive Functions and Effective Computability*. New York; McGraw-Hill, 1967.
- M. Sipser. *Introduction to the Theory of Computation*. Boston, Mass. ; PWS Publishers, 1996.
- J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Reading, Mass. ; Addison-Wesley, 1979.
- C. H. Papadimitriou. *Computational Complexity*. Reading, Mass. ; Addison-Wesley, 1994.
- H. Hermes. *Enumerability, Decidability, Computability*. New York, Springer-Verlag, 1996 (从 1965 年德文版翻译)。

我们对组合 Turing 机的概念和记号受到上述最后一本书影响。

在特点上与 2.4 节的“随机存取 Turing 机”类似的随机存取机器在下列文章里被研究过:

- S. A. Cook and R. A. Reckhow. Time-bounded random-access machines. *Journal of Computer and Systems Sciences*, 7(4), pp. 354—375, 1973.

原始和 μ 递归函数要归功于 Kleene:

- S. C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112, pp. 727—742, 1936.

Markov 算法(习题 2.6.5)来自:

- A. A. Markov. *Theory of Algorithms*. Trudy Math. Inst. V. A. Steklova, 1954. 英译本: *Isreal Program for Scientific Translations*, Jerusalem, 1961.

第 5 章 不可判定性

5.1 Church-Turing 论题

在本书里我们提出这样的问题：什么东西是可计算的？（而且更关心：什么东西是不可计算的？）我们已介绍了计算过程的各式各样的数学模型，它们完成具体的计算任务——判定语言、半判定语言、生成语言和计算函数等。在前一章我们看到，Turing 机可完成这些惊人的复杂的任务。我们也看到，考虑添加包括随机存取能力在内的某些附加特性到基本 Turing 机模型上，并没有增加 Turing 机可完成的任务。另外，沿完全不同的路线（即试图推广上下文无关文法），我们得出了能力恰与 Turing 机相同的语言生成器类。最后，通过试图形式化我们认为哪些数值函数可计算的直觉，定义了恰好就是递归函数的函数类。

所有这一切都表明我们已达到关于计算装置可被设计去干什么的自然上限，我们已胜利完成对计算过程（算法）的终极的和最一般的数学概念的探索——Turing 机就是正确的答案。不过在上一章我们也看到有些 Turing 机不应当称为“算法”；我们论证过，半判定语言，即通过永不停机来拒绝的 Turing 机不是有用的计算装置。相比之下，判定语言与计算函数（因此在所有输入上停机）的 Turing 机是有用的计算装置。算法的概念必须排除在某些输入上不停机的 Turing 机。

因此我们建议采用在所有输入上停机的 Turing 机作为与“算法”的直觉化概念相对应的精确的形式化概念。不能被变换成在所有输入上停机的 Turing 机的任何东西都不能被认为是算法，而所有这样的机器都被正确地称为算法。这个原理以 Church-Turing 论题而闻名。它是论题，不是定理，因为它不是数学结果；它只断言某个非形式化概念（算法）对应于某个数学对象（Turing 机）。因为不是数学命题，所以 Church-Turing 论题不能被证明。不过在理论上有可能在未来某一天 Church-Turing 论题被证伪，假设有人提出另一种计算模型，公认它是有说服力的和合理的，并且证明它能完成用 Turing 机不能完成的计算。没有人认为这是可能的。

通过采用关于算法的精确的数学概念，开辟出令人感兴趣的可能性，即形式化地证明某些计算问题不能被任何算法所解决。我们已知道有许多问题正期待这种可能性。在第 1 章里我们论证过，若利用字符串去表示语言则不能表示出所有语言；在任何字母表上只有可数个字符串，却有不可数个语言。有穷自动机、下推自动机、上下文无关文法、无限制文法和 Turing 机都是可用来规定语言的有穷对象的例子，并且它们自身可用字符串来描述（在下一节我们详细阐述把 Turing 机表示成字符串的具体方法）。相应地在任何字母表上只有可数多个递归和递归可枚举语言。所以，虽然我们尽量扩充计算机的能力，但是从根本上说，在所有可能的语言里只有无穷小的部分可用计算机去半判定或判定。

利用基数论证法去证明计算方法的局限性是平凡的,发现在模型里不能完成的计算任务的具体例子是更有趣和有益的。在前几章里我们确实成功地发现过非正则和非上下文无关的语言;在本章我们同样要发现非递归语言。不过有两点主要区别。首先,这些新的否定性结果不仅仅是暂时的挫折,等着下一章定义更强的计算装置来克服;根据 Church-Turing 论题,不能用 Turing 机完成的计算任务是不可能的,没有希望的,不可判定的。其次用来证明语言是非递归的方法不得不有别于为发掘上下文无关文法和有穷自动机的弱点而用过的“泵”定理。为了暴露 Turing 机的局限性,我们必须设计出发掘它们重要的能力的技术。将被探索的那个方面的 Turing 机能力是它们拥有的反省能力;我们指出 Turing 机都可收到 Turing 机的编码作为输入并且用有趣的方式操纵这些编码。我们要问,当 Turing 机操纵它自身的编码时会发生什么事情——这是对角化原理巧妙但不失为简单的应用。因此,如何编码 Turing 机使得另一台(或同一台!)Turing 机可操纵它是下一步的主题。

5.2 通用 Turing 机

是硬件还是软件构成计算的基础?你可能有关于这件事情——以及关于问题是否有意义和有成效——的观点。但是事实上我们在上一章介绍和开发的对算法的形式化——即 Turing 机——是“不可编程的”硬件,它专门解决一个具体的问题,它的指令都是“在工厂里固化了的”。

现在我们采取相反的观点。我们论证 Turing 机也是软件。即,证明存在某台大致上像通用计算机那样可编程的“一般的”Turing 机,以解决用 Turing 机可解决的任何问题。控制这台一般机器去模仿特殊机器 M 所用的“程序”不得不是对 M 的描述。换句话说,我们认为 Turing 机的形式化是可用来写程序的编程语言。然后用这种语言写的程序被通用 Turing 机——即用同样语言写的另一段程序——解释执行。用语言写的程序可解释执行用同样语言写的另一段程序,这并不是非常新鲜的想法,它是“自举式”语言处理器这种经典方法的基础。^① 但是为了继续我们在本书里的计划,我们必须在 Turing 机的背景里把这个要点精确化。

首先,我们必须提供说明 Turing 机的一般方法,使得 Turing 机的描述可用来作为其他 Turing 机的输入。即,我们必须定义一个语言,它的字符串都是 Turing 机的合法表示。一个问题自动出现了,无论我们为这种表示选择多么大的字母表,总是存在有更多状态和更多带符号的 Turing 机。显然必须把状态和带符号编码成固定字母表上的字符串。我们采取下列约定:表示 Turing 机状态的字符串形如 $\{q\}\{0,1\}^*$,即字母 q 后面跟着二进制字符串。同理,带符号总是表示成 $\{a\}\{0,1\}^*$ 里的字符串。

设 $M=(K,\Sigma,\delta,s,H)$ 是 Turing 机,设 i 和 j 是满足 $2^i \geq |K|$ 和 $2^j \geq |\Sigma|+2$ 的最小整

^① 语言实现者经常用同样的编程语言写编程语言的翻译程序。但是翻译程序自身如何被翻译呢?做这件事情的一种方法如下,用同样编程语言的简单子集写翻译程序,避免使用语言的较复杂的(难以翻译的)特性。然后用甚至更加简化型的同样语言为这个子集写翻译程序——更简化的任务。以这种方式继续下去,直到你的语言像汇编语言那样简单和清楚为止,因此它就被直接翻译出来。

数。于是, K 里的每个状态都表示成 q 后面跟着长度为 i 的二进制字符串, Σ 里的每个符号类似地表示成字母 a 后面跟着 j 位的二进制字符串。另外, 带头方向 \leftarrow 和 \rightarrow 被当作“名义上的带符号”(它们是在 j 的定义里有“+2”这一项的原因)。对特殊符号 $\sqcup, \triangleright, \leftarrow, \rightarrow$ 的表示, 分别被固定成 4 个在字典序下最小的符号: \sqcup 总是表示成 $a0^j$, \triangleright 表示成 $a0^{j-1}1$, \leftarrow 表示成 $a0^{j-2}10$, \rightarrow 表示成 $a0^{j-2}11$ 。初始状态总是表示成在字典序下的第一种状态, 即 $q0^i$ 。注意我们要求在符号 a 和 q 后面跟着的字符串里, 用前缀的 0 来保证总长度达到所要求的长度。

我们把整台 Turing 机 M 的表示记作“ M ”。“ M ”包括状态转移表 δ , 即它是形如 (q, a, p, b) 的字符串的序列, 其中 q 和 p 表示状态而 a 和 b 表示符号, 用逗号分隔并包含在圆括号里。我们约定从 $\delta(s, \sqcup)$ 开始, 以字典序下的升序排列四元组。停机状态集 H 将被直接确定, 因为停机状态不出现在“ M ”的任何四元组的第一分量上。若 M 判定语言, 因而 $H = \{y, n\}$, 则我们约定 y 是这两种停机状态里在字典序下较小的。

通过这种方法, 任意的 Turing 机都可被表示出来。我们用同样方法表示 Turing 机的字母表里的字符串。任何字符串 $w \in \Sigma^*$ 都有唯一的表示, 即它的符号的表示的并置, 也记作“ w ”。

例 5.2.1 考虑 $M = (K, \Sigma, \delta, s, \{h\})$, 其中 $K = \{s, q, h\}$, $\Sigma = \{\sqcup, \triangleright, a\}$, 并且 δ 如表 5-1 所示。

表 5-1

状态	符号	δ
s	a	(q, \sqcup)
s	\sqcup	(h, \sqcup)
s	\triangleright	(s, \rightarrow)
q	a	(s, a)
q	\sqcup	(s, \rightarrow)
q	\triangleright	(q, \rightarrow)

表 5-2

状态/符号	表示
s	$q00$
q	$q01$
h	$q11$
\sqcup	$a000$
\triangleright	$a001$
\leftarrow	$a010$
\rightarrow	$a011$
a	$a100$

因为有三种状态属于 K 和三种符号属于 Σ , 所以 $i=2$ 和 $j=3$ 。这些数是满足 $2^i \geq 3$ 和 $2^j \geq 3+2$ 的最小整数。状态和符号表示如表 5-2 所示。

因此, 字符串 $\triangleright aa \sqcup a$ 的表示是

$$“\triangleright aa \sqcup a” = a001a100a100a000a100$$

Turing 机 M 的表示“ M ”是下列字符串: “ M ” = $(q00, a100, q01, a000), (q00, a000, q11, a000), (q00, a001, q00, a011), (q01, a100, q00, a011), (q01, a000, q00, a011), (q01, a001, q01, a011)$ 。◇

现在我们准备就绪讨论通用 Turing 机 U , 它用其他机器的编码作为程序来指导它的操作。直观上, U 收到两个自变量, 分别是机器 M 的描述“ M ”和输入字符串 w 的描述“ w ”。我们希望 U 具有下列性质: U 在输入“ M ”“ w ”上停机当且仅当 M 在输入 w 上停机。用我们在上一章里开发的 Turing 机的函数记号, 就是

$$U(“M”“w”) = “M(w)”$$

我们实际上描述的不是单带机器 U , 而是密切相关的 3 带机器 U' (然后 U 将是模拟 U' 的单带 Turing 机)。具体地说, U' 如下利用三条带: 第一条带包含 M 当前带内容的编码, 第二条带包含 M 自身的编码, 第三条带包含在被模拟的计算中 M 当前的状态的编码。

机器 U' 启动时有某个字符串 " M " " w " 在第一条带上, 另外两条带都是空格 (若输入字符串不是这种形式, 则它如何操作是无关紧要的)。首先 U' 把 " M " 移动到第二条带上, 并且把 " w " 平移到第一条带的左端, 让它前面与 " $\triangleright \sqcup$ " 相邻。因此在这个时刻第一条带包含 " $\triangleright \sqcup w$ "。 U' 在第三条带上写下 M 的初始状态 s 的编码, 它总是 q_0' (U' 通过检查 " M " 可轻易地确定 i 和 j)。现在 U' 开始模拟 M 的每步计算。在这样被模拟的每步之间, U' 把第二和第三条带上的带头保持在带的左端, 让第一条带的带头扫描 M 在相应的时刻正在扫描的符号的编码里的 a 。

U' 如下模拟 M 的每步: U' 扫描第二条带直到发现下述的四元组为止, 这个四元组的第一分量与写在第三条带上的状态编码相匹配, 第二分量与在第一条带里扫描的符号编码相匹配。若它发现这样的四元组, 则 U' 把第三条带上的状态改成四元组的第三分量, 并且在第一条带里完成四元组的第四分量所提示的操作。若第四分量是 M 的带字母表的符号的编码, 则在第一条带里写上这个符号。若第四分量是 $a_0^{j-2}10$, 即 \leftarrow 的编码, 则 U' 把第一只带头向左移动到下一个 a 符号上, 若第四分量是 \rightarrow 的编码, 则带头向右移动到下一个 a 符号上。若遇到 \sqcup , 则 U' 必须把它转换成 a_0' , 即 M 的空格的编码。

若在某步在第二条带里找不到规定的状态与符号的组合, 则意味着第三条带上的状态是停机状态。 U' 也在适当的状态里停机。这样就完成了对 U' 的操作的描述。

习 题

5.2.1 回忆例 4.1.1 里的 Turing 机 M 。

- (a) 字符串 " M " 是什么?
- (b) 字符串 aaa 的表示是什么?
- (c) 假设本章描述的通用 (3 带) Turing 机 U' 模拟 M 在输入 aaa 上的操作。在模拟开始时 U' 的带的内容是什么? 在开始模拟 M 的第三步时呢?

5.2.2 通过与通用 Turing 机的类比, 我们可能希望设计接受语言 $\{ \langle M \rangle \langle w \rangle : w \in L(M) \}$ 的通用有穷自动机 U 。解释为什么不可能存在通用有穷自动机。

5.3 停机问题

假设你用喜欢的编程语言写了完成下列不寻常操作的程序: 它收到用同样语言写的程序 P 和这个程序的输入 X 作为输入。通过某些聪明的分析, 你的程序总是正确地判定在输入 X 上程序 P 是否停机 (若 P 停机则它返回“是”), 或者 P 是否死循环 (若 P 死循环则它返回“否”)。你把这段程序命名为 $\text{halts}(P, X)$ 。

这是最有价值的程序。它发现使得其他程序在某些输入上死循环的所有微妙的编程错误。你可用它完成许多不寻常的事情。这里是一个有点微妙的例子: 你可用它写用不祥的名字 $\text{diagonal}(X)$ 命名的另一段程序 (回忆第 1.5 节里用对角化证明 $2^{\mathbb{N}}$ 不可数):

$\text{diagonal}(X)$

$a : \text{if } \text{halts}(X, X) \text{ then goto } a \text{ else halt}$

注意 $\text{diagonal}(X)$ 做什么事情: 如果你的 halts 程序断定当程序 X 用它自身 X 作为输入时 X 停机, 那么 $\text{diagonal}(X)$ 死循环; 否则它停机。

现在出现无法回答的问题: $\text{diagonal}(\text{diagonal})$ 是否停机? 它停机当且仅当对 $\text{halts}(\text{diagonal}, \text{diagonal})$ 的调用返回“否”; 换句话说, 它停机当且仅当它不停机。这是矛盾, 我们必须得出结论说把我们领向此路的唯一假设是假的, 程序 $\text{halts}(P, X)$ 其实并不存在。也就是说, 没有程序, 没有算法可解决假设 halts 解决的问题; 辨别任意的程序是停机还是死循环。

这种论证是熟悉的, 它不仅来源于你在过去对计算机科学的接触, 而且来源于一般的 20 世纪的文化。要点在于, 为了给出形式化的在数学上严格的这种类型的悖论, 现在我们已介绍了所有必要的概念和方法。我们有完备的算法记号, 某种“编程语言”; Turing 机。事实上, 在上一节里我们介绍了所需要的最后一个特性; 我们开发了允许我们的“程序”去操纵其他程序和它们的输入的框架, 恰恰像我们虚构的程序 $\text{halts}(P, X)$ 所做的那样。因此我们准备就绪定义非递归的语言并且证明它非递归。设

$H = \{ \langle M \rangle \langle w \rangle : \text{Turing 机 } M \text{ 在输入 } w \text{ 上停机} \}$

首先注意 H 是递归可枚举的; 它恰好是用上一节里的通用 Turing 机半判定的语言。确实, 在输入 $\langle M \rangle \langle w \rangle$ 上, 恰好当输入属于 H 时 U 才停机。

另外, 若 H 是递归的, 则每一个递归可枚举语言都是递归的。换句话说, H 掌握着我们在 4.2 节里所提问题的答案, 是否所有递归可枚举语言也是 Turing 机可判定的: 答案是肯定的当且仅当 H 是递归的。因为假设某个 Turing 机 M_0 确实判定 H , 则给定半判定语言 $L(M)$ 的任何具体的 Turing 机 M , 我们可设计判定 $L(M)$ 的 Turing 机 M' 如下: 首先, M' 把输入带从 $\triangleright \sqcup w \sqcup$ 变换成 $\triangleright \langle M \rangle \langle w \rangle \sqcup$, 然后在这个输入上模拟 M_0 。根据假设, M_0 正确地判定出 M 是否接受 w 。提前使用在第 7 章里处理的内容, 则我们说存在从所有递归可枚举语言到 H 的归约, 因而 H 是关于递归可枚举语言类完全的。

但是通过形式化上述对 $\text{halts}(P, X)$ 的论证, 我们可证明 H 不是递归的。首先, 如果假设 H 是递归的, 那么

$H_1 = \{ \langle M \rangle : \text{Turing 机 } M \text{ 在输入字符串 } \langle M \rangle \text{ 上停机} \}$

也是递归的。(H_1 表示对角化程序的 $\text{halts}(X, X)$ 部分。) 假设存在判定 H 的 Turing 机 M_0 , 那么判定 H_1 的 Turing 机 M_1 只需要把输入字符串 $\triangleright \sqcup \langle M \rangle \sqcup$ 变换成 $\triangleright \sqcup \langle M \rangle \langle M \rangle \sqcup$, 然后再把控制交给 M_0 。因此, 证明 H_1 不是递归的就足够了。

其次, 假设 H_1 是递归的, 那么 H_1 的补也是递归的:

$\overline{H_1} = \{ w : \text{要么 } w \text{ 不是一台 Turing 机的编码,}$

$\text{要么 } w \text{ 是一台在 } \langle M \rangle \text{ 上不停机的 Turing 机 } M \text{ 的编码} \}$

这是因为递归语言类在补运算下封闭(定理 4.2.2)。顺便说一句, $\overline{H_1}$ 是对角化语言, 类似于 diagonal 程序, 也是证明的最后一个步骤。

但是 $\overline{H_1}$ 甚至不可能是递归可枚举的, 就更不用说递归了。因为假设 M^* 是半判定 $\overline{H_1}$ 的 Turing 机。“ M^* ”是否属于 $\overline{H_1}$? 根据 $\overline{H_1}$ 的定义, $M^* \in \overline{H_1}$ 当且仅当 M^* 不接受输入字符

串“ M^* ”。但是假设 M^* 半判定 $\overline{H_1}$, 所以 $M^* \in \overline{H_1}$ 当且仅当 M^* 接受“ M^* ”。我们得出结论说 M^* 接受“ M^* ”当且仅当 M^* 不接受“ M^* ”。这是荒谬的, 所以 M_0 存在的假设必定是错误的。

让我们总结本节内容的发展。我们希望发现是否每一个递归可枚举语言也是递归的。我们观察到这个命题为真当且仅当具体的递归可枚举语言 H 是递归的。我们分两步从 H 推导出语言 $\overline{H_1}$, 为了让 H 是递归的, $\overline{H_1}$ 不得不是递归的。但是通过对角化, $\overline{H_1}$ 是递归的假设导致了逻辑矛盾。因此我们已证明了下列最重要的定理。

定理 5.3.1 语言 H 不是递归的, 所以, 递归语言类是递归可枚举语言类的真子集。

我们以前说过, 这种论证是在 1.5 节里用来证明 $2^{\mathbb{N}}$ 不可数的对角化原理的实例。为了看出为什么以及为了再次强调证明的实质, 让我们在用来编码 Turing 机的字母表上定义字符串之间的二元关系 $R: (u, w) \in R$ 当且仅当对某个接受 w 的 Turing 机 M , $u = "M"$ 。(R 是改头换面的 H 。) 现在对每个字符串 u , 设

$$R_u = \{w : (u, w) \in R\}$$

(这些 R_u 对应于递归可枚举语言), 并且考虑 R 的对角化, 即

$$D = \{w : (w, w) \notin R\}$$

(D 是 $\overline{H_1}$)。根据对角化原理, 对所有 u , $D \neq R_u$, 即 $\overline{H_1}$ 是与任何递归可枚举语言都不同的语言。

为什么对所有 u , $D \neq R_u$? 因为正是根据 D 的构造, D 与每个 R_u (因此与每个递归可枚举语言) 至少在一个字符串上, 即 u 上, 不同。

我们在 4.2 节末尾提出过两个问题(“是否每一个递归可枚举语言也是递归的?”以及“递归可枚举语言类在补运算下是否封闭?”), 定理 5.3.1 否定地回答了两个问题中的第一个问题。但是同样的证明给出了对另一个问题的回答。容易看出, H_1 像 H 那样是递归可枚举的, 并且我们证明了 $\overline{H_1}$ 不是递归可枚举的。因此我们证明了下列结果。

定理 5.3.2 递归可枚举语言类在补运算下不封闭。

习 题

5.3.1 我们可不用对角化论证发现非递归语言的具体例子。忙碌的海狸函数 $\beta: \mathbb{N} \mapsto \mathbb{N}$ 定义如下: 对每个整数 n , $\beta(n)$ 是最大的数 m 使得存在有字母表 $\{\triangleright, \sqcup, a, b\}$ 并且恰有 n 个状态的 Turing 机, 当它在空格带上启动时, 它最终在格局 $(h, \triangleright \sqcup a^m)$ 停机。

(a) 证明若 f 是任何递归函数, 则存在整数 k_f 使得 $\beta(n + k_f) \geq f(n)$ 。(k_f 是 Turing 机 M_f 里的状态数, 若 M_f 在输入 a^n 上启动, 则 M_f 停机时在带上有 $a^{f(n)}$ 。)

(b) 证明 β 不是递归的。(假设它是, 那么 $f(n) = \beta(2n)$ 也是。应用上面(a)里的结果。)

5.3.2 我们知道递归可枚举语言类在补运算下不封闭。证明它在并和交运算下封闭。

5.3.3 证明递归语言的类在并、补、交、连接和 Kleene 星号等运算下封闭。

5.4 与 Turing 机有关的不可判定问题

我们已证明了一个重大的结果。但是考虑到 Church-Turing 论题,还是让我们回过头来在直觉层次上看看它说些什么。因为证明得出了 H 不是递归的,而且我们承认任何算法可转变成在所有输入上停机的 Turing 机的原理,所以我们必须得出结论说没有对任意给定的 Turing 机 M 和输入字符串 w 判定 M 是否接受 w 的算法。没有算法的问题称为不可判定的或不可解的,我们将在本章看到许多这样的问题。最有名的和最基本的不可判定问题是辨别给定的 Turing 机在给定的输入上是否停机的问题,我们刚刚证明了它的不可判定性。这个问题通常称为 Turing 机的停机问题。

注意,停机问题的不可判定性丝毫没有蕴涵,在某些情形下有可能预测 Turing 机是否在输入上停机也是不可能的。在例 4.1.2 里我们设法得出结论说,某台简单机器在某个输入上必定永不停机。或者检查 Turing 机的状态转移表,例如验证停机状态是否在任何地方出现过。如果没有,那么机器在任何输入字符串上都不停机。这样的或更复杂的分析可对某些情形产生某些有用信息,但是我们的定理表明任何这样的分析最终必定是在某些情况下要么没有结果,要么产生错误结果;没有正确地判定所有情形的完全地一般方法。

一旦我们通过对角化证明了停机问题是不可判定的,就可由此得出一大批问题的不可判定性。证明这些结果不是继续通过对角化,而是通过归约;我们在每种情形里证明假设某个语言 L_2 是递归的,那么某个语言 L_1 也是递归的,但是已知 L_1 不是递归的。

定义 5.4.1 设 $L_1, L_2 \subseteq \Sigma^*$ 是两个语言。从 L_1 到 L_2 的归约是递归函数 $\tau: \Sigma^* \rightarrow \Sigma^*$ 使得 $x \in L_1$ 当且仅当 $\tau(x) \in L_2$ 。

读者必须小心理解归约被应用的“方向”。为了证明语言 L_2 不是递归的,我们必须找到一个已知非递归的语言 L_1 ,然后把 L_1 归约到 L_2 。把 L_2 归约到 L_1 等于什么事情也没做;这只证明若我们可判定 L_1 则 L_2 是可判定的——我们知道我们不能判定 L_1 。

形式地,在不可判定性证明里归约的正确使用方法如下:

定理 5.4.1 若 L_1 不是递归的,并且存在从 L_1 到 L_2 的归约,则 L_2 也不是递归的。

证明: 假设 L_2 是递归的,比方说用 Turing 机 M_2 判定,并设 T 是计算归约 τ 的 Turing 机。那么 Turing 机 TM_2 判定 L_1 。但是 L_1 是不可判定的,矛盾。 ■

下一步我们用归约证明与 Turing 机有关的几个问题是不可判定的。

定理 5.4.2 与 Turing 机有关的下列问题是不可判定的。

- (a) 给定 Turing 机 M 和输入字符串 w , M 是否在输入 w 上停机?
- (b) 给定 Turing 机 M , M 是否在空带上停机?
- (c) 给定 Turing 机 M , 究竟有没有 M 在上面停机的字符串?
- (d) 给定 Turing 机 M , M 是否在每一个输入字符串上都停机?
- (e) 给定两台 Turing 机 M_1 和 M_2 , 它们是否在同样的字符串上停机?
- (f) 给定 Turing 机 M , M 半判定的语言是否正则? 是否上下文无关? 是否递归?
- (g) 另外,存在某台固定机器 M_0 , 对它下列问题是不可判定的: 给定 w , M_0 是否在 w 上停机?

证明: (a) 在前一节证明过。

(b) 我们描述从 H 到语言 L 的归约, 其中

$$L = \{ \langle M \rangle : M \text{ 在 } e \text{ 上停机} \}$$

给定对 Turing 机 M 和输入 w 的描述, 我们的归约仅仅构造对 Turing 机 M_w 的描述, M_w 如下操作: 当 M_w 在空格带上 (即在格局 (s, \triangleright) 里) 启动时, M_w 把 w 写在带上然后开始模拟 M 。换句话说, 若 $w = a_1 \cdots a_n$, 则 M_w 仅仅是机器

$$Ra_1Ra_2R \cdots Ra_nL \sqcup M$$

容易看出把 “ M ” “ w ” 映射成 “ M_w ” 的函数 τ 确实是递归的。

(c) 我们把在 (b) 部分里证明是非递归的语言 L 归约到语言 $L' = \{ \langle M \rangle : M \text{ 在某个输入上停机} \}$ 上, 具体作法如下。给定对任何 Turing 机 M 的表示, 我们的归约构造对 Turing 机 M' 的表示, M' 删除给定的任何输入, 然后在空字符串上模拟 M 。显然, M' 在某个字符串上停机当且仅当它在所有字符串上停机, 当且仅当 M 在空字符串上停机。

(d) 对 (c) 部分的论证在这里同样有效, 因为 M' 被构造使它接受某个输入当且仅当它接受每一个输入。

(e) 我们把在 (d) 部分里的问题归约到这个问题上。给定对机器 M 的描述, 我们的归约构造字符串

$$\tau(\langle M \rangle) = \langle M \rangle \langle y \rangle$$

其中 “ y ” 是对立即接受任何输入的机器的描述。显然, M 和 y 这两台机器接受同样字符串当且仅当 M 接受所有字符串。

(f) 我们把在上面 (b) 部分里的问题归约到现在这个问题上。我们证明如何修改任何 Turing 机 M 来获得 Turing 机 M' , 使得根据 M 在空字符串上是否停机, M' 要么在属于 H 的字符串上停机, 要么不在任何字符串上停机。因为没有算法辨别 M 是否在空字符串上停机, 所以没有算法辨别 $L(M)$ 是等于 \emptyset (\emptyset 是正则的、上下文无关的和递归的), 还是等于 H (H 不属于上面三种)。首先, M' 保存输入字符串并且开始模拟 M 在输入 e 上的所有动作。当 M 停机时 M' 恢复它的输入并且在那个输入上完成通用 Turing 机 U 的操作。因此, M' 要么不在任何字符串上停机, 因为它永远完不成在输入 e 上对 M 的模拟, 要么它恰好在属于 H 的字符串上停机。

(g) 在定理的叙述里提到的固定机器 M_0 恰好就是通用 Turing 机 U 。 ■

习 题

5.4.1 说 Turing 机 M 在输入字符串 w 上使用 k 个带方格, 当且仅当存在 M 的格局 $(q, u \underline{a} v)$, 使得 $(s, \triangleright \sqcup w) \vdash_M^* (q, u \underline{a} v)$ 并且 $|u \underline{a} v| \geq k$ 。

(a) 证明下列问题是可解的: 给定 Turing 机 M , 输入字符串 w 和数字 k , M 是否在输入 w 上使用 k 个带方格?

(b) 假设 $f: \mathbb{N} \rightarrow \mathbb{N}$ 是递归的。证明下列问题是可解的: 给定 Turing 机 M 和输入字符串 w , M 是否在输入 w 上使用 $f(|w|)$ 个带方格?

(c) 证明下列问题是不可判定的: 给定 Turing 机 M 和输入字符串 w , 是否存在 $k \geq 0$ 使得 M 在输入 w 上不使用 k 个带方格? (即 M 是否在输入 w 上只使用有穷多个带方格?)

- 5.4.2 与 Turing 机有关的下列问题哪些是可解的?哪些是不可判定的?仔细地解释你的回答。
- (a) 给定 Turing 机 M 、状态 q 和字符串 w ,判定当 M 在输入 w 上从初始状态启动时, M 是否至少到达一次状态 q 。
 - (b) 给定 Turing 机 M 和两个状态 p 和 q ,判定是否存在带有状态 p 的格局,它产生带有状态 q 的格局。
 - (c) 给定 Turing 机 M 和状态 q ,判定究竟是否存在产生带有状态 q 的格局的格局。
 - (d) 给定 Turing 机 M 和符号 a ,判定当 M 在空格带上启动时, M 是否至少写一次符号 a 。
 - (e) 给定 Turing 机 M ,判定当 M 在空格带上启动时, M 是否至少写一次非空格符。
 - (f) 给定 Turing 机 M 和字符串 w ,判定当 M 在输入 w 上启动时, M 是否至少把带头向左移动一次。
 - (g) 给定两台 Turing 机,判定是否其中一台半判定的语言等于另一台半判定的语言的补。
 - (h) 给定两台 Turing 机,判定是否存在让它们都停机的输入字符串。
 - (i) 给定 Turing 机 M ,判定 M 半判定的语言是否为有穷的。
- 5.4.3 证明这是不可判定的问题:给定 Turing 机 M ,判定是否存在某个字符串 w 使得 M 在输入 w 上的计算里进入每个状态。
- 5.4.4 证明即使限制在状态数很少并且固定的 Turing 机上,Turing 机的停机问题仍然是不可判定的。(若要求状态数固定但是并不少,则通用 Turing 机的存在性就证明了这个结果。证明任意的 Turing 机如何在某种适当意义下,被有大约半打(6个)状态但是有非常大的字母表的 Turing 机所模拟。实际上,三个状态就足够了;若我们允许机器在一步里又写又移动,则两个状态也足够了。)
- 5.4.5 证明在某种意义下(我们把它留给你去具体说明),用没有带但是有两只计数器的自动机可模拟任何 Turing 机。计数器是只有一个符号的下推存储器,另外还有永不被删除的可识别的底端标记。因此可认为计数器是容纳一进制数的寄存器。在计数器上的可能操作如下:加 1;看它是否包含 0;若它不包含 0 则减 1。证明这些双计数器机器的停机问题是不可判定的。(提示:首先证明如何用有两个符号的两只下推存储器去模拟 Turing 机的一条带;其次通过用一进制来编码下推存储器的内容,证明可用四只计数器来模拟两只下推存储器;最后通过把四个数 a, b, c, d 编码成 $2^a 3^b 5^c 7^d$,用两只计数器去模拟四只计数器。)

5.5 与文法有关的不可解问题

不可解问题不但出现在 Turing 机的领域里,而且事实上出现在所有数学领域里。例如存在多个与文法有关的不可判定问题,总结如下。

定理 5.5.1 下列每个问题都是不可判定的:

- (a) 对给定的文法 G 和字符串 w ,判定是否 $w \in L(G)$ 。

(b) 对给定的文法 G , 判定是否 $\epsilon \in L(G)$ 。

(c) 对两个给定的文法 G_1 和 G_2 , 判定是否 $L(G_1) = L(G_2)$ 。

(d) 对任意的文法 G , 判定是否 $L(G) = \emptyset$ 。

(e) 另外, 存在某个固定的文法 G_0 , 使得判定任意给定的字符串 w 是否属于 $L(G_0)$, 这是不可判定的。

证明: 我们证明从停机问题到 (a) 的归约, 非常类似的归约证明其余的部分。给定任意 Turing 机 M 和字符串 w , 我们仅仅对 M 应用在定理 4.6.1 的证明里给出的构造, 以产生文法 G 使得 $L(G)$ 是 M 半判定的语言。由此得出 $w \in L(G)$ 当且仅当 M 在输入 w 上停机。■

因为我们已证明了文法恰好与 Turing 机同样强 (定理 4.6.1), 所以上面的不可判定性结果大概不会令人吃惊。不过更令人惊讶的是与上下文无关文法及其相关系统——这是一个相当简单和有限的领域——有关的类似问题是不可判定的。自然, 不可判定问题不包括辨别是否 $w \in L(G)$ 或是否 $L(G) = \emptyset$, 这些问题可用算法解决, 并且事实上这些算法是有效的 (回忆定理 3.6.1)。不过许多其他问题是不可解的。

定理 5.5.2 下列每个问题都是不可判定的:

(a) 给定上下文无关文法 G , 是否 $L(G) = \Sigma^*$?

(b) 给定两个上下文无关文法 G_1 和 G_2 , 是否 $L(G_1) = L(G_2)$?

(c) 给定两台推自动机 M_1 和 M_2 , 它们是否恰好接受同样的语言?

(d) 给定下推自动机 M , 找出状态数最少的等价的下推自动机。

证明: 主要的技术困难是证明 (a), 其余部分很容易由此得出。我们把在定理 5.5.1 (d) 部分里证明是不可判定的问题, 即判定给定的一般性文法究竟是否生成任何字符串的问题, 归约到 (a) 部分。

设 $G_1 = (V_1, \Sigma_1, R_1, S_1)$ 是一般性文法。我们首先修改这个文法如下: 设对 $i = 1, \dots, |R_1|$, G_1 的规则是 $\alpha_i \rightarrow \beta_i$ 。我们向 V_1 添加 $|R_1|$ 个新的非终结符 $A_i, i = 1, \dots, |R_1|$, R_1 里的每条规则对应一个非终结符, 并且对 $i = 1, \dots, |R_1|$, 用两条规则 $\alpha_i \rightarrow A_i$ 和 $A_i \rightarrow \beta_i$ 去替换第 i 条规则。让我们把得出的文法称为 $G'_1 = (V'_1, \Sigma_1, R'_1, S_1)$ 。显然 $L(G'_1) = L(G_1)$; 在 G_1 里字符串的任何推导可变成同样字符串在 G'_1 里的标准推导, 其中每个奇数步应用形如 $u_i \rightarrow A_i$ 的规则, 而随后的偶数步应用形如 $A_i \rightarrow v_i$ 的规则。

我们从 G'_1 出发在字母表 Σ 上构造上下文无关文法 G_2 , 使得 $L(G_2) = \Sigma^*$ 当且仅当 $L(G_1) = \emptyset$ 。于是这个归约就证明 (a) 部分。

假设存在在 G'_1 里的终结符字符串的推导; 根据上述讨论, 我们假设它是标准推导, 即

$$S \Rightarrow x_1 \Rightarrow x_2 \Rightarrow x_3 \Rightarrow \dots \Rightarrow x_n$$

其中对所有 $i, x_i \in V'_1^*$, 并且 $x_n \in \Sigma^*$, n 是偶数, i 为奇数的每个 x_i 恰好是一个 A_j 非终结符, 而 i 为偶数的每个 x_i 不包含 A_j 非终结符。这样的推导可表示成在字母表 $\Sigma = V \cup \{\Rightarrow\}$ 里的字符串, 恰恰就是上面显示的字符串。事实上, 由于我们很快就会清楚的原因, 我们更关心牛回头型^① 的与标准推导相对应的字符串, 其中奇数编号的 x_i (即 x_1, x_3 , 等等)

^① 牛回头, 来源于希腊文单词 *Boustrophedon*, 意思是“像牛来回调头”, 它表示以相反方向写轮流两行的书写格式, 一行从左向右而下一行从右向左。

是倒转的:

$$S \Rightarrow x_1^R \Rightarrow x_2^R \Rightarrow x_3^R \Rightarrow \cdots \Rightarrow x_{n-1}^R \Rightarrow x_n$$

现在考虑语言 $D_{G_1} \subseteq \Sigma^*$, 它包括在 G_1 里的终结符字符串的所有这种牛回头型的标准推导。显然 $D_{G_1} = \emptyset$ 当且仅当 $L(G_1) = \emptyset$ 。换句话说, 就补语言 $\overline{D_{G_1}} = \Sigma^* - D_{G_1}$ 而言,

$$\overline{D_{G_1}} = \Sigma^* \quad \text{当且仅当} \quad L(G_1) = \emptyset$$

因此, 为了完成证明, 我们必须证明的全部内容就是 $\overline{D_{G_1}}$ 是上下文无关的。

为了达到这个目的, 让我们更仔细地查看语言 $\overline{D_{G_1}}$ 。字符串 w 要满足什么条件才属于这个语言? 即字符串 w 何时不是 G_1 里的终结符字符串的牛回头型的标准推导? 回答是当且仅当至少下列条件之一成立:

- (1) w 开头不是 $S_1 \Rightarrow$ 。
- (2) w 结尾不是 $\Rightarrow v$, 其中 $v \in \Sigma_1^*$ 。
- (3) w 含有奇数个 \Rightarrow 。
- (4) w 形如 $u \Rightarrow y \Rightarrow v$ 或 $u \Rightarrow y$, 其中
 - (a) u 含有 \Rightarrow 的偶数次出现。
 - (b) y 恰好含有 \Rightarrow 的一次出现, 并且
 - (c) y 不形如 $y = y_1 A_i y_2 \Rightarrow y_2^R \beta_i y_1^R$, 其中 $i \leq |R_1|$, $y_1, y_2 \in \Sigma_1^*$, β_i 是 G_1 的第 i 条规则的右方。
- (5) w 形如 $u \Rightarrow y \Rightarrow v$, 其中
 - (a) u 含有 \Rightarrow 的奇数次出现。
 - (b) y 恰好含有 \Rightarrow 的一次出现, 并且
 - (c) y 不形如 $y = y_1 \alpha_i y_2 \Rightarrow y_2^R A_i y_1^R$, 其中 $i \leq |R_1|$, $y_1, y_2 \in \Sigma_1^*$, α_i 是 G_1 的第 i 条规则的左方。

若 w 满足这 5 个条件里的任何一个条件, 并且只有这样, 则 w 属于 $\overline{D_{G_1}}$ 。即, $\overline{D_{G_1}}$ 是上面从 (1) 到 (5) 描述的 5 个语言 L_1, L_2, L_3, L_4 和 L_5 的并。我们断言所有这 5 个语言都是上下文无关的。事实上, 我们可构造出生成它们的上下文无关方法。对前三个语言, 因为它们碰巧是正则语言, 所以这是平凡的。

对 L_4 和 L_5 , 我们的论证是间接的; 可为它们每个都设计出非确定型下推自动机。根据定理 3.6.1(b), 存在着从任何下推自动机 M 构造上下文无关文法 G 使得 $L(G) = L(M)$ 的算法。

接受 L_4 的下推自动机 M_4 分两个阶段工作。在第一阶段里, 它从左向右扫描输入 w , 总是记忆着它看到奇数个或偶数个 \Rightarrow (这可用两个状态来实现)。当遇到偶数个 \Rightarrow 时, M_4 有两种选择, 并且在这两种选择之间非确定性地选择: 要么继续模 2 计数 \Rightarrow 的个数, 要么进入第二阶段。

在第二阶段里, M_4 把输入收集在栈里, 直到发现 \Rightarrow 为止。如果没有推入任何 A_i 符号, 或者推入超过一个 A_i 符号, 那么 M_4 接受, 即输入属于 L_4 。否则, M_4 把栈的内容 (从上向下读是形如 $y_2^R A_i y_1^R$ 的字符串) 与输入里直到下一个 \Rightarrow 为止的部分进行比较。如果在遇到 A_i 之前就发现不匹配, 或者如果发现在输入里没有用 β_i 替换 α_i (M_4 在它的状态空间里记

忆字符串 β_i), 或者如果在此之后发现不匹配, 或者如果在栈变空之后没有立即发现下一个 \Rightarrow (或者输入的结尾), 那么 M_4 同样接受。否则它拒绝。

这样就完成了对 M_4 的描述, 显然 $L(M_4) = L_4$ 。对 L_5 的构造是非常相似的。因此我们可设计出生成从 L_1 到 L_5 的每个语言的上下文无关文法, 所以我们可设计出生成它们的并的上下文无关文法。

因此, 给定任何一般性文法 G_1 , 我们可构造上下文无关文法 G_2 使得 $L(G_2) = \overline{D_{G_1}}$ 。但是我们知道 $\overline{D_{G_1}} = \Sigma^*$ 当且仅当 $L(G_1) = \emptyset$ 。我们得出结论说, 如果有判定任何给定的上下文无关文法是否生成全部 Σ^* 的算法, 那么可用这个算法判定给定的一般性文法是否生成空语言, 我们知道最后这件事是不可能的。(a)部分的证明就完成了。

(b)假如我们可辨别两个上下文无关文法是否生成同样的语言, 那么可设法辨别上下文无关文法是否生成 Σ^* ; 让第二个文法是确实生成 Σ^* 的平凡文法。

(c)假如我们可辨别两台下推自动机是否等价, 那么通过把两个上下文无关文法变换成接受同样语言的两台下推自动机, 然后测试这两台下推自动机的等价性, 可设法辨别两个上下文无关文法是否等价。

(d)假如存在极小化任意下推自动机的状态数的算法, 就像对有穷自动机存在这种算法那样, 那么我们可设法辨别给定的下推自动机是否接受 Σ^* ; 它接受当且仅当优化过的下推自动机只有一个状态并且接受 Σ^* 。单状态下推自动机是否接受 Σ^* , 是可判定的 (在习题 5.5.1 里证明)。

习 题

5.5.1 证明以下问题是可判定的: 给定单状态下推自动机 M , 判定是否 $L(M) = \Sigma^*$ 。(提示: 证明这样的自动机接受所有字符串当且仅当它接受所有长度为 1 的字符串。)

5.5.2 Post 对应系统是非空字符串的有序对的有穷集 P , 即 P 是 $\Sigma^* \times \Sigma^*$ 的有穷子集。 P 的匹配是任何字符串 $w \in \Sigma^*$ 使得, 对某个 $n > 0$ 和某些 (不必相异的) 有序对 $(u_1, v_1), (u_2, v_2), \dots, (u_n, v_n), w = u_1 u_2 \dots u_n = v_1 v_2 \dots v_n$ 。

(a) 证明以下问题是不可判定的: 给定 Post 对应系统, 判定它是否有匹配。(从受限制的情形开始, 其中匹配必须用 P 里特殊的有序对来开头。)

(b) 用上述的 (a) 去找出定理 5.5.2 的另一种证明。

5.5.3 (非确定型) 双头有穷自动机 (简称 THFA) 包括有穷控制器、输入带和在带上只读不写并且只能从左向右移动的两只带头。机器在初始状态启动时, 两只带头都在带的最左边的方格里。每步转移都形如 (q, a, b, p) , 其中 q 和 p 是状态, a 和 b 是符号或 ϵ 。这步转移意味着当 M 在状态 q 里用第一只带头读到 a 并且用第二只带头读到 b 时, M 进入状态 p 。通过把两只带头都移出带的右端, 同时进入指定的终止状态, M 接受输入字符串。

(a) 证明这个问题是可解的: 给定 THFA M 和输入字符串 w , 判定 M 是否接受 w 。

(b) 证明这个问题是不可解的: 给定 THFA M , 判定是否存在 M 接受的任何字符串。(利用前一个问题。)

5.6 不可解的铺砖问题

给定砖的有穷集,每块砖是单位正方形。要求我们用这些砖的复制品来铺满平面的第一象限,如图 5-1 所示。我们有每块砖的无穷多块复制品。

仅有的限制是必须在左下角放特殊的砖(即原点砖),只有特定的成对的砖才可水平地彼此相邻,只有特定的成对的砖才可垂直地彼此相邻(砖不可被旋转或翻面。)。给定砖的有穷集、原点砖和相邻规则,有没有判定第一象限能否被铺满的算法?

这个问题可形式化如下。铺砖系统是四元组 $\mathcal{D} = (D, d_0, H, V)$, 其中 D 是砖的有穷集, $d_0 \in D$, 并且 $H, V \subseteq D \times D$ 。根据 \mathcal{D} 的铺砖是函数 $f: \mathbb{N} \times \mathbb{N} \rightarrow D$ 使得下列关系成立:

$$f(0,0) = d_0,$$

$$(f(m,n), f(m+1,n)) \in H \quad \text{对于所有的 } m, n \in \mathbb{N},$$

$$(f(m,n), f(m,n+1)) \in V \quad \text{对于所有的 } m, n \in \mathbb{N}.$$

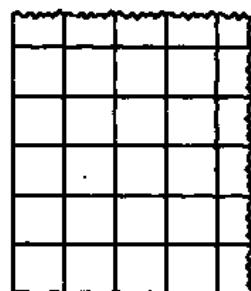


图 5-1

定理 5.6.1 给定铺砖系统,判定是否存在根据这个系统的铺砖,这个问题是不可判定的。

证明: 我们把给定 Turing 机 M , 判定 M 是否在输入 e 上不停机的问题归约到铺砖问题上。这个问题是停机问题的补,所以是不可判定问题。若这个问题被归约到铺砖问题上,则铺砖问题肯定是不可判定的。

基本想法是从任意 Turing 机 M 构造出铺砖系统 \mathcal{D} , 使得根据 \mathcal{D} 的铺砖(如果存在这种铺砖的话)表示 M 从空格带开始的无穷计算。 M 的格局水平地表示在铺砖里,后继的格局出现在前一个格局的上面。即水平维度表示 M 的带,垂直维度表示时间。若 M 在空输入上永不停机,则后继的行铺到无穷;但是若 M 在 k 步后停机,则铺到超过 k 行是不可能的。

在构造关系 H 和 V 时认为砖的边被标记了某种信息,这对我们是有帮助的,只有相邻边上的标记相同时,我们才允许砖与砖水平地或垂直地相邻。在水平的边上,这些标记要么是来自 M 的字母表的符号,要么是状态与符号的组合。铺砖系统被安排成如果铺砖是可行的,那么通过查看在第 n 行和第 $n+1$ 行之间的水平边标记,我们可读出 M 在 $n-1$ 步计算之后的格局。因此沿这样的边界只有一条边用状态与符号的有序对来标记,其他的边都用单个符号来标记。

在砖的垂直边上的标记要么空缺(它只能匹配同样无标记的垂直边),要么包含 M 的状态和我们用箭头指示的“方向”指针。(两种例外在下面(e)里给出)。在垂直的边上的这些标记都用来表示带头向左或向右地从一块砖移动到下一块砖。

具体地说,设 $M = (K, \Sigma, \delta, s, H)$ 。于是 $\mathcal{D} = (D, d_0, H, V)$, 其中 D 包含下列的砖:

(a) 对每个 $a \in \Sigma$, 如图 5-2 所示的砖,它仅仅表示从格局到格局向上时任何未改变的符号。

(b) 对每个 $a \in \Sigma$ 和 $q \in K$ 使得 $\delta(q, a) = (p, b)$, 其中 $p \in K, b \in \Sigma$, 如图 5-3 所示的

砖。这块砖表示带头位置向上并且也适当地改变状态和被扫描符号。

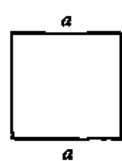


图 5-2



图 5-3

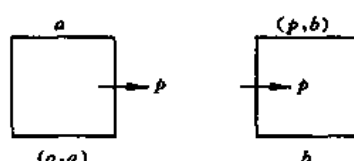


图 5-4

(c) 对每个 $a \in \Sigma$ 和 $q \in K$, 使得 $\delta(q, a) = (p, \rightarrow)$, 以及对每个 $b \in \Sigma - \{\triangleright\}$, 如图 5-3 所示的砖。这些砖表示带头位置从左向右移动一格, 同时适当地改变状态。注意我们自然不允许任何状态从左方进入左端符 \triangleright 。

(d) $\delta(q, a) = (p, \leftarrow)$ 的情形与 (c) 类似。如图 5-5 所示。这里不会遇到符号 \triangleright 。

这些砖是用 \mathcal{D} 模拟 M 的主体。剩下来只要指定启动计算的某些砖并且保证正确铺好最底下一行。

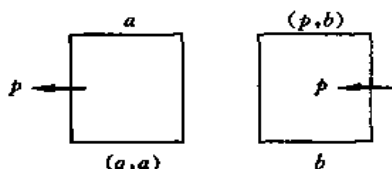


图 5-5

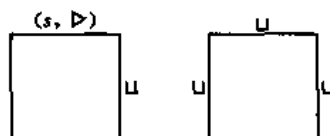


图 5-6

(e) 原点砖 d_0 如图 5-6 所示。它顶上的边指定 M 的初始状态 s 和符号 \triangleright 。即 M 不是在格局 $(s, \triangleright \sqcup)$ 上启动, 而是在 (s, \triangleright) 上启动; 根据关于 \triangleright 的约定, 下一个格局肯定是 $(s, \triangleright \sqcup)$ 。原点砖右侧的边用空格符号标记, 这条边只能用图 5-6(b) 所示最后一种砖左侧的边去匹配, 这种砖继续向右传播信息说在最底下一行每块砖顶上的边都用空格标记。

这就完成了对 \mathcal{D} 的构造。(e) 里的砖确保开头两行之间的边界被标记成 $(s, \triangleright) \sqcup \sqcup \sqcup \dots$; 其他的砖迫使每个后继边界被正确地标记。注意没有砖给出停机状态, 所以若 M 在 n 步之后停机, 则只能铺到 n 行。

例 5.6.1 考虑 Turing 机 $(K, \Sigma, \delta, s, \{h\})$, 其中 $\Sigma = \{\triangleright, \sqcup\}$, $K = \{s, h\}$, 并且给定 δ 是

$$\delta(s, \triangleright) = (s, \rightarrow)$$

$$\delta(s, \sqcup) = (s, \leftarrow)$$

这台机器仅仅从左到右再从右到左地来回移动带头, 永不移动到超出第一个带方格。与 M 的无穷计算相关联的平面铺砖如图 5-7 所示。

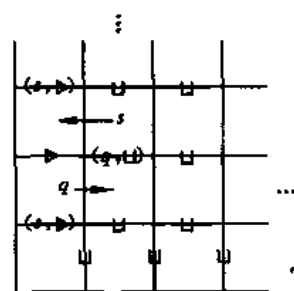


图 5-7

习 题

5.6.1 设 $M = (\{s, h\}, \{a, \sqcup, \triangleright\}, \delta, s, \{h\})$, 其中 $\delta(s, \sqcup) = (s, a)$, $\delta(s, a) = (s, \rightarrow)$ 。用本节里的构造找出与 M 关联的砖的集合, 并且说明利用这些砖对平面铺砖的开头四行。

- 5.6.2 证明存在某个固定的砖集 D 以及相邻规则集 H 和 V , 使得下列问题是不可判定的: 给定部分铺砖, 即对某个有穷子集 $S \subseteq \mathbb{N} \times \mathbb{N}$, 给定映射 $f: S \rightarrow D$ 使得 f 服从相邻规则, 能否把 f 扩充成整个平面的铺砖?
- 5.6.3 假设铺砖游戏的规则变成不是指定在原点上所放的特殊砖, 而是指定特殊的砖集并规定只可用这些砖来铺第一行。证明铺砖问题仍然是不可判定的。
- 5.6.4 假设铺砖游戏的规则改变如下: 砖不是完美的正方形, 而是可能沿着边界有各种凸凹。仅当两块砖的边界像拼图玩具的零件那样完美地吻合在一起时, 它们才可以彼此相邻地铺下来; 并且只有边界是直线的砖才可铺在边缘上。证明即使允许砖被旋转或翻面, 铺砖问题也仍然是不可判定的。(在这种形式的问题里没有指定的“原点砖”。)
- 5.6.5 假设我们认为方砖是通过四条边的颜色来确定的, 并且两条边可相邻的条件是它们有相同的颜色。证明如果允许砖被旋转或翻面, 那么可用任意非空的砖集来铺满整个第一象限(即使要求在原点上放特殊的砖)。

5.7 递归语言的性质

我们已看到每一个递归语言都是递归可枚举的, 但是这两个类是不一样的: 语言 H 就是见证不一样的一个例子。哪些递归可枚举语言是递归的? 回答这个问题的方式有许多种, 下面给出其中一种。

定理 5.7.1 语言是递归的当且仅当它和它的补都是递归可枚举的。

证明: 如果 L 是递归的, 那么根据定理 4.2.1, L 是递归可枚举的; 另外根据定理 4.2.2, \bar{L} 是递归的, 因此是递归可枚举的。

对另一个方向, 假设 M_1 半判定 L 并且 M_2 半判定 \bar{L} 。于是我们可构造判定 L 的 Turing 机 M 。为了方便, 我们把 M 描述成 2 带 Turing 机, 根据定理 4.3.1, M 可用 1 带 Turing 机来模拟。机器 M 首先在两条带上都写上输入字符串 w , 并且把带头都放在所复制的输入的左端。然后 M 平行地模拟 M_1 和 M_2 ; 在 M 的每步操作里, 在第一条带上完成 M_1 的一步计算, 在第二条带上完成 M_2 的一步计算。因为要么 M_1 要么 M_2 在 w 上必须停机, 但是不能都停机, 所以 M 最终达到这样的情况: 被模拟的 M_1 或 M_2 停机。当这种情况发生时, M 判定是 M_1 还是 M_2 停机, 并且相应地在状态 y 或 n 停机。 ■

存在对递归可枚举语言的另一种有趣的刻画: 递归可枚举语言恰好是那些可用 Turing 机枚举的语言。

定义 5.7.1 我们说 Turing 机 M 枚举语言 L 当且仅当对 M 的某个固定状态 q ,

$$L = \{w : (s, \triangleright \sqcup) \vdash_M^*(q, \triangleright \sqcup w)\}.$$

语言是 Turing 可枚举的当且仅当存在枚举它的 Turing 机。

也就是说, M 通过从空格带开始并且计算下去, 周期性地经过特殊状态 q (一个非停机状态) 来枚举 L 。 进入状态 q 标志着当前在 M 带上的字符串是 L 的成员; 然后 M 离开状态 q 并且当随后重新进入它时在带上又有 L 的某个其他成员; 注意 L 的成员们可按照任意顺序排列并且可重复。

定理 5.7.2 语言是递归可枚举的当且仅当它是 Turing 可枚举的。

证明: 假设 Turing 机 M 半可判定 L 。于是我们可设计 Turing 机 M' , 它不用字符串作为输入而是从空带启动, 系统地生成(例如按照字典序)在 L 的字母表上的所有字符串, 并且在每个字符串上完成 M 所完成的同样的计算。不幸的是达到这个目标的明显方式行不通: 新机器 M' 不能指望在开始下一个字符串上的计算之前, 先结束在上一个字符串上的计算, 因为即使存在 M 接受的但 M' 尚未生成的其他字符串, M' 有可能永远“挂起”在 M 的计算不停机的某个字符串上。(假如 M 判定 L , 这样的策略当然就行得通了, 见下一条定理的证明。)

解决的办法是基于我们在 1.4 节里用来证明可数多个集合的并集依然可数的“制作棒头”过程的翻版。 M' 不是试图完成生成每个字符串的计算, 而是完成下列操作序列:

(阶段 1) 首先 M' 在 M 字母表上(字典序下)的第一个字符串上完成 M 的一步计算。

(阶段 2) 然后 M' 在前两个字符串里的每个上完成 M 的两步计算。

(阶段 3) 然后 M' 在前三个字符串里的每个上完成 M 的三步计算, 等等。

当 M' 第一次找到 M 接受的字符串, 比方说 w_1 时, M' 把 w_1 写在带上, 并且在状态 q 里暂停以标志 $w_1 \in L$ 。

一般在找到 L 里第 i 个字符串 w_i 之后, M' 首先在状态 q 里显示它, 然后从阶段 1 一切重新开始, 等等, 在带上保存 w_i 。每当它找到 M 接受的字符串就首先与 w_i 比较, 若它们不相等则继续。若它发现刚刚找到的字符串与 w_i 相同, 则它寻找 M 接受的下一个字符串。这个下一个字符串是 w_{i+1} 。同样 w_{i+1} 在状态 q 里被显示, 然后被记忆, 并且被比较。显然 L 的任何成员最终都被显示出来。

另一个方向有点更容易。若 M 枚举 L , 则我们修改 M 去半可判定 L 如下: 重新设计 M 使得它在开始枚举过程之前, 保存提供给它的任何输入字符串。另外, M 每次进入特殊状态 q 时, 修改的机器就比较当前带内容与保存的字符串。若找到匹配, 则接受输入字符串; 否则继续枚举过程。于是新的机器恰恰半可判定 M 所枚举的语言。 ■

关于递归语言情况又如何呢? 事实上它们可用更加有序的方式来枚举。

定义 5.7.2 设 M 是枚举语言 L 的 Turing 机。若下列关系为真, 其中 q 是特殊的“显示”状态: 每当 $(q, \triangleright \sqcup w) \vdash \hat{M} (q, \triangleright \sqcup w')$ 时, w' 就在字典序下排在 w 之后, 则我们说 M 以字典序枚举 L 。语言是以字典序 Turing 可枚举的当且仅当存在以字典序枚举它的 Turing 机。

定理 5.7.3 语言是递归的当且仅当它是以字典序 Turing 可枚举的。

证明: 假设 M 是判定 L 的 Turing 机。于是下列 Turing 机 M' (M' 是我们在证明定理 5.7.2 开始时第一次不成功的尝试) 以字典序枚举 L : M' 在字典序下一个接一个地生成 L 字母表上的所有字符串, 并且在每个字符串上运行 M 。每当 M 接受时, M' 就显示字符串并且继续处理下一个字符串。若 M 拒绝, 则 M' 不经过显示状态, 直接继续处理下一个字符串。

对另一个方向, 假设用 Turing 机 M 以字典序枚举 L , 有两种情形: 如果 L 是有穷的, 那么无须证明任何东西, 因为在这种情形里 L 肯定是递归的(也是上下文无关的、正则的等等)。所以假设 L 是无穷的。下列机器 M' 判定 L : 在输入 w 上, 启动枚举机器 M 。等待直

到要么显示 w 要么显示字典序下排在 w 之后的任何字符串为止。在第一种情形接受 w ；在第二种情形拒绝它。因为存在有穷多个在字典序下排在 w 之前的字符串(少于 $(|\Sigma| + 1)^{|w|}$)，并且 L 是无穷的，所以我们知道两种情形之一将发生。 ■

每一台 Turing 机 M 都半判定表示成 $L(M)$ 的唯一语言，即它上面停机的所有字符串的集合。但是从对 M 的平凡改动(例如状态重新编号的 M ，或者就在停机之前毫无意义地“跳”进新状态，而在别处则与 M 相同的机器)，到非常微妙的变种(读者应当设法提供几个)，在这个范围里的许多其他 Turing 机也半判定 L 。换句话说，这个从所有 Turing 机的集合到递归可枚举语言类的函数远远不是同构，因为它把无穷多个非常不同的机器映射到同样的语言上。事实上，我们知道，这个映射是否把两台给定机器映射成同样的语言是不可判定的。下列结果提示说，这个映射是如此地复杂，以致于在下面解释清楚的某种意义下，所有想到的关于它的问题都是不可判定的。

定理 5.7.4 (Rice 定理) 假设 \mathcal{C} 是所有递归可枚举语言组成的类的非空真子集，那么下列问题是不可判定的：给定 Turing 机 M ，是否 $L(M) \in \mathcal{C}$ ？

证明：我们假设 $\emptyset \notin \mathcal{C}$ (否则所有不属于 \mathcal{C} 的递归可枚举语言也是递归可枚举语言的非空真子集，对它重复其余的论证)。下一步因为 \mathcal{C} 非空，所以我们假设存在机器 M_L 半判定的语言 $L \in \mathcal{C}$ 。

我们把停机问题归约到判定给定的 Turing 机半判定的语言是否属于 \mathcal{C} 的问题上。于是假设给定 Turing 机 M 和输入 w ，我们希望判定 M 是否在 w 上停机。为了完成这个任务我们构造 Turing 机 $T_{M,w}$ ，使得它半判定的语言要么是上述的固定语言 L ，要么是语言 \emptyset 。在输入 x 上 $T_{M,w}$ 在输入“ M ”“ w ”上模拟通用 Turing 机 U ，如果 U 停机，那么 $T_{M,w}$ 不是停机而是接着在输入 x 上模拟 M_L ，根据 M_L 在 x 上的行为， $T_{M,w}$ 要么停机并接受、要么死循环。自然若 $U(\text{“}M\text{”“}w\text{”}) = \nearrow$ 则同样 $T_{M,w} = \nearrow$ 。总之 $T_{M,w}$ 是这样的机器：

$$T_{M,w}(x) : \text{if } U(\text{“}M\text{”“}w\text{”}) \neq \nearrow \text{ then } M_L(x) \text{ else } \nearrow$$

断言： $T_{M,w}$ 半判定的语言属于 \mathcal{C} 当且仅当 M 在输入 w 上停机。

注意这个断言说从 M 和 w 出发对 $T_{M,w}$ 的构造，是从停机问题到辨别给定 Turing 机所半判定的语言是否属于 \mathcal{C} 的问题的归约。这个断言完成了对定理的证明。

对断言的证明：假设 M 在输入 w 上停机。于是 $T_{M,w}$ 在输入 x 上判定这件事情，然后总是接受 x 当且仅当 $x \in L$ 。因此在这种情形里 $T_{M,w}$ 半判定的语言是 L ， L 属于 \mathcal{C} 。

然后假设 $M(w) = \nearrow$ 。在这种情形里 M 永不停机，因此 $T_{M,w}$ 半判定语言 \emptyset ，但已知 \emptyset 不属于 \mathcal{C} 。 ■

从 Rice 定理得出许多问题的不可判定性：给定 Turing 机 M ，它半判定的语言 $L(M)$ 是否正则？上下文无关？有穷？ Σ^* ？递归？等等。

习 题

- 5.7.1 证明 L 是递归可枚举的当且仅当对某个非确定型 Turing 机 $M, L = \{w : (s, \triangleright \sqcup) \vdash_M^* (q, \triangleright \sqcup w)\}$ ，其中 s 是 M 的初始状态。
- 5.7.2 证明如果语言是递归可枚举的，那么存在枚举它的 Turing 机，在枚举时从不重复语言的元素。

- 5.7.3 (a) 设 Σ 是不含“,”的字母表,并假设 $L \subseteq \Sigma^*$, Σ^* 是递归可枚举的。证明语言 $L' = \{x \in \Sigma^* : \text{对某个 } y \in \Sigma^*, x, y \in L\}$ 是递归可枚举的。
 (b) 若 L 是递归的,则在(a)里的 L' 是否一定是递归的?
- 5.7.4 文法称为是上下文有关的当且仅当每条规则都形如 $u \rightarrow v$, 其中 $|u| \geq |v|$ 。上下文有关语言是上下文有关文法生成的语言。
 (a) 证明每一个上下文有关语言都是递归的。(提示:推导可有多长?)
 (b) 证明语言是上下文有关的当且仅当它由每条规则都形如 $uAv \rightarrow uvw$ 的文法生成,其中 A 是非终结符,并且 $w \neq \epsilon$ 。
 (c) 原地接受器(或线性界限自动机)是非确定型 Turing 机使得带头永不访问除输入的左右两端相邻的两个空格之外的空格。证明语言是上下文有关的当且仅当它是用原地接受器半判定的(提示:两个方向都是定理 4.6.1 的证明的具体化。)
- 5.7.5 在前一题里介绍并在(c)部分里用非确定型原地 Turing 机另外刻画过的上下文有关语言类完成了 Chomsky 分层,它是用来描述语言生成器的表达能力和自动机能力的有影响的框架模型,由语言学家 Noam Chomsky 在 20 世纪 60 年代提出。Chomsky 分层包括这样五类语言(按照本书介绍它们的顺序):正则语言、上下文无关语言,递归语言,递归可枚举语言,上下文有关语言。按照一般性递增的顺序来排列这些语言类(使得在排列里的每个类真包含所有前面的类),并且在每类旁边写出对应的自动机和、或文法。
- 5.7.6 假设 $f: \Sigma_0^* \mapsto \Sigma_1^*$ 是递归满函数。证明存在递归函数 $g: \Sigma_1^* \mapsto \Sigma_0^*$ 使得,对每个 $w \in \Sigma_0^*$, $f(g(w)) = w$ 。
- 5.7.7 证明下列问题都是不可判定的:给定的 Turing 机所半可判定的语言是否
 (a) 有穷。
 (b) 正则。
 (c) 上下文无关。
 (d) 递归。
 (e) 等于语言 $\{ww^R; w \in \{a,b\}^*\}$ 。
- 5.7.8 本章展示的非递归语言 L 具有性质:要么 L 是递归可枚举的,要么 \bar{L} 是递归可枚举的。
 (a) 用计数论证法证明存在语言 L 使得 L 和 \bar{L} 都不是递归可枚举的。
 (b) 给出这样的语言的例子。
- 5.7.9 本题是习题 3.7.10 的继续:扩充包含(a)正则语言,(b)上下文无关语言,(c)确定型上下文无关语言及(d)上下文无关语言的补的文氏图,使得它包括下列三个新类:
 (e) 递归语言,
 (f) 递归可枚举语言,
 (g) 递归可枚举语言的补。
 在文氏图的每个区域里给出语言的例子。

- 5.7.10 描述有下列性质的 Turing 机 M : 在任何输入上, 它输出“ M ”, 即它自身的描述。
(提示: 首先用程序设计语言写一个自己打印自己的程序。然后, 关于更复杂的论证, 见下一个习题。)
- 5.7.11 (a) 论证存在 Turing 机 G , 当向它提供 Turing 机 M 的描述时, 它计算出对另一台 Turing 机 M^2 的描述, M^2 在任何输入 x 上, 首先在输入“ M ”上模拟 M , 然后若 M 停机并且在 M 的带上有 Turing 机 N 的有效描述, 则 M^2 在输入 x 上模拟 N 。
- (b) 论证存在 Turing 机 C , 当向它提供两台 Turing 机 M_1 和 M_2 的描述时, 它计算出 M_1 和 M_2 的复合 Turing 机, 即在任何输入 x 上, 机器首先在 x 上模拟 M_2 , 然后在结果上模拟 M_1 。
- (c) 假设 M 是一台 Turing 机, 当它的输入是 Turing 机的有效描述时, 它输出 Turing 机的另一个有效描述。证明任何这样的 M 都有不动点, 即具有下列性质的 Turing 机 F : F 和用 $M(F)$ 所表示的机器在所有输入上的行为都恰恰相同。(假设把 M 和 G (见(a)) 的复合 (见(b)) 作为给 G 的输入。验证用输出作为它的描述的机器就是所需要的不动点 F)。
- (d) 设 M 是任何 Turing 机。论证存在具有下列性质的另一台 Turing 机 M' : 对任何输入 x , 如果 M 在输入 x 上停机并输出 y , 那么 M' 也在输入 x 上停机并输出 yM' , 即 M' 把它自身的描述——它的“签名”——添加在 M 的输出后面。

参 考 文 献

停机问题的不可判定性是 A. M. Turing 在他的 1936 年的文章里证明的 (见第 4 章的参考文献)。与上下文无关文法有关的问题的不可判定性是在第 3 章结尾引用的 Bar-Hillel, Perles 和 Shamir 的文章里证明的。Post 对应问题 (见习题 5.5.2) 来自:

- E. L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Math. Society*, 52, pp. 264—268, 1946.

Chomsky 分层 (见习题 5.7.5) 归功于 Noam Chomsky:

- N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3), pp. 113—124, 1956.

铺砖问题 (5.6 节) 来自:

- H. Wang. Proving theorems by pattern recognition. *Bell System Technical Journal*, 40, pp. 1—141, 1961.

关于许多其他不可解问题, 见上一章的参考文献, 特别是 Martin Davis 的书。

第 6 章 计算复杂性

6.1 P 类

在前一章我们看到了存在不能用算法解决的良好定义的判定问题,还给出了这样的问题的某些具体例子。因此所有计算问题可分成两类:能用算法解决的和不能用算法解决的。过去几十年计算机技术的长足进步使你有理由希望,所有前一类问题现在都用令人满意的方式解决了。不幸的是计算实践揭示许多问题尽管在原理上可解,但在任何实践意义上还是不能用计算机解决,原因是过度的时间需求。

假设你的任务是安排旅行商访问 10 个地方部门。给你标有 10 个城市和城市间距离(以英里为单位)的地图,要求你求出经过的总距离最短的巡回路线。你一定希望用计算机解决这个任务。而且从理论上讲这个问题肯定是可解的。若存在 n 个要访问的城市,则可能的巡回路线的数目是有穷的,准确说是 $(n-1)!$,即 $1 \cdot 2 \cdot 3 \cdots (n-1)$ 。因此可轻松地设计出检查所有巡回路线以找出最短路线的算法。甚至可设计出计算最短路线的 Turing 机。

尽管这样,你对这个算法的感觉还是不轻松。需要检查的路线太多了。对于有 10 个城市的中等问题就不得不检查 $9! = 362\,880$ 条巡回路线。假如有耐心的话,这个工作还可使用计算机完成。但是假如有 40 个城市要访问,你又怎么办呢?现在巡回路线的数目是巨大的: $39!$,它大于 10^{45} 。即使我们每秒检查 10^{15} 条路线,这个速度远远超过最强的超级计算机(无论是存在的还是规划的),完成这个计算所需要的时间是宇宙寿命的几十亿倍!

显然问题在理论上可解并不立即蕴涵它在实践上是实际可解的。本章的目标是建立形式化的数学理论——Church-Turing 论题的定量细化——它刻画“实际可行算法”这个直觉概念。问题是哪些算法,以及哪些 Turing 机,应当被认为是实际可行的?

像旅行商问题所揭示的那样,这里的限制参数是算法在给定输入上需要的时间或步数。旅行商问题的 $(n-1)!$ 算法是不实际的,这是因为过度的指数增长的时间需求(容易看出函数 $(n-1)!$ 甚至增长得比 2^n 还快)。对比之下,像我们在本书各部分里介绍的算法那样,多项式增长率的算法显然是更有吸引力的。

为了刻画“实际可行算法”的概念,似乎必须限制计算装置只运行到输入长度的多项式界限的步数。但是我们究竟选择何种计算装置作为这个重要理论的基础呢? Turing 机,它的多带变种,多维 Turing 机,或者也许随机存取模型? 4.3 节里的模拟结果告诉我们,因为我们对多项式增长感兴趣,所以这个选择是无关紧要的——只要排除有明显指数能力的非确定型模型就行,回忆 4.5 节并且看 6.4 节。若这些确定型种类里任意一种 Turing 机在多项式步数之后停机,则存在其他任意一种等价的 Turing 机也在多项式步数之后停机,仅仅这两个多项式不同。所以我们还是决定用最简单的模型,即标准 Turing 机。这种

“模型无关性”是选择多项式增长率作为“有效性”概念所带来的重要额外好处。

因此我们得到下列定义。

定义 6.1.1 对 Turing 机 $M = (K, \Sigma, \delta, s, H)$, 若存在多项式 $p(n)$ 使下列关系为真: 对任意输入 x , 不存在格局 C 满足 $(s, \triangleright \sqcup x) \vdash_M^{p(|x|)+1} C$, 则 M 称为是多项式界限的。换句话说, 机器到最多 $p(n)$ 步之后总是停机, 其中 n 是输入长度。

对语言, 若存在判定它的多项式界限的 Turing 机, 则语言称为多项式可判定的。所有多项式可判定语言的类表示成 P 。

Church-Turing 论题的定量细化现在陈述如下: 多项式界限的 Turing 机和 P 类分别适当刻画了实际可行算法和实际可解问题的直觉概念。

换句话说, 我们建议把 P 作为递归语言类的定量类比。事实上, P 确实具有递归语言类的某些性质。

定理 6.1.1 P 在补运算下封闭。

证明: 如果语言 L 被多项式界限的 Turing 机 M 判定, 那么它的补被交换 M 的 y 和 n 得到的 Turing 机判定。显然, 多项式界限不受影响。■

另外我们用对角化展示不属于 P 的某些简单的递归语言。考虑“停机”语言 H 的下列定量类比(回忆 5.3 节):

$$E = \{ \langle M \rangle \langle w \rangle; M \text{ 在最多 } 2^{|w|} \text{ 步之后接受 } \langle w \rangle \}$$

定理 6.1.2 $E \notin P$ 。

证明: 这个证明几乎照搬了对停机问题不可判定性的证明(定理 5.3.1)。假设 $E \in P$ 。于是下列语言也属于 P :

$$E_1 = \{ \langle M \rangle; M \text{ 在最多 } 2^{|\langle M \rangle|} \text{ 步之后接受 } \langle M \rangle \}$$

根据定理 6.1.1, 它的补 \bar{E}_1 也属于 P 。因此存在 Turing 机 M^* , M^* 接受所有在 2^n 时间之内不接受它们自身的描述的 Turing 机的描述(其中 n 是描述的长度), 而拒绝所有其他输入。另外 M^* 在 $p(n)$ 步之内总是停机, 其中 $p(n)$ 是多项式。可以假设 M^* 是单带机, 因为否则把它转换成单带机并不改变多项式性质。

回忆一下, 因为 $p(n)$ 是多项式, 所以存在正整数 n_0 使得对所有 $n \geq n_0$, $p(n) \leq 2^n$ 。另外假设 M^* 的编码长度至少是 n_0 , 即 $|\langle M^* \rangle| \geq n_0$ 。否则就向 M^* 添加 n_0 个在计算里永不可达的无用状态。

现在问题是这样: 当给 M^* 它自身的描述 $\langle M^* \rangle$ 时, 它怎么办? 接受还是拒绝? (它被构造造成二者择其一。)两种回答都导致矛盾: 如果 M^* 接受 $\langle M^* \rangle$, 那么因为 M^* 判定 \bar{E}_1 , 这意味着 M^* 不在 $2^{|\langle M^* \rangle|}$ 步之内接受 $\langle M^* \rangle$; 但是 M^* 被构造造成在长度为 n 的输入上在 $p(n)$ 步之内总是停机, 并且 $2^{|\langle M^* \rangle|} > p(|\langle M^* \rangle|)$, 所以它必须拒绝。同理, 假设 M^* 拒绝它自身的描述, 我们推出它接受该描述。因为导致这个矛盾的唯一假设是 $E \in P$, 所以我们必然得出 $E \notin P$ 。■

习 题

6.1.1 证明 P 也在并、交和连接运算下封闭。

6.1.2 证明 P 在 Kleene 星号运算下封闭。(这比前一个问题的封闭性证明更难。对某个

$L \in P$, 为了辨别是否 $x \in L^*$, 你不得不考虑 x 的所有子串, 从长度为 1 的子串开始并向越来越长的子串前进。这非常像有关上下文无关识别的动态规划算法, 回忆 3.6 节。)

6.2 若干问题

P 类对“可令人满意地解决的问题”这个直觉概念的刻画有多好? 有多少人接受多项式算法恰恰就是经验可行算法这个论题? 公正地说尽管这个论题是该领域里唯一严肃的建议, 但它还是在各种背景下受到挑战。^① 例如可以论证, 时间需求为 n^{100} 或者甚至 $10^{100}n^2$ 的算法, 虽然是多项式时间算法但根本不是“实际可行的”。但是时间需求为 $n^{\log \log n}$ 的算法, 尽管时间需求的增长没有任何多项式界限, 还是可被认为在实践中是完全可行的。为我们的论题提供辩护的经验论据是, 这样极端的时间界限虽然在理论上是可能的, 但是在实践中很少出现。从计算实践得出的多项式算法通常有小的幂次数和令人满意的常数系数, 但是非多项式算法通常是使人绝望的指数算法, 因此在实践中用途非常有限。

对我们论题的另一种批评意见是, 它对算法的分类仅仅根据最坏情形性能(在所有长度为 n 的输入上的最大运行时间)。按理说平均情形方法——例如坚持要求当在所有长度为 n 的输入上平均时, Turing 机的时间需求有 $p(n)$ 界限——是对算法实际用途的更好预测。尽管平均情形分析似乎是非常合理的替代方法, 但事实上它自身容易受到更致命的挑战。例如在平均情形分析里应当采取什么样的输入分布? 对此似乎没有令人满意的回答。

不过尽管有这些保留意见, 多项式界限计算还是有吸引力和有成效的概念, 它导致优美和有用的理论。过分关注它边界上的灰色区域, 常常使你忘记它是多么有用的分类工具, 忘记它主要包括实际可行的算法和主要排除不实际的算法。为了熟悉多项式时间计算、它的真正范围和它的限度等, 最好的方法莫过于介绍一族有趣的已知属于 P 的计算问题。对比这样的问题与似乎不属于 P 的难解问题的某些例子, 这也是有教育意义的。困难问题与容易问题常常看起来非常相似。

我们已经见过 P 的某些非常有趣的典型问题和子类。例如, 所有正则语言和所有上下文无关语言都属于 P(回忆定理 2.6.2 和 3.6.1 的(c)部分)。我们还知道可在多项式时间里计算关系的自反传递闭包(1.6 节)。下一步我们检查这个问题的有趣变种。

可达性问题定义如下。

可达性: 给定有向图 $G \subseteq V \times V$, 其中 $V = \{v_1, \dots, v_n\}$ 是有穷集, 以及两个顶点 $v_i, v_j \in V$, 是否存在从 v_i 到 v_j 的路径?

(在计算问题的背景下提到的所有图当然都是有穷的。)可达性是否属于 P? 严格地说, 因为 P 只包含语言, 所以它与可达性无关。可达性是我们称为问题的那种东西。问题是输入的集合(这个集合在典型情况下是无穷的), 以及对每个输入问的是或否问题(输入

^① 甚至 Church-Turing 论题也始终受到大量挑战, 挑战来自两方面, 一些数学家认为 Turing 可判定性是太严格的概念, 另一些数学家则认为它是太自由的概念。事实上可认为作为本章和下一章主题的复杂性理论是最新和最严格的后一种类型的挑战。

可能有也可能无的性质)。在可达性这个例子里输入集是所有三元组 (G, v_i, v_j) 的集合, 其中 G 是有穷图, v_i, v_j 是 G 的两个顶点。问的问题是在 G 里是否存在从 v_i 到 v_j 的路径。

这不是我们第一次见到问题。停机问题肯定是问题: 它的输入是 Turing 机和字符串, 问的问题是当给定的 Turing 机在这个输入字符串上启动时它是否停机。

停机问题: 给定 Turing 机 M 和输入字符串 w , M 是否接受 w ?

在第 5 章里我们研究过停机问题的“语言替身”, 即语言

$$H = \{ \langle M \rangle \langle w \rangle; \text{Turing 机 } M \text{ 在字符串 } w \text{ 上停机} \}$$

同理, 我们研究可达性的语言

$$R = \{ \langle \kappa(G) \rangle \langle b(i) \rangle \langle b(j) \rangle; \text{在 } G \text{ 里有从 } v_i \text{ 到 } v_j \text{ 的路径} \}$$

其中 $b(i)$ 表示整数 i 的二进制编码, κ 是把图编码成字符串的某种合理方法。可以想到图的多种自然编码。让我们把图 $G \subseteq V \times V$ 编码成相邻矩阵, 然后线性化成字符串 (回忆例 4.4.3 和图 4-21)。从下面讨论中得出的要点是编码的准确细节几乎无关紧要。

语言是对问题的编码。当然, 任何语言 $L \subseteq \Sigma^*$ 也被认为是问题:

L 的判定问题: 给定字符串 $x \in \Sigma^*$, $x \in L$ 吗?

对问题和关联的语言做交替思考是有益的。语言更适合于联系 Turing 机, 但是问题更清楚地陈述了我们必须为之发展算法的有关的实际计算任务。在下面几页里我们深入介绍和讨论许多有趣的问题, 我们把问题和对应的语言当作同一个事物的两个不同方面。例如, 下一步我们指出可达性属于 P。这样说的意思是上面定义的对语言 R 属于 P。

确实, 解决可达性的方法是首先计算 G 的自反传递闭包, 这可用例 4.4.3 的随机存取 Turing 机在 $O(n^3)$ 时间里完成。然后检查 G 的自反传递闭包里对应于 v_i 和 v_j 的项, 它告诉我们在 G 里是否存在从 v_i 到 v_j 的路径。因为我们知道随机存取 Turing 机可用普通 Turing 机在多项式时间里模拟, 因此得出可达性属于 P。

注意因为我们只想确定时间界限是否多项式, 所以我们允许时间界限表示成顶点数 $n = |V|$ 的函数, 而不表示成输入长度的函数, 输入长度是 $m = |\kappa(G) \langle b(i) \rangle \langle b(j) \rangle|$ 。因为容易看出 $m = O(n^3)$, 所以这是另一个无关紧要的不精确之处, 它不影响我们认为重要的东西。

6.2.1 Euler 图和 Hamilton 图

18 世纪伟大的数学家 Leonard Euler 研究并解决的历史上关于图的第一个问题是:

Euler 回路: 给定图 G , 在 G 里是否存在经过每条边恰好一次的回路?

寻找的回路可经过每个顶点多次 (或者若图中有孤立点, 即没有边进出的顶点, 则甚至根本不经过这些顶点)。包含这种回路的图称为 **Euler 图** 或 **一笔画**。例如图 6-1(a) 所示的图是 Euler 图, 因为存在回路 $(v_1, v_2, v_3, v_4, v_3, v_2, v_4, v_1, v_3, v_1)$, 而图 6-1(b) 不是 Euler 图。

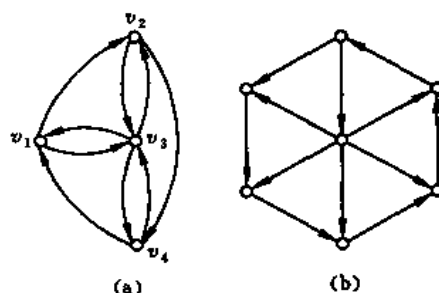


图 6-1

不难看出 Euler 回路属于 P, 我们这样说的意思当然是指对应的语言

$$L = \{ \langle G \rangle, G \text{ 是 Euler 图} \}$$

属于 P。这个结论来自对 Euler 图的下列简洁刻画, 它归功于 Euler。图里的顶点称为孤立的, 当且仅当它没有关联的边。

图 G 是 Euler 图当且仅当它有下列两条性质:

- (a) 对任意一对都不是孤立的顶点 $u, v \in V$, 存在从 u 到 v 的通路; 以及
- (b) 所有顶点都有同样数目的入边和出边。

无须多想就可知道这两个条件对成为 Euler 图都是必要的。我们把充分性证明留作练习(习题 6.2.1)。

因此非常容易验证图是否 Euler 图。首先, 我们在多项式时间里确定是否除孤立点外所有顶点都是连通的。方法是先计算图的自反传递闭包, 再验证是否除孤立点外所有顶点都连通(归根结底, 对所有可能的关于图的连通性的问题, 回答都包含在图的自反传递闭包里)。我们知道可在多项式步数里计算自反传递闭包。其次, 我们验证是否所有顶点都有同样数目的入边和出边, 这显然也在多项式时间里完成。

顺便说一句, 这是下面几页里没完没了地重复的模式实例; 利用前面证明过的问题(在这个情形里是可达性)属于 P 这个事实, 我们证明另一个问题(Euler 回路)属于 P, 即把要证明的问题归约到已解决的问题上。

也许, 本章和下一章的要点是存在许多自然的、可判定的、陈述简单的问题, 我们不知道或者不相信它们属于 P。通常这样的问题非常像另一个已知属于 P 的问题! 例如考虑另一位著名数学家——这次是 19 世纪的——William Rowan Hamilton 以及在他之后的许多数学家研究过的下列问题:

Hamilton 圈: 给定图 G , 是否存在经过 G 的每个顶点恰好一次的圈?

这样的圈称为 Hamilton 圈, 有 Hamilton 圈的图称为 Hamilton 图。注意区别: 现在是顶点而不是边必须被经过恰好一次, 不必经过所有边。例如图 6-1(b) 里的图是 Hamilton 图但不是 Euler 图, 而图 6-1(a) 里的图既是 Hamilton 图又是 Euler 图。

尽管在 Euler 回路和 Hamilton 圈这两个问题之间有表面的相似性, 但在它们之间似乎还是有极大的差别。经过许多天才数学家一个半世纪的仔细研究之后, 还是没有人发现 Hamilton 圈的多项式时间算法。自然, 下列算法确实解决这个问题:

检查所有可能的顶点排列;

对每种排列验证它是否是 Hamilton 圈。

不幸的是, 做了所有简单的改进和加速之后, 它仍不是多项式的。

6.2.2 优化问题

在本章开头非形式化地介绍过的旅行商问题是另一个陈述简单的问题, 尽管经过数十年的深入研究, 还是没有已知的多项式时间算法。给定城市的集合 $\{c_1, c_2, \dots, c_n\}$, 以及非负整数构成的 $n \times n$ 矩阵 d_{ij} , 其中 d_{ij} 表示城市 c_i 与城市 c_j 之间的距离。假定对所有 $i, j, d_{ii} = 0$ 并且 $d_{ij} = d_{ji}$ 。要求我们找出这些城市的最短巡回路线, 即从集合 $\{1, 2, \dots, n\}$ 到它自身的双射 π (其中 $\pi(i)$ 直观上是巡回路线的第 i 个访问城市), 使得

$$c(\pi) = d_{\pi(1)\pi(2)} + d_{\pi(2)\pi(3)} + \cdots + d_{\pi(n-1)\pi(n)} + d_{\pi(n)\pi(1)}$$

尽可能小。

我们在问题和语言的框架里研究旅行商问题遇到严重障碍。与在本章里见过的所有其他问题不同,旅行商问题不是要求“是”或“否”的回答,因此不是可用语言来研究的那种问题。它是优化问题,因为它要求从许多可行解里找出最好的(根据某个成本函数)。

存在把优化问题转变成语言使得我们可以研究它的复杂性的有用的一般性方法:给每个输入附加上成本函数的界限。在目前情形里考虑下列问题:

旅行商问题: 给定整数 $n \geq 2$, $n \times n$ 距离矩阵 d_{ij} , 以及整数 $B \geq 0$ (直观上, B 是旅行商的预算), 是否存在 $\{1, 2, \dots, n\}$ 的排列 π 使得 $c(\pi) \leq B$ 。

如果我们能在多项式时间里解决原来的优化问题,即我们有计算最便宜路线的算法,那么显然我们可设法在多项式时间里解决这个“预算”型问题:仅仅计算出最便宜路线的成本并且与 B 比较。因此关于刚才定义的旅行商问题的复杂性的任何否定性结果,都会对解决原来的优化型问题的前景投下阴影。

我们用这种方法把许多有趣的优化问题纳入到语言框架中。在最大化问题的情形里,我们不提供预算 B 而是用目标 K 代替。例如下列问题是通过这种方法变换的重要的极大化问题:

独立集: 给定无向图 G 和整数 $K \geq 2$, 是否存在 V 的子集 C 满足 $|C| \geq K$ 使得对所有 $v_i, v_j \in C$, 在 v_i 和 v_j 之间没有边?

独立集又是另一个自然的和陈述简单的问题,尽管研究者对它有长期的、浓厚的兴趣,还是没有发现多项式时间算法。

让我们介绍另外两个关于无向图的优化问题。下一个问题在某种意义上与独立集恰好相反:

团: 给定无向图 G 和整数 $K \geq 2$, 是否存在 V 的子集 C 满足 $|C| \geq K$ 使得对所有 $v_i, v_j \in C$, 在 v_i 和 v_j 之间有边?

对于下一个问题,若一些顶点的集合至少包含某条边的一个端点,则我们说这个集合覆盖这条边。

顶点覆盖: 给定无向图 G 和整数 $B \geq 2$, 是否存在 V 的子集 C 满足 $|C| \leq B$ 使得 C 覆盖 G 的所有边?

我们设想无向图的顶点是博物馆的房间,每条边是连接两个房间的又长又直的走廊。于是用顶点覆盖来安排尽量少的房间警卫,使得所有走廊都有警卫监视。

为了解释这些有趣的问题,在图 6-2 里,图的最大独立集有三个顶点,最大团有四个顶点,最小顶点覆盖有六个顶点。你能否找出它们? 你能否让自己相信它们是最优的?

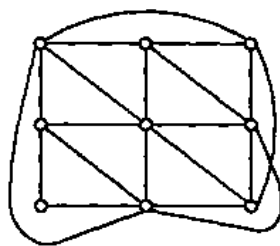


图 6-2

6.2.3 整数划分

假设给定多个正整数,比如 38, 17, 52, 61, 21, 88, 25, 问能否把它们划分成两个不相交的集合,使得这两个集合包含所有的数,并且各自之和都等于同样的数。在上面的例子回答为“是”,因为 $38 + 52 + 61 = 17 + 21 + 88 + 25 = 151$ 。一般的问题是这样的:

划分:给定用二进制表示的 n 个非负整数 a_1, \dots, a_n 的集合, 是否存在子集 $P \subseteq \{1, \dots, n\}$ 使得 $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$?

这个问题由如下简单算法来解决, 首先设 H 是 $S = \{a_1, \dots, a_n\}$ 里所有整数之和除以 2 (如果这个数不是整数, 那么 S 里的数加起来是奇数, 因此不能被划分成两个相等的和, 所以我们立即回答“否”)。对每个 $i, 0 \leq i \leq n$, 定义数的集合:

$$B(i) = \{b \leq H; b \text{ 是 } \{a_1, \dots, a_i\} \text{ 的某个子集的和}\}$$

若我们知道 $B(n)$, 则仅通过验证是否 $H \in B(n)$ 就轻易解决了划分。若是, 则有加起来等于 H 的部分和, 因此回答“是”; 否则, 回答“否”。

另外注意到 $B(n)$ 可用下述算法来计算:

$$B(0) := \{0\}.$$

for $i=1, 2, \dots, n$ do

$$B(i) := B(i-1),$$

for $j=a_i, a_i+1, a_i+2, \dots, H$ do

if $j-a_i \in B(i-1)$ then add j to $B(i)$

例如在上面给出的划分的实例里, $a_1=38, a_2=17, a_3=52, a_4=61, a_5=21, a_6=88, a_7=25$, $B(i)$ 集合如下:

$$B(0) = \{0\}$$

$$B(1) = \{0, 38\}$$

$$B(2) = \{0, 17, 38, 55\}$$

$$B(3) = \{0, 17, 38, 52, 55, 69, 90, 107\}$$

$$B(4) = \{0, 17, 38, 52, 55, 61, 69, 78, 90, 107, 113, 116, 130, 151\}$$

$$B(5) = \{0, 17, 21, 38, 52, 55, 59, 61, 69, 73, 76, 78, 82, 90, 99, 107, 111, 113, 116, 128, 130, 134, 137, 151\}$$

$$B(6) = \{0, 17, 21, 38, 52, 55, 59, 61, 69, 73, 76, 78, 82, 88, 90, 99, 105, 107, 109, 111, 113, 116, 126, 128, 130, 134, 137, 140, 143, 147, 149, 151\}$$

$$B(7) = \{0, 17, 21, 25, 38, 42, 46, 52, 55, 59, 61, 63, 69, 73, 76, 77, 78, 80, 82, 84, 86, 88, 90, 94, 98, 99, 101, 103, 105, 107, 109, 111, 113, 115, 116, 124, 126, 128, 130, 133, 134, 136, 137, 138, 140, 141, 143, 147, 149, 151\}$$

划分的这个实例是“是”实例, 因为半和 $H=151$ 包含在 $B(7)$ 里。

通过对 i 归纳容易证明对 $i=0, \dots, n$, 这个算法不仅正确计算 $B(i)$, 而且在 $\mathcal{O}(nH)$ 时间里完成 (见习题 6.2.5)。那么我们是否证明了划分属于 P?

界限 $\mathcal{O}(nH)$ 尽管有完美的多项式外表, 却不是输入长度的多项式。原因是整数 a_1, \dots, a_n 是用二进制给出的, 因此它们的和在一般情况下与输入长度相比是指数大小的。

例如, 如果在划分的实例里所有 n 个整数都大约是 2^n , 那么 H 近似地是 $\frac{n}{2} 2^n$, 而输入长度仅仅是 $\mathcal{O}(n^2)$ 。事实上, 划分是著名的困难问题之一, 这些困难问题包括旅行商问题、Hamilton 圈和独立集等。对于它们, 没有真正的多项式算法是已知的或者是会有希望会很快出现的 (它们是下一章的主题)。

不过上述算法还是证明了下列问题确实属于 P:

一进制划分: 给定用一进制表示的 n 个非负整数 a_1, \dots, a_n 的集合, 是否存在子集 $P \subseteq \{1, \dots, n\}$ 使得 $\sum_{i \in P} a_i = \sum_{i \notin P} a_i$?

这是因为一进制问题的输入长度大约是 H , 所以 $\mathcal{O}(nH)$ 算法突然变成“有效的”。

划分和一进制划分这对问题的复杂性是截然不同的, 它们说明关于输入表示的下列重要事实: 作为问题输入的数学对象, 比如图、自动机、Turing 机等等, 对它们的精确表示几乎不影响对应语言是否属于 P, 因为对同一个对象的所有合理表示都是多项式相关的。唯一的例外是整数应当用二进制编码,^①而不用一进制。

6.2.4 有穷自动机的等价性

在第 2 章我们看到关于有穷自动机的下列重要问题属于 P(定理 2.6.1(e)):

确定型有穷自动机的等价性: 给定两个确定型有穷自动机 M_1 和 M_2 , $L(M_1) = L(M_2)$ 吗?

相比之下我们注意到, 我们只知道如何在指数时间里验证非确定型有穷自动机的等价性。即我们不知道下列两个问题是否有一个属于 P:

非确定型有穷自动机的等价性: 给定两个非确定型有穷自动机 M_1 和 M_2 , $L(M_1) = L(M_2)$ 吗?

正则表达式的等价性: 给定两个正则表达式 R_1 和 R_2 , $L(R_1) = L(R_2)$ 吗?

通过把两个给定的非确定型有穷自动机(或正则表达式)变换成两个确定型有穷自动机(定理 2.6.1), 然后验证所得出的两个确定型有穷自动机的等价性, 就可以解决这两个问题中的任何一个。当然问题是从正则表达式或非确定型有穷自动机到确定型有穷自动机的变换可能指数地增加自动机的状态数(回忆习题 2.5.4)。所以这样的方法不能证明非确定型有穷自动机的等价性或正则表达式的等价性属于 P, 事实上复杂得多的方法也不能证明它们属于 P。

习 题

6.2.1 证明图是 Euler 图当且仅当它是连通的并且每个顶点的入度等于出度。(提示: 一个方向是容易的。对另一个方向, 从顶点出发经过边到达另一个顶点。继续经过新的边直到找不到新的边为止, 这时你又回到了出发顶点(为什么?)。证明在剩下来还未经过的图的碎片里你如何重新出发并经过它们。)

6.2.2 证明正文里给出的算法在 $\mathcal{O}(nH)$ 步里解决划分。

6.2.3 解决有五座城市 A, B, C, D 和 E 的旅行商问题, 距离矩阵如下:

	B	C	D	E
A	8	4	5	9
B		1	7	3
C			6	2
D				5

^① 或者用十进制, 十六进制, 或者任何其他基数系统。同一个整数的所有这样的表示, 它们的长度彼此相差常数倍。

6.2.4 我们利用通过“预算” B 或“目标” K 定义的语言来研究优化问题。选择在本节介绍的优化问题,证明原来的问题存在多项式算法当且仅当它的“是或否”问题存在多项式算法。(一个方向是平凡的。另一个方向需要二分搜索,以及这些问题具有的称为自归约性的性质。)

6.2.5 图 E6-1 中的无向图是否存在 Hamilton 圈? 什么是这个图的最大团、最大独立集和最小顶点覆盖?

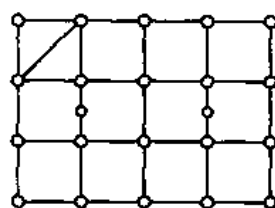


图 E6-1

6.3 布尔可满足性

在前一节我们看到属于 P 的许多有趣计算问题,以及另外一些被怀疑不属于 P 的问题。也许在这两类问题中最基本的问题都与布尔逻辑有关。

布尔逻辑是我们熟悉的数学记号,它用来表达复合命题,比如“要么现在不下雨,要么拐杖不在角落里”。在布尔逻辑里我们用布尔变元 x_1, x_2, \dots 表示个别命题,比如“现在下雨”。即,每个变元表示独立于其他命题的真值、在原则上为真或为假的命题。我们用布尔联结词来组合布尔变元并构成更复杂的布尔公式。为了我们在本书的目标,只需考虑下面定义的特殊布尔公式。

定义 6.3.1 设 $X = \{x_1, x_2, \dots, x_n\}$ 是布尔变元的有穷集,并设 $\bar{X} = \{\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n\}$, 其中新的符号 $\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n$ 表示 x_1, x_2, \dots, x_n 的非,或者否定。我们把 $X \cup \bar{X}$ 的元素都称为文字,变元都是正文字,而变元的非都是负文字。子句 C 是文字的非空集: $C \subseteq X \cup \bar{X}$ 。最后,合取范式形式的布尔公式(或者简称布尔公式,因为我们在本书里不处理其他类型的公式)是在 X 上定义的子句的集合。

例 6.3.1 设 $X = \{x_1, x_2, \dots, x_n\}$, 因此 $\bar{X} = \{\bar{x}_1, \bar{x}_2, \dots, \bar{x}_n\}$ 。 $C = \{x_1, \bar{x}_2, x_3\}$ 是子句。虽然子句是文字的集合,但是我们在书写子句时采用特殊的记号:我们用圆括号代替通常的集合花括号,用分隔符 \vee (读作或)代替逗号在子句里分隔不同的文字。像通常的集合那样,元素的顺序并不重要,但是不允许元素的重复。例如上述的子句 C 写成 $C = (x_1 \vee \bar{x}_2 \vee x_3)$ 。

下面是合取范式形式的布尔公式:

$$F = \{(x_1 \vee \bar{x}_2 \vee x_3), (\bar{x}_1), (x_2 \vee \bar{x}_2)\} \quad (1)$$

它包含 3 个子句,其中一个上面的 C 。 \diamond

定义 6.3.2 迄今为止我们仅仅定义了布尔公式的语法,或表面结构。下一步我们定义这样的公式的语义,或意义。设 F 是在 $X = \{x_1, x_2, \dots, x_n\}$ 里的变元上定义的合取范式形式的布尔公式。 F 的真值赋值是从 X 到集合 $\{\top, \perp\}$ 的映射,其中 \top 和 \perp 是两个新的符号,分别读作真和假。设 T 是真值赋值并且 F 是布尔公式,若下列关系成立:对每个子句 $C \in F$,至少存在一个变元 x_i 使得要么 (a) $T(x_i) = \top$ 并且 $x_i \in C$, 要么 (b) $T(x_i) = \perp$ 并且 $\bar{x}_i \in C$, 则我们说 T 满足 F 。即,若子句包含至少一个真文字则它被满足,其中认为 x_i 为真当且仅当 $T(x_i) = \top$, 认为 \bar{x}_i 为真当且仅当 $T(x_i) = \perp$ 。

最后,若存在满足 F 的真值赋值,则 F 称为可满足的。

例 6.3.2 真值赋值 T 满足上面式(1)里的布尔公式 F , 其中 $T(x_1) = \perp, T(x_2) = \top, T(x_3) = \top$ 。这个真值赋值满足子句 $C_1 = (x_1 \vee \overline{x_2} \vee x_3)$, 因为 $T(x_3) = \top$ 并且 $x_3 \in C_1$; 满足子句 $C_2 = (\overline{x_1})$, 因为 $\overline{x_1} \in C_1$ 并且 $T(x_1) = \perp$; 最后它满足第三个子句 $C_3 = (x_2 \vee \overline{x_2})$ (这并不奇怪, 因为任何真值赋值都满足 C_3)。存在许多不满足 F 的真值赋值: 例如任何让 $T'(x_1) = \perp$ 的真值赋值 T' 都不满足 C_2 , 因此不满足 F 。不过 F 是可满足的, 因为至少存在一个满足它的真值赋值。 \diamond

例 6.3.3 现在考虑公式:

$$F' = \{(x_1 \vee x_2 \vee x_3), (\overline{x_1} \vee x_2), (\overline{x_2} \vee x_3), (\overline{x_3} \vee x_1), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})\}$$

它是否可满足? 在这里正确回答是“否”。让我们证明它。

子句 $(x_1 \vee x_2 \vee x_3)$ 要求变元 x_1, x_2, x_3 中至少有一个是 \top 。同理, 最后一个子句要求它们中至少有一个是 \perp 。然后考虑剩下的三个子句, 假设 $T(x_1) = \top$ 。于是为了满足子句 $(\overline{x_1} \vee x_2)$, $T(x_2)$ 必须是 \top ; 而为了满足 $(\overline{x_2} \vee x_3)$, 我们必须有 $T(x_3) = \top$ 。另一方面, 如果 $T(x_1) = \perp$, 那么子句 $(\overline{x_3} \vee x_1)$ 强迫 $T(x_3)$ 是 \perp , 而子句 $(\overline{x_2} \vee x_3)$ 又使得 $T(x_2)$ 必须是 \perp 。所以无论 $T(x_1)$ 是什么值, 如果想要满足这个公式, 那么三个变元的真值必须相同。

总之, 满足 F' 的真值赋值必须 (a) 至少把一个变元映射到 \top ; (b) 至少把一个变元映射到 \perp ; (c) 把所有三个变元映射到同样的真值。显然这样的真值赋值是不可能的, 所以 F' 是一个矛盾式的编码; 它是不可满足的。 \diamond

这个例子提示下列重要问题:

可满足性: 给定合取范式形式的布尔公式, 它是否可满足?

这是一个相当棘手的问题, 这也许在前一个例子就已经变得明显。确实, 对这个基本的和被非常深入研究过的问题, 至今没有已知的多项式时间算法, 并且人们普遍相信不存在这样的算法。

6.3.1 二元可满足性

假设我们把可满足性的实例限制在所有子句只有两个或更少文字的公式上, 于是得到我们称为二元可满足性的新问题。二元可满足性是可满足性的特殊情形。这样说的意思是所有可能的输入只是可满足性的输入的子集合, 并且在两个问题里对每个公共输入的回答都是同样的。二元可满足性的典型实例如下所示。

$$\{(x_1 \vee x_2), (x_3 \vee \overline{x_2}), (x_1), (\overline{x_1} \vee \overline{x_2}), (x_3 \vee x_4), (\overline{x_3} \vee x_5), (\overline{x_4} \vee \overline{x_5}), (x_4 \vee \overline{x_3})\} \quad (2)$$

下一步我们描述对这样的公式搜索满足的真值赋值的合理方法。在搜索过程中, 一些变元已经赋值为 \top 或 \perp , 其余变元还没有赋值。起初没有任何变元赋有真值。

假设在我们的公式里存在只有一个文字的子句, 比方说在式(2)里的第三个子句 (x_1) 。那么显然这个文字在任何满足的真值赋值里都必须是 \top , 因此我们立即决定这个变元的值。即在本例中我们立即决定 $T(x_1) = \top$, 然后继续进行。既然我们知道 $T(x_1) = \top$, 我们就从公式里删除包含 x_1 作为文字的所有子句, 因为这些子句已经被满足了 (在本例中我们删除第一个子句)。不过如果子句包含否定文字 $\overline{x_1}$, 那么我们就从子句里删除这个文字, 因为这个文字是 \perp 因此它不能用来满足子句。在本例中第一个子句被删除,

第四个子句被 (\bar{x}_2) 替换。因此对在子句里单个出现的文字赋真值,可能产生新的单文字子句,所以我们必须重复进行(在我们的例子(2)里下一步我们设置 $T(x_2)=\perp$)。

我们把寻找单文字子句直到不存在这样的子句为止的过程称为清洗。如果在清洗的任何一步产生了空子句——假定因为对某个 i ,子句 (x_i) 和 (\bar{x}_i) 都在前一步出现——那么我们说清洗已经失败。无论如何,在 $\mathcal{O}(n)$ 步之后,其中 n 是给定公式的子句数,这个清洗过程必定要么失败(在这种情形里我们判定公式是不可满足的),要么停止并且剩下一组子句,其中每个子句都有两个不同的文字。例如,在式(2)里初始清洗结束时设置 $T(x_1)=\top, T(x_2)=\perp$,并删除前四个子句。

因此假定我们的公式在每个子句里恰有两个文字。那么如何寻找可满足真值赋值呢?这里是简单的想法:选择还没有赋真值的任何变元,试验设置它的真值是 \top 并完成清洗;然后把公式恢复原状,把同一个变元设置成 \perp 并再次完成清洗。若两次清洗都失败则搜索结束,公式是不可满足的。若两次清洗中至少有一次成功,则设置变元等于成功的清洗中的真值并继续。在我们的例子里在第一遍清洗之后剩下四个子句

$$\{(x_3 \vee x_4), (\bar{x}_3 \vee x_5), (\bar{x}_4 \vee \bar{x}_5), (x_4 \vee \bar{x}_3)\} \quad (3)$$

试验真值 $T(x_3)=\top$ 启动新的清洗,它在设置 $T(x_5)=\top$ 之后失败(产生子句 (x_4) 和 (\bar{x}_4)),所以我们必须恢复在式(3)里的四个子句并试验 $T(x_3)=\perp$ 。这次成功地找到整个公式的可满足真值赋值(没有剩下任何子句),即 $T(x_4)=\top$ 和 $T(x_5)=\perp$ 。

容易看出(问题 6.3.2)当每个子句最多有两个文字时,这个简单算法正确地解决可满足性问题。因为算法对每个变元最多完成两遍清洗并且每遍清洗都在多项式时间里完成,所以由此得出二元可满足性属于 P。

习 题

- 6.3.1 试找出包括下述子句的布尔公式的所有满足的真值赋值: $(x_1 \vee \bar{x}_2 \vee x_3), (\bar{x}_1 \vee x_4), (x_2 \vee \bar{x}_3 \vee \bar{x}_4)$ 。
- 6.3.2 (a) 证明正文里描述的清洗算法在多项式时间里正确解决二元可满足性的任何实例。(提示:假设清洗算法判定公式不可满足,但是存在可满足真值赋值。那么清洗算法是如何遗漏这个赋值的?)
- (b) 对这个算法你能证明的最低多项式界限是多少?
- (c) 清洗算法在下述公式上如何工作? $(x_1 \vee \bar{x}_2), (\bar{x}_1 \vee \bar{x}_4), (x_2 \vee \bar{x}_3), (x_1 \vee x_4), (x_3 \vee x_4)$ 。
- 6.3.3 这是对二元可满足性属于 P 的另一个证明。可认为有两个文字的任何子句,比方说 $(x \vee y)$,是两个蕴涵式,即 $(\bar{x} \rightarrow y)$ 和 $(\bar{y} \rightarrow x)$ (可认为子句 (x) 是 $(\bar{x} \rightarrow x)$)。因此从二元可满足性的任何实例开始,构造用所有文字作为顶点并且画出所有这些蕴涵式作为边的有向图。证明二元可满足性的实例是不可满足的当且仅当存在变元 x 使得在这个图里存在从 x 到 \bar{x} 的通路和从 \bar{x} 到 x 的通路。由此证明二元可满足性属于 P。

6.4 NP 类

复杂性理论的主要目标之一是发现帮助证明我们感兴趣的问题不属于 P 的数学方法。我们已经见过这样的方法：用来证明 $E \notin P$ 的对角化论证，其中 E 是语言

$$E = \{ \langle M, w \rangle; M \text{ 在最多 } 2^{|w|} \text{ 步之后接受输入 } w \}$$

(定理 6.1.2)，这完全是对停机问题 H 的不可判定性的类比。不过这样的结果很难令人满意。原因是与可判定性的概念不同，P 和多项式时间计算是受到世俗和实践推动的概念。拿出像 E 那样与停机问题类似的人造问题并论证它不属于 P，这是不够的。我们想要认出自然的、合理的、在实践中重要的、不属于 P 的问题。

在上一节我们见过一批自然的、合理的、令人信服是有实际意义的、似乎不属于 P 的问题：Hamilton 圈，旅行商问题，独立集，划分，可满足性等。尽管数学家和计算机科学家们做出了长期巨大的努力对这些问题中的每一个寻找多项式时间算法，但还是没有找到任何这样的算法。于是最有价值的事情莫过于利用计算复杂性的思想和方法去证明这样的多项式时间算法是不可能的，从而把我们的下一代科学家从更多的不幸尝试中拯救出来。

不幸的是，在发现这样的不可能性证明上有难以捉摸的困难。我们下一步将看到，困难的原因是所有这些问题可用多项式界限非确定型 Turing 机解决。在多项式时间水平上分离确定性与非确定性是今日计算机科学里最重要和最深奥的悬而未决问题之一。

在第 4 章证明过如果语言 L 被几类(单带，多带，二维，甚至随机存取) Turing 机中的一类在多项式时间里判定，那么 L 被其他任意一类 Turing 机在多项式时间里判定。在第 4 章介绍过的 Turing 机模型的最后变种，即非确定型 Turing 机(4.6 节)，它的情况又怎样呢？是否在相差多项式的意义上它也等价于其余的种类？为了讨论这个重要问题，首先让我们形式化定义语言被非确定型 Turing 机在多项式时间界限里判定是什么意思。

定义 6.4.1 对非确定型 Turing 机 $M = (K, \Sigma, \Delta, s, H)$ ，若存在多项式 $p(n)$ 使得对任何输入 x ，不存在 M 的任何格局 C 满足 $(s, \sqcup x) \vdash_{\Delta}^{|x|+1} C$ ，则说 M 是多项式界限的，即这台机器的计算不会持续到超过多项式那么多步。定义 NP(表示非确定型多项式)是被多项式界限非确定型 Turing 机判定的所有语言的类。

此时此刻重要的是回忆对非确定型 Turing 机判定语言 L 是什么意思所下的独特定义：对每个不属于 L 的输入，机器的所有计算都必须拒绝输入；对每个属于 L 的输入，我们仅仅要求存在至少一个计算接受输入——只要存在一个接受计算，在其余的计算里就可以没有、或有些、或大多数、或全部都拒绝这个输入。

非确定型 Turing 机在给定输入上所有可能的计算最好是画成树形的结构(见图 6-3)。顶点表示格局，向下的线表示步。非确定性选择表示成有多条线离开的顶点。用垂直维度量时间。例如在图 6-3 里，输入在 5 步之后被接受。

这样的图形使得非确定性是如此强的计算方式的原因变得一目了然：在非常短的时间里(从根上算起的垂直距离)产生了天文数字那么多的格局。我们在下一个例子将看到用非确定性的这种能力来“解决”在上一节我们看到的某些棘手问题。

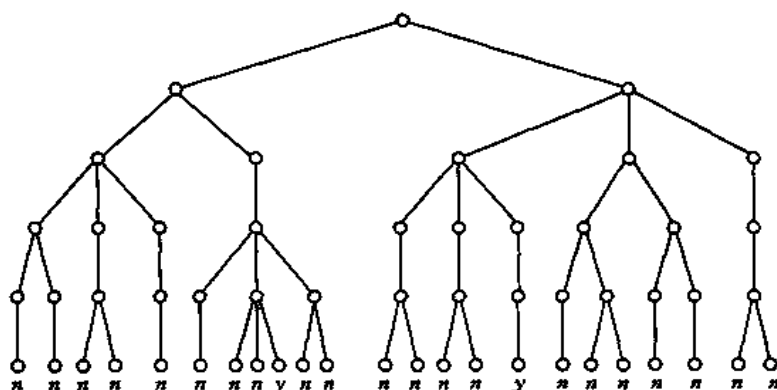


图 6-3

例 6.4.1 我们在前一节指出普遍相信可满足性不属于 P。现在让我们证明它属于 NP。我们设计在多项式非确定性时间里判定合取范式形式的可满足布尔公式的所有编码的非确定型 Turing 机 M 。

M 操作如下：在输入 w 上它首先验证 w 是否确实是合取范式形式的布尔公式的编码（若不是则它立即拒绝），并且计数在 w 里出现的变元的个数 n 。这是容易在多项式时间里完成的。在这个第一阶段的结尾， M 的第二条带包含字符串 $\triangleright I^n$ ，其中 I 与公式中的变元一样多。

然后 M 进入非确定性阶段，在这个阶段里 M 在第二条带上把所有 I 改写成长度为 n 的 \top 和 \perp 的序列。究竟是 \top 和 \perp 的哪个序列？回答是“任意序列，非确定性地”。更精确的回答是“所有序列，每个序列属于非确定性计算树的不同分枝”。容易设计出完成这件工作的非确定型机器：仅仅向 K 添加新状态 q ，向 Δ 添加转移（忽略其余的带，那里没有发生操作） $(q, I, q, \top), (q, I, q, \perp), (q, \top, q, \rightarrow), (q, \perp, q, \rightarrow), (q, \sqcup, q', \sqcup)$ ，其中 q' 是继续计算的状态。

M 的最后阶段是确定性的： M 把第二条带上 $\{\top, \perp\}^n$ 里的字符串解释成输入公式的真值赋值。然后它逐个访问输入里的每个子句，并验证子句是否含有在这个真值赋值下是 \top 的文字。若发现所有子句都有 \top 文字，则 M 接受。否则，若发现一个不满足的子句，则 M 拒绝。

容易看出，上面描述的 M 证明可满足性属于 NP。首先，所有计算的长度都有某个小的多项式界限。关键部分是，如果输入是可满足布尔公式的编码，那么 M 在非确定型计算的某个分枝上“猜测”满足赋值，因此至少有一个接受计算。除此之外，可能有许多拒绝计算。因此输入被接受。如果输入公式是不可满足的或者根本不是公式，那么所有计算最终都拒绝。

例 6.4.2 旅行商问题（像在 6.2 节用给定“预算” B 来定义的那样）也属于 NP。证明这个结论的非确定型 Turing 机在第二条带上非确定性地写与输入的长度相等的 0, 1 和 \sqcup 的字符串。然后机器进入确定性阶段，其中它验证写在第二条带上的字符串是否碰巧是整数 $1, \dots, n$ 的双射 π 的编码，其中 n 是给定输入里的城市数。双射 π 编码成用二进制写的 $\pi(1), \pi(2), \dots$ ，用 \sqcup 分隔。若字符串确实是双射的编码，则机器继续确定性地计算巡回

路线的成本,并且与输入里的“预算” B 比较。若成本不超过 B ,则机器接受;在所有其他的最终结局里(若所猜测的字符串不是双射的编码,或者若它表示成本大于 B 的巡回路线)这台机器都拒绝。显然字符串属于这台机器所判定的语言当且仅当它编码旅行商问题的“是”实例。

同理,容易证明我们在前一节遇到的其他明显的困难问题,包括独立集、Hamilton 圈和划分等(但是没有非确定型有穷自动机的等价性),也都属于 NP。◇

注意前两个例子中的非确定性“算法”是如何聪明地利用了在非确定性时间界限计算的定义里的基本非对称性。它们在独立的计算里试验所处理的问题的所有可能解,一旦发现可行解就立即接受它,忘掉其他的非可行解。

与递归可枚举语言作类比在这里是有诱惑力的,它的定义在接受与拒绝之间有类似的非对称性。像递归可枚举语言那样,根本就不清楚 NP 在补运算下是否封闭(但是在 P 的情形里和在递归函数类的情形里这是清楚的)。另外,显然 $P \subseteq NP$ (每一个递归语言都是递归可枚举这一事实的类比)。这是因为确定型机器只不过是转移关系碰巧是函数的非确定型机器。

P 是否等于 NP? 换句话说,对于在多项式时间里判定的语言类来说,非确定型 Turing 机是与其余的型号等价的另一种型号的 Turing 机吗? 通过第一印象你获得的直觉感觉是,非确定性是如此强和“不同的”特性使得这种情形不会成立。表示非确定型 Turing 机的计算树(回忆图 6-3)有许多顶点(即格局),所有这些顶点都是在中等的深度上。就可达到的格局数而言,确定型 Turing 机与非确定型 Turing 机竞争的唯一方法是通过操作指数多步。判定可满足性和旅行商问题的上述非确定型机器相当不费力地搜索了指数多种可能性,假如可以用有条理的确定性方式在多项式时间里达到同样的效果,那么这是真正值得注意的。

在利用确定型 Turing 机搜索大范围的“解”时遇到的这种困难也反映在定理 4.5.1 的证明里。在那里证明了非确定型 Turing 机可被确定型 Turing 机模拟,但是那个模拟不是直接的一步对一步的模拟,就像定理 4.3.1 和 4.3.2 里的模拟那样(我们设法证明了这两种模拟的多项式界限)。对非确定型 Turing 机的模拟需要借助对所有可能性的穷尽检查。同样,你得到的直觉感觉这是非确定性所固有的,因为它允许在每步上的多种选择,所以存在指数多种要验证的可能计算。

为了在时间性能上比较非确定型的和确定型的机器,我们必须首先定义更一般的语言类。

定义 6.4.2 设 $M = (K, \Sigma, \delta, s, H)$ 是确定型 Turing 机,若存在多项式 $p(n)$ 使得下列关系为真:对任意输入 x ,都不存在格局 C 使得 $(s, \triangleright \sqcup x) \vdash_{\delta}^{p(|x|)+1} C$,则说 M 是指数界限的。也就是说,机器在最多指数多步之后总是停机。

最后定义 EXP 是用指数界限确定型 Turing 机判定的所有语言的类。

定理 6.4.1 若 $L \in NP$,则 $L \in EXP$ 。

证明: 假设给定我们在时间界限 $p(n)$ 里判定 L 的非确定型多项式界限 Turing 机 M 。我们证明如何构造对某个常量 c ,在时间 $c^{p(n)}$ 里判定同一个语言的确定型 Turing 机 M' (通过对满足 $2^k > c$ 的某个 k 考虑多项式 $k \cdot p(n)$,然后得出定理)。 M' 恰好是在定理

4.5.1 的证明里构造的机器。 M' 在长度为 1 的所有可能的计算上模拟 M , 然后在长度为 2 的所有可能的计算上, 等等, 直到长度为 $p(n)+1$ 为止, 在这个时候要么发现了接受计算, 要么所有计算都已停机并拒绝。为了模拟长度为 l 的 M 的计算, M' 需要 $\mathcal{O}(l)$ 步——复制输入, 产生 $\{1, 2, \dots, r\}^l$ 里的下一个字符串 (其中 r 是 M 的非确定性度, 它是只依赖于 M 的固定数值, 等于在 Δ 里拥有相同的前两个分量的四元组的最大可能数), 以及按照这个字符串所提示的选择来模拟 M 。因此 M' 可在时间

$$\sum_{i=1}^{p(n)+1} r^i \leq (r+1)r^{p(n)+1}$$

里完成在长度为 n 的输入上对 M 的模拟, 设 $c=r+2$ 就完成了证明。 ■

我们已经指出, 是否 $P=NP$ 是在复杂性理论里具有核心重要性的目前还悬而未决的问题。是否 $NP=EXP$, 即在上述推论里的包含关系是否真包含, 是另一个悬而未决的问题, 不过重要性要低一些。但是我们确实知道下列结论: 在包含关系之链

$$P \subseteq NP \subseteq EXP$$

里, 第三个类真包含第一个类。原因是在定理 6.1.2 里证明不属于 P 的语言 E 肯定属于 EXP ; Turing 机在指数时间里在输入 w 上可模拟 M 到 $2^{|w|}$ 步。因此, 虽然我们猜想上面显示的两个包含关系都是真包含, 但是目前我们能证明的全部东西就是其中至少一个是真包含, 而且我们不知道是哪个……。

6.4.1 简短的证书

在例 6.4.1-2 里为判定可满足性和旅行商问题而设计的非确定型 Turing 机都是相当简单和有些相似的; 它们首先非确定性地产生字符串, 然后确定性地验证所产生的字符串是否具有与输入相关的某种需要的性质。若输入属于这个语言则至少存在一个合适的字符串。若输入不属于这个语言则找不到具有所需要性质的字符串。

这样的字符串称为证书, 或者见证。我们将看到所有 NP 里的问题都有证书, 并且只有 NP 里的问题才有证书。证书必须是多项式那么短的, 即它的长度最多是输入长度的多项式。它还必须是在多项式时间里可验证的。在可满足性的情形里验证证书就是要求验证真值赋值是否满足输入公式的所有子句; 在旅行商问题的情形里就是要求验证建议的巡回路线的总成本是否在预算之内; 对于独立集就是验证给定的顶点集是否大小合适并且在它们之间没有边等等。最后, 问题的所有“是”输入都必须有至少一个证书, 而所有“否”输入都必须没有任何证书。

证书的想法可在语言的领域里形式化, 因此提供对 NP 的有趣的另一种定义。

定义 6.4.3 设 Σ 是字母表, 并设 “,” 是不属于 Σ 的符号。考虑语言 $L' \subseteq \Sigma^*$; Σ^* 。若存在多项式 $p(n)$ 使得如果 $x, y \in L'$ 那么 $|y| \leq p(|x|)$, 则我们说 L' 是多项式平衡的。

定理 6.4.2 设 $L \subseteq \Sigma^*$ 是语言, 其中, $\notin \Sigma$, 并且 $|\Sigma| \geq 2$ 。那么 $L \in NP$ 当且仅当存在多项式平衡的语言 $L' \subseteq \Sigma^*$; Σ^* 使得 $L' \in P$, 并且 $L = \{x: \text{存在一个 } y \in \Sigma^* \text{ 使得 } x, y \in L'\}$ 。

证明: 直观上语言 L' 包括了所有输入的所有证书。即

$$L' = \{x, y: y \text{ 是 } x \text{ 的证书}\}$$

对每个 $x \in \Sigma^*$, 存在若干个 y 使得 $x, y \in L'$, 这些 y 组成 x 的证书的集合。若 $x \in L$ 则它

的证书集合至少有一个元素,若 $x \notin L$ 则这个证书集合是空的。

如果这样的语言 L' 存在,那么通过尝试所有的证书(非常像判定可满足性的非确定型 Turing 机所做的那样)并且利用判定 L' 的确定型 Turing 机,非确定型 Turing 机就可以判定 L 。反之,任何判定 L 的非确定型 Turing 机都为 L 的证书提供了坚实的框架:输入 x 的证书恰恰是 M 在 x 上的任何接受计算,它既是短的也是多项式可验证的。

形式化的证明留作练习(习题 6.4.5)。 ■

简短的证书这个概念最好还是用合数集合 $C \subseteq \mathbb{N}$ 来说明(回忆例 4.5.1 里的讨论)。假设给定我们用通常的十进制表示的自然数,例如 4 294 967 297,并问它是否合数。没有回答这样的问题的清楚有效的方法。不过每一个属于 C 的数确实都有简短的证书。例如,4 294 967 297 碰巧是合数,就有整数对 6 700 417 和 641 来作为证书,这对整数的乘积是 4 294 967 297。为了验证证书的有效性,你只要完成乘法就相信 $4\,294\,967\,297 \in C$ 。这正是证书的微妙之处:一旦你发现了它,你就能有效地展示它的有效性。但是发现它却是不容易的。上述 4 294 967 297 的因子分解,是数学家 Leonard Euler(1707—1783)在 1732 年首次发现的,这距离另一个伟大数学家 Pierre de Fermat(1601—1665)猜测这样的因子分解不存在过去了整整 92 年!

习 题

- 6.4.1 证明 NP 在并、交、连接和 Kleene 星号运算下封闭。(对 NP 的证明比对 P 的证明简单得多。)
- 6.4.2 定义 co NP 是下列语言类 $\{\bar{L}; L \in \text{NP}\}$ 。是否 $\text{NP} = \text{co NP}$,即是否 NP 在补运算下封闭,这是重要的悬而未决问题。证明如果 $\text{NP} \neq \text{co NP}$,那么 $P \neq \text{NP}$ 。
- 6.4.3 如果同态 h (回忆习题 2.3.11 和 3.5.3)不把任何符号映射到 ϵ ,那么它称为是非删除的。证明 NP 在非删除的同态下封闭。

你可能想要思考这些问题:NP 是否在一般的同态下封闭?对递归语言类情况怎样呢?对递归可枚举语言类呢?对 P 类呢?对最后一个问题,见下一章的习题 7.2.4。

- 6.4.4 给出划分和团的适当的简短证书。
- 6.4.5 证明定理 6.4.2。

参 考 文 献

在 20 世纪 50 年代末和 60 年代初已初露端倪的计算复杂性理论,是 Hartmanis 和 Stearns 在他们的下述文章里正式提出的,

- J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. Transaction of the American Mathematical Society, 117, pp. 285—305, 1965.

定理 6.1.2 由这篇文章里的结果得出。P 类恰当地刻画了“有效可解问题”这个概念的论题主要出现在 20 世纪 60 年代中期的这两篇文章里,它也蕴涵在更早的工作中,

- A. Cobham. The intrinsic computational difficulty of functions. Proceedings of the 1964 Congress for Logic, Mathematics and the Philosophy of Science, pp. 24—30, New York: North Holland, 1964.

1964.

- J. Edmonds. Paths, trees and flowers. Canadian Journal of Mathematics, 17(3), pp. 449—467, 1965.

在上述最后一篇文章里还非正式地介绍了 NP 类(利用证书,回忆定理 6.4.2),并首次猜测 $P \neq NP$ 。对计算复杂性的更深入的处理,见,

- C. H. Papadimitriou. Computational Complexity. Reading, Mass. : Addison-Wesley, 1994.

第 7 章 NP 完全性

7.1 多项式时间归约

我们在复杂性理论里使用的许多概念和技术,是我们为研究不可判定性而开发的概念和技术的有时间界限的类比。我们在定理 6.1.2 的证明里看到,为证明特定的语言不属于 P 而使用了对角化的多项式时间类比。下一步我们介绍多项式时间归约,它是在第 5 章里为证明不可判定性而使用的归约的类比。我们用多项式时间归约去研究几个重要而似乎困难的 NP 问题的复杂性,它们是在前一章里介绍的:可满足性,旅行商问题,独立集,划分和其他的问题等。我们将看到这些问题有下列重要的完全性性质:通过多项式时间归约,所有 NP 问题都可归约到它们——与把所有递归可枚举语言都归约到停机问题几乎同样的方式。我们把这样的问题称为是 NP 完全的。

不幸的是,与停机问题的类比到此为止。似乎不存在证明 NP 完全问题不属于 P 的简单的对角化论证;对角化论证似乎只适用于因太困难和太不自然而无关紧要的语言,比如定理 6.1.2 里的指数时间语言 E 。

尽管事实上这些 NP 完全问题没有提供 $P \neq NP$ 的证明,它们确实还是在我们的复杂性研究里占有重要的一席之地:如果我们假设 $P \neq NP$ ——这是普遍接受的猜想,虽然距离被证明还很遥远——那么所有 NP 完全问题确实不属于 P (这在下面的定理 7.1.1 里有清楚的说明)。在尚未证明 $P \neq NP$ 的情况下,这几分间接的困难性的证据就是我们对 NP 里问题的主要期望。

现在我们定义归约的多项式时间变种(与定义 5.4.1 比较)。

定义 7.1.1 如果存在计算函数 $f: \Sigma^* \rightarrow \Sigma^*$ 的多项式界限的 Turing 机 M , 那么 f 称为多项式时间可计算的。

现在设 $L_1, L_2 \subseteq \Sigma^*$ 是语言, 设 $\tau: \Sigma^* \rightarrow \Sigma^*$ 是多项式时间可计算的函数。如果对每个 $x \in \Sigma^*$ 下列关系成立: $x \in L_1$ 当且仅当 $\tau(x) \in L_2$, 那么 τ 称为从 L_1 到 L_2 的多项式归约。

多项式归约是重要的,因为它们揭示出计算问题之间的有趣关系。严格地说像上述那样定义的多项式归约联系的是两个语言而不是两个问题。不过从 6.2 节的讨论我们知道可用语言编码所有类型的重要计算问题,比如 Hamilton 圈、可满足性和独立集等。在这个意义下若 τ 是对应的语言之间的多项式归约,则我们说 τ 是从问题 A 到问题 B 的多项式归约。即 τ 以某种方式在多项式时间里把问题 A 的实例变换成问题 B 的实例,使得 x 是问题 A 的“是”实例当且仅当 $\tau(x)$ 是问题 B 的“是”实例。

当拥有从问题 A 到问题 B 的多项式归约 τ 时,可修改 B 问题的任何多项式时间算法

去获得 A 问题的多项式时间算法(如图 7-1 所示)。为了辨别 A 问题的任何给定实例 x 是

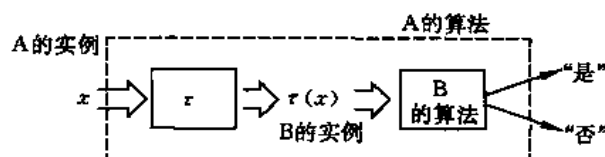


图 7-1

否确实是 A 的“是”实例,首先计算 $\tau(x)$,然后检验它是否 B 的“是”实例。如果拥有 B 的多项式时间算法,那么这种求解 A 的方法也是多项式的,因为归约步骤和解决得出的 B 的实例的算法都可以在多项式时间里完成。换句话说,从 A 到 B 的多项式时间归约的存在性就是 B 至少像 A 一样难的证据。若 B 是有效可解的则 A 也必然是有效可解的;若 A 需要指数时间则 B 也需要指数时间。

我们在下面给出归约的若干例子。

例 7.1.1 让我们描述从 Hamilton 圈到可满足性的多项式时间归约。假设给定我们 Hamilton 圈的实例,即图 $G \subseteq V \times V$, 其中 $V = \{1, 2, \dots, n\}$ 。我们描述算法 τ , 它产生合取范式形式的布尔公式 $\tau(G)$, 使得 G 有 Hamilton 圈(经过 G 的每个顶点恰好一次的回路)当且仅当 $\tau(G)$ 是可满足的。

公式 $\tau(G)$ 包含 n^2 个表示成 x_{ij} 的布尔变元, 其中 $1 \leq i, j \leq n$ 。直观上, 布尔变元 x_{ij} 具有这样的预期含义: “ G 的顶点 i 是 G 的 Hamilton 圈上的第 j 个顶点”。于是 $\tau(G)$ 的各种子句用布尔逻辑的语言去表达 Hamilton 圈必须满足的各种要求。

x_{ij} 必须满足什么要求才确实定义 G 的 Hamilton 圈? 我们把每个这样的要求都安排成子句。首先对 $j=1, \dots, n$, 有子句

$$(x_{1j} \vee x_{2j} \vee \dots \vee x_{nj})$$

满足这个子句的真值赋值必须在变元 $x_{1j}, x_{2j}, \dots, x_{nj}$ 里至少设置一个变元是 1, 因此按照布尔变元的“预期含义”, 这个子句表达在 Hamilton 圈的第 j 个位置上出现至少一个顶点的要求。

但是 Hamilton 圈的第 j 个位置上只能出现一个顶点。因此向布尔公式里加入所有 $O(n^2)$ 个如下形式的子句

$$(\overline{x_{ij}} \vee \overline{x_{kj}})$$

其中 $i, j, k=1, \dots, n$ 并且 $i \neq k$ 。因为必须满足这个子句里的至少一个文字, 所以这些子句成功地表达出在圈的第 j 个位置上不能同时出现顶点 i 和顶点 k 的要求。

迄今为止的子句保证 Hamilton 圈的第 j 个位置上恰好出现一个顶点。但是我们还必须要求顶点 i 恰有一次出现在圈上。达到这个要求是通过下列子句实现的

$$(x_{i1} \vee x_{i2} \vee \dots \vee x_{in}), \quad i = 1, \dots, n$$

和

$$(\overline{x_{ij}} \vee \overline{x_{ik}}), \quad i, j, k = 1, \dots, n \text{ 且 } j \neq k.$$

迄今为止这些子句保证 x_{ij} 表示 G 的顶点之间的双射或置换。下一步我们必须用新的

子句表达这个置换确实是 G 的圈的要求。我们达到这个要求是通过对于 $j=1, \dots, n$ 和每对顶点 (i, k) , 这里 (i, k) 不是 G 的边, 加入下列子句

$$(\overline{x_{ij}} \vee \overline{x_{k,j+1}}) \quad (1)$$

在这里 $x_{k,n+1}$ 表示布尔变元 x_{k1} , 即假定第二个下标的加法是模 n 的。直观上, (1) 里的子句说如果在 G 里不存在 (i, k) 边, 那么 i 和 k 不能以这样的顺序连续地出现在所断言的 Hamilton 圈里。对 $\tau(G)$ 的构造到此完成。

容易论证, $\tau(G)$ 的构造可在多项式时间里完成。构造的子句有 $\mathcal{O}(n^3)$ 个, 文字的总数是 $\mathcal{O}(n^3)$ 个。子句的结构极其简单, 要么只依赖于 n , 要么以相当直截了当的方式依赖于 G 的边。构造计算函数 τ 的多项式时间 Turing 机是直截了当的。

下一步我们需要证明 G 有 Hamilton 圈当且仅当 $\tau(G)$ 是可满足的。假设存在满足 $\tau(G)$ 的真值赋值 T 。因为 T 必须在除上述 (1) 外的子句里让至少一个文字成为 \top , 所以 T 是 G 的顶点上的双射的编码, 即对每个 i 恰有一个 $T(x_{ij})$ 是 \top , 对每个 j 恰有一个 $T(x_{ij})$ 是 \top , 用 $\pi(j)$ 表示满足 $T(x_{i\pi(j)}) = \top$ 的唯一的 i 。

上述 (1) 里的全部子句也必须被满足。这意味着每当 $i = \pi(j)$ 和 $k = \pi(j+1)$ 时 (其中 $n+1$ 还是意味着 1), 那么 (j, k) 必须是 G 的边。所以 $(\pi(1), \pi(2), \dots, \pi(n))$ 确实是 G 的 Hamilton 圈。当的方向已经被证明了。

反之, 假设 G 有 Hamilton 圈, 比方说 $(\pi(1), \pi(2), \dots, \pi(n))$ 。于是容易看出, 真值赋值 T 满足 $\tau(G)$ 里的全部子句, 其中 $T(x_{ij}) = \top$ 当且仅当 $i = \pi(j)$ 。证毕。

以后在本章里我们将看到相反方向的归约 (定理 7.3.1 和 7.3.2)。

就所涉及的问题的困难性而言, 正确地解释多项式归约所揭示的信息是迷惑人的、需要小心谨慎和经验的工作。此时此刻我们鼓励读者思考下列问题: 关于 Hamilton 圈的复杂性, 这个归约是好消息还是坏消息? 关于可满足性呢? 假如我们有 Hamilton 圈的多项式时间算法, 那么根据这个归约, 关于可满足性我们能得出什么结论? 假如我们有可满足性的多项式时间算法, 那么能得出什么结论? 如果我们知道 Hamilton 圈是困难问题, 那么我们能得出什么结论? 如果可满足性是困难问题呢? \diamond

例 7.1.2 你必须在两台机器上调度 n 个作业。两台机器速度相同, 每个作业可在任何机器上执行, 不限制作业的执行顺序。给定作业的执行时间 a_1, a_2, \dots, a_n 和截止时间 D , 所有数字都用二进制表示。你能否在截止时间之前在两台机器上完成所有这些作业?

陈述这个问题的另一种方式如下: 是否存在把给定的二进制数字划分成两组的方法, 使得每组数字加起来小于或等于 D ? 我们把这个问题称为双机调度, 它与在上一章里定义的划分问题的联系也许是清楚的。我们还介绍与划分密切相关的另一个问题。

背包问题: 给定非负整数的集合 $S = \{a_1, a_2, \dots, a_n\}$ 和整数 K , 所有数字用二进制表示, 是否存在子集 $P \subseteq S$ 使得 $\sum_{a_i \in P} a_i = K$?

(这个形象的名字让人想到步行者正试图用不同重量的物品把她的背包装满为止。)

这三个问题 (划分、背包问题和双机调度) 是如何通过多项式归约联系起来的? 我们证明存在六个多项式归约, 把三个问题里的任何一个归约到其中另外一个!

假设我们有带有整数 a_1, a_2, \dots, a_n 和 K 的背包问题的实例, 我们必须把它归约到划

分的等价实例。如果 K 碰巧等于给定的整数的半和 $H = \frac{1}{2} \sum_{i=1}^n a_i$, 那么归约需要做的所有工作就是从输入里删除 K , 所得出的划分的实例等价于给定的背包问题的实例。当然, 问题是一般情况下 K 不等于 H 。但是这容易解决: 向 a_i 的集合里加入两个新的大整数 $a_{n+1} = 2H + 2K$ 和 $a_{n+2} = 4H$ (注意即使 H 不是整数, 这两个数也是整数。)。现在把得出的整数集合 $\{a_1, \dots, a_{n+2}\}$ 看作划分的实例——归约完成了。显然这个归约可在多项式时间里完成。

现在我们必须证明归约可行, 即划分的实例有解当且仅当原来的背包的实例有解。首先注意如果新的整数集合可划分成有相等的和的两个集合, 那么两个新整数 a_{n+1} 和 a_{n+2} 必须落在划分的不同集合里 (这二数之和超过其余整数之和)。设 P 是原来 a_1, \dots, a_n 里与 $a_{n+2} = 4H$ 在同一个集合的整数。我们有

$$4H + \sum_{a_i \in P} a_i = 2H + 2K + \sum_{a_j \in S-P} a_j$$

在两边加上 $\sum_{a_i \in P} a_i$ 得到 (因为 $\sum_{a_i \in S} a_i = 2H$)

$$4H + 2 \sum_{a_i \in P} a_i = 4H + 2K$$

或者 $\sum_{a_i \in P} a_i = K$ 。因此如果得出的划分的实例有解, 那么原来的背包问题的实例就有解。反之, 如果我们有背包问题的解, 那么向它加入 a_{n+2} 就产生划分的实例的解。

这是从背包问题到划分的归约。我们从背包问题的任意实例出发构造出划分的等价实例。相反方向的归约是平凡的, 因为划分是背包问题的特殊情形。给定带有整数 a_1, \dots, a_n 的划分的任何实例, 到背包问题的归约把划分的给定实例变换成带有同样数字

和容量 $K = \frac{1}{2} \sum_{i=1}^n a_i$ 的背包问题的实例。

但是如果 K 不是整数 (即给定的整数之和是奇数) 怎么办呢? 那么就让归约产生我们所希望的背包问题的任何不可能实例, 比如 $n=1, a_1=2$ 和 $K=1$ 的实例, 此时给定的划分的实例也是不可能的。

从划分到双机调度的归约也是容易的: 给定带有整数 a_1, \dots, a_n 的划分的实例, 归约产生的双机调度的实例带有 n 个作业, 执行时间 a_1, \dots, a_n , 和截止时间 $D = \lfloor \frac{1}{2} \sum_{i=1}^n a_i \rfloor$ (如果 $\frac{1}{2} \sum_{i=1}^n a_i$ 不是整数, 那么这已经是不可能实例)。容易看出所得出的双机调度的实例有解当且仅当原来的划分的实例有解。这是因为作业可划分成各自的和不超过 D 的两个集合当且仅当它们可划分成各自的和恰好是 D 的两个集合。

从双机调度到划分的归约要更复杂一些。假设给定带有作业长度 a_1, \dots, a_n 和截止时间 D 的双机调度的实例。考虑数字 $I = 2D - \sum_{i=1}^n a_i$ 。直观上 I 是任何合法调度的全部空闲时间 (如图 7-2 所示)。它是在求解该调度问题时我们拥有的松弛量。现在我们向作业长度的集合里加入几个新数字使得 (a) 新数字之和为 I ; (b) 更重要的是在 0 与 I 之间的任何数字可表示成新数字的子集之和。不难看出如果我们设法这样做了, 那么得出的划分的实

例就等价于原来的双机调度的实例,因为通过向两个集合里分别加入新引进数字的子集使得两个集合里数字之和都是 D ,我们就把原来作业的任何可行调度变换成新数字集合的公平合理的划分。

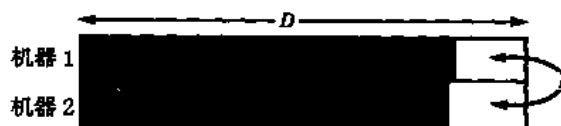


图 7-2

现在我们需要做的全部事情是提供总和为 I 的整数,使得在零与 I 之间的任何数字是这些整数的子集之和。表面上看这样做似乎是平凡的:加入 I 个整数 1。不过这个归约的错误是它不是多项式时间归约,这与我们在前一章里简述的有关划分的算法不是多项式的是出于同样的原因:整数 I 不能用输入规模的多项式来限界,因为输入包括作业长度和截止时间,都用二进制编码,数字 I 可能是输入规模的指数大小。

解决办法要更复杂一些:加入的数字都是小于 $\frac{1}{2}I$ 的 2 的方幂,外加让总和等于 I 的另一个数字。例如,如果 $I=56$,那么新加入的数字是 1, 2, 4, 8, 16 和 25。在 0 与 56 之间的所有整数,并且只有这些整数,可表示成所加入的整数的某个子集之和。这就完成了对从双机调度到划分的归约的描述。这个归约是多项式的,因为我们引进的整数的个数不超过 I 的对数,它小于输入规模。◇

在上述例子里我们没有讨论从背包问题到双机调度和相反方向的直接归约。但是并没有这个必要:正如下面所示,多项式归约以传递的方式来合成。回忆两个函数 $f: A \mapsto B$ 和 $g: B \mapsto C$ 的合成是函数 $f \circ g: A \mapsto C$,其中对所有的 $x \in A$, $f \circ g(x) = g(f(x))$ 。

引理 7.1.1 如果 τ_1 是从 L_1 到 L_2 的多项式归约并且 τ_2 是从 L_2 到 L_3 的多项式归约,那么它们的合成 $\tau_1 \circ \tau_2$ 是从 L_1 到 L_3 的多项式归约。

证明: 假设 τ_1 用 Turing 机 M_1 在时间 p_1 里计算, p_1 是多项式,并且 τ_2 用 M_2 在时间 p_2 里计算, p_2 也是多项式。那么 $\tau_1 \circ \tau_2$ 可用机器 $M_1 M_2$ 计算。在输入 $x \in \Sigma^*$ 上 $M_1 M_2$ 输出 $\tau_1 \circ \tau_2(x)$,所用时间不超过多项式 $p_1(|x|) + p_2(p_1(|x|))$ 。后一项所反映出的事实是 $|\tau_1(x)|$ 不大于 $p_1(|x|)$ 。

还剩下证明 $x \in L_1$ 当且仅当 $\tau_1 \circ \tau_2(x) \in L_3$ 。这是显然的: $x \in L_1$ 当且仅当 $\tau_1(x) \in L_2$, 当且仅当 $\tau_1 \circ \tau_2(x) \in L_3$ 。■

最后,我们得出下述重要定义。

定义 7.1.2 设 $L \subseteq \Sigma^*$ 是语言,如果

(a) $L \in \text{NP}$; 并且

(b) 对每一个语言 $L' \in \text{NP}$, 存在从 L' 到 L 的多项式归约,

那么 L 称为是 NP 完全的。

就像 H 的可判定性刻画了 Turing 可接受语言的可判定性的整个问题那样,任何 NP 完全语言是否属于 P 的问题事实上等价于整个 $P = \text{NP}$ 问题:

定理 7.1.1 设 L 是 NP 完全语言。那么 $P = \text{NP}$ 当且仅当 $L \in P$ 。

证明: (仅当)假设 $P=NP$ 。因为 L 是 NP 完全的, 所以 $L \in NP$ (回忆 NP 完全语言必须属于 NP), 由此得出 $L \in P$ 。

(当)假设 L 是 NP 完全语言, 它用确定型 Turing 机 M_1 在时间 $p_1(n)$ 里判定, $p_1(n)$ 是多项式。又设 L' 是 NP 里的任意语言, 我们证明 $L' \in P$ 。

因为 L 是 NP 完全的并且 $L' \in NP$, 所以存在从 L' 到 L 的多项式归约 τ (现在使用 NP 完全语言的定义的第二部分)。假设 τ 用某个 Turing 机 M_2 在时间 $p_2(n)$ 里计算, $p_2(n)$ 也是多项式。那么我们断言 M_2M_1 在多项式时间里判定 L' 。为了看出 M_2M_1 判定 L' , 注意 M_2M_1 接受 x 当且仅当 $\tau(x) \in L$; 以及因为 τ 是多项式归约, 所以 $\tau(x) \in L$ 当且仅当 $x \in L'$ 。

最后为了分析 M_2M_1 的时间需求, 注意它初始的 M_2 部分在输入 x 上花费时间 $p_2(|x|)$ 来产生给 M_1 的输入。这个输入最多长 $p_2(|x|)$, 因为 M_2 每步不能写多于一个的符号。因此 M_1 在这个输入上最多花费时间 $p_1(p_2(|x|))$ 。整个机器在输入 x 上在时间 $p_2(|x|) + p_1(p_2(|x|))$ 之内停机, 这个时间是 $|x|$ 的多项式。

因为 L' 是在 NP 里选取的任意语言而且我们得出结论说 $L' \in P$, 所以由此得出 $NP=P$ 。 ■

习 题

7.1.1 在可着 3 色问题里给定无向图, 问是否可用 3 种颜色对顶点着色使得没有任何两个相邻顶点有相同颜色。

(a) 证明可着 3 色属于 NP。

(b) 描述从可着 3 色到可满足性的多项式时间归约。

7.1.2 有些作者定义通常称为**多项式 Turing 归约**的更一般的归约概念。设 L_1 和 L_2 是语言。从 L_1 到 L_2 的**多项式 Turing 归约**是带有三个不同状态 q_1 , q_1 和 q_0 的 2 带 Turing 机, 它在多项式时间里判定 L_1 , 并且它的计算是用相当特殊的方式来定义的。所有不含状态 q_1 的格局之间的产生关系的定义与普通 Turing 机完全一样。而对 q_1 , 我们说 $(q_1, \triangleright u_1 \underline{a_1} v_1, \triangleright u_2 \underline{a_2} v_2) \vdash M (q, \triangleright u'_1 \underline{a'_1} v'_1, \triangleright u'_2 \underline{a'_2} v'_2)$ 当且仅当 (1) $u_1 = u'_1$, $a'_1 = a_1$, $v'_1 = v_1$, $u_2 = u'_2$, $a'_2 = a_2$, $v'_2 = v_2$, 并且

(2) 下列之一成立:

(2a) $u_2 a_2 v_2 \in L_2$ 并且 $q' = q_1$, 或者

(2b) $u_2 a_2 v_2 \notin L_2$ 并且 $q' = q_0$ 。

换句话说, 从状态 q_1 出发 M 永不改变带上的任何内容, 它只根据第二条带上的字符串是否属于 L_2 来进入状态 q_1 或 q_0 。另外对 M 来说, 这只算作一步。

(a) 证明: 如果存在从 L_1 到 L_2 和从 L_2 到 L_3 的多项式 Turing 归约, 那么存在从 L_1 到 L_3 的多项式 Turing 归约。

(b) 证明: 如果存在从 L_1 到 L_2 的多项式 Turing 归约并且 $L_2 \in P$, 那么 $L_1 \in P$ 。

(c) 给出从 Hamilton 圈到 Hamilton 通路 (在本问题里我们不要求经过每个顶点恰好一次的通路是封闭的) 的多项式 Turing 归约。你能否找到在这两个问题之间的直接的 (即不使用下面定理 7.3.2 的证明里的归约) 多项式归约?

7.2 Cook 定理

我们还没有证明 NP 完全语言存在,但它们确实存在。在过去的 20 年里,计算复杂性研究已经发现了数以百计的这样的 NP 完全语言(或者 NP 完全问题,因为我们继续不区别计算问题,比如可满足性和 Hamilton 圈等,与编码它们的语言)。大量的 NP 完全问题是来源于运筹学、逻辑学、组合学、人工智能以及其他似乎互不关联的应用领域的重要实际问题。在发现 NP 完全性之前,许多研究努力白白地投入到寻找有关这些问题的多项式时间算法上。通过证明除非 $P=NP$ ——似乎与直觉和经验都矛盾的情况——否则这些问题都不是多项式时间算法可解的,NP 完全性的概念就统一了在这些不同领域里的研究者的经验。出现这个结果所带来的好处是,把过去集中在解决特殊的 NP 完全问题上的研究努力转移到其他更容易的目标上,这些都是 7.4 节的主题。研究努力的转向一直是计算理论对计算实践的最重要的影响。

7.2.1 有界铺砖

一旦我们证明了第一个 NP 完全问题,通过把已知的 NP 完全问题归约到其他的问题上,并利用多项式归约的传递性(回忆引理 7.1.1),就可以证明其他的问题是 NP 完全的。但是第一个 NP 完全性的证明必须是对定义的应用;我们必须证明 NP 里的所有问题都归约到这个问题。历史上 Stephen A. Cook 在 1971 年证明的第一个 NP 完全问题是可满足性。我们不直接给出那个证明,而是从铺砖问题的变种开始,在第 5 章已证明铺砖问题是不可判定的。

在原来的铺砖问题里给定铺砖系统 \mathcal{D} ,问我们是否有办法铺满无穷的第一象限,使得任何两块水平地或垂直地相邻的砖都以规定的方式相关,并且在原点上放给定的砖。我们定义称为有界铺砖的不那么宏大的问题,在这个问题里问是否有不是铺满整个象限,而是铺满 $s \times s$ 的正方形的合法铺砖,其中 $s > 0$ 是给定的整数。这次,用指定整个第一行的砖取代仅仅指定放在 $(0,0)$ 上的砖。即给定我们铺砖系统 $\mathcal{D} = (D, H, V)$ (其中我们省略初始砖 d_0 ,现在它无关紧要),整数 $s > 0$ 以及函数 $f_0: \{0, \dots, s-1\} \mapsto \mathcal{D}$ 。问我们是否有按照 \mathcal{D} 扩充 f_0 的 $s \times s$ 的铺砖,即函数 $f_0: \{0, \dots, s-1\} \times \{0, \dots, s-1\} \mapsto \mathcal{D}$ 使得

对所有 $m < s, f(m, 0) = f_0(m)$;

对所有 $m < s-1, n < s, (f(m, n), f(m+1, n)) \in H$;

对所有 $m < s, n < s-1, (f(m, n), f(m, n+1)) \in V$ 。

有界铺砖是这样的:

有界铺砖: 给定铺砖系统 \mathcal{D} , 整数 $s > 0$, 以及用值的序列 $(f_0(0), \dots, f_0(s-1))$ 来表示的函数 $f_0: \{0, \dots, s-1\} \mapsto D$, 是否有按照 \mathcal{D} 扩充 f_0 的 $s \times s$ 的铺砖?

定理 7.2.1 有界铺砖是 NP 完全的。

证明: 让我们首先论证它属于 NP。这个问题的证书是铺砖函数 f 的 s^2 个值的完整列表。这样的函数可在多项式时间里被验证是否服从上述三项要求。另外它是短的: 它的总长度是 s^2 乘以用来表示每块砖的符号数, 而且输入的长度是 s 的上界, 因为输入包括

f_0 的列表。实际上对铺砖的形式化的这种技巧恰恰是为了保证问题属于 NP。如果我们仅仅指定一块初始的砖,那么问题变得难得多——可证明它不属于 P;见习题 7.2.2。

现在我们必须证明通过多项式归约, NP 里的所有语言都归约到有界铺砖上。因此考虑任意语言 $L \in \text{NP}$ 。我们必须产生从 L 到有界铺砖上的多项式归约,即多项式时间可计算的函数 τ 使得对每个 $x \in \Sigma^*$, $\tau(x)$ 是铺砖系统 $\mathcal{D} = (D, H, V)$ 的编码,外加整数 $s > 0$ 和函数 f_0 的编码,它们带有这样的性质:存在根据 \mathcal{D} 扩充 f_0 的 $s \times s$ 的铺砖当且仅当 $x \in L$ 。

为了获得这样的归约,必须以某种方式探讨关于 L 我们拥有的少量信息。关于 L 我们的全部所知就是它是 NP 里的语言,即存在非确定型 Turing 机 $M = (K, \Sigma, \Delta, s, H)$ 使得 (a) 对某个多项式 p , M 在输入 x 上的所有计算都在 $p(|x|)$ 步里停机, (b) 在输入 x 上存在接受计算当且仅当 $x \in L$ 。

我们首先描述 τ 在输入 x 上构造的整数 s ;它是 $s = p(|x|) + 2$,它比 M 在输入 x 上的时间界限多 2。

在 $\tau(x)$ 里描述的铺砖系统 \mathcal{D} 非常类似于在无界铺砖问题的不可判定性证明里构造的铺砖系统(定理 5.6.1)。像在那个构造里那样,我们用砖边的标记来描述 \mathcal{D} 里的砖;同样,在 t 与 $t+1$ 行之间的水平边标记表示 M 在输入 x 上合法计算在恰好第 t 步之后的带内容(因为 M 是非确定型的,所以存在多种这样的计算,因此存在铺满 $s \times s$ 正方形的多种可行的合法方式)。

按照 f_0 的规定, $s \times s$ 正方形的第 0 行由表示初始格局 $(s, \triangleright \sqcup x)$ 的砖占据。即, $f_0(0)$ 是上方的边标记了 \triangleright 的砖, $f_0(1)$ 是上方的边标记了 (s, \sqcup) 的砖,以及对 $i = 1, \dots, |x|$, $f_0(i+1)$ 是上方的边标记了 x_i 的砖, x_i 是输入 x 的第 i 个字母。最后对所有 $i > |x| + 1$, $f_0(i)$ 是上方的边标记了 \sqcup 的砖(如图 7-3 所示)。因此在第 0 与第 1 行之间的水平边的标记表示 M 在输入 x 上的初始格局。

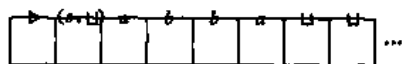


图 7-3

\mathcal{D} 里其余的砖都恰恰是像在定理 5.6.1 的证明里那样的。因为机器是非确定型的,所以对应于当 M 在状态 q 里扫描 a 时可能的多种选择,存在着带有下方水平标记 $(q, a) \in \Sigma \times \Sigma$ 的多块砖,它们中的每块都像在那个证明里那样被构造出来。只有一个小小的差别:对每个符号 a ,存在带有同样的上方和下方标记 (q, a) 的砖,当计算在状态 y 上停机之后,这些砖有效地允许铺砖继续下去,但是若计算在状态 n 上停机则没有这些砖。这样就完成了对有界铺砖的实例 $\tau(x)$ 的构造。对实例的构造可在 $|x|$ 的多项式时间里完成这是清楚的。

现在我们必须证明存在按照 \mathcal{D} 的 $s \times s$ 的铺砖当且仅当 $x \in L$ 。假定存在 $s \times s$ 的铺砖。 f_0 的定义保证在第 0 与第 1 行之间的水平边标记必须表示 M 在输入 x 上的初始格局。通过归纳得出在第 n 与 $n+1$ 行之间的水平边标记表示 M 在输入 x 上某种合法计算在恰好第 n 步之后的格局。因为 M 在输入 x 上没有计算持续到超过 $p(|x|) = s - 2$ 步,所以第 $s-2$ 行的上方标记必须包含符号 y 和 n 之一。因为存在第 $s-1$ 行,并且没有带有下方标记 n 的砖,所以我们必然得出结论说符号 y 出现,因此计算是接受的。我们得出结论说,如果存在按照 \mathcal{D} 对 $s \times s$ 正方形的铺砖,那么 M 接受 x 。

反之,若 M 在输入 x 上有接受计算,则它可被铺砖轻易地模拟(若计算在小于

$p(|x|)$ 步里在状态 y 上停机,则多次地重复最后一行)。证毕。 ■

现在我们利用有界铺砖去证明本节的主要结果。

定理 7.2.2(Cook 定理) 可满足性是 NP 完全的。

证明: 我们已经论证了问题属于 NP, 下一步我们把有界铺砖归约到可满足性上。

给定有界铺砖的任意实例, 比方说铺砖系统 $\mathcal{D} = (D, H, V)$, 边长 s 以及最底行 f_0 , 其中 $D = \{d_1, \dots, d_k\}$, 我们证明如何构造布尔公式 $\tau(\mathcal{D}, s, f_0)$, 使得存在按照 \mathcal{D} 的 $s \times s$ 的铺砖当且仅当 $\tau(\mathcal{D}, s, f_0)$ 是可满足的。

对每个 $0 \leq m, n < s$ 和 $d \in D$, 在 $\tau(\mathcal{D}, s, f_0)$ 里的布尔变元是 x_{mnd} 。在这些变元与铺砖问题之间预期的对应关系是变元 x_{mnd} 为 \top 当且仅当 $f(m, n) = d$ 。下一步我们描述保证 f 确实是合法的按照 \mathcal{D} 的 $s \times s$ 铺砖的子句。

首先对每个 $m, n < s$, 子句

$$(x_{mnd_1} \vee x_{mnd_2} \vee \dots \vee x_{mnd_k})$$

说每个位置至少有一块砖。对每个 $m, n < s$ 以及每两块不同的砖 $d \neq d' \in D$, 我们有子句 $(\overline{x_{mnd}} \vee \overline{x_{mnd'}})$, 它说一个位置不能有超过一块的砖。迄今为止描述的子句保证了 x_{mnd} 表示从 $\{0, \dots, s-1\} \times \{0, \dots, s-1\}$ 到 D 的函数 f 。

下一步我们必须构造子句保证全体 x_{mnd} 描述的函数是合法的按照 \mathcal{D} 的铺砖。首先对 $i = 0, \dots, s-1$, 我们有子句 $(x_{i0f_0(i)})$, 它迫使 $f(i, 0)$ 是 $f_0(i)$ 。然后对每个 $n < s$ 和 $m < s-1$, 以及对每个 $(d, d') \in D^2 - H$, 我们有子句

$$(\overline{x_{mnd}} \vee \overline{x_{m+1,n,d'}})$$

它禁止非水平相容的两块砖水平地彼此相邻。关于垂直相容性, 对每个 $n < s-1$ 和 $m < s$, 以及对每个 $(d, d') \in D^2 - V$, 我们有子句 $(\overline{x_{mnd}} \vee \overline{x_{m,n+1,d'}})$ 。这样就完成了对 $\tau(\mathcal{D}, s, f_0)$ 里的子句的构造。剩下来要证明 $\tau(\mathcal{D}, s, f_0)$ 是可满足的当且仅当存在按照 \mathcal{D} 扩充 f_0 的 $s \times s$ 的铺砖。

于是, 假设 $\tau(\mathcal{D}, s, f_0)$ 是可满足的, 比方说被赋值 T 满足。因为大的析取式都被 T 满足, 所以对每个 m 和 n , 至少存在一个 $d \in D$ 使得 $T(x_{mnd}) = \top$ 。因为子句 $(\overline{x_{mnd}} \vee \overline{x_{mnd'}})$ 都被 T 满足, 所以不存在 m 和 n 使得两个或者更多的 x_{mnd} 在 T 下是 \top 。因此 T 表示函数 $f: \{0, \dots, s-1\} \times \{0, \dots, s-1\} \rightarrow D$ 。

我们断言 $f(m, n)$ 是合法的扩充 f_0 的 $s \times s$ 的铺砖。首先, 因为子句 $(x_{i0f_0(i)})$ 都被满足, 所以必然像所要求的那样 $f(i, 0) = f_0(i)$ 。其次, 水平相邻约束条件必然被满足, 因为如果它在位置 (m, n) 和 $(m+1, n)$ 上不被满足, 那么剩下子句 $(\overline{x_{mnd}} \vee \overline{x_{m+1,n,d'}})$ 之一不被满足。对于垂直相邻同理可证, 因此 $f(m, n)$ 确实是合法的扩充 f_0 的 $s \times s$ 的铺砖。

反之, 假设存在合法的扩充 f_0 的 $s \times s$ 铺砖。于是定义下列真值赋值 $T: T(x_{mnd}) = \top$ 当且仅当 $f(m, n) = d$ 。容易验证 T 满足所有子句, 证毕。 ■

因此除非 $P = NP$, 否则不存在可满足性的多项式时间算法。我们在 6.3 节看到特殊情形的二元可满足性可在多项式时间里解决。下列定理提示相邻的下一个最密切的情形已经是难解的。通过与二元可满足性类比, 设三元可满足性是所有子句都碰巧包含三个或更少的文字的可满足性的特殊情形。

定理 7.2.3 三元可满足性是 NP 完全的。

证明: 它当然属于 NP, 因为它是属于 NP 的问题的特殊情形。

为了证明完全性, 我们把可满足性归约到三元可满足性上。这是相当普通类型的归约, 在这里把问题归约到它自身的特殊情形上。通过证明从一般问题的任意实例开始, 我们设法消除妨碍这个实例落进特殊情形里的特性, 这样的归约就可行了。在目前情况下我们必须证明从任意子句集 F 出发, 如何在多项式时间里得到在每个子句里最多有三个文字的子句集 $\tau(F)$ 。

归约是简单的。对 F 里每个有 $k > 3$ 个文字的子句, 比方说

$$C = (\lambda_1 \vee \lambda_2 \vee \cdots \vee \lambda_k)$$

做下列工作: 设 y_1, \dots, y_{k-3} 是不在布尔公式 $\tau(F)$ 的别处出现的新的布尔变元, 我们把子句 C 换成下列子句集:

$$(\lambda_1 \vee \lambda_2 \vee y_1), (\neg y_1 \vee \lambda_3 \vee y_2), (\neg y_2 \vee \lambda_4 \vee y_3), \dots, \\ (\neg y_{k-4} \vee \lambda_{k-2} \vee y_{k-3}), (\neg y_{k-3} \vee \lambda_{k-1} \vee \lambda_k)$$

我们以这种方式拆碎 F 的所有“长的”子句, 对每个子句使用不同的 y_i 变元集。我们照原样保留“短的”子句。得出的布尔公式就是 $\tau(F)$ 。容易看出 τ 可在多项式时间里完成。

我们断言 $\tau(F)$ 是可满足的当且仅当 F 是可满足的。在直观上, 把变元 y_i 解释成“文字 $\lambda_{i+2}, \dots, \lambda_k$ 至少有一个为真”, 并且把子句 $(\neg y_i \vee \lambda_{i+2} \vee y_{i+1})$ 解释成说“如果 y_i 为真, 那么要么 λ_{i+2} 为真, 要么 y_{i+1} 为真”。

形式地, 假设赋值 T 满足 $\tau(F)$ 。我们证明 T 也满足 F 的每个子句。对短的子句这是平凡的; 如果 T 把原来的长子句的所有 k 个文字都映射成 \perp , 那么 y_i 变元就无法靠它们自身来满足所有得出的子句: 第一个这样的子句迫使 y_1 是 \top , 第二个迫使 y_2 是 \top , 最后倒数第二个子句导致 y_{k-3} 是 \top , 这与最后一个子句矛盾 (顺便说一句, 注意这恰恰是解决二元可满足性的这个实例的清洗算法)。

反之, 如果存在满足 F 的真值赋值 T , 那么 T 可被扩充成满足 $\tau(F)$ 的赋值 T' 如下: 对 F 的每个长的子句 $C = (\lambda_1 \vee \lambda_2 \vee \cdots \vee \lambda_k)$, 设 j 是满足 $T(\lambda_j) = \top$ 的最小指标 (因为假设 T 满足 F , 所以这样的 j 存在)。于是我们把新变元 y_1, \dots, y_{k-3} 的真值设置成: 若 $i \leq j-2$ 则 $T'(y_i) = \top$, 否则 $T'(y_i) = \perp$ 。容易看出, 现在 T' 满足 $\tau(F)$, 等价性证毕。 ■

最后考虑下列优化型的可满足性。

最大可满足性: 给定子句集 F 和整数 K , 是否存在满足至少 K 个子句的真值赋值?

定理 7.2.4 最大可满足性是 NP 完全的。

证明: 属于 NP 是显然的。我们把可满足性归约到最大可满足性上。

虽然这个归约是极其简单的归约, 但是它属于在证明 NP 完全性结果方面非常普通和非常有用的类型 (关于其他例子, 见习题 7.3.4)。我们只需观察到最大可满足性是对可满足性的推广, 即可满足性的每一个实例是最大可满足性的特殊类型的实例。这是真的: 可满足性的实例可被认为是参数 K 碰巧等于子句数的最大可满足性的实例。

形式地, 归约是这样的: 给定可满足性带有 m 个子句的实例 F , 通过仅仅向 F 附加容易计算的参数 $K = m$, 我们构造最大可满足性的等价实例 (F, m) 。显然存在至少满足 F 的 $K = m$ 个子句的真值赋值 (F 恰有 m 个子句) 当且仅当存在满足 F 的所有子句的真值赋值。 ■

事实上,把最大可满足性限制在最多有两个文字的子句上,也被证明是 NP 完全的(与二元可满足性比较)。

习 题

- 7.2.1 让我们考虑当 L 属于 P 时,即当在定理 7.2.1 的证明里机器 M 其实是确定型时,在从 L 到有界铺砖的归约里发生什么事情。事实上在这种情形里所得出的铺砖系统可表示成在某些关系下的集合的闭包。

设 M 是在时间 $p(n)$ 里判定语言 L 的确定型 Turing 机, $p(n)$ 是多项式; 设 x 是 M 的输入; 并设 $s, (D, H, V)$ 和 f_0 是把定理 7.2.1 的证明里的归约应用到 x 上所得出的有界铺砖的实例的成分。考虑集合 $P = \{0, 1, 2, \dots, s-1\}$ 和 $S = P \times P \times D$ 。设 $S_0 \subseteq S$ 是下列集合:

$$\{(m, 0, f_0(m)), 0 \leq m < s\}$$

设 $R_H \subseteq S \times S$ 是下列关系:

$$R\{((m-1, n, d), (m, n, d')), 1 \leq m, n < s, (d, d') \in H\}$$

以及 $R_V \subseteq S \times S$ 是这样的:

$$R\{((m, n-1, d), (m, n, d')), 0 \leq m < s, 2 \leq n < s, (d, d') \in V\}$$

证明: $x \in L$ 当且仅当对每个 $0 \leq i, j < s$ 和某个 $d \in D$, 在 R_H 和 R_V 下 S_0 的闭包包含三元组 (i, j, d) 。换句话说,不但任意的闭包性质可在多项式时间里被计算出来(这个结论在接近 1.6 节的结尾处被证明过),而且反之,任何多项式计算都可表示成闭包性质。当然,使用巨大的关系,比如 R_H ,使得这个结论似乎有点人造的感觉,但是事实上它也在更有意义得多的情形里成立(看本章结尾的文献)。

- 7.2.2 考虑二进制有界铺砖,它是有界铺砖的变种,其中给定的不是第一行的砖而是仅仅在原点上的砖 d_0 ,要铺的正方形的大小 s 用二进制给定。

(a) 证明:存在从语言

$$E_1 = \{ \langle M \rangle; M \text{ 在空字符串上在 } 2^{|\langle M \rangle|} \text{ 步内停机} \}$$

到二进制有界铺砖的归约。

(b) 证明:二进制有界铺砖不属于 P。

(c) 设 NEXP 是对某个 $k > 0$, 非确定型 Turing 机在时间 2^k 里判定的所有语言的类。证明:(i)二进制有界铺砖属于 NEXP, 以及(ii)所有 NEXP 里的语言都多项式归约到二进制有界铺砖,即二进制有界铺砖可称为是 NEXP 完全的。

- 7.2.3 (a) 证明:即使限制每个变元最多出现三次的实例上,可满足性也还是 NP 完全的。(提示:把变元比方说 x 的出现都换成新的变元 x_1, \dots, x_k 。然后添加每个新变元出现两次并且表示“所有这些变元都等价”的公式。)

(b) 如果每个变元最多出现两次,那么发生什么事情?

- 7.2.4 回忆习题 6.4.3 里 NP 类在非删除同态运算下封闭。证明 P 在非删除同态运算下封闭当且仅当 $P = NP$ 。(提示:从(a)得出一个方向。对另一个方向,考虑显然属于 P 的下列语言:

$L = \{xy; x \in \{0,1\}^* \text{ 是一个布尔公式 } F \text{ 的编码,}$
 并且 $y \in \{\top, \perp\}^* \text{ 是满足 } F \text{ 的一个赋值}\}.$

7.2.5 考虑最大可满足性的下列特殊情形:

最大二元可满足性: 给定每个子句最多有两个文字的子句集 F 以及整数 K , 是否存在满足至少 K 个子句的真值赋值?

证明: 最大二元可满足性是 NP 完全的。(这是困难的。考虑子句 $(x), (y), (z), (w), (\bar{x} \vee \bar{y}), (\bar{y} \vee \bar{z}), (\bar{z} \vee \bar{x}), (x \vee \bar{w}), (y \vee \bar{w}), (z \vee \bar{w})$ 。证明这 10 个子句的集合有下列性质: 在 x, y, z 上的所有可满足真值赋值, 除一个外都可被扩充到满足 7 个子句并且不能满足更多的子句, 剩下的一个可被扩充到满足 6 个子句。你能否利用这样的“小装置”把三元可满足性归约到最大二元可满足性上?

7.3 其他的 NP 完全问题

一旦我们证明了第一个 NP 完全问题, 就可以把它归约到其他问题, 并且从那些问题再归约到更多的问题, 证明它们都是 NP 完全的。关于对在本节和上一节证明的归约的描绘, 见图 7-4, 关于许多其他 NP 完全问题, 见习题和参考文献。

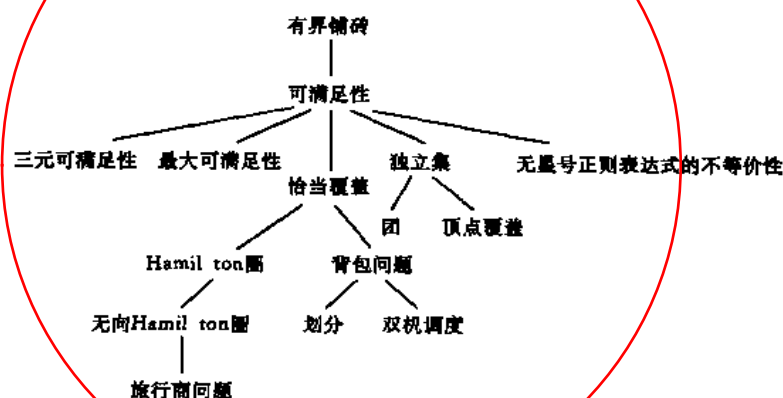


图 7-4

NP 完全问题来源于涉及复杂计算的所有领域和应用。设法认出 NP 完全问题是重要的技能——要么认出它们是已知的 NP 完全问题,^① 要么一切从头开始证明它们是 NP 完全的。这样的知识让研究人员和程序员摆脱在不可能目标上毫无结果的尝试, 把他们的努力重新定向到更有希望的路径上(关于对付 NP 完全性, 在 7.4 节里回顾)。NP 完全问题, 比如图的可着色性(见习题 7.1.1)、可满足性和独立集等是重要的, 因为它们在各种伪装下频繁地出现在应用里。像旅行商问题这样的问题是重要的, 不仅因为它们的实际用途, 而且因为它们已经被如此深入地研究过。还有其他的问题, 比如下面介绍的问题是重要

^① 这并不总是容易的, 因为 NP 完全问题往往在各种迷惑人的伪装下出现在应用里。

的,因为它们常常是 NP 完全性归约的方便的出发点 (可满足性是重要的,因为所有这三种理由)。

给定有穷集 $U = \{u_1, \dots, u_n\}$ (即论域), 以及 m 个 U 的子集的族 $\mathcal{S} = \{S_1, \dots, S_m\}$ 。问是否存在恰当覆盖, 即子族 $\mathcal{C} \subseteq \mathcal{S}$, 使得 \mathcal{C} 里的集合都是不相交的并且 U 是它们的并集。我们把这个问题称为 恰当覆盖。

例如, 设论域是 $U = \{u_1, u_2, u_3, u_4, u_5, u_6\}$, 子集族是 $\mathcal{S} = \{\{u_1, u_3\}, \{u_2, u_3, u_6\}, \{u_1, u_5\}, \{u_2, u_3, u_4\}, \{u_5, u_6\}, \{u_2, u_4\}\}$ 。在这个实例里存在恰当覆盖, 它是 $\mathcal{C} = \{\{u_1, u_3\}, \{u_5, u_6\}, \{u_2, u_4\}\}$ 。

定理 7.3.1 恰当覆盖是 NP 完全的。

证明: 恰当覆盖属于 NP 这是清楚的: 给定这个问题的实例 (U, \mathcal{S}) , 所求的子集族就是有效的证书。按照输入规模来衡量, 这个证书是多项式短的 (因为它是 \mathcal{S} 的子集, \mathcal{S} 是输入的一部分), 并且可以在多项式时间里验证是否 U 的所有元素确实在 \mathcal{C} 的集合里恰好出现一次。

为了证明 NP 完全性, 我们把可满足性归约到恰当覆盖上。即, 给定带有在布尔变元 x_1, \dots, x_n 上的子句 $\{C_1, \dots, C_l\}$ 的布尔公式 F , 我们必须证明如何在多项式时间里构造恰当覆盖的等价实例 $\tau(F)$ 。我们把子句 C_j 里的文字记作 $\lambda_{jk}, k=1, \dots, m_j$, 其中 m_j 是 C_j 里的文字数。

$\tau(F)$ 的论域是集合

$$U = \{x_i: 1 \leq i \leq n\} \cup \{C_j: j = 1, \dots, l\} \cup \{p_{jk}: 1 \leq j \leq l, k = 1, \dots, m_j\}.$$

即, 对每个布尔变元、每个子句以及每个子句里的每个位置都存在一个元素。

现在构造 \mathcal{S} 里的集合。首先对每个元素 p_{jk} , 我们在 \mathcal{S} 里有集合 $\{p_{jk}\}$ 。即 p_{jk} 都非常容易覆盖。困难在于覆盖与布尔变元和子句相对应的元素。每个变元 x_i 属于 \mathcal{S} 里的两个集合: 集合

$$T_{i,\neg} = \{x_i\} \cup \{p_{jk}: \lambda_{jk} = \overline{x_i}\},$$

它包含 x_i 的所有否定出现, 集合

$$T_{i,+} = \{x_i\} \cup \{p_{jk}: \lambda_{jk} = x_i\},$$

它包含 x_i 的所有肯定出现 (注意符号的反转)。最后每个子句 C_j 属于 m_j 个集合, 对 C_j 里的每个文字存在一个集合, 即 $\{C_j, p_{jk}\}, k=1, \dots, m_j$ 。这些是 \mathcal{S} 里的所有集合, 这就完成了对 $\tau(F)$ 的描述。

让我们说明对给定的布尔公式 F 的归约, 它带有子句 $C_1 = (x_1 \vee \overline{x_2}), C_2 = (\overline{x_1} \vee x_2 \vee x_3), C_3 = (x_2)$ 以及 $C_4 = (\overline{x_2} \vee \overline{x_3})$ 。 $\tau(F)$ 的论域是

$$U = \{x_1, x_2, x_3, C_1, C_2, C_3, C_4, p_{11}, p_{12}, p_{21}, p_{22}, p_{23}, p_{31}, p_{41}, p_{42}\}$$

并且集族 \mathcal{S} 包含下述集合:

$$\begin{aligned} & \{p_{11}\}, \{p_{12}\}, \{p_{21}\}, \{p_{22}\}, \{p_{23}\}, \{p_{31}\}, \{p_{41}\}, \{p_{42}\}, \\ & T_{1,+} = \{x_1, p_{11}\}, \\ & T_{1,\neg} = \{x_1, p_{21}\}, \\ & T_{2,+} = \{x_2, p_{22}, p_{31}\}, \end{aligned}$$

$$\begin{aligned}
T_{2,\top} &= \{x_2, p_{12}, p_{41}\}, \\
T_{3,\perp} &= \{x_3, p_{23}\}, \\
T_{3,\top} &= \{x_3, p_{42}\}, \\
\{C_1, p_{11}\}, \{C_1, p_{12}\}, \{C_2, p_{21}\}, \{C_2, p_{22}\}, \\
\{C_3, p_{23}\}, \{C_3, p_{31}\}, \{C_4, p_{41}\}, \{C_4, p_{42}\}.
\end{aligned}$$

我们断言 $\tau(F)$ 有恰当覆盖当且仅当 F 是可满足的。假设存在恰当覆盖 \mathcal{C} 。因为每个 x_i 必须被覆盖, 所以对包含 x_i 的两个集合 $T_{i,\top}$ 和 $T_{i,\perp}$, 其中恰有一个必须属于 \mathcal{C} 。认为 $T_{i,\top} \in \mathcal{C}$ 意味着 $T(x_i) = \top$, 认为 $T_{i,\perp} \in \mathcal{C}$ 意味着 $T(x_i) = \perp$, 这样定义了真值赋值 T 。我们断言 T 满足 F 。在证明里考虑子句 C_j 。与这个子句对应的 U 里的元素必须被集合 $\{C_j, p_{jk}\}$ 覆盖, 其中 $1 \leq k \leq m_j$ 。这意味着元素 p_{jk} 不属于恰当覆盖 \mathcal{C} 里的任何其他集合。特别地, 它不属于 \mathcal{C} 里包含 (否定地或肯定地) 出现在 C_j 的第 k 个文字上的变元的集合。但是这意味着 T 把 C_j 的第 k 个文字映射成 \top , 因此 C_j 被 T 满足。所以 F 是可满足的。

反之, 假设存在满足 F 的真值赋值 T 。那么我们构造下列恰当覆盖 \mathcal{C} 。对每个 x_i , 若 $T(x_i) = \top$, 则 \mathcal{C} 包含集合 $T_{i,\top}$; 若 $T(x_i) = \perp$, 则它包含集合 $T_{i,\perp}$ 。另外对每个子句 C_j , \mathcal{C} 包含集合 $\{C_j, p_{jk}\}$, 使得 C_j 的第 k 个文字被 T 映射到 \top , 因此 p_{jk} 不属于迄今为止在 \mathcal{C} 里所选择的任何集合。根据 T 满足 F 的假设, 我们知道存在这样的 k 。最后在覆盖了所有 x_i 和 C_j 之后, \mathcal{C} 包含足够的单元元素集, 去覆盖任何剩下没有被其他集合覆盖的 p_{jk} 元素。在上述说明里, 对应于满足的真值赋值 $T(x_1) = \top, T(x_2) = \top, T(x_3) = \perp$, 恰当覆盖包含这些集合: $T_{1,\top}, T_{2,\top}, T_{3,\perp}, \{C_1, p_{11}\}, \{C_2, p_{22}\}, \{C_3, p_{31}\}, \{C_4, p_{42}\}$, 以及单元集 $\{p_{12}\}, \{p_{21}\}, \{p_{23}\}, \{p_{41}\}$ 。我们得出结论说 $\tau(F)$ 有恰当覆盖, 证毕。 ■

7.3.1 旅行商问题

我们利用恰当覆盖去证明 Hamilton 圈的 NP 完全性。

定理 7.3.2 Hamilton 圈是 NP 完全的。

证明: 我们已知它属于 NP。现在我们证明如何把恰当覆盖归约到 Hamilton 圈上。我们描述多项式时间算法, 当给定恰当覆盖的实例 (U, \mathcal{S}) 时, 这个算法产生有向图 $G = \tau(U, \mathcal{S})$ 使得 G 有 Hamilton 圈当且仅当 (U, \mathcal{S}) 有恰当覆盖。

构造是基于具有关于 Hamilton 圈的有趣性质的某种简单图, 在 NP 完全性的术语里, 这样的图被称为小装置。图 7-5(a) 显示这种小装置。设想它是更大的图 G 的一部分, 通过画成实心圆点的四个顶点, 它连接到 G 的其余部分。换句话说, 虽然存在其他的顶点和边通向这个图, 但是除所画出的边外没有任何别的边与中间的三个顶点相关联。另外假设 G 有 Hamilton 圈, 即经过 G 的每个顶点恰好一次的圈。问题是这条 Hamilton 圈用哪些边经过中间这三个顶点? 容易看出其实只有两种可能性: 要么边 $(a, u), (u, v), (v, w), (w, b)$ 属于 Hamilton 圈, 要么边 $(c, w), (w, v), (v, u), (u, d)$ 属于 Hamilton 圈。所有其他的可能性都剩下这三个顶点中的一两个没有被 Hamilton 圈经过, 因此这个圈根本不是 Hamilton 圈。

换一种方式说, 这个简单装置可被认为是两条边 (a, b) 和 (c, d) , 它们带有下列附加的约束条件: 在整个图 G 的任何 Hamilton 圈里, 要么 (a, b) 被经过, 要么 (c, d) 被经过, 但是不

能二者都被经过。这种情况如图 7-5(b)所示,其中两条本来无关的边被异或符号所连接,这意味着其中恰有一条边要被任何 Hamilton 圈所经过。每当我们在此图的描绘里画这个构造时,我们知道事实上这个图包含着在图 7-5(a)里显示的整个子图。事实上我们可利用这种子图把多条边连接到同一条边上(如图 7-5(c)所示)。结果是同样的:要么在一侧的所有边都被经过,要么在另一侧的那条边被经过,但是不能两侧的边都被经过。

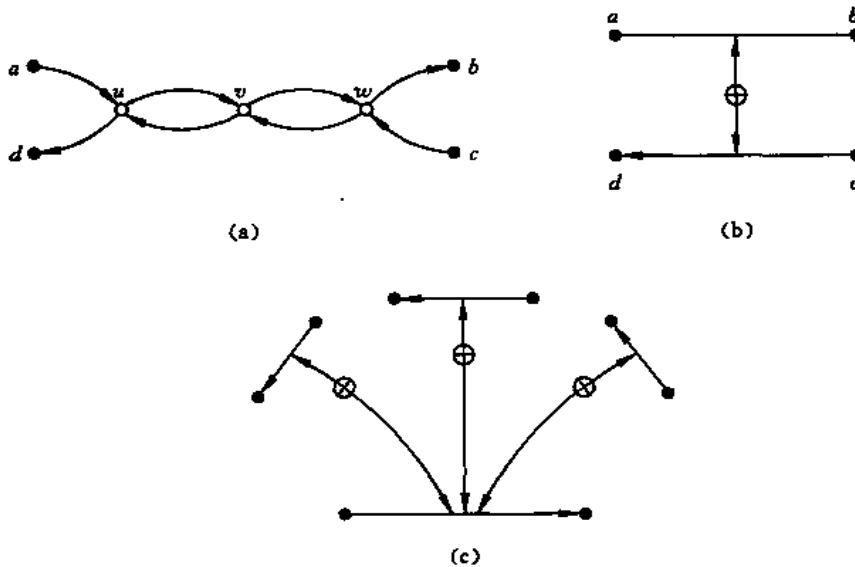


图 7-5

这种装置是我们对图 $G = \tau(U, \mathcal{S})$ 的构造的核心, G 对应于恰当覆盖的实例, 其中 $U = \{u_1, \dots, u_n\}$ 以及 $\mathcal{S} = \{S_1, \dots, S_m\}$ 。下一步我们描述图 G 。 G 有顶点 u_0, u_1, \dots, u_n 和 S_0, S_1, \dots, S_m , 即对论域里的每个元素和给定实例里的每个集合都存在一个顶点, 外加两个顶点。对 $i = 1, \dots, m$, 存在两条边 (S_{i-1}, S_i) 。当然在图里用两条不同的边连接同样一对顶点是毫无意义的。在目前情形里我们允许它的唯一理由是这些边随后会被“异或”子图连接, 像在图 7-6 里那样, 因此当构造完成时就不会存在任何“平行”边。在两条边 (S_{i-1}, S_i) 里, 一条称为长的边, 另一条称为短的边。对 $j = 1, \dots, n$, 在顶点 u_{j-1} 和 u_j 之间存在的边数, 与包含元素 u_j 的 \mathcal{S} 里的集合一样多。通过这种方式我们可认为边 (u_{j-1}, u_j) 的每个复制品对应于 u_j 在集合里的一次出现。最后我们添加边 (u_n, S_0) 和 (S_m, u_0) , 因此就“封闭了圈”。

注意迄今为止的构造仅仅依赖于论域的规模以及集合的个数和规模, 它不依赖于究竟哪些集合正好包含了每个元素。实例的这种精细结构影响我们的构造如下: 由于边 (u_{j-1}, u_j) 的每个复制品对应着 u_j 在某个集合 $S_i \in \mathcal{S}$ 里的一次出现, 其中 $u_j \in S_i$, 我们利用“异或”子图, 把边 (u_{j-1}, u_j) 的这个复制品与长的边 (S_{i-1}, S_i) 相连接(对这种连接的说明, 见图 7-6)。这样就完成了对图 $\tau(U, \mathcal{S})$ 的构造。

我们断言图 $\tau(U, \mathcal{S})$ 有 Hamilton 圈当且仅当 (U, \mathcal{S}) 有恰当覆盖。 首先假设存在 Hamilton 圈。它必须按照顺序 S_0, S_1, \dots, S_m 经过与这些集合对应的顶点, 然后经过边 (S_m, u_0) , 然后经过顶点 u_0, u_1, \dots, u_n , 最后沿着边 (u_n, S_0) 结束(如图 7-6 所示)。问题在于

对每个集合 S_j , Hamilton 圈是经过短的还是长的边 (S_{j-1}, S_j) ? 设 \mathcal{C} 是使短的边 (S_{j-1}, S_j) 被 Hamilton 圈经过的所有集合 T_j 的集合。我们证明 \mathcal{C} 是合法的集合覆盖, 即它包含所有元素并且没有重叠。不难看出: 根据“异或”子图的性质, \mathcal{C} 里的集合里的元素都恰恰是形如 (u_{i-1}, u_i) 的边的复制品, 这些边都被 Hamilton 圈经过; 对每个元素 $u_j \in U$, Hamilton 圈恰好经过一条这样的边。由此得出每个元素 $u_j \in U$ 恰好被包含在一个属于 \mathcal{C} 的集合里, 因此 \mathcal{C} 是恰当覆盖。

反之, 假设存在恰当覆盖 \mathcal{C} 。于是在图 $\tau(U, \mathcal{C})$ 里的 Hamilton 圈可如下构造: 对所有 $S_j \in \mathcal{C}$, 经过边 (S_{j-1}, S_j) 的短的复制品, 对所有其他集合, 经过长的边。然后对每个元素 u_i , 经过边 (u_{i-1}, u_i) 的复制品, 它对应于包含 u_i 的 \mathcal{C} 里的唯一集合。用边 (u_n, S_0) 和 (S_m, u_0) 完成 Hamilton 圈。

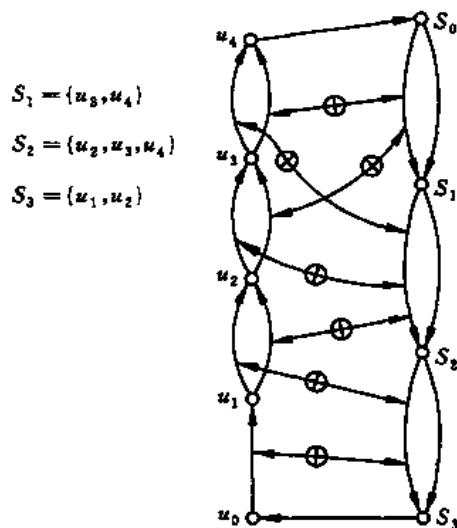


图 7-6

一旦问题被证明是 NP 完全的, 研究常常就集中在解决问题的令人感兴趣并且易解的特殊情形上。NP 完全性证明常常产生目标问题的复杂的“不自然的”实例。在实践中令人感兴趣的简单得多的实例是否也不能被某些有效算法所解决, 这样的问题常常困扰着我们。在另一方面, 常常可证明即使受到极大限制的问题也仍然是 NP 完全的。我们在可满足性上观察到了这两种模式, 特殊情形二元可满足性可在多项式时间里解决, 但是特殊情形三元可满足性是 NP 完全的。为了介绍 Hamilton 圈的有趣的特殊情形, 定义无向 Hamilton 圈是把 Hamilton 圈限制到无向图上, 即没有自环边的对称图。

定理 7.3.3 无向 Hamilton 圈是 NP 完全的。

证明: 我们把普通的 Hamilton 圈归约到它。给定图 $G \subseteq V \times V$, 我们构造没有自环边的对称图 $G' \subseteq V' \times V'$, 使得 G 有 Hamilton 圈当且仅当 G' 也有。在图 7-7 里说明的构造是这样的: 首先 $V' = \{v_0, v_1, v_2; v \in V\}$, 即对 G 的每个顶点 v , G' 有三个顶点 v_0, v_1 和 v_2 。非形式化地说, 其中 v_0 是入口顶点, 进入 v 的边都指向它, 而 v_2 是出口顶点, 从 v 出发的边都从它发出。

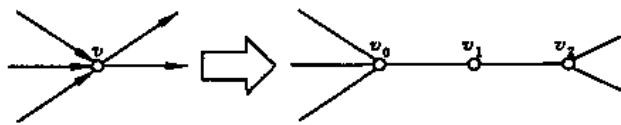


图 7-7

因此 G' 里的边是这些 (见图 7-7; 回忆通过无向线段连接顶点可更方便地描绘无向图):

$$\{(u_2, v_0), (v_0, u_2); (u, v) \in G\} \cup \{(v_0, v_1), (v_1, v_0), (v_1, v_2), (v_2, v_1); v \in V\}.$$

即按照 v_0, v_1, v_2 的顺序用通路连接这三个顶点, 并且每当 $(u, v) \in G$ 时, 在 u_2 和 v_0 之间就存在无向边。这样就完成了对 G' 的构造。

现在我们必须证明 G' 有 Hamilton 圈当且仅当 G 也有。假设 G' 的 Hamilton 圈从形如 (u_2, v_0) 的边到达顶点 v_0 。如果这条 Hamilton 圈通过不是 (v_0, v_1) 的边离开顶点 v_0 , 那么它不能以任何其他方式“带上”顶点 v_1 , 所以假设它是 Hamilton 圈就是错误的。因此边 (v_0, v_1) 必须属于这条圈, 而且对 (v_1, v_2) 也是同样的。于是这条圈必须向前通过边 (v_2, w_0) , 其中 $(v, w) \in G$, 从这里到 w_1, w_2 , 到某个 z_0 , 其中 $(w, z) \in G$, 等等。因此在 G' 的这条 Hamilton 圈里形如 (u_0, v_2) 的边事实上组成了 G 的 Hamilton 圈。反之, G 的任何 Hamilton 圈 $(v^1, v^2, \dots, v^{|V|})$, 可如下地转换成 G' 的 Hamilton 圈: $(v_0^1, v_1^1, v_2^1, v_0^2, v_1^2, v_2^2, \dots, v_0^{|V|}, v_1^{|V|}, v_2^{|V|})$ 。我们得出结论说, G 有 Hamilton 圈当且仅当 G' 有 Hamilton 圈, 证毕。 ■

我们的下一个结果涉及出了名的旅行商问题——它的“是否”型, 像在 6.2 节里定义的那样, 每个实例附带预算 B 。

定理 7.3.4 旅行商问题是 NP 完全的。

证明: 我们已知问题属于 NP。为了证明完全性, 我们把无向 Hamilton 圈归约到旅行商问题上。给定对称图 G , 其中不失一般性 $V = \{v_1, \dots, v_{|V|}\}$, 构造旅行商问题的下列实例: 城市数 n 是 $|V|$, 任何两个城市 i 和 j 之间的距离 d_{ij} 是

$$d_{ij} = \begin{cases} 0 & \text{若 } i = j; \\ 1 & \text{若 } (v_i, v_j) \in G; \\ 2 & \text{否则。} \end{cases}$$

因为 G 是无环的对称图, 所以这个距离函数本身是对称的, 即对所有城市 i 和 j , 像所需要的那样 $d_{ij} = d_{ji}$ 。最后, 预算为 $B = n$ 。

显然这些城市的任何巡回路线的成本等于数字 n 加上所经过的相邻城市之间不是 G 的边的个数。因此存在成本等于或小于 B 的巡回路线当且仅当它使用的“非边”的个数为零, 即当且仅当巡回路线是 G 的 Hamilton 圈。 ■

7.3.2 划分与团

恰当覆盖也提供了对划分的 NP 完全性的良好证明。最容易的莫过于从密切相关的背包问题开始, 其中给定要达到的任意的和 K (回忆例 7.1.2 里它的定义)。

定理 7.3.5 背包问题是 NP 完全的。

证明: 背包问题属于 NP 这是清楚的: 给定背包问题的实例 a_1, \dots, a_n 和 K , 满足 $\sum_{i \in P} a_i = K$ 的 $\{1, \dots, n\}$ 的子集 P 可作为对给定实例的回答是“是”的证书。它是多项式短的, 并且它可在多项式时间里通过二进制加法来验证。

现在我们把恰当覆盖归约到背包问题上。给定论域 $U = \{u_1, \dots, u_n\}$ 以及 U 的子集族 $\mathcal{S} = \{S_1, \dots, S_m\}$ 。我们构造背包问题的实例 $\tau(U, \mathcal{S})$, 即非负整数 a_1, \dots, a_n 和 K , 使得存在满足 $\sum_{i \in P} a_i = K$ 的子集 $P \subseteq \{1, \dots, n\}$ 当且仅当存在子集的集合 $\mathcal{C} \subseteq \mathcal{S}$, 这些子集都是不相交的并且合起来覆盖全部 U 。

这个构造特别简单,因为它是基于在集合并运算与整数加法运算之间的出人意料的
关系。 n 元集合的子集,比如 \mathcal{S} 里的那些集合,可表示成 $\{0,1\}^n$ 上的字符串(如图 7-8 所
示)。然后这些字符串可解释成写成二进制的在 0 和 2^n-1 之间的整数。现在在这样的集
合都不相交的前提下求它们的并集就等于把对应的整数相加。因为在恰当覆盖里我们问
是否有一些不相交的集合的并组成了整个的 U ,这似乎等于问是否在给定的整数里有一
些整数加起来等于 $K=1+2+4+\cdots+2^{n-1}$ ——即只包含 n 个 1 的二进制整数。这非常接
近于背包问题的实例了。

$S_1 = \{u_3, u_4\}$	$S_1 = 0011$	0011 ✓
$S_2 = \{u_2, u_3, u_4\}$	$S_2 = 0111$	0111
$S_3 = \{u_1, u_2\}$	$S_3 = 1100$	+ 1100 ✓
		1111

(a)
(b)
(c)

图 7-8 从集合(a)到位向量(b)到 m 进制整数加法(c)

在这个简单的归约里存在一个问题:在集合并与整数加之间的密切对应关系有时不
成立,因为在整数加法里我们可能有进位。例如考虑和 $11+13+15+24=63$,用二进制表
示就是 $001011+001101+001111+011000=111111$ 。若我们往回翻译出 $\{u_1, \dots, u_6\}$ 的子
集,即集合 $\{u_3, u_5, u_6\}$, $\{u_3, u_4, u_6\}$, $\{u_3, u_4, u_5, u_6\}$ 和 $\{u_2, u_3\}$,则它们既非不相交也非覆盖
全部 U 。换句话说,进位使得在并与加之间的翻译出了差错。

这个问题可非常容易地解决如下,不再认为 $\{0,1\}^n$ 里的字符串是二进制整数,而认
为它们是 m 进制整数,其中 m 是 \mathcal{S} 里的集合个数。即我们有 m 个整数 a_1, \dots, a_n ,其中 a_i
 $= \sum_{u_j \in S_i} m^{j-1}$ 。我们问是否存在加起来是 $K = \sum_{j=1}^n m^{j-1}$ 的子集。通过这种方式加法就
不成问题了,因为在每个数字不是 0 就是 1 的前提下,少于 m 个数字相加是永不产生进
位的。现在清楚的是,得出的背包问题的实例有解当且仅当原来的恰当覆盖的实例有解。

推论 划分和双机调度都是 NP 完全的。

证明: 从背包问题到这两个问题的归约都是存在的,回忆例 7.1.2。

下一步我们转向在 6.2 节里介绍过的三个图论问题:独立集,团和顶点覆盖。

定理 7.3.6 独立集是 NP 完全的。

证明: 它显然属于 NP,我们把三元可满足性归约到它。

假设给定带有子句 C_1, \dots, C_m ,每个子句最多有三个文字的布尔公式 F 。事实上,假设
 F 的所有子句都恰好有三个文字;如果子句只有一个或两个文字,那么我们允许重复文字
使得文字总数达到三个。构造无向图 G 和整数 K ,使得在 G 里存在 K 个顶点并且在它们
之间没有边当且仅当 F 是可满足的。

归约在图 7-9 里说明。对于 F 的每个子句,我们在 G 里有用边连接成三角形的三个
顶点,与子句 C_i 对应的三角形的顶点称为 c_{i1}, c_{i2}, c_{i3} 。这些顶点就是 G 的所有顶点,总共有
 $3m$ 个。目标是 $K=m$,它等于子句数。为定义 G 的剩下的边,顶点 c_{ij} 等同于子句 C_i 的第 j
个文字。最后,两个顶点用边连接当且仅当它们的文字互为否定。这样就完成了对归约的

描述,图 7-9 给出了一个例子。

假设存在 G 的有 $K=m$ 个顶点的独立集 I 。因为在同一个三角形里任何两个顶点都用边连接,所以显然在每个三角形里恰有一个顶点属于 I 。回忆一下,顶点都对应着文字。现在考虑下述事实:顶点属于 I 意味着它对应的文字是 \top 。因为在 I 里的顶点之间不存在任何边,所以由此得出任何两个这样的文字都不是互为否定的,因此它们可以作为真值赋值 T 的基础。注意 T 可能没有在所有变元上被完整地定义出来,因为 I 里的顶点的集合可能没有涉及到所有的变元。例如在图 7-9 里,用实心圆圈表示的独立集没有决定变元 x_3 的值。 T 在这样的“丢失的”变元上可取任意真值。所得出的真值赋值 T 满足所有子句,因为每个子句至少有一个变元被 T 满足。反之,给定满足 F 的任意真值赋值 T ,通过为每个子句挑选对应着被满足的文字的顶点,我们可获得规模为 m 的独立集。 ■

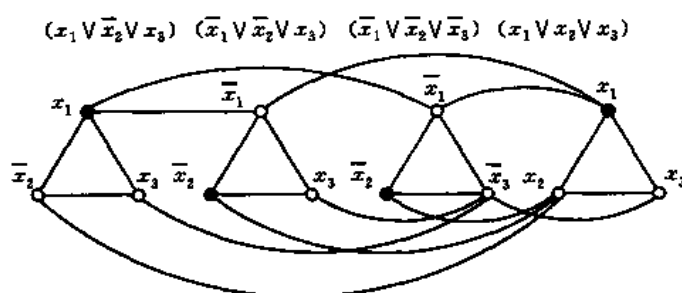


图 7-9

其他两个图论问题的 NP 完全性现在唾手可得。

定理 7.3.7 团和顶点覆盖都是 NP 完全的。

证明: 它们都显然属于 NP。

团要求在集合里任何两个顶点之间都有边,在某种意义上它与独立集恰好相反。归约使这种意义精确化。给定独立集的实例 (G, K) , 其中 $G \subseteq V \times V$ 是无向图并且 $K \geq 2$ 是目标,只要取 $G' = V \times V - \{(i, i) : i \in V\} - G$, 并保持同样的目标 $K' = K$, 就构造出团的等价实例 (G', K') 。这是可行的,因为(相当容易验证) G 的最大独立集恰好是 G 的补图里的最大团, G 的补图是由所有不属于 G 的非环边所组成的图(见图 7-10)。

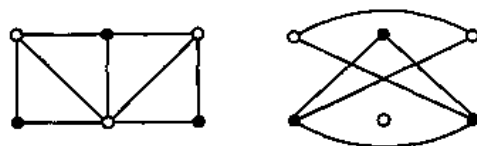


图 7-10

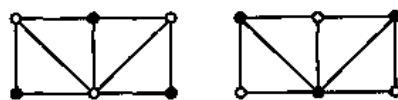


图 7-11

最后,在另一种不同的意义下顶点覆盖与独立集恰好相反:因为在顶点覆盖 $N \subseteq V$ 里的顶点合起来“击中”所有边,所以集合 $V - N$ 中的顶点之间必然没有边,因此是独立集(见图 7-11)。因此, $N \subseteq V$ 是 G 的顶点覆盖当且仅当 $V - N$ 是 G 的独立集。所以, G 的最大独立集有规模 K 当且仅当 G 的最小顶点覆盖有规模 $|V| - K$ 或更小。从独立集到顶点覆盖的归约让图保持原样,仅仅用 $|V| - K$ 代替 K 。 ■

7.3.3 有穷自动机

我们最后的 NP 完全性结果涉及到在本书里研究的某些最初级的并且显然是最简单的数学对象:非确定型有穷自动机和正则表达式。

在上一章里我们介绍的所有问题中间,只有两个问题的 NP 成员性不是明显的:正则表达式的等价性,以及密切相关的非确定型有穷自动机的等价性。给定两个正则表达式,什么是令人信服的它们等价性的证书?想不出任何短的证书。

不过,如果我们定义它的补问题

正则表达式的不等价性: 给定两个正则表达式 R_1 和 R_2 , 是否 $L(R_1) \neq L(R_2)$?

那么似乎可能有证书了:对两个正则表达式的不等价性的证书是属于其中一个表达式产生的语言但不属于另一个表达式产生的语言的字符串,即它是 $(L(R_1) - L(R_2)) \cup (L(R_2) - L(R_1))$ 里的任何元素。确实,这个集合是非空的当且仅当这两个表达式是不等价的。

但是,现在真正的困难出现了:这样的证书在几乎所有方面都是合法的,除了关键的多项式简短性性质。给定两个正则表达式 R_1 和 R_2 , 对属于 $(L(R_1) - L(R_2)) \cup (L(R_2) - L(R_1))$ 的最短字符串的长度,没有任何明显的多项式上界。合格的界限应当是这两个表达式的长度的多项式,这两个表达式的长度是 $|R_1| + |R_2|$, 即为表示它们而需要的符号数,符号是比如 $a, b, \cup, *$ 和圆括号等。事实上存在着许多对不等价的正则表达式,它们只在表达式规模的指数那样长的字符串上才是不同的!

为了获得 NP 问题,我们必须查看受到限制的特殊情形:**无星号正则表达式**,即在并运算和连接运算上的正则表达式,其中不出现 Kleene 星号运算。考虑无星号正则表达式,比如

$$R = (0 \cup 1)00(0 \cup 1) \cup 010(0 \cup 1)0$$

现在容易看出,如果 x 是它产生的语言里的字符串(比方说对上面的 $R, x = 1001$), 那么 $|x| \leq |R|$ 。作为这种观察的结果,下列问题属于 NP:

无星号正则表达式的不等价性: 给定两个无星号正则表达式 R_1 和 R_2 , 是否 $L(R_1) \neq L(R_2)$?

有效证书是 $(L(R_1) - L(R_2)) \cup (L(R_2) - L(R_1))$ 里的任何字符串,并且所有这样的字符串都是短的,比 $|R_1| + |R_2|$ 更短。对于在非确定型有穷自动机领域里的类似问题,见习题 7.3.7。

事实上,我们可以证明下述结果:

定理 7.3.8 无星号正则表达式的不等价性是 NP 完全的。

证明: 我们已经论证问题属于 NP。现在我们把可满足性归约到无星号正则表达式的不等价性。

给定带有布尔变元 x_1, \dots, x_n 和子句 C_1, \dots, C_m 的任意布尔公式,我们产生两个在字母表 $\Sigma = \{0, 1\}$ 上的正则表达式 R_1 和 R_2 , 它们都没有应用 Kleene 星号运算,使得 $L(R_1) \neq L(R_2)$ 当且仅当给定的布尔公式是可满足的。

第二个正则表达式 R_2 是非常简单的:

$$(0 \cup 1)(0 \cup 1) \cdots (0 \cup 1)$$

其中把表达式 $(0 \cup 1)$ 重复 n 次。 R_2 产生的语言显然是所有长度为 n 的字符串的集合,即

$$L(R_2) = \{0, 1\}^n.$$

现在构造 R_1 。与 R_2 对照, R_1 严重地依赖于给定的布尔公式。具体地, R_1 是 m 个正则表达式的并:

$$R_1 = a_1 \cup a_2 \cup \dots \cup a_m$$

其中正则表达式 a_i 依赖于子句 C_i 。每个 a_i 是 n 个正则表达式的连接:

$$a_i = a_{i1} a_{i2} \dots a_{in}$$

其中

$$a_{ij} = \begin{cases} 0 & \text{若 } x_j \text{ 是 } C_i \text{ 的一个文字;} \\ 1 & \text{若 } \bar{x}_j \text{ 是 } C_i \text{ 的一个文字;} \\ (0 \cup 1) & \text{否则。} \end{cases}$$

假如我们暂时放弃在 0-1 和 \top - \perp 之间的区别, 那么可认为 $\{0, 1\}^n$ 里的字符串是对布尔变元 $\{x_1, \dots, x_n\}$ 的真值赋值。按照这种解释, $L(a_i)$ 恰恰是不满足 C_i 的所有真值赋值的集合。因此 $L(R_1)$ 是至少不满足给定布尔公式中一个子句的所有真值赋值的集合。因此给定的布尔公式是可满足的当且仅当 $L(R_1)$ 与 $\{0, 1\}^n$ ——它恰恰是 $L(R_2)$ ——不同。证毕。 ■

有关一般正则表达式和非确定型有穷自动机的等价性问题当然只能是更难(关于它们准确的复杂性的信息, 见参考文献), 有关非确定型有穷自动机的状态极小化问题是类似的。除非 $P=NP$, 否则这些陈述起来简单的关于最初级计算模型的问题都不能被有效解决。

但是关于下列雄心较少的目标情况如何呢? 假设给定非确定型有穷自动机, 我们希望找出等价的状态数最少的确定型有穷自动机。我们知道任何这样的算法在最坏情形里必然是指数的, 因为输出可能不得不是输入规模的指数那么长的(回忆例 2.5.4)。但是有没有运行时间是输入和输出规模的多项式的算法? 虽然当输出很大时这样的算法花费指数时间, 但是它快速地输出小的自动机。(首先完成子集构造法然后极小化所得出的确定型自动机的明显算法是行不通的, 因为即使最终输出——极小等价确定型自动机——是多项式的, 子集构造法还是可以产生指数大小的中间结果。)

不幸的是, 我们证明即使这样的算法也不像是存在的。

推论: 除非 $P=NP$, 否则不存在这样的算法: 给定正则表达式或非确定型有穷自动机, 在输入和输出规模的多项式时间里构造状态最少的等价确定型有穷自动机。

证明: 设 M_n 表示接受 $\{0, 1\}^n$ 的简单的 $n+1$ 个状态的有穷自动机。在定理的证明中的归约里, 给定的带有 n 个变元的布尔公式是不可满足的当且仅当等价于 R_1 的状态最少的确定型有穷自动机恰恰是 M_n 。

现在假设存在像在推论的陈述里描述的那样的算法, 它带有形如 $p(|x| + |y|)$ 的时间界限, 其中 p 是多项式, x 是它的输入, y 是它的输出。那么我们可以解决可满足性如下。

给定带有 n 个变元的任意布尔公式 F , 我们首先完成在定理的证明里描述的归约以获得正则表达式 R_1 。然后在输入 R_1 上运行声称的算法到 $p(|R_1| + |M_n|)$ 步, 其中 $|M_n|$ 是 M_n 的编码的长度。如果算法在分配的时间里终止, 那么我们知道如何回答原来的可满足性问题: 若输出是 M_n , 则我们回答“否”; 在任何其他输出的情况下, 我们回答“是”。不过若算法在 $p(|R_1| + |M_n|)$ 步之后不停机, 则因为我们知道它总是在 $p(|x| + |y|)$ 步之后停

机,所以我们得出结论说它的输出比 $|M_n|$ 更大。因此算法的输出不是 M_n ,我们对原来的可满足性问题自信地回答“是”。

上面描述的算法是有关可满足性的多项式时间算法——时间界限 $p(|R_1| + |M|)$ 是布尔公式 F 的规模的多项式。因为可满足性是 NP 完全的,所以我们必须根据定理 7.1.1 得出结论说 $P=NP$,这样就完成了证明。 ■

习 题

- 7.3.1 (a) 证明即使所有集合都有不超过 3 个元素,并且每个元素最多出现在 3 个集合里,恰当覆盖也仍然是 NP 完全的。
 (b) 如果其中一个数字是 2,那么发生什么事情?
- 7.3.2 如果恰当覆盖的给定实例包括论域 $\{u_1, u_2\}$ 和集族 $\mathcal{S} = \{\{u_1\}, \{u_1, u_2\}\}$,给出从恰当覆盖到 Hamilton 圈的归约产生的完整的图(没有对异或小装置的缩写)。
- 7.3.3 (a) 证明 Hamilton 通路是 NP 完全的。(1)通过把 Hamilton 圈问题归约到它上;
 (2)通过稍微修改定理 7.3.2 的证明里的构造。
 (b) 对在两个指定顶点之间的 Hamilton 通路问题(定义是显然的)重复上一问。
- 7.3.4 下列每个问题都是一个 NP 完全问题的推广,因此也是 NP 完全的。即如果以某种方式固定问题的某些参数,那么所处理的问题成为已知的 NP 完全问题(回忆定理 7.2.4 的证明)。通过仅仅引入新的参数,并且让实例的其余部分保持原样,你就可把任何问题归约到它的推广上。

对下面每个问题,通过证明它是某个 NP 完全问题的推广来证明它是 NP 完全的。在每个情形里给出适当的参数限制。

- (a) 最长回路:给定图和整数 K ,是否存在没有重复顶点并且长度至少是 K 的回路?(提示:如果限制 K 等于图的顶点数,那么发生什么事情?)
- (b) 子图同构:给定两个无向图 G 和 H , G 是 H 的子图吗?(即,如果 G 有顶点 v_1, \dots, v_n ,你能否找到 H 里的不同顶点 u_1, \dots, u_n ,使得每当 $[v_i, v_j]$ 是 G 里的边时, $[u_i, u_j]$ 就是 H 里的边?)
- (c) 导出子图同构:给定两个无向图 G 和 H , G 是 H 的导出子图吗?(即,如果 G 有顶点 v_1, \dots, v_n ,你能找到 H 里的不同顶点 u_1, \dots, u_n ,使得 $[u_i, u_j]$ 是 H 里的边当且仅当 $[v_i, v_j]$ 是 G 里的边?)
- (d) 可圈图:给定带有顶点 v_1, \dots, v_n 的无向图 G ,自然数的 $n \times n$ 对称矩阵 R_{ij} ,以及整数 B ,是否存在带有下列性质的 G 的 B 条边的集合 S :在每一对顶点 v_i 和 v_j ($v_i \neq v_j$)之间,至少存在 R_{ij} 条不相交的由 S 里的边组成的通路(即除端点外没有其他公共顶点的通路)。(提示:如果对所有 $i, j, R_{ij} = 2$,并且 $B = n$,那么发生什么事情?)
- (e) 整数规划:给定关于 n 个变元的 m 个方程

$$\sum_{j=1}^n a_{ij}x_j = b_i, i = 1, \dots, m$$

系数 a_{ij} 和 b_i 都是整数,它是否有所有 x_j 为 0 或 1 的解?(其实,这是对我们见

过的许多 NP 完全问题的共同的推广,你能找到其中的多少个?)

(f) **出租车乘客**:给定边带有正长度 d_{ij} 的有向图 G ,两个顶点 1 和 n 以及整数 K ,是否存在从 1 到 n 的任何顶点不重复两次并且总长度至少为 K 的通路?

(g) **命中集**:给定集族 $\{S_1, S_2, \dots, S_n\}$ 以及整数 B ,是否存在至多包含 B 个元素的集合 H ,使得 H 与族里的所有集合都相交?

(h) **装箱**:给定正整数的集合 $A = \{a_1, \dots, a_n\}$ 以及另外两个整数 B 和 K ,是否 A 里的整数可划分成 B 个子集合(“箱子”)使得在每个箱子里的数之和都等于或小于 K ?

(i) **集合覆盖**:给定论域 U 的子集族 \mathcal{S} 以及整数 K ,是否存在 \mathcal{S} 里的 K 个集合,它们的并集等于 U ?

7.3.5 证明:即使所寻找的独立集的规模 K 等于 $\lceil n/2 \rceil$,其中 n 是顶点数,独立集也是 NP 完全的。

7.3.6 证明下列问题是 NP 完全的。**支配集**:给定有向图 G 和整数 B ,是否存在 B 个 G 的顶点的集合 S ,使得对每个 G 的顶点 $u \notin S$,存在顶点 $v \in S$ 使得 (v, u) 是 G 的边。

7.3.7 对非确定型有穷自动机 $M = (K, \Sigma, \Delta, s, F)$ 来说,若不存在状态 q 和字符串 $w \neq \epsilon$ 使得 $(q, w) \vdash_M^* (q, \epsilon)$,则 M 称为无圈的。证明辨别两台无圈的非确定型有穷自动机是否不等价的问题是 NP 完全的。

7.4 对付 NP 完全性

问题不会在它们被证明是 NP 完全的时候就消失掉。但是,一旦我们知道所关注的问题是 NP 完全问题,就更愿意降低标准去采用非最优解,去采用不总是多项式的或者不在所有可能实例上工作的算法。在本节里我们回顾某些最有用的这类技术。

7.4.1 特殊情形

一旦我们的问题被证明是 NP 完全的,首先要问的问题就是这样的:我们是否真正需要在完全一般性的情况下解决这个问题?在这种完全一般性的情况下这个问题被定义出来,并且被证明是 NP 完全的。NP 完全性归约经常产生问题的不自然的复杂的实例。也许真正需要解决的是问题的更容易解决的特殊情形。

例如,我们看到对可满足性存在重要的可被有效轻易地解决的特殊情形:二元可满足性(回忆 6.3 节)。如果所有必须解决的可满足性的实例都具有这类子句,那么一般的可满足性问题是 NP 完全的这个事实就无关紧要。但是,常常我们关注的特殊情形本身也是 NP 完全的,例如三元可满足性就是这样的情形,回忆定理 7.2.3。下面我们看另一个例子。

例 7.4.1 当无向图是树时,即它无回路(如图 7-12 所示),大多数关于这个无向图的问题变得容易解决。回顾我们收集的 NP 完全的图论问题,Hamilton 圈在树的情形里当然是平凡的(树没有圈,无论是 Hamilton 圈还是别的什么圈)。Hamilton 通路也是这样的。树有 Hamilton 通路仅当它是一条 Hamilton 通路。团也变成平凡的,树没有超过两个顶点的团。

当图是树时独立集也是容易解决的。用来解决它的方法是利用树的“分层结构”。常

规的有用作法是挑选树里的任意一个顶点并指定它作为根(如图 7-12 所示);一旦这样做过之后,树里的每个顶点 u 自身就变成子树 $T(u)$ 的根。子树 $T(u)$ 是到根的(唯一)通路经过 u 的所有顶点组成的集合,见图 7-12。于是问题可自底向上地解决,从树叶(包含一个顶点的子树)到越来越大的子树,直到整棵树(根的子树)被处理完为止。对每个顶点 u ,可定义它的儿子的集合 $C(u)$ ——在 u 的子树里与它相邻的顶点,不包括 u 自己——以及它的孙子的集合 $G(u)$ ——它的儿子的儿子。自然,这些集合可能为空。例如在图 7-12 里用 r 表示的根具有两个儿子和五个孙子。没有儿子的顶点称为树叶。

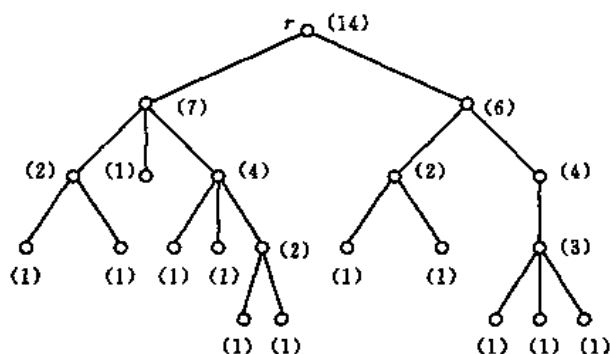


图 7-12

现在通过对每个顶点 u 计算数值 $I(u)$ 来求出树的最大独立集的规模,其中 $I(u)$ 是 $T(u)$ 的最大独立集的规模。显然下列等式成立:

$$I(u) = \max \left\{ \sum_{v \in C(u)} I(v), 1 + \sum_{v \in G(u)} I(v) \right\} \quad (2)$$

这个等式说在确定 $T(u)$ 的最大独立集的过程里我们拥有两种选择:要么(这是 \max 里的第一项)不把 u 放进独立集,这时可把 u 的儿子们的子树的最大独立集合并到一起,要么(这是 \max 里的第二项)把 u 放进独立集,这时必须删去 u 的儿子们,而把它的孙子们的子树的最大独立集合并到一起。

现在容易看出,动态规划算法可在多项式时间里解决在树的特殊情形里的独立集。算法开始于树叶上(在这里 $I(u)$ 平凡地是 1),对越来越大的子树计算 $I(u)$ 。在根上的 I 值就是树的最大独立集的规模。算法是多项式的,因为对每个顶点 u ,只需花费线性时间去计算(2)里的表达式。例如在图 7-12 里在括号里标出了 I 的值。树的最大独立集的规模是 14。

不言而喻,用同样方法也可解决密切相关的顶点覆盖(回忆在顶点覆盖与独立集之间的归约)。因此,如果我们关注的图碰巧是树,那么顶点覆盖与独立集都是 NP 完全的这个事实就无关紧要。当专门研究树时可利用类似算法去解决许多其他与图有关的 NP 完全问题,例如习题 7.4.1。◇

7.4.2 近似算法

当面对 NP 完全优化问题时,我们想要考虑虽然不产生最优解但产生保证接近最优的解的算法。假设对某个优化问题,无论是最大化还是最小化,我们想要获得这样的解。对这个问题的每个实例 x ,存在值为 $\text{opt}(x)$ 的最优解;假设 $\text{opt}(x)$ 总是正整数(我们这里研

究的所有优化问题都是这样的；我们可轻易地识别和解决 opt 为零的实例）。

现在假设我们有多项式算法 A ，当输入优化问题的实例 x 时，算法 A 返回带有值 $A(x)$ 的某个解。因为问题是 NP 完全的并且 A 是多项式的，所以我们不能非现实地希望 $A(x)$ 总是最优值。但是假设我们知道下列不等式总是成立：

$$\frac{|\text{opt}(x) - A(x)|}{\text{opt}(x)} \leq \epsilon$$

其中 ϵ 是某个正实数，希望它非常小，它从上方限制算法 A 在最坏情形下的相对误差。（不等式里的绝对值允许我们用同样框架处理最小化和最大化问题。）如果对问题的所有实例 x ，算法 A 满足上述不等式，那么 A 就称为是 ϵ 近似算法。

一旦优化问题被证明是 NP 完全的，下列问题就变成是最重要的：是否存在有关这个问题的 ϵ 近似算法？如果存在的话，那么 ϵ 可有多小？注意只有假设 $P \neq \text{NP}$ ，这些问题才是有意义的，因为如果 $P = \text{NP}$ ，那么优化问题可被精确地解决，即 $\epsilon = 0$ 。

因此全部 NP 完全优化问题可细分成三个大类：

(a) **完全可近似问题**，即对所有无论多么小的 $\epsilon > 0$ ，存在有关这些问题的多项式时间 ϵ 近似算法。在我们见过的 NP 完全优化问题里，只有双机调度（其中我们希望最小化完成时间 D ）落进这个最幸运的类里。

(b) **部分可近似问题**，即对某个范围的 ϵ ，存在有关这些问题的多项式时间 ϵ 近似算法，但是这个范围不会像完全可近似问题那样一直降到零。在我们见过的 NP 完全优化问题里，顶点覆盖和最大可满足性问题落进了这个类里。

(c) **不可近似问题**，即无论 ϵ 多么大，也不存在有关这些问题的多项式时间 ϵ 近似算法，当然除非 $P = \text{NP}$ 。我们在本章里见过的 NP 完全优化问题里，有许多都很不幸地落进这个类里：旅行商问题，团，独立集，以及在输出的多项式时间里，与给定正则表达式等价的确定型自动机的状态数最小化问题（回忆定理 7.3.8 的推论）。

例 7.4.2 让我们描述有关顶点覆盖的 1 近似算法，即对任意图，返回最多两倍于最优解规模的顶点覆盖的算法。算法非常简单：

$C := \emptyset$

while 在 G 里存在剩余的边 $[u, v]$ do

 把 u 和 v 添加到 C 里并从 G 里删除它们

例如在图 7-13 里，算法开始选择边 $[a, b]$ 并把两个端点都插入到 C 里；然后从 G 里删除这两个顶点（当然还有它们所关联的边）。然后选择 $[e, f]$ ，最后选择 $[g, h]$ 。所得出的集合 C 是顶点覆盖，因为在 G 里的每条边必须接触到它里面的一个顶点（要么是因为这个算法选择了它，要么是因为这个算法删除了它）。在这个例子里， $C = \{a, b, e, f, g, h\}$ 有 6 个顶点，它最多两倍于最优值，最优值是 4。

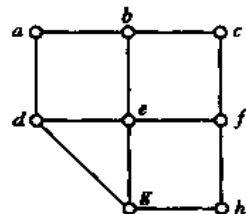


图 7-13

为证明“最多两倍”的保证，考虑算法所返回的覆盖 C ，并设 \hat{C} 是最优顶点覆盖。 $|C|$ 恰好两倍于算法所选择的边数。而通过上述方式所选择的这些边都没有公共顶点，并且对它们中的每条来说至少有一个顶点必

须属于 \hat{C} , 因为 \hat{C} 是顶点覆盖。由此得出算法所选择的边数不大于最优顶点覆盖中的顶点数, 因此 $|C| \leq 2 \cdot |\hat{C}|$, 所以这个算法确实是 1 近似算法。

我们能否做得更好? 令人失望的是, 这个简单算法是有关顶点覆盖的已知的最好算法。仅仅在不久以前我们才证明除非 $P=NP$, 对任意 $\epsilon < \frac{1}{6}$, 不存在有关顶点覆盖的 ϵ 近似算法。 \diamond

例 7.4.3 不过对双机调度来说, 关于我们可多么接近最优值并不存在任何界限; 对任意 $\epsilon > 0$ 都存在有关这个问题的 ϵ 近似算法。

这样一族算法是基于我们见过的想法: 回忆一下, 划分可在时间 $\mathcal{O}(nS)$ 里解决 (其中 n 是整数个数, S 是它们的和, 见 6.2 节)。非常容易看出这个算法可相当平凡地用来解决双机调度 (找出最小的 D): $B(i)$ 集合被扩充到包含直到 S 的和 (不仅达到 $H = \frac{1}{2}S$)。在 $B(n)$ 里大于等于 $\frac{1}{2}S$ 的最小和就是所需要的最小的 D 。

为了得出我们的近似算法还需要另一个想法。考虑双机调度的有下述任务长度的实例:

45362, 134537, 85879, 56390, 145627, 197342, 83625, 126789, 38562, 75402

其中 $n=10, S \approx 10^6$ 。用我们的精确的 $\mathcal{O}(nS)$ 算法解决它, 要花费令我们望而却步的 10^7 步。但是, 假设取而代之, 我们把任务长度都舍入成下一个最近的整百数字。我们得到数字

45400, 134600, 85900, 56400, 145700, 197400, 83700, 126800, 38600, 75500

这其实等于

454, 1346, 859, 564, 1457, 1974, 837, 1268, 386, 755

(都除以 100); 因此现在我们可在大约 10^5 步里解决这个实例。通过牺牲一点精确度 (新问题的最优解显然距离原始问题的最优解不是很远), 我们把时间要求减少了上百倍!

容易证明, 若我们舍入到最近的下一个 10 的 k 次方, 则两个最优解之间的差不超过 $n10^k$ 。为了计算相对误差, 这个量必须除以最优解, 显然最优解不小于 $\frac{S}{2}$ 。因此我们有 $\frac{2n10^k}{S}$ 近似算法, 它的运行时间是 $\mathcal{O}\left(\frac{nS}{10^k}\right)$ 。通过设置 $\frac{2n10^k}{S}$ 等于任意需要的 $\epsilon > 0$, 我们就得到运行时间是 $\mathcal{O}\left(\frac{n^2}{\epsilon}\right)$ 的算法, 这肯定是多项式的。 \diamond

例 7.4.4 如何证明问题不可近似 (或者不是完全可近似)? 对于所关注的大多数优化问题来说, 这个问题一直是最顽固的悬而未决的问题之一, 需要开发新的思想和数学方法 (看本章结尾的参考文献)。但是让我们查看其中相对容易的情形, 即对旅行商问题的不可近似性的证明。

假设给定某个大数 ϵ , 我们必须证明除非 $P=NP$, 否则就不存在有关旅行商问题的 ϵ 近似算法。我们知道 Hamilton 圈是 NP 完全的; 我们将证明如果存在旅行商问题的 ϵ 近似算法, 那么存在 Hamilton 圈的多项式时间算法。让我们从 Hamilton 圈的有 n 个顶点的实例 G 开始。我们对它应用从 Hamilton 圈到旅行商问题的简单归约 (回忆定理 7.3.4 的证明), 但是需要一点技巧: 定义距离 d_{ij} 如下 (比较定理 7.3.4 的证明)

$$d_{ij} = \begin{cases} 0 & \text{如果 } i = j; \\ 1 & \text{如果 } (v_i, v_j) \in G; \\ 2 + n\epsilon & \text{否则。} \end{cases}$$

所构造的实例具有下列有趣性质:如果 G 有 Hamilton 圈,那么巡回路线的最优成本是 n ;不过如果 G 没有 Hamilton 圈,那么最优成本大于 $n(1+\epsilon)$,因为必须至少经过一个是 $2+n\epsilon$ 的距离 $2+n\epsilon$ 和另外 $n-1$ 个至少是 1 的距离。

假设我们有旅行商问题的多项式时间 ϵ 近似算法 A 。那么我们可以如下辨别 G 是否有 Hamilton 圈,在给定的旅行商问题的实例上运行算法 A 。于是有下列两种情形:

(a) 如果返回的解有成本 $\geq n(1+\epsilon)+1$,那么我们知道最优解不是 n ,否则 A 的相对误差至少是

$$\frac{|n(1+\epsilon)+1-n|}{n} > \epsilon$$

这与 A 是 ϵ 近似算法的假设矛盾。因为最优解是大于 n 的,所以我们得出结论说 G 没有 Hamilton 圈。

(b) 如果 A 返回的解有成本 $\leq n(1+\epsilon)$,那么我们知道最优解必定是 n 。这是因为我们的实例被设计成不能有成本在 $n+1$ 和 $n(1+\epsilon)$ 之间的巡回路线。因此在这种情形里 G 有 Hamilton 圈。

由此得出通过把算法 A 应用到我们在多项式时间里构造出的旅行商问题的实例上,我们就可辨别 G 是否有 Hamilton 圈——这蕴涵着 $P=NP$ 。因为对无论多大的任意 $\epsilon>0$,这样的论证都可被完成,所以我们必须得出结论说旅行商问题是不可近似的。◇

对付 NP 完全性的方法常常很好地融合起来:一旦我们认识到旅行商问题是不可近似的,我们就想要近似这个问题的特殊情形。确实,让我们考虑距离 d_{ij} 满足三角不等式的特殊情形:

$$d_{ij} \leq d_{ik} + d_{kj} \quad \text{对于每个 } i, j, k$$

这是相当自然的关于距离矩阵的假设,它在大多数来源于实践的旅行商问题实例里成立。像事实上那样,这种特殊情形是部分可近似的,而且已知的最好误差界限是 $\frac{1}{2}$ 。另外,当城市都被限制成是在带有通常欧氏距离的平面上的点时——这是另一种有明显吸引力和相关性的特殊情形——则这个问题变成完全可近似的!这两种特殊情形都已知是 NP 完全的(对三角不等式情形的证明,见习题 7.4.3)。

7.4.3 回溯和分支限界算法

所有 NP 完全问题根据定义都可用多项式界限的非确定型 Turing 机解决;但不幸的是,我们仅仅知道用指数方法来模拟这些机器。下一步我们检查试图利用聪明的和与问题有关的策略去改进这种指数性能算法类。这样的方法典型地产生虽然在最坏情形下是指数的、但是通常性能更好的算法。

典型的 NP 完全问题问在“候选证书”或“候选见证”(真值赋值、顶点集、顶点排列等等,回忆 6.4 节)的大集合 S_0 中是否有一个成员满足实例所指定的某些约束条件(满足所有子句,是规模为 K 的团,是 Hamilton 通路等)。我们把这些候选证书或见证称为解。对

所有有趣的问题,所有可能的解的集合 S_0 的规模典型地是指数大小的,并且仅仅与给定的实例 x 有关(它的规模指数地依赖于公式里的变元数、图里的顶点数等等)。

现在,“解”这样的 NP 完全问题的实例的非确定型 Turing 机产生格局的树(回忆图 6-3)。这些格局中的每个都对应到潜在解集合 S_0 的一个子集上,称这个子集为 S 。这些格局面对的“任务”是判定在 S 里是否存在满足 x 的约束条件的解。因此 S_0 是对应于初始格局的集合。辨别 S 是否包含解,这经常是与原来问题相差无几的问题。因此我们把在树里的每个格局看作与原来问题同类的子问题(NP 完全问题的这种有用的“自相似性”性质称为自归约性)。在格局里做非确定性选择,比方说导致 r 个可能的下一个格局,对应着把 S 替换成 r 个集合 S_1, \dots, S_r , 它们的并集必须是 S , 使得没有任何候选解落在这些小集合之外。

这样的想法提示出下面这种解决 NP 完全问题的算法:我们总是保持一个活子问题的集合,把它称为 \mathcal{A} 。起初, \mathcal{A} 仅仅包含初始问题 S_0 , 即 $\mathcal{A} = \{S_0\}$ 。在每个时刻,我们从 \mathcal{A} 里选择一个子问题(假设它是对我们似乎最“有希望”的子问题),从 \mathcal{A} 里删除它,并且把它替换成若干个更小的子问题(这些子问题的候选解的并集必须覆盖刚刚删除的子问题的候选解)。这称为分支。

下一步把每个新生成的子问题提交给快速的启发式测试。这样的测试检查子问题,并且获得三种回答之一:

(a) 它获得回答“空”,这意味着所考虑的子问题没有满足输入的约束条件的解,因此可删除它。这样的事件称为回溯。

(b) 它获得包含在当前子问题里的原来问题的真正解(原来的公式的可满足真值赋值,原来的图的 Hamilton 圈等等),在这种情形里算法成功地终止。

(c) 因为问题是 NP 完全的,所以我们不能希望存在总是获得上述回答之一的快速启发式测试(否则,我们就把原来的子问题 S_0 提交给它)。因此测试经常回答“?”,这意味着它既不能证明子问题是空的,也不能在子问题里找到快速解;在这种情形,我们把所处理的子问题添加到活子问题的集合 \mathcal{A} 里。

我们寄希望于测试足够经常地获得前两种回答中的一种,因此大大地减少我们不得不检查的子问题的个数,最终减少算法的运行时间。

现在我们给出完整的回溯算法:

```
 $\mathcal{A} := \{S_0\}$ 
while  $\mathcal{A}$  非空 do
  从  $\mathcal{A}$  中选择子问题  $S$  并删除它
  选择让  $S$  分支的方式,比方说分成子问题  $S_1, \dots, S_r$ 
  对这个表里的每个子问题  $S_i$  do
    if test( $S_i$ ) 返回“找到解” then 停机
    else if test( $S_i$ ) 返回“?” then 把  $S_i$  添加到  $\mathcal{A}$  中
return “无解”
```

回溯算法终止是因为到了最后子问题变成如此地小而特殊,以致于它们只包含一个候选解(这些是非确定型计算树里的叶子);在这种情形里测试就设法快速地判定这个解是否满足实例的约束条件。

回溯算法的有效性依赖三个重要的“设计决定”：

- (1) 如何选择下一个要被分支的子问题？
- (2) 选中的子问题如何继续分裂成更小的子问题？
- (3) 使用何种测试？

例 7.4.5 为了设计可满足性的回溯算法，我们必须作出上述从(1)到(3)的设计决定。

在可满足性里分裂子问题的最自然方式是选择变元 x 并生成两个子问题：一个是让 $x = \top$ ，另一个是让 $x = \perp$ 。像所许诺的那样，每个子问题都是与原来的问题同类的，还是子句的集合，但是变元更少（不出现在当前子问题里的原来问题的每个变元有固定的真值赋值）。在 $x = \top$ 的子问题里，出现 x 的子句都被删去，出现在子句里的 \bar{x} 都被删去；在 $x = \perp$ 的子问题里恰恰相反。

有关设计决定(2)的问题是如何选择分支的变元 x 。让我们用下列规则：选择在最短的子句里出现的变元（若有几个这样的子句，则任选其中一个）。这是合理的策略，因为子句越短，约束条件就“越严”，就越快地导致回溯。特别地，空子句是不可满足性的清楚无误的信号。

现在考虑设计决定(1)，即如何选择下一个子问题。向我们对(2)的策略看齐，让我们选择包含最短子句的子问题（若又有几个这样的子问题，还是任选其中一个）。

最后，测试（设计决定(3)）非常简单：

如果存在空子句，返回“子问题为空”；

如果不存在子句，返回“找到解”；

否则返回“？”

见图 7-14，把上述回溯算法应用到实例

$$(x \vee y \vee z), (\bar{x} \vee y), (\bar{y} \vee z), (\bar{x} \vee x), (\bar{x} \vee \bar{y} \vee \bar{z}),$$

我们知道它是不可满足的（回忆习题 6.3.3）。事实上，这个算法是以 Davis-Putnam 过程而闻名的著名算法的变种。颇有意义的是，当每个子句最多有两个文字时，回溯算法恰恰成为 6.3 节的多项式清洗算法。◇

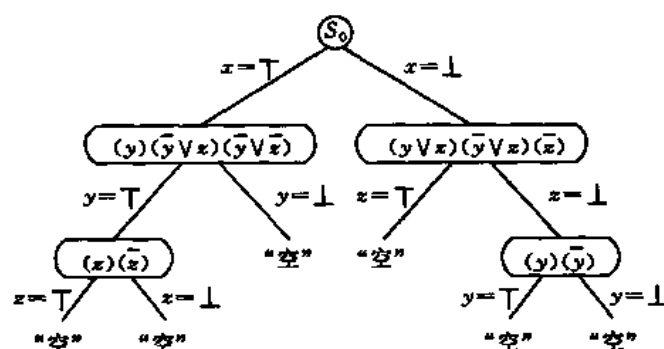


图 7-14

例 7.4.6 现在让我们设计 Hamilton 圈的回溯算法。在每个子问题里我们已经有一条通路，比方说，它的端点是 a 和 b ，并且经过顶点集合 $T \subseteq V - \{a, b\}$ 。我们寻找从 a 到 b

经过 V 里剩余顶点的 Hamilton 通路, 以完成 Hamilton 图。起初 $a=b$ 是任意顶点, 而 $T=\emptyset$ 。

分支是容易的, 我们仅仅选择如何用新边来扩充通路, 比方说用 $[a, c]$, 它从 a 到顶点 $c \notin T$ 。这样的顶点 c 在子问题里成为 a 的新值 (在算法里顶点 b 从头到尾是固定的)。我们把对分支子问题的选择留下来不作规定 (我们从 \mathcal{A} 里挑出任意子问题)。最后, 测试如下 (记住在子问题里, 我们在图 $G-T$ 里寻找从 a 到 b 的通路, 其中 $G-T$ 是从原来的图里删除 T 里的顶点):

如果 $G-T-\{a, b\}$ 不连通, 或如果 $G-T$ 有不是 a 或 b 的 1 度顶点, 那么返回“子问题为空”;

如果 $G-T$ 是从 a 到 b 的通路, 那么返回“找到解”;

否则返回“?”

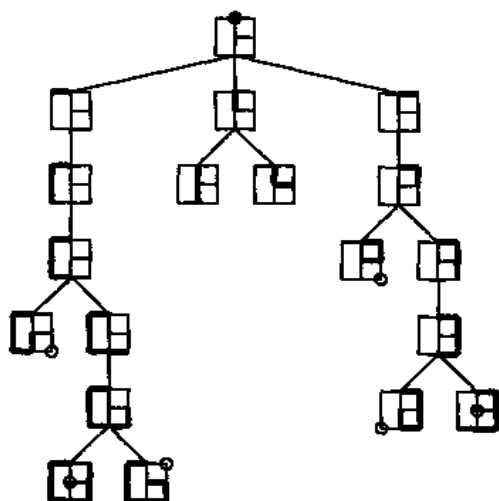


图 7-15 Hamilton 图的回溯算法在根结点所示的图上的执行情况。起初 a 和 b 都与实心顶点重合。在叶子 (回溯) 结点里 1 度顶点用空心圆表示 (在中间的叶子里有许多选择)。总共有 19 个子问题被考虑

这个算法对简单图的应用如图 7-15 所示。尽管构造出的部分解的数目似乎太大 (19), 但是与完整的非确定型“算法”对同样实例所检查的数目 (这个数字是 $(n-1)! = 5040$) 相比, 它还是非常小的。不言而喻, 有可能设计出比这里使用的更复杂也更有效的分支规则和测试。◇

确定最好的设计决定 (1) 到 (3) 不仅主要依赖问题, 而且依赖所关注的实例, 通常这要求大量实验。

当解决“是否”问题时回溯算法是有益的。对优化问题常常使用称为分支限界的回溯法的有趣变种。在优化问题里我们也认为有一个指数规模的候选解集合, 不过这次每个解带有与它关联的成本^①, 我们希望在 S_0 里找出有最小成本的候选解。分支限界算法一般

① 假定所讨论的优化问题是最小化问题; 最大化问题可用非常相似的方式来处理。

是如下所示的算法(虽然所示算法仅仅返回最优成本,但是容易把它修改成返回最优解):

```
 $\mathcal{S} := \{S_0\}, \text{bestsofar} := \infty$ 
while  $\mathcal{S}$  非空 do
  从  $\mathcal{S}$  中选择子问题  $S$  并删除它
  选择让  $S$  分支的方式,比方说分成子问题  $S_1, \dots, S_r$ 
  for 这个表里的每个子问题  $S_i$  do
    if  $|S_i| = 1$  (即  $S_i$  是完全解) then 更新 bestsofar
    else if  $\text{lowerbound}(S_i) < \text{bestsofar}$  then 把  $S_i$  添加到  $\mathcal{S}$  中
return bestsofar
```

算法总是记住迄今为止见过的最小成本,起初是 ∞ (若 bestsofar 初始化成另一个启发式方法所获得的解的成本,则性能通常改进许多)。每一次找到原来问题的完整解,就更新 bestsofar。分支限界算法的关键部分(除了它与回溯法共享的设计决定(1)和(2)之外)是获得子问题 S 里任何解的成本的下界的方法。即函数 $\text{lowerbound}(S)$ 返回保证小于或等于 S 里任何解的最小成本的数值。上述的分支限界算法总是在终止时得出最优解。这是因为剩下没有考虑的子问题仅仅是那些 $\text{lowerbound}(S_i) \geq \text{bestsofar}$ 的子问题,即已证明其最优解不优于我们迄今为止见过的最好解的那些子问题。

自然,有许多方式去获得下界($\text{lowerbound}(S) = 0$ 通常可行……)。要点在于,若 $\text{lowerbound}(S)$ 是返回通常非常接近 S 里最优解的值的复杂算法,则分支限界算法就很有可能具有非常好的性能,即合理地快速终止。

例 7.4.7 让我们用为 Hamilton 圈开发的回溯算法来获得有关旅行商问题的分支限界算法。像以前那样,子问题 S 是用从 a 到 b 经过城市集合 T 的通路来刻画的。什么是合理的下界? 这里有一种想法:对 $T \cup \{a, b\}$ 之外的每个城市,计算它到 T 之外的另一个城市的两个最短距离之和。对 a 和 b ,计算它们到 T 之外的另一个城市的最短距离。不难证明(见习题 7.4.4),这些数字的半和,加上从 a 到 b 经过 T 的固定通路的成本,就是子问题 S 里任何路线的成本的有效下界。现在分支限界算法完全确定了。

关于旅行商问题有复杂得多的下界。◇

7.4.4 局部改进算法

我们的最后一族算法是受了生物进化的启发:如果我们让优化问题的解有小的改变,并且若新解的成本有所改进则我们就采用它,那么会发生什么情况?具体地说,设 S_0 是优化问题实例里候选解的集合(我们还是假设它是最小化问题)。在解的集合上定义邻域关系 $N \subseteq S_0 \times S_0$,它刻画了“有小的改变”的直觉概念。对 $s \in S_0$,集合 $\{s' : (s, s') \in N\}$ 称为 s 的邻域。

算法仅仅是这样的(见图 7-16 对局部改进算法的运行的提示性描绘):

```
 $s := \text{initialsolution}$ 
while 存在解  $s'$  使得
   $N(s, s')$  并且  $\text{cost}(s') < \text{cost}(s)$  do  $s := s'$ 
return  $s$ 
```


即,算法通过用有更好成本的相邻解 s' 取代 s 来持续地改进 s ,直到在 s 的邻域里不存在有更好成本的 s' 为止;在最后的情形里我们说 s 是局部最优解。显然局部最优解不保证是最优解,除非 $N=S_0 \times S_0$ 。所获得的局部最优解的质量和算法的运行时间都严重地依赖 N ;邻域越大局部最优解越好;另一方面,大的邻域蕴涵着算法的迭代(while 循环的执行和搜索当前解 s 的邻域)更慢。局部改进算法在这种交换关系中寻求好的平衡点。通常没有指导我们设计好的邻域的一般原理;选择似乎非常受问题影响,甚至受实例影响,最好是通过实验来做选择。

影响局部改进算法性能的另一个因素是在发现 s' 上所使用的方法。我们是采用在 s 的邻域里发现的第一个更好的解,还是等到发现最好的解?

更长的迭代是否被下降的速度所补偿,以及我们想要快速的下降吗?最后,局部改进算法的性能也受过程 `initialsolution` 影响。根本就不清楚更好的初始解是否导致更好的性能,通常不太好的出发点是更可取的,因为它为算法提供了更多自由度去探索解空间(见图 7-16)。顺便提一句,过程 `initialsolution` 最好随机化,即当多次调用时设法生成不同的初始解。这种方法允许我们多次重新启动上面的局部改进算法,获得许多局部最优解。

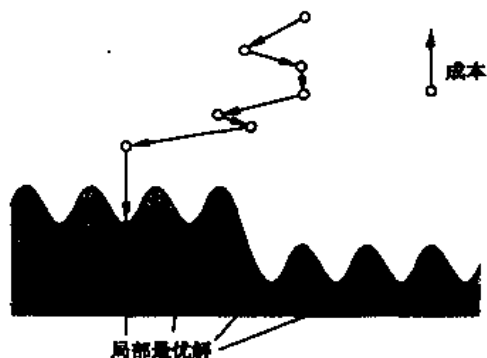


图 7-16 一旦邻域关系固定下来,优化问题的解可描绘成能量风景画,其中局部最优解画成山谷。局部改进启发式算法从解跳跃到解,直到发现局部最优解为止

例 7.4.8 让我们再用旅行商问题作示范。我们何时认为两条巡回路线是相邻的?



图 7-17

因为巡回路线可被认为是 n 个城市之间的无向的“链条”的集合,所以一种似乎有理的回答是当除了非常少数的几个链条之外,它们的所有链条都相同时。两条巡回路线相差的链条数最小可能是 2,这个数字提示了我们称为 **2 交换** 的有关旅行商问题的著名的邻域关系(如图 7-17 所示)。即两条巡回路线

被 N 关联当且仅当它们仅仅相差两个链条。利用 2 交换邻域的局部改进算法在实践中性能相当好。不过,通过采用 3 交换邻域可得到好得多的结果;另外在文献里有报道称,4 交换没有返回足够好的巡回路线以补偿在迭代时间上的增加。

也许当前已知最好的有关旅行商问题的启发式算法是 **Lin-Kernighan 算法**,它依赖于 λ 交换,这种邻域结构是如此的深奥和复杂,以致于它甚至不在我们的框架之内(两个解是否相邻这要由距离来决定)。像它的名字所提示的那样, λ 交换允许在一步里交换任意多个链条(当然不是所有可能交换都被探索到了,这样做可能导致迭代速度指数地降低)。

例 7.4.9 为了开发局部改进算法解决最大可满足性(它是可满足性的变种,我们希望满足尽可能多的子句。回忆定理 7.2.4),我们可以考虑若两个真值赋值仅仅有一个变元的值不同则它们被 N 关联。这就立即定义了有关最大可满足性的有趣的局部改进算法,实验证明它是成功的。在这种情形里明显有利的是采用 s 的最好的相邻解作为 s' ,去

代替采用第一个被发现的比 s 更好的相邻解。另外有报道称值得去做“横向移动”(甚至当算法第三行里的不等式不是严格成立时,也采用这个解)。这个启发式方法被认为是获得最大可满足性最好解的非常有效的方式,并且常常被用来解决可满足性(这时,寄希望于算法结束时返回满足所有子句的真值赋值)。

关于局部改进算法的有趣的技巧是称为模拟退火的方法。像名字所提示的那样,灵感来自于固体冷却的物理过程。通过偶尔执行让成本增加的变化,模拟退火允许算法“避开”坏的局部最优解(见图 7-18,并与图 7-17 比较)。

```

 $s := \text{initialsolution}, T := T_0$ 
repeat
  生成随机解  $s'$  使得  $N(s, s')$ , 并令  $\Delta := \text{cost}(s') - \text{cost}(s)$ 
  if  $\Delta \leq 0$  then  $s := s'$ , else
    以概率  $e^{-\frac{\Delta}{T}}$  令  $s := s'$ 
  update( $T$ )
until  $T = 0$ 
return 见到的最好的解
  
```

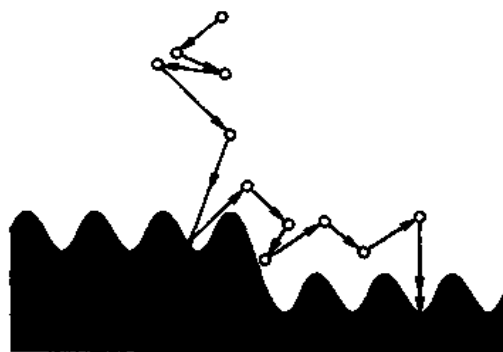


图 7-18 模拟退火比基本的局部改进算法优越,因为它的偶尔让成本增加的移动帮助它避免过早地收敛到坏的局部最优解。这常常引起效率上的巨大损失

直观上,采取让成本增加的变化的概率取决于成本增加的数量 Δ 和重要的参数 T ,即温度。温度越高就越积极地搜索成本较高的解。在算法倒数第三行里更新 T 的方式称作算法的退火时间表,也许是这些算法里最关键的设计决定,当然还有邻域的选择。

还有多种其他的相关种类的局部改进方法,像我们在这里描述的方法那样,它们中的许多都建立在对物理或生物系统的某些类比上(遗传算法,神经网络等等;见参考文献)。

从我们在本书里开发的形式标准的观点来看,局部改进算法和它们的许多变种都是完全没有吸引力的:它们在一般情况下不返回最优解,它们倾向于有指数的最坏情形复杂性,它们甚至不保证返回在任何良定义的意义下“接近”最优解的解。尽管这样,对许多 NP 完全问题,这些算法在实践中还是经常被证明是性能最好的!解释和预言某些这种算法给人以深刻印象的经验上的成功,是今日计算理论里最有挑战性的前沿之一。

习 题

- 7.4.1 在树(看作对称有向图)的特殊情形里,给出支配集问题(回忆习题 7.3.6)的多项式算法。
- 7.4.2 假设在可满足性的实例里所有子句包含最多一个正文字,这样的子句称为 Horn 子句。证明:如果布尔公式的所有子句都是 Horn 子句,那么这个实例的可满足性问题可在多项式时间里解决。(提示:Horn 子句里的变元何时不得不赋值成 T?)
- 7.4.3 证明:即使要求距离服从三角不等式,旅行商问题也还是 NP 完全的。(提示:回顾我们对旅行商问题是 NP 完全的原始证明。)

7.4.4 假设在旅行商问题的带有城市 $1, 2, \dots, n$ 和距离矩阵 d_{ij} 的实例里, 我们仅仅考虑这样的巡回路线, 它从 a 出发, 由长度为 L 的某个通路经过集合 $T \subseteq \{1, 2, \dots, n\}$ 里的城市, 到达另一个城市 b , 然后访问其余的城市并返回 a 。让我们把这样的巡回路线的集合称为 S 。

(a) 对每个城市 $i \in \{1, 2, \dots, n\} - T - \{a, b\}$, 设 m_i 是从 i 到 $\{1, 2, \dots, n\} - T - \{a, b\}$ 里另一个城市的最短和次最短距离之和。设 s 是从 a 到 $\{1, 2, \dots, n\} - T - \{a, b\}$ 里任何城市的最短距离加上从 b 开始的相应的最短距离。证明 S 里的任何巡回路线的成本至少是

$$L + \frac{1}{2} \left[\sum_{i \in \{1, 2, \dots, n\} - T - \{a, b\}} m_i + s \right]$$

即, 上述公式是 S 的有效下界。

(b) n 个城市的最小生成树是以这些城市作为顶点集的最小树, 它可被非常有效地计算出来。从这个信息里推导出有关 S 的更好的下界。

7.4.5 n 个城市的巡回路线有多少个 2 交换的邻居? 多少个 3 交换的邻居? 多少个 4 交换的邻居?

7.4.6 (a) 假设在模拟退火算法里温度保持为 0。证明这是基本的局部改进算法。

(b) 什么是温度保持为无穷大的模拟退火算法?

(c) 现在假设温度在几次迭代里为 0, 然后在几次迭代里为无穷大, 然后又为 0, 等等。所得出的算法与基本型局部改进算法的关系如何?

参 考 文 献

Stephen A. Cook 是证明 NP 完全语言的第一人, 他的文章是:

- S. A. Cook. The Complexity of the Theorem-Proving Procedures. Proceedings of the Third Annual ACM Symposium on the Theory of Computing, pp. 151—158. New York: Association for Computing Machinery, 1971.

Richard M. Karp 在下述文章里建立了 NP 完全性的普遍性与重要性, 证明了定理 7.3.1—7.3.7, 习题 7.3.4 和 7.3.6 里的结果以及一组其他结果。

- R. M. Karp. Reducibility among Combinatorial Problems. In: Complexity of Computer Computations, (pp. 85—104), R. E. Miller, J. W. Thatcher (ed.). New York: Plenum Press, 1972.

Leonid Levin 独立地发现了 NP 完全性:

- L. A. Levin. Universal Sorting Problems. Problemi Peredachi Informatsii, 9(3), pp. 265—266 (in Russian), 1973.

下面的书包含对许多不同领域的 300 多个 NP 完全问题的有用分类, 在它问世后又有许多其他问题被证明是 NP 完全的。

- M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-completeness. New York: Freeman, 1979.

这本书还是关于具体问题的复杂性和关于近似算法的早期信息来源。对最后这个专题的更接近当

前和更深入的处理,见:

- D. Hochbaum (ed.). *Approximation Algorithms for NP-hard Problems*. Boston, Mass: PWS Publishers, 1996.
- 关于对付 NP 完全性的其他方法的有关信息,见:
- C. R. Reeves (ed.). *Modern Heuristic Techniques for Combinatorial Problems*. New York: John Wiley, 1993.
- C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Englewood Cliffs, N. J. :Prentice-Hall, 1982; second edition, New York; Dover, 1997.

中英对照名词索引

(名词后给出所在章节号)

一 画

一对一函数,	one-to-one function,	1. 2
一元函数,	unary function,	1. 2
一进制表示,	unary notation,	2. 4
一进制划分,	unary partition,	6. 2
一步产生, \vdash ,	yields in one step, \vdash ,	2. 2, 3. 3, 4. 4
一笔画,	unicursal,	6. 2

二 画

二元关系,	binary relation,	1. 2
二元有界铺砖,	binary bounded tiling,	7. 2
二元可满足性,	2-satisfiability,	6. 3, 7. 2~7. 4
二元可满足性问题的清洗算法,	purge algorithm for 2-SAT,	6. 3
二元最大可满足性,	MAX 2-SAT,	7. 2, 7. 4
二进制字母表,	binary alphabet,	1. 7

三 画

三元关系,	ternary relation,	1. 2
三元可满足性,	3-satisfiability,	7. 2~7. 4
子图同构,	subgraph isomorphism	7. 3
子序列,	subsequence,	1. 7
子集,	subset,	1. 1
子串,	substring,	1. 7
子句,	clause,	6. 3
上下文,	context,	3. 1, 4. 6
上下文无关文法,	context-free grammar,	3. 1
歧义 \sim ,	ambiguous \sim ,	3. 2
自嵌入 \sim ,	self-embedding \sim ,	3. 5
\sim 与下推自动机,	\sim s and pushdown automata,	3. 4
$LL(1)\sim$,	$LL(1)\sim$,	3. 7
弱优先 \sim ,	weak precedence \sim ,	3. 7
关于 \sim 问题的不可判定性,	undecidability of problems about \sim s,	5. 5
上下文无关文法的扇出,	fanout of a context-free grammar,	3. 5
上下文无关语言,	context-free language,	3. 1~3. 7

确定型~,	deterministic ~,	3. 6.
固有歧义~,	inherently ambiguous ~,	3. 2
上下文有关语言,	context-sensitive language,	5. 7
下推自动机,	pushdown automaton,	3. 3
确定型~,	deterministic ~,	3. 7
~与上下文无关语言,	~ and context-free languages,	3. 4
简单~,	simple ~,	3. 4
下推存储器,栈,	pushdown store, stack,	3. 3
小装置,	gadget,	7. 3

四 画

五元组,	quintuple,	1. 2
六元组,	sextuple,	1. 2
长度,	length	
序列~,	~ of a sequence,	1. 2
通路~,	~ of a path,	1. 3
字符串~,	~ of a string,	1. 7
计算~,	~ of a computation,	3. 3, 3. 5, 4. 1
推导~,	~ of a derivation,	3. 1
计算,	computation,	引言
用文法和其他系统~,	~ by grammars and other systems,	4. 6
用随机存取 Turing 机~,	~ by a random access Turing	
	machine,	4. 4
用 Turing 机~,	~ by a Turing machine,	4. 1, 4. 2
计数器机器,	counter machine,	5. 4
不可数集合,	uncountable set,	1. 4
不相交的集合,	disjoint sets,	1. 1
不可解问题,	unsolvable problem,	5. 4~5. 7
不可判定语言,	undecidable language,	5. 4~5. 7
不可近似问题,	inapproximable problem,	7. 4
不可满足的布尔公式,	unsatisfiable Boolean formula,	6. 3
无穷集合,	infinite set,	1. 4
无星号正则表达式,	*-free regular expressions,	7. 3
无星号正则表达式的不等价性,	inequivalence of *-free regular	
	expressions,	7. 3
无限制文法,	unrestricted grammar,	4. 6
无向图,	undirected graph,	1. 3
无向 Hamilton 图,	undirected Hamilton cycle,	7. 3
文法,无限制文法,	grammar, unrestricted grammar,	4. 6
文法可计算函数,	grammatically computable function,	4. 6
文字,正文字与负文字,	literal, positive and negative,	6. 3
双射,	bijection,	1. 2

双向有穷自动机,	two-way finite automaton,	2.5
双带有穷自动机,	2-tape finite automaton,	2.1
双机调度,	two-machine scheduling,	7.1, 7.3, 7.4
反对称关系,	antisymmetric relation,	1.3
反转, ^R ,	reversal, ^R ,	1.7
以字典序枚举,	lexicographic enumeration,	5.7
以字典序 Turing 可枚举语言,	lexicographically Turing-enumerable language,	5.7
牛回头型语言,	boustrophedon language,	5.5
从语言到语言的归约,	reduction from a language to another,	5.4
从语言到语言的多项式归约,	polynomial \sim ,	7.1
见证,证书,	witness, certificate,	6.4
支配集,	dominating set,	7.4
分支限界算法,	branch-and-bound algorithm,	7.4

五 画

四元组,	quadruple,	1.2
可数集合,	countable set,	1.4
可数无穷集合,	countably infinite set,	1.4
可极小化函数,	minimalizable function,	4.6
可达性,	reachability,	6.2
可满足的布尔公式,	satisfiable Boolean formula,	6.3
可满足性,	satisfiability,	6.3, 7.1~7.3
可着三色,	3-coloring,	7.1
可靠图,	reliable graph,	7.3
正则表达式,	regular expression,	1.8
正则表达式的析取范式,	disjunctive normal form of a regular expression,	1.8
正则表达式的星号深度,	star height of a regular expression,	1.8
正则表达式的等价性,	equivalence of regular expressions,	6.2
正则表达式的不等价性,	inequivalence of regular expressions,	7.3
正则语言,	regular language,	1.8, 2.3, 2.4, 3.1
正则语言的标准自动机,	standard automaton for a regular language,	2.5
正文字,	positive literal,	6.3
布尔变元,	Boolean variable,	6.3
布尔联结词,	Boolean connectives,	6.3
布尔公式,	Boolean formula,	6.3
合取范式形式的 \sim ,	\sim in conjunction normal form,	6.3
布尔逻辑,	Boolean logic,	6.3
左因子分解,	left factoring,	3.7

左递归,	left recursive,	3.7
左端符, \triangleright ,	left-end symbol, \triangleright ,	4.1
左线性文法,	left-linear grammar,	3.1
右线性文法,	right-linear grammar,	3.1
对称关系,	symmetric relation,	1.3
归纳定义,	definition by induction,	1.7
出现,	occurrence,	1.7
击中集,	hitting set,	7.3
半判定,	semidecides,	4.1, 4.4, 4.5

六 画

字母表,	alphabet,	1.7, 3.1, 4.1
字符串,	string,	1.7
字符串的连接,	concatenation of strings,	1.7
字符串机器,	string machine,	2.6
字典顺序,	lexicographic ordering,	1.7
有穷集合,	finite set,	1.4
有穷控制,	finite control,	2.1
有穷自动机,	finite automaton,	2.1
非确定型 \sim ,	nondeterministic \sim ,	2.2
双向 \sim ,	two-way \sim ,	2.5
双带头 \sim ,	2-head \sim ,	2.4, 5.5
双带 \sim ,	2-tape \sim ,	2.1
有穷状态机器,	finite-state machine,	2.1
有序对,	ordered pair,	1.2
有序组,	ordered tuple,	1.2
有序三元组,	ordered triple,	1.2
有向图,	directed graph,	1.3
有界铺砖,	bounded tiling,	7.2
自反关系,	reflexive relation,	1.3
自反传递闭包,	reflexive transitive closure,	1.6
自嵌入文法,	self-embedding grammar,	3.5
自归约性,	self-reducibility,	6.2, 7.4
自底向上的语法分析,	bottom-up parsing,	3.7
自顶向下的语法分析,	top-down parser,	3.7
多项式时间算法,	polynomial-time algorithm,	6.1
多项式可判定的,	polynomially decidable,	6.1
多项式界限 Turing 机,	polynomially bounded Turing machine,	6.1, 6.4
多项式 Turing 归约,	polynomial Turing reduction,	7.1
多项式平衡语言,	polynomially balanced language,	6.4
问题,	problem,	6.2
问题的解,	solution of a problem,	7.4

问题的推广,	generalization of a problem,	7. 2
在指定两点之间的 Hamilton 通路,	Hamilton path between two specified nodes,	7. 3
在两个语言(问题)之间的多项式归约,	polynomial reduction between two languages, or problems,	7. 1
产生,	generates,	3. 1
产生, \vdash^* ,	yields, \vdash^* ,	2. 1, 4. 1, 4. 4
死结束格局,	dead-end configuration,	3. 6
负文字,	negative literal,	6. 3
闭包,	closure,	1. 6
后缀,	suffix,	1. 7, 2. 3
传递关系,	transitive relation,	2. 2, 3. 3, 4. 5
全序,	total order,	1. 3
先于, $<$,	precedes, $<$,	3. 2
后继函数,	successor function,	4. 7
合数,	composite number,	4. 5, 6. 4
忙碌的海狸函数,	busy-beaver function,	5. 3
同态,	homomorphism,	2. 3, 3. 5, 7. 2
非删除 \sim ,	nonerasing \sim ,	6. 4
导出子图同构,	induced subgraph isomorphism,	7. 3
团,	clique,	6. 2, 7. 3, 7. 4
出租车宰客,	taxicab ripoff,	7. 3
动态规划算法,	dynamic programming algorithm,	3. 6, 6. 1, 7. 4
回溯算法,	backtracking algorithms,	7. 4
优先关系,	precedence relation,	3. 7
机器的输出,	output of a machine,	4. 2
划分,	partition,	6. 2, 6. 4, 7. 1, 7. 3
托出	pop	3. 3

七 画

序列,	sequence,	1. 2
步,	step,	3. 1, 3. 3, 4. 1, 6. 1
状态,	state,	2. 1, 2. 2, 3. 1, 3. 3, 4. 1
状态图,	state diagram,	2. 1
初始状态,	initial state,	2. 1, 2. 2, 3. 3, 4. 1
初始格局,	initial configuration,	4. 2, 4. 4
拒绝,	rejects,	4. 2, 4. 4
拒绝格局,	rejecting configuration,	4. 2

判定,	decides,	4.2, 4.4, 4.5
证书, 见证,	certificate, witness,	6.4
近似算法,	approximation algorithms,	7.4
完全可近似问题,	fully approximable problem,	7.4
邻域, 邻域关系,	neighborhood, neighborhood relation,	7.4
极小化,	minimalization,	4.7

八 画

单元集,	singleton,	1.1
单轮下推自动机,	single-turn pushdown automaton,	3.4
转移,	transition,	2.2, 3.3
转移函数,	transition function,	2.1, 4.1, 4.3
转移关系,	transition relation,	2.2, 3.3, 4.5
函数,	function,	1.2
基本~,	basic ~,	4.7
分情形定义的~,	~ defined by cases,	4.7
递归定义的~,	~ defined recursively,	4.7
原始递归~,	primitive recursive ~,	4.7
μ 递归~,	μ -recursive ~,	4.7
函数的自变量,	arguments of a function,	1.2
函数的象,	image of a function,	1.2
函数值,	value of a function,	1.2
函数的值域,	range of a function,	1.2
函数的定义域,	domain of a function,	1.2
函数的逆,	inverse of a function,	1.2
函数的合成,	composition of functions,	4.7
函数的阶, $\mathcal{O}(\cdot)$,	order of a function, $\mathcal{O}(\cdot)$,	1.6
函数的增长率,	rate of growth of a function,	1.6
非确定型有穷自动机的等价性,	equivalence of nondeterministic finite automata,	6.2, 6.4, 7.3
非确定性,	nondeterminism,	引言, 2.2, 3.7, 4.5, 6.4
非确定型有穷自动机,	nondeterministic finite automaton,	2.2
非确定型双带有穷自动机,	nondeterministic 2-tape finite automaton,	2.3
非确定型 Turing 机,	nondeterministic Turing machine,	4.5
非确定性多项式时间, NP,	nondeterministic polynomial time, NP,	6.4
非终结符,	nonterminal,	3.1, 4.6
非删除同态,	nonerasing homomorphism,	6.4
图,	graph,	1.3
图的边,	edge of a graph,	1.3
图中的圈(回路),	cycle in a graph,	1.3

Euler 回路,	Euler cycle,	6. 2
Hamilton 圈,	Hamilton cycle,	6. 2
图的着色,	graph coloring,	7. 3
顶点,	node,	1. 3, 3. 2
顶点覆盖,	node cover,	6. 2, 7. 3, 7. 4
空集,	empty set,	1. 1
空串,	empty string,	1. 7
空格符, \square ,	blank symbol, \square ,	4. 1
线性的,	linear,	3. 4
线性界限自动机,	linear-bounded automaton,	5. 7
歧义文法,	ambiguous grammar,	3. 2
固有歧义语言,	inherently ambiguous language,	3. 2
终结符,	terminal symbol,	3. 1, 4. 6
终结状态,	final states,	2. 1, 2. 2, 3. 3
定语言,	definite language,	2. 3
枚举 Turing 机,	enumerating Turing machine,	5. 7
或, \vee ,	or, \vee ,	6. 3

九 画

语言,	language,	1. 7
正则~,	regular ~,	1. 7, 2. 3
上下文无关~,	context-free ~,	3. 1~3. 7
确定型上下文无关~,	deterministic context-free ~,	3. 7
递归~,	recursive ~,	4. 2, 5. 7
递归可枚举~,	recursively enumerable ~,	4. 2, 5. 7
有穷自动机接受的~,	~ accepted by finite automata,	2. 1, 2. 2
以空栈方式接受的~,	~ accepted by empty store,	3. 3
以终结状态方式接受的~,	~ accepted by final state,	3. 3
文法产生的~,	~ generated by a grammar,	3. 1, 4. 6
~与问题,	~ s vs. problems,	6. 2
语言的商,	quotient of languages,	2. 5
语言的右商,	right quotient of languages,	2. 3, 3. 5
语言生成器,	language generator,	1. 8, 3. 1
语言识别装置,	language recognition device,	1. 8, 3. 1
语言的连接,	concatenation of languages,	1. 7
语法规则,	rule of a grammar,	3. 1, 4. 6
语法分析树,	parse tree,	3. 2
语法分析树的根,	root of a parse tree,	3. 2
语法分析树的叶子,	leaves of a parse tree,	3. 2
语法分析树的高度,	height of a parse tree,	3. 2
语法分析树的结果,	yield of a parse tree,	3. 2
语法分析器,	parser,	2. 1, 3. 7

语法分析器的化简动作,	reduce move of a parser,	3.7
语法分析器的移动动作,	shift move of a parser,	3.7
相容字符串,	consistent strings,	3.7
相对于 L 的等价字符串,	equivalent strings with respect to L ,	2.5
相对于 M 的等价字符串,	equivalent strings with respect to M ,	2.5
相似的,	similar,	3.2
树,	tree,	7.4
树根,	root of a tree,	7.4
带,	tape,	2.1, 4.1, 4.3, 4.4
前缀,	prefix,	1.7, 2.3
封闭性,	closure property,	1.6, 2.3, 3.5
恰当覆盖,	exact cover,	7.3, 7.4
恒等函数,	identity function,	4.7
重写系统,	rewriting system,	4.6
独立集,	independent set,	6.2, 6.4, 7.1, 7.3, 7.4
背包问题,	knapsack,	7.1, 7.3
栈符号,	stack symbols,	3.3
标准推导,	standard derivation,	5.5
标记,	label,	3.2
指数界限 Turing 机,	exponentially bounded Turing machine,	6.4

十 画

格局,	configuration,	
有穷自动机的~,	~ of a finite automaton,	2.1, 2.2
下推自动机的~,	~ of a pushdown automaton,	3.3
Turing 机的~,	~ of a Turing machine,	4.3
随机存取 Turing 机的~,	~ of a random access Turing machine,	4.4
格局的变换,	transformation of a configuration,	4.1
递归函数,	recursive function,	4.2
递归语言,	recursive language,	4.2
递归可枚举语言,	recursively enumerable language,	4.2
原始递归函数,	primitive recursive function,	4.7
原始递归谓词,	primitive recursive predicate,	4.7
原地接受器,线性界限自动机,	in-place acceptor, linear-bounded automaton,	5.7
通路,	path,	1.3, 3.5
通用 Turing 机,	universal Turing machine,	5.2
读头,	reading head,	2.1
弱优先文法,	weak precedence grammar,	3.7

真, \top ,	true, \top ,	6.3
真子集,	proper subset,	1.1
真值赋值,	truth assignment,	6.3
旅行商问题,	traveling salesman problem,	6.1, 6.2, 6.4, 7.1, 7.3, 7.4
部分可近似问题,	partly approximable problem,	7.4
兼容的转移,	compatible transitions,	3.7
起始符,	start symbol,	3.1, 4.6

十一 画

笛卡儿积,	Cartesian product,	1.2
符号,	symbol,	1.7
接受,	acceptance,	2.1
被有穷自动机 \sim ,	\sim by finite automata,	2.1, 2.2
被非确定型有穷自动机 \sim ,	\sim by nondeterministic finite automata,	2.2
被下推自动机 \sim ,	\sim by pushdown automata,	3.3
以空栈方式 \sim ,	\sim by empty store,	3.3
以终结状态方式 \sim ,	\sim by final state,	3.3
被 Turing 机 \sim ,	\sim by Turing machine,	4.2
被随机存取 Turing 机 \sim ,	\sim by random access Turing machine,	4.4
被非确定型 Turing 机 \sim ,	\sim by nondeterministic Turing machine,	4.5
接受格局,	accepting configuration,	4.2
推导,	derivation,	3.1, 4.6
最左 \sim ,	leftmost \sim ,	3.2
最右 \sim ,	rightmost \sim ,	3.2
推入,	push,	3.3
停机格局,	halted configuration,	4.1, 4.4
停机问题,	halting problem,	6.2
停机状态,	halting states,	4.1
随机存取 Turing 机,	random access Turing machine,	4.4
随机存取 Turing 机程序,	program of a random access Turing machine,	4.4
随机存取 Turing 机指令,	instructions of a random access Turing machine,	4.4
随机存取 Turing 机寄存器,	register of a random access Turing machine,	4.4
偏序,	partial order,	1.3
偏序的极小元,	minimal element of a partial order,	1.3
基本函数,	basic functions,	4.7
假, \perp ,	false, \perp ,	6.3

十二 画

集合,	set,	1. 1
集合的元素,	element of a set,	1. 1
集合的成员,集合的元素,	member, or element of a set,	1. 1
集合的划分,	partition of a set,	1. 1
集合的交,	intersection of sets,	1. 1
集合的并,	union of sets,	1. 1
集合的差,	difference of sets,	1. 1
集合的补,	complement of a set,	1. 7
在 \sim 下封闭的正则语言,	regular languages closed under \sim ,	2. 3
在 \sim 下不封闭的上下文无关语言,	context-free languages not closed under \sim ,	3. 5
在 \sim 下封闭的递归语言,	recursive languages closed under \sim ,	4. 2
在 \sim 下不封闭的递归可枚举语言,	recursive enumerable languages not closed under \sim ,	5. 3
在 \sim 下封闭的 P,	P closed under \sim ,	2. 3
集合覆盖,	set cover,	7. 3
幂集,	power set,	1. 1
确定型有穷自动机,	deterministic finite automaton,	2. 1
确定型有穷状态转换器,	deterministic finite-state transducer,	2. 1
确定型下推自动机,	deterministic pushdown automaton,	3. 7
确定型上下文无关文法,	deterministic context-free language,	3. 7
确定型有穷自动机的等价性,	equivalence of deterministic finite automata,	6. 2
等势的集合,	equinumerous set,	1. 4
等价关系,	equivalence relation,	1. 3
等价类,	equivalence class,	1. 3
等价关系的细化,	refinement of an equivalence relation,	1. 3, 2. 5
等价的有穷自动机,	equivalent finite automata,	2. 2
最左推导,	leftmost derivation,	3. 2
最长圈问题,	longest cycle,	7. 3
最大可满足性,	MAX SAT,	7. 2, 7. 4
最小等价有穷自动机,	minimum equivalent finite automaton,	2. 6, 7. 3
最小生成树,	minimum spanning tree,	7. 4
最右推导,	rightmost derivation,	3. 2
铺砖,	tile,	5. 6
铺砖问题,	tiling problem,	5. 6, 7. 2
铺砖系统,	tiling system,	5. 6
装箱,	bin packing,	7. 3
温度,	temperature,	7. 4

编译程序,	compiler,	导言, 2.1, 3.1, 3.7
程序计数器,	program counter,	4.4

十三 画

满足的真值赋值,	satisfying truth assignment,	6.3
输入字母表,	input alphabet,	4.1
输入符号,	input symbols,	3.3
输入带,	input tape,	2.1
零函数,	zero function,	4.7

十四画以上

算法,	algorithms,	导言, 1.6
有关有穷自动机的~,	~ for finite automata,	2.6
有关上下文无关文法的~,	~ for context-free grammars,	3.6
作为~的 Turing 机,	Turing machines as ~,	4.1, 5.1
有效~,	efficient ~,	6.1~6.3
多项式时间~,	polynomial-time ~,	6.1~6.3
近似~, ϵ 近似~,	approximation ~, ϵ -approximation ~,	7.4
动态规划~,	dynamic programming ~,	3.6, 7.4
回溯与分支限界~,	backtracking and branch-and-bound ~,	7.4
局部改进与模拟退火~,	local improvement and simulated annealing ~,	7.4
算术级数,	arithmetic progression,	2.4
模拟退火,	simulated annealing,	7.4
整数的二进制表示,	binary representation of integers,	4.2, 4.4, 6.2, 7.2
整数规划,	integer programming,	7.3

其 他

Aho, A. V. ,		3 参
Bar-Hillel, V. ,		2 参, 3 参
Bird, M. ,		2 参
Brainerd, W. S. ,		4 参
Brassard, G. ,		1 参
Bratley, P. ,		1 参
Cantor, G. ,		1.5, 1 参
Chomsky, N. ,		3 参, 5 参
Chomsky 分层,	Chomsky hierarchy,	5.7
Chomsky 范式,	Chomsky normal form,	3.6

Church-Turing 论题,	Church-Turing thesis,	5.1
~的量的细化,	quantitative refinement of~,	6.1
Cobham, A. ,		6 参
Cook, S. A. ,		4 参, 7 参
Cook 定理,	Cook's Theorem,	7.2
Cormen, T. H. ,		1 参
Davis, M. ,		4 参
Davis-Putnam 过程,	Davis-Putnam procedure,	7.4
ϵ 近似算法,	ϵ -approximation algorithm,	7.4
Earley, J. ,		3 参
Edmonds, J. ,		6 参
Euler, L. ,		6.2, 6.4
Euler 回路,	Euler cycle,	6.2
Euler 图,	Eulerian graph,	6.2
Evey, J. ,		3 参
EXP, 指数时间,	EXP, or exponential time,	6.4
Fermat, P. de,		6.4
4 交换邻域	4-change neighborhood	7.4
Garey, M. R. ,		7 参
Ginsburg, S. ,		2 参, 3 参
Greibach 范式,	Greibach normal formal,	3.5
Greibach, S. ,		3 参
Halmos, P. ,		1 参
Turing 机的停机问题,	halting problem for Turing machine,	5.3
Hamilton, W. R. ,		6.2
Hamilton 图,	Hamilton cycle,	6.2, 6.4, 7.1 ~
		7.4
Hamilton 通路,	Hamilton path,	7.2, 7.4
Harrison, M. A. ,		1 参, 3 参
Hartmanis, J. ,		6 参
Hennie, F. C. ,		4 参
Hermes, H. ,		4 参
Hochbaum, D. ,		7 参
Hopcroft, J. E. ,		2 参, 4 参
Horn 子句,	Horn clause,	7.4
Ichbiah, J. H. ,		3 参
Johnson, D. S,		7 参
Karp, R. M. ,		7 参
Kasami, T. ,		3 参
Kleene, S. C. ,		2 参, 4 参
Kleene 星号,	Kleene star,	1.7
Knuth, D. E. ,		1 参, 2 参, 3 参

k 带 Turing 机,	k -tape Turing machine,	4.3
Landweber, L. H. ,		4 参
λ 交换邻域,	λ -change neighborhood,	7.4
Leiserson, C. E. ,		1 参
Levin, L. A. ,		7 参
Lewis, P. M. ,II,		3 参
Lin-Kernighan 算法,	Lin-Kernighan algorithm,	7.4
Liu, C. L. ,		1 参
$LL(1)$ 文法,	$LL(1)$ grammar,	3.7
Machtey, M. ,		4 参
Markov, A. A. ,		4 参
Markov 系统, Markov 算法,	Markov system, Markov algorithm,	4.6
McNaughton, R. ,		2 参
Mealy, G. H. ,		2 参
Miller, G. A. ,		3 参
Minsky, M. L. ,		4 参
Moore, E. F. ,		2 参
Morris, J. H. ,Jr,		2 参
Morse, S. P. ,		3 参
μ 递归函数,	μ -recursive function,	4.7
n 元关系,	n -ary relation,	1.2
Naur, P. ,		3 参
Nerode, A. ,		2 参
n 重笛卡儿积,	n -fold Cartesian product,	1.2
NP,		6.4
NP 完全问题,	NP-complete problem,	7.1~7.4
Oettinger, A. G. ,		3 参
Ogden, W. G. ,		3 参
P,		2.3
Papadimitriou, C. H. ,		4 参, 6 参, 7 参
Perles, M. ,		2 参, 3 参
Polya, G. ,		1 参
Post 对应系统,	Post correspondence system,	5.5
Post, E. L. ,		4 参, 5 参
Pratt, V. R. ,		2 参
Rabin, M. O. ,		2 参
Reckhow, R. A. ,		4 参
Reeves, C. R. ,		7 参
Rivest, R. L. ,		1 参
Rogers, H. ,Jr. ,		4 参
Salomaa, A. ,		1 参
Schutzenberger, M. P. ,		3 参

Scott, D. ,	2 参
Sethi, R. ,	3 参
Shamir, E. ,	2 参, 3 参
Shepherdson, J. C. ,	2 参
Sipser, M. ,	4 参
Stearns, R. E. ,	3 参, 6 参
Steiglitz, K. ,	7 参
Thompson, K. ,	2 参
Turing, A. M. ,	导言, 4. 1, 4 参
Turing 机,	4. 1~4. 5
作为算法的~,	~ as algorithm, 4. 1, 5. 1
用~计算,	computation by a ~, 4. 2
k 带~,	k-tape ~, 4. 3
带多带头的~,	~ with multiple heads, 4. 3
带二维带的~,	~ with two-dimensional tapes, 4. 3
带随机存取的~,	~ with random access, 4. 4
非确定型~,	nondeterministic ~, 4. 5, 6. 4
通用~,	universal ~, 5. 2
有效~,	efficient ~, 6. 1~6. 4
多项式界限~,	polynomially bounded ~, 6. 1~6. 4
指数界限~,	exponentially bounded ~, 6. 4
Turing 机的删除动作,	erasing move of a Turing machine, 4. 1
Turing 可枚举语言,	Turing-enumerable language, 5. 7
Ullian, J. S. ,	3 参, 4 参
Valiant, L. G. ,	3 参
Wang, H. ,	5 参
Warshall, S. ,	1 参
Yamada, H. ,	2 参
Young, P. R. ,	4 参
Younger, D. H. ,	3 参

计算理论基础(第2版)

ELEMENTS OF THE THEORY OF COMPUTATION (SECOND EDITION)



计算理论是计算机科学的基础。本书介绍了计算理论最核心、最基本的内容,包括形式语言与自动机、可计算性和计算复杂性三大部分。全书共分七章,分别为:集合、关系和语言;有穷自动机;上下文无关语言;Turing机;不可判定性;计算复杂性;NP完全性。本书突出了算法,从而使计算机专业的学生更易接受,也更有收益。

世界著名计算机教材精选

- 计算机网络(第3版)
Computer Networks (Third Edition)
- 数据结构:C++语言描述
Data Structures with C++
- 计算机组织与结构:性能设计(第4版)
Computer Organization and Architecture: Design for Performance (Fourth Edition)
- 数据库系统基础教程
A First Course in Database Systems
- 面向对象系统分析与设计
Object-Oriented Systems Analysis and Design
- 计算理论基础(第2版)
Elements of the Theory of Computation (Second Edition)
- 多媒体技术:计算、通信及应用
Multimedia: Computing, Communications & Applications

ISBN 7-302-03948-8



9 787302 039488 >

定价: 29.00 元

责任编辑 薛 慧 / 封面设计 魏瑞华