

**交通系统建模与算法**

**Transportation System: Modelling and Algorithm**

**2022年9月**

**交通网络优化课题组（编）**

目 录

[1 数值计算 8](#_Toc120016618)

[1.1 数值积分 8](#_Toc120016619)

[1.1.1 数值积分方法 8](#_Toc120016620)

[1.1.2 测试问题 8](#_Toc120016621)

[1.2 数值微分 9](#_Toc120016622)

[1.2.1 一元函数的数值微分方法 9](#_Toc120016623)

[1.2.2 多元函数的数值微分方法 10](#_Toc120016624)

[1.2.3 测试问题 10](#_Toc120016625)

[1.3 常微分方程的数值解法 10](#_Toc120016626)

[1.3.1 Euler方法 11](#_Toc120016627)

[1.3.2 改进的Euler方法 11](#_Toc120016628)

[1.3.3 龙格库塔方法 12](#_Toc120016629)

[1.3.4 测试问题 13](#_Toc120016630)

[2 排序算法 14](#_Toc120016631)

[2.1 排序算法概述 14](#_Toc120016632)

[2.1.1 排序问题 14](#_Toc120016633)

[2.1.2 排序评价 14](#_Toc120016634)

[2.1.3 排序算法分类 15](#_Toc120016635)

[2.2 分治法与递归 16](#_Toc120016636)

[2.3 直接选择排序 16](#_Toc120016637)

[2.4 冒泡排序 17](#_Toc120016638)

[2.5 快速排序 17](#_Toc120016639)

[2.6 改进的快速排序 19](#_Toc120016640)

[2.7 堆排序 21](#_Toc120016641)

[2.7.1 二叉树 21](#_Toc120016642)

[2.7.2 堆 21](#_Toc120016643)

[2.7.3 排序算法 22](#_Toc120016644)

[2.8 测试问题 25](#_Toc120016645)

[2.8.1 小规模数组排序 25](#_Toc120016646)

[2.8.2 大规模数组排序 25](#_Toc120016647)

[3 元胞自动机 27](#_Toc120016648)

[3.1 概述 27](#_Toc120016649)

[3.2 Wolfram的初等元胞自动机 28](#_Toc120016650)

[3.2.1 简介 28](#_Toc120016651)

[3.2.2 演化规则 28](#_Toc120016652)

[3.2.3 仿真算法 29](#_Toc120016653)

[3.2.4 仿真结果 29](#_Toc120016654)

[3.3 单车道NaSch模型 31](#_Toc120016655)

[3.3.1 简介 32](#_Toc120016656)

[3.3.2 演化规则 32](#_Toc120016657)

[3.3.3 仿真算法 32](#_Toc120016658)

[3.3.4 仿真结果 33](#_Toc120016659)

[3.4 双车道NaSch模型 33](#_Toc120016660)

[3.4.1 仿真算法 34](#_Toc120016661)

[3.4.2 仿真结果 35](#_Toc120016662)

[4 线性代数\* 36](#_Toc120016663)

[4.1 矩阵相乘 36](#_Toc120016664)

[4.2 矩阵求逆 36](#_Toc120016665)

[4.3 计算矩阵的行列式 38](#_Toc120016666)

[4.3.1 行列式的相关定义定理 38](#_Toc120016667)

[4.3.2 详细算法 39](#_Toc120016668)

[4.4 计算矩阵的秩 40](#_Toc120016669)

[4.5 平面魔方矩阵的生成 41](#_Toc120016670)

[4.5.1 奇数魔方的生成 41](#_Toc120016671)

[4.5.2 双偶魔方的生成 42](#_Toc120016672)

[4.5.3 单偶魔方的生成 43](#_Toc120016673)

[4.6 测试问题 43](#_Toc120016674)

[4.6.1 矩阵相乘 43](#_Toc120016675)

[4.6.2 矩阵的逆 44](#_Toc120016676)

[4.6.3 魔方矩阵的生成和求逆 45](#_Toc120016677)

[4.6.4 矩阵的行列式 46](#_Toc120016678)

[4.6.5 矩阵的秩 47](#_Toc120016679)

[5 基于Cplex的数学规划问题求解\* 48](#_Toc120016680)

[5.1 概述 48](#_Toc120016681)

[5.2 运输问题 48](#_Toc120016682)

[5.3 指派问题 49](#_Toc120016683)

[5.4 最大并发流量问题 49](#_Toc120016684)

[5.4.1 基于路段的模型 49](#_Toc120016685)

[5.4.2 基于路径的模型及列生成算法\* 50](#_Toc120016686)

[5.4.3 等价的基于路径的模型\* 52](#_Toc120016687)

[5.5 最小费用最大并发流量问题 53](#_Toc120016688)

[5.5.1 基于路段的模型 53](#_Toc120016689)

[5.5.2 基于路径的模型及列生成算法\* 53](#_Toc120016690)

[5.6 测试问题 54](#_Toc120016691)

[5.6.1 产销平衡运输问题 54](#_Toc120016692)

[5.6.2 产销不平衡运输问题 55](#_Toc120016693)

[5.6.3 指派问题 55](#_Toc120016694)

[5.6.4 最大并发流量问题 56](#_Toc120016695)

[5.6.5 最小费用最大并发流量问题 56](#_Toc120016696)

[6 最短路问题 58](#_Toc120016697)

[6.1 概述 58](#_Toc120016698)

[6.2 单点对单点最短路问题的数学模型 58](#_Toc120016699)

[6.3 标号设置算法 59](#_Toc120016700)

[6.3.1 Dijkstra算法 59](#_Toc120016701)

[6.3.2 T标号排序的Dijkstra算法 61](#_Toc120016702)

[6.3.3 T标号堆排序的Dijkstra算法 61](#_Toc120016703)

[6.4 标号修正算法 63](#_Toc120016704)

[6.5 有向无环网络上的最短路问题 65](#_Toc120016705)

[6.5.1 拓扑排序 65](#_Toc120016706)

[6.5.2 拓扑排序下的最短路算法 66](#_Toc120016707)

[6.6 测试问题 66](#_Toc120016708)

[6.6.1 Label-setting算法和Label-correcting算法 67](#_Toc120016709)

[6.6.2 有向无环网络Label-correcting算法 67](#_Toc120016710)

[6.6.3 最短路算法计算效率的对比 68](#_Toc120016711)

[6.7 参考文献 69](#_Toc120016712)

[7 旅行商问题 70](#_Toc120016713)

[7.1 概述 70](#_Toc120016714)

[7.2 遗传算法 71](#_Toc120016715)

[8.2.1解的编码和评估 71](#_Toc120016716)

[8.2.2初始解的生成 72](#_Toc120016717)

[8.2.3交叉算子 72](#_Toc120016718)

[8.2.4变异算子 73](#_Toc120016719)

[8.2.5交叉概率和变异概率的自适应调整 75](#_Toc120016720)

[7.3 禁忌搜索算法 75](#_Toc120016721)

[8.3.1解的编码和评估及禁忌的定义 77](#_Toc120016722)

[8.3.2初始解的生成 77](#_Toc120016723)

[8.3.3邻域搜索 77](#_Toc120016724)

[8.3.4禁忌表的更新 79](#_Toc120016725)

[7.4 测试问题 79](#_Toc120016726)

[8.5.1算例生成 79](#_Toc120016727)

[8.5.2两种算法性能的对比 80](#_Toc120016728)

[8.5.3两种算法在不同算例中的性能对比 81](#_Toc120016729)

[8.5参考文献 81](#_Toc120016730)

[8 车辆路径问题\* 82](#_Toc120016731)

[8.1 概述 82](#_Toc120016732)

[8.2 数学优化模型 82](#_Toc120016733)

[8.3 人工蜂群算法 83](#_Toc120016734)

[8.3.1 算法概述 83](#_Toc120016735)

[8.3.2 解的编码 84](#_Toc120016736)

[8.3.3 解的评估 84](#_Toc120016737)

[8.3.4 初始解的生成 84](#_Toc120016738)

[8.3.5 邻域算子 85](#_Toc120016739)

[8.3.6 轮盘赌选择的方法 86](#_Toc120016740)

[8.3.7 详细算法 86](#_Toc120016741)

[8.4 变邻域搜索算法 87](#_Toc120016742)

[8.4.1 算法概述 87](#_Toc120016743)

[8.4.2 解的编码和评估 87](#_Toc120016744)

[8.4.3 初始可行解的生成 87](#_Toc120016745)

[8.4.4 扰动 89](#_Toc120016746)

[8.4.5 局部搜索 90](#_Toc120016747)

[8.4.6 详细算法 92](#_Toc120016748)

[8.5 自适应大邻域搜索算法 92](#_Toc120016749)

[8.5.1 算法概述 92](#_Toc120016750)

[8.5.2 解的编码和评估 93](#_Toc120016751)

[8.5.3 初始可行解的生成 93](#_Toc120016752)

[8.5.4 破坏算子 93](#_Toc120016753)

[8.5.5 修复算子 95](#_Toc120016754)

[8.5.6 更新算子的得分和选择概率 96](#_Toc120016755)

[8.5.7 详细算法 97](#_Toc120016756)

[8.6 测试问题 98](#_Toc120016757)

[8.6.1 算例生成 98](#_Toc120016758)

[8.6.2 三种算法的性能的对比 98](#_Toc120016759)

[8.6.3 三种算法在不同算例中的性能对比 99](#_Toc120016760)

[8.7 参考文献 100](#_Toc120016761)

[9 大规模整数规划问题\* 101](#_Toc120016762)

[10 非线性规划 102](#_Toc120016763)

[10.1 概述 102](#_Toc120016764)

[10.2 凸集、凸函数、凸规划 102](#_Toc120016765)

[10.3 优化问题的最优性条件 104](#_Toc120016766)

[10.3.1 无约束优化问题的最优性条件 104](#_Toc120016767)

[10.3.2 约束优化问题的最优性条件 104](#_Toc120016768)

[10.4 优化问题的一般迭代算法 104](#_Toc120016769)

[10.5 一维搜索 105](#_Toc120016770)

[10.5.1 黄金分割法 106](#_Toc120016771)

[10.5.2 二分法 107](#_Toc120016772)

[10.5.3 非精确一维搜索 108](#_Toc120016773)

[10.5.4 测试问题 109](#_Toc120016774)

[10.6 无约束优化问题的求解算法 109](#_Toc120016775)

[10.6.1 最速下降法 109](#_Toc120016776)

[10.6.2 牛顿型算法 110](#_Toc120016777)

[10.6.3 BFGS方法 111](#_Toc120016778)

[10.6.4 最小二乘法问题 111](#_Toc120016779)

[10.7 线性约束的非线性规划问题的求解方法 115](#_Toc120016780)

[10.7.1 Zoutendijk的可行方向法 115](#_Toc120016781)

[10.7.2 Frank-Wolfe算法 116](#_Toc120016782)

[10.7.3 MSA算法 117](#_Toc120016783)

[10.8 参考文献 117](#_Toc120016784)

[11 用户平衡交通分配问题 119](#_Toc120016785)

[11.1 概述 119](#_Toc120016786)

[11.2 符号及变量定义 119](#_Toc120016787)

[11.3 数学模型 120](#_Toc120016788)

[11.3.1 Beckmann模型 120](#_Toc120016789)

[11.3.2 变分不等式模型 120](#_Toc120016790)

[11.4 间隙函数 120](#_Toc120016791)

[11.5 Frank-wolfe算法 121](#_Toc120016792)

[11.6 投影算法 123](#_Toc120016793)

[11.6.1 投影的计算 123](#_Toc120016794)

[11.6.2 线性投影 123](#_Toc120016795)

[11.6.3 求解UE交通分配问题的投影算法 124](#_Toc120016796)

[11.6.4 求解UE交通分配问题的双投影算法 125](#_Toc120016797)

[11.7 系统最优交通分配 125](#_Toc120016798)

[11.7.1 系统最优交通分配模型 125](#_Toc120016799)

[11.7.2 UE模型与SO模型之间的相互转换 126](#_Toc120016800)

[11.8 Greedy算法 126](#_Toc120016801)

[11.8.1 Greedy算法基本原理 126](#_Toc120016802)

[11.8.2 Greedy算法基本框架 127](#_Toc120016803)

[11.8.3 Greedy单O-D子问题求解方法 129](#_Toc120016804)

[11.9 灵敏度分析 131](#_Toc120016805)

[11.9.1 灵敏度分析计算步骤 131](#_Toc120016806)

[11.9.2 算例分析 133](#_Toc120016807)

[11.10 测试问题 135](#_Toc120016808)

[11.10.1 对称的UE交通分配问题（Frank-wolf算法和投影算法） 135](#_Toc120016809)

[11.10.2 非对称的UE交通分配问题（投影算法） 136](#_Toc120016810)

[11.10.3 SO交通分配问题 137](#_Toc120016811)

[11.10.4 Greedy算法求解UE交通问题 139](#_Toc120016812)

[11.10.5 灵敏度分析 140](#_Toc120016813)

[11.11 参考文献 142](#_Toc120016814)

[12 基于Logit的随机用户平衡交通分配问题 143](#_Toc120016815)

[12.1 概述 143](#_Toc120016816)

[12.2 符号及变量定义 143](#_Toc120016817)

[12.3 数学模型 143](#_Toc120016818)

[12.3.1 路径集的生成 143](#_Toc120016819)

[12.3.2 Fisk模型 145](#_Toc120016820)

[12.3.3 无约束优化模型 145](#_Toc120016821)

[12.4 自适应平均法 146](#_Toc120016822)

[12.5 测试问题 146](#_Toc120016823)

[12.5.1 路径集的生成 146](#_Toc120016824)

[12.5.2 SUE交通分配问题求解 146](#_Toc120016825)

[12.6 参考文献 147](#_Toc120016826)

[13 连续网络设计问题 149](#_Toc120016827)

[13.1 概述 149](#_Toc120016828)

[13.2 连续网络设计问题的双层规划模型 150](#_Toc120016829)

[13.2.1 符号及变量定义 150](#_Toc120016830)

[13.2.2 基于用户均衡的连续网络设计的数学模型 150](#_Toc120016831)

[13.2.3 连续网络设计问题的求解算法概述 151](#_Toc120016832)

[13.3 遗传算法 151](#_Toc120016833)

[13.3.1 解的编码和适应度计算 151](#_Toc120016834)

[13.3.2 初始解的生成 152](#_Toc120016835)

[13.3.3 交叉操作算子 152](#_Toc120016836)

[13.3.4 变异操作算子 153](#_Toc120016837)

[13.4 基于响应面模型的求解算法 154](#_Toc120016838)

[13.4.1 算法概述 154](#_Toc120016839)

[13.4.2 响应面模型 155](#_Toc120016840)

[13.4.3 初始评估点集的生成 156](#_Toc120016841)

[13.4.4 构造候选点集合 156](#_Toc120016842)

[13.4.5 候选点评分 156](#_Toc120016843)

[13.4.6 基于响应面模型的CNDP求解算法 157](#_Toc120016844)

[13.5 测试问题 158](#_Toc120016845)

[13.6 参考文献 161](#_Toc120016846)

[14 离散网络设计问题 162](#_Toc120016847)

[14.1 离散网络设计问题 162](#_Toc120016848)

[14.2 遗传算法 163](#_Toc120016849)

[14.2.1 解的编码和适应度计算 163](#_Toc120016850)

[14.2.2 初始解的生成 163](#_Toc120016851)

[14.2.3 交叉操作算子 163](#_Toc120016852)

[14.2.4 变异操作算子 164](#_Toc120016853)

[14.3 基于响应面模型的求解算法 165](#_Toc120016854)

[14.3.1 初始评估点集的生成 165](#_Toc120016855)

[14.3.2 构造候选点集合 166](#_Toc120016856)

[14.3.3 基于响应面模型的DNDP求解算法 166](#_Toc120016857)

[14.4 测试问题 167](#_Toc120016858)

[14.5 参考文献 169](#_Toc120016859)

[附 录 170](#_Toc120016860)

[附录A：Sioux Falls 网络数据 170](#_Toc120016861)

[A.1 Sioux Falls 网络 170](#_Toc120016862)

[A.2 UE交通分配算法的数据结构 170](#_Toc120016863)

[A.2.1 Frank-wolf算法 170](#_Toc120016864)

[A.2.2 双投影算法 172](#_Toc120016865)

[附录B：旅行商问题网络数据 174](#_Toc120016866)

[B.1 eil51网络节点数据 174](#_Toc120016867)

[B.2 kroA100网络节点数据 175](#_Toc120016868)

# 数值计算

* 1. 数值积分

我们考虑一个给定的函数，求该函数在积分区间[*a,b*]上的定积分。如果能够求得不定积分，则很容易得到定积分。然而，现实当中存在很多函数求不出其不定积分。针对这种情况，我们可以采用数值积分的方法来求定积分的近似值。

* + 1. 数值积分方法

数值积分方法通常把积分区间[*a,b*]均匀离散成很多小的积分区间。令为离散的区间数量，[]为第个积分区间，为区间长度。依据定义，我们有和。为了描述方便，定义。依据积分的可加性，我们有

|  |  |  |
| --- | --- | --- |
|  |  |  |

对于任意区间[]，在该区间上的积分可以采用如下三种方法来近似：

* 中矩形公式

* 梯形公式

* 辛普森公式

把式带入式，我们可以得到**复化中矩公式**：

把式带入式，我们可以得到**复化梯形公式**：

把式带入式，我们可以得到**复化辛普森公式**：

* + 1. 测试问题

（1）选取和，分别采用复化中矩形公式、复化梯形公式、复化辛普森公式求定积分和。

参考答案：下表给出了和时三种复化积分方法的结果。可以看出，在数值计算精度方面，复化辛普森公式最好，其次是复化中矩公式，复化梯形公式最差。

|  |  |  |  |
| --- | --- | --- | --- |
|  | 积分方法 |  |  |
| 10 | 复化中矩公式 | 0.6928 | 0.5204 |
| 10 | 复化梯形公式 | 0.7166 | 0.5346 |
| 10 | 复化辛普森公式 | 0.6931 | 0.5205 |
| 100 | 复化中矩公式 | 0.6931 | 0.5205 |
| 100 | 复化梯形公式 | 0.6956 | 0.5221 |
| 100 | 复化辛普森公式 | 0.6931 | 0.5205 |
|  | 理论值 | 0.6931 | 0.5205 |

（2）设置从1到100，采用复化中矩公式求定积分，给出定积分的值和误差。

我们把区间数量取1000，采用复化中矩公式估计得到定积分的值为0.3591。图1.1给出了该定积分在不同区间离散数量下的估计值及其误差。从图1.1中可以看出，随着离散区间数量的增大，数值积分的精度越来越高。



图1.1 积分值和Gap随离散区间数量的变化情况

* 1. 数值微分
     1. 一元函数的数值微分方法

我们考虑一个给定的一元函数，求该函数在处的导数。在实际中，有时函数没有显示表达式，没法推导出其一阶导。此时，我们可以采用数值微分的方法来得到导数。依据导数的定义，我们有：

当公式中的取值足够小且的时候，我们可以近似计算导数如下：

如果，则称公式的计算方法为右侧差分；否则，称公式的计算方法为左侧差分。为了提高数值微分的计算精度，我们可以采用如下中心差分的方法来近似计算导数：

* + 1. 多元函数的数值微分方法

我们考虑一个给定的多元函数，其中。利用公式和，我们可以对处的梯度分别近似如下：

其中。中的第个元素，其它个元素都为0。

* + 1. 测试问题

（1）已知函数的数值如下表：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 |
|  | 0.591716 | 0.510204 | 0.444444 | 0.390265 | 0.346021 |

分别取和，采用公式和求在处的一阶导数的近似值。

参考答案：在处的一阶导数的精确值。两种数值微分方法得到的近似值和误差如下表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 右侧差分 | 误差（%） | 中心差分 | 误差（%） |
| 0.1 | -0.541794 | 8.57 | -0.614238 | 3.65 |
| 0.2 | -0.492095 | 16.96 | -0.599695 | 1.20 |

（2）采用数值微分的方法求函数在处的梯度。

参考答案：在处的梯度为。两种数值微分方法得到梯度的近似值如下表：

|  |  |  |
| --- | --- | --- |
|  | 右侧差分 | 中心差分 |
| 0.05 |  |  |
| 0.10 |  |  |
| 0.20 |  |  |

* 1. 常微分方程的数值解法

科学研究和工程技术中的许多问题在数学上可以描述为微分方程的求解问题。除特殊情形外，微分方程一般求不出解析解。在实际应用中，往往也只要求得到某处的函数值。为了求解这类问题，有两种方法逼近原方程的解。第一种方法是：将原微分方程化简为可以准确求解的微分方程，然后用化简后的方程的解近似原方程的解。第二种方法是：采用数值解法求解原微分方程，这是实际中最常用的方法。本节介绍Euler方法、改进的Euler方法以及四阶Runge-Kutta方法等三种数值方法来求解如下常微分方程：

采用数值解法求解常微分方程时，通常需要对区间[]进行离散化。令和分别为离散区间的长度和份数，为等距节点。依据定义，我们有和。为了表述简单起见，我们令。

在区间[]上对两边积分，我们有：

* + 1. Euler方法

可以采用左矩形求积公式来近似式右侧的第二项。于是，我们有

把式代入式，我们可以得到求解常微分方程的Euler迭代格式

* + 1. 改进的Euler方法

可以采用梯形求积公式来近似式右侧的第二项。于是，我们有

把式代入式，我们可以得到求解常微分方程的如下梯形迭代格式

因为梯形求积公式比矩形求积公式具有更高的精度，所以采用梯形迭代格式可以比Euler迭代格式得到常微分方程更高的数值解。然而，梯形迭代格式是隐式方法，不便于计算。使用梯形迭代格式求解常微分方程时，在每次迭代中都需要求解一个非线性方程以得到的值。为了克服这个困难，我们可以先用Euler迭代格式求得一个初步的近似值，称之为预测值；然后利用公式做一次迭代得到，即对做一次校正。依据该思路，我们可以得到改进的Euler迭代方法，即对进行如下计算：

* + 1. 龙格库塔方法

采用Euler方法每一步只计算一次函数的值，其整体截断误差为。而采用改进的Euler方法每一步需要计算两次函数的值，其整体截断误差为。基于这两种情况的启发，形成了Runge-Kutta方法的基本思想：通过每次迭代多计算几次函数的值，然后对这些值做线性组合，使整体截断误差的阶数升高，从而达到提高精度的目的。

如下一般迭代格式通过计算两次函数值来实现对常微分方程的求解，被称为二阶Runge-Kutta方法：

其中。通过适当选择参数、、，可以在成立的条件下，使得局部截断误差的阶尽可能高。将在点处作Taylor展开，可以推导得到

将在处作Taylor展开，我们有

只要两个展开式和的前三项相同，便有局部截断误差。要达成该条件，参数必须满足条件：和。

在二阶Runge-Kutta方法中，、、等三个参数常用的取值方法有如下三种：

* 取和。此时迭代格式变成了改进的Euler方法。
* 取、和。此时迭代格式称之为中点方法（midpoint method）。
* 取、和。此时迭代格式称之为Heun方法。

如下一般迭代格式通过计算三次函数值来实现对常微分方程的求解，被称为三阶Runge-Kutta方法：

其中。通过适当选择参数、、、、、、，可以在成立的条件下，使得局部截断误差的阶尽可能高。类似于二阶Runge-Kutta法，将在点处作Taylor展开，然后再将在处作Taylor展开。只要两个展开式的前四项相同，便有局部截断误差。要达成该条件，参数必须满足如下条件：

在三阶Runge-Kutta方法中，参数常用的取值方法为：、、、、、、。

类似地，通过计算四次函数值来实现对常微分方程的求解，可以得到四阶Runge-Kutta法。以下直接给出四阶经典Runge-Kutta公式，即对分别进行如下计算：

其中。

* + 1. 测试问题

取步长，分别采用Euler方法、改进的Euler方法和四阶Runge-Kutta方法求解如下常微分方程：

下表给出了三种不同方法得到的数值解和精确解。由结果可知，四阶Runge-Kutta方法具有最好的求解精度，而改进的Euler方法的求解精度也很不错。相比而言，Euler方法的精度最差。此外，我们还可以观察到，距离初值点越远，常微分方程在处的数值解的误差越大。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | Euler方法 | 改进的Euler方法 | 三阶Runge-Kutta法 | 四阶Runge-Kutta法 | 精确值 |
|  |  |  |  |  |  |  |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1.1 | 0.2718282 | 0.3423778 | 0.3456111 | 0.3459103 | 0.3459199 |
| 2 | 1.2 | 0.6847556 | 0.8583145 | 0.859449 | 0.8666217 | 0.8666425 |
| 3 | 1.3 | 1.2769783 | 1.5927496 | 1.6060447 | 1.6071813 | 1.6072151 |
| 4 | 1.4 | 2.0935477 | 2.5982982 | 2.6186280 | 2.6203113 | 2.6203596 |
| 5 | 1.5 | 3.1874451 | 3.9364441 | 3.9652805 | 3.9676019 | 3.9676663 |
| 6 | 1.6 | 4.6208178 | 5.6789071 | 5.7178234 | 5.7208793 | 5.7209615 |
| 7 | 1.7 | 6.4663964 | 7.9092092 | 7.9598797 | 7.9637718 | 7.9638735 |
| 8 | 1.8 | 8.8091197 | 10.7244671 | 10.7886661 | 10.7935018 | 10.7936247 |
| 9 | 1.9 | 11.7479965 | 14.2374418 | 14.3170429 | 14.3229357 | 14.3230815 |
| 10 | 2.0 | 15.3982357 | 18.5788825 | 18.6758568 | 18.6829266 | 18.6830971 |

# 排序算法

* 1. 排序算法概述
     1. 排序问题

排序就是对一串杂乱无章的数据通过某种方法，使其数值按一定顺序排列起来。例如：“3, 5, 9, 7, 8”按从小到大的顺序进行排列，得到的结果为“3, 5, 7, 8, 9”。采用什么样的方法来得到这组从小到大的数据就是排序所要研究的问题。它是计算机程序设计中一项很重要的技术，是数据处理领域经常使用的一种运算。这类算法在所有数据库程序、编译程序、解释程序和操作系统中得到了广泛的使用。数据排序的主要目的一般都是使查找更快速、更方便。

* + 1. 排序评价

我们考虑对一个长度为的数组进行排序，令和是数组中的两个元素。我们有如下定义：

**定义2.1**（稳定和不稳定排序算法）：对于一个给定的排序算法，如果原本在前面，且，排序之后可能会出现在的后面，则称排序算法不稳定；否则，称排序算法稳定。

**定义2.2**（排序算法的时间复杂度）：对于一个给定的排序算法，完成数组排序所需总的操作次数随数据规模变化而变化的函数称为该算法的时间复杂度。

**定义2.3**（排序算法的空间复杂度）：对于一个给定的排序算法，完成数组排序所需的计算机存储空间随数据规模变化而变化的函数称为该算法的空间复杂度。

排序算法的时间复杂度和空间复杂度都可以表示为数据规模的函数，它们和算法本身的复杂程度一起构成了评价排序算法好坏的主要标准。其中，排序的时间开销是算法好坏的最重要标志。排序的时间开销主要表现在两方面：(i)算法执行中的比较次数和(ii)算法执行中的移动次数。排序算法的时间复杂度反映了代码执行时间随数据规模增长的变化趋势，可以粗略地估计算法的执行效率。

令和分别表示算法执行所要时间的增长速度和算法需要执行的运算次数。如果存在常数使得当时始终有，则记。常见的算法时间复杂度量级主要有如下7类：

|  |  |  |
| --- | --- | --- |
| **分类** | **记作** | **说明** |
| 常量阶 |  | 算法中语句执行次数为一个常数 |
| 对数阶 |  | 算法中语句执行次数与成正比 |
| 线性阶 |  | 算法中语句执行次数与成正比 |
| 线性对数阶 |  | 算法中语句执行次数与成正比 |
| n方阶 |  | 算法中语句执行次数与成正比 |
| 指数阶 |  | 算法中语句执行次数与成正比 |
| 阶乘阶 |  | 算法中语句执行次数与成正比 |

一般而言，算法时间复杂度具备如下关系：。

* + 1. 排序算法分类

表2.1列举了10种主要的排序算法：冒泡排序、选择排序、插入排序、希尔排序、归并排序、快速排序、基数排序、堆排序、计数排序、桶排序。这些排序方法依据不同的准则，可以进行如下三种分类：

（1）内部排序和外部排序

根据待排序的数据大小不同，使得排序过程中所涉及的存储器不同，可分为：(i)**内部排序**和(ii)**外部排序**。将需要处理的所有数据都加载到内部存储器中进行排序称为内部排序；数据量过大，无法全部加载到内存中，需要借助外部存储进行排序称为外部排序。内部排序有冒泡排序、选择排序、插入排序、归并排序、堆排序、快速排序、希尔排序等。外部排序有多路归并排序。

（2）稳定排序和不稳定排序

排序关键字可能出现重复，根据重复关键字的排序情况可分为：(i)**稳定排序**和(ii)**不稳定排序**。待排序的记录序列中可能存在两个或两个以上关键字相等的记录。排序前的序列中领先于（即）。若在排序后的序列中仍然领先于，则称所用的方法是稳定的，反之则为不稳定。稳定排序有插入排序、基数排序、归并排序、冒泡排序、计数排序。不稳定排序有快速排序、希尔排序、简单选择排序、堆排序。

（3）比较排序和非比较排序

根据待排序的数据元素之间的次序是否依赖于它们之间的比较可分为：(i)**比较排序**和(ii)**非比较排序**。排序依赖于元素之间的比较的排序称为比较排序反之称为非比较排序。比较排序有冒泡排序、选择排序、插入排序、归并排序、堆排序、快速排序、希尔排序。非比较排序有计数排序、基数排序、桶排序。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 类别 | 排序方法 | 时间复杂度 | | | 空间复杂度 | 稳定性 | 复杂性 |
| 最好 | 平均 | 最坏 | 辅助存储 |  |  |
| 插入排序 | 直接插入 |  |  |  |  | 稳定 | 简单 |
| 希尔排序 |  |  |  |  | 不稳定 | 复杂 |
| 选择排序 | 直接选择 |  |  |  |  | 稳定 | 简单 |
| 堆排序 |  |  |  |  | 不稳定 | 复杂 |
| 交换排序 | 冒泡排序 |  |  |  |  | 稳定 | 简单 |
| 快速排序 |  |  |  |  | 不稳定 | 复杂 |
| 归并排序 | |  |  |  |  | 稳定 | 复杂 |
| 基数排序 | |  |  |  |  | 稳定 | 复杂 |
| 桶排序 | |  |  |  |  | 稳定 | 复杂 |

本章考虑对数组从小到大进行排序。没有特别说明，数组中的元素序号从0开始，即，排序都是从小到大进行排序。要进行从大到小排序，只需要把所有数据添加一个负号，然后采用从小到大排序即可。本章主要介绍四种排序算法:直接选择排序、冒泡排序、快速排序、堆排序。快速排序在实践中是最好用也是最常用的方法，其期望时间复杂度是。在仅需要寻找堆中最小元素时，相比较严格的快速排序，堆排序更为合适。

* 1. 分治法与递归

分治法（Divide and Conquer）与递归（Recurrences）是一些高效算法常采用的策略。分治法的思想是把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题，直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。这个技巧是很多高效算法的基础，如快速排序算法。分治法的主要步骤就是**分解、解决**以及**合并**，详述如下：

* **分解**：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；
* **解决**：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题；
* **合并**：将各个子问题的解合并为原问题的解。

递归是一种自己调用自己的算法，其特点如下：

* 使用递增归策略时，必须有一个明确的递归结束条件，称为递归出口；
* 递归算法解题通常显得很简洁，但递归算法解题的运行效率较低；
* 在递归调用的过程当中系统为每一层的返回点、局部量等开辟了栈来存储。递归次数过多容易造成栈溢出等情况。
  1. 直接选择排序

直接选择排序方法是一种最简单的排序方法。这种方法的基本思想是：它的基本思想是：第一次从中选取最小值，与交换；第二次从中选取最小值，与交换；第次从中选取最小值，与交换；第次从中选取最小值，与交换。总共通过次，得到一个从小到大排列的有序序列。简而言之，直接选择排序方法先找出数组中最小的元素，然后找次小元素、次次小元素，依此类推。算法2.1.给出了直接选择排序算法。

|  |
| --- |
| **算法2.1. DirectSort()** |
| **输入：**数组 **//** 数组的长度为，数组编码从0开始 |
| 1: **for** to **do** |
| 2: **for** to **do** |
| 3: **if** |
| 4: 置 |
| 5: 置和 **//** 和做交换 |
| 6: **end if** |
| 7: **end for** |
| 8: **end for**  **//** 循环结束后变成及以后的最小元素  **输出：**更新后的数组 |

* 1. 冒泡排序

冒泡排序方法是另一种最简单的排序方法。这种方法的基本思想是：将待排序的元素看作是竖着排列的“气泡”，较小的元素比较轻，从而要往上浮。在冒泡排序算法中我们要对这个“气泡”序列处理若干遍。所谓一遍处理，就是自底向上检查一遍这个序列，并时刻注意两个相邻的元素的顺序是否正确。如果发现两个相邻元素的顺序不对，即“轻”的元素在下面，就交换它们的位置。显然，处理一遍之后，“最轻”的元素就浮到了最高位置；处理二遍之后，“次轻”的元素就浮到了次高位置。在作第二遍处理时，由于最高位置上的元素已是“最轻”元素，所以不必检查。一般地，第遍处理时，不必检查第高位置以上的元素，因为经过前面遍的处理，它们已正确地排好序了。冒泡排序方法本质上是先找出数组中最大的元素，然后找次大元素、次次大元素，依此类推。算法2.2.给出了冒泡排序算法。

|  |
| --- |
| **算法2.2. BubbleSort()** |
| **输入：**数组 **//** 数组的长度为，数组编码从0开始 |
| 1: **for** to **do** |
| 2: **for** to **do** |
| 3: **if** |
| 4: 置 |
| 5: 置和 **//** 和做交换 |
| 6: **end if** |
| 7: **end for** |
| 8: **end for**  **//** 循环结束后变成及以后的最小元素  **输出：**更新后的数组 |

* 1. 快速排序

快速排序是对冒泡排序的一种本质改进，它应用分治法的思想来实现排序。它的基本思想是通过一趟扫描后，使得排序序列的长度能大幅度地减少。在冒泡排序中，一次扫描只能确保最小数值的数移到正确位置，而待排序序列的长度可能只减少1。快速排序通过一趟扫描，就能确保某个数（以它为基准点）的左边各数都比它小，右边各数都比它大。然后又用同样的方法处理它左右两边的数，直到基准点的左右只有一个元素为止。在快排算法中，基准点位置的标定是关键。采用快速排序方法对个数值进行排序时，在最差情况下的时间复杂度为。尽管如此，**快排在实践中仍然是最好用的排序方法。**因为该方法的期望时间复杂度为，而大多数应用场景中会在期望时间内完成快速排序。

任意选取数组中的连续片段，其中和分别为片段的起始位置和结束位置。以选取片段的起点元素为基准点，采用算法2.2对片段进行一趟扫描，就能确保基准点左边各数都比它小，右边各数都比它大。

|  |
| --- |
| **算法2.2** |
| **Input:** 数组，位置和 // 是起始位置和是结束位置  1: 置, , // 以起始位置元素为基准,是左侧位置和是右侧位置 |
| 2: **while** **do** |
| 3: **while** 且 **do** |
| 4: 置  5: **end** **while**  6: **if** **then**  7: 置和 // 和做交换，交换前 |
| 8:  9: **end** **if**  10: **while** 且 **do**  11: 置 |
| 12: **end** **while**  13: **if** **then**  14: 置和 // 和做交换，交换前 |
| 15:  16: **end** **if**  17: **end** **while** |
| **Output:** 更新后的数组和位置 |

算法2.2需要对片段全部扫描一遍，扫描的顺序是先右侧再左侧，交替进行。算法中，和分别为左侧和右侧扫描到的位置。第1行设置基准点取片段的第一个元素，和分别设置为片段的开始和结束位置。算法第3-6行先从片段的右侧开始扫描，此时。当找到比基准点小的元素时，交换和。算法第8-11行先从片段的左侧开始扫描，此时。当找到比基准点大的元素时，交换和。算法2.2固定取片段的第一个元素为基准点。如果想选取其它点做基准点（满足），则可以先交换和，然后再执行算法2.2。

图2.2给出了一个有9个元素的数组的例子来展示了算法2.2如何使基准点左边各数都比它小，右边各数都比它大。如图2.2所示，算法2.2经过4次扫描和3次元素交换完成了基准点的位置标定。如图2.2(a)所示，算法2.2初始化和，有和。算法2.2先从最右侧开始扫描，当扫描到时，满足的条件，右侧扫描终止。然后，执行和的交换操作，并且设置。操作之后的数组如图2.2(b)所示。算法2.2接着从最左侧扫描，当扫描到时，满足的条件，左侧扫描终止。然后，执行和的交换操作，并且设置。操作之后的数组如图2.2(c)所示。接着，算法2.2先从最右侧开始扫描，当扫描到时，满足的条件，右侧扫描终止。然后，执行和的交换操作，并且设置。操作之后的数组如图2.2(d)所示。此时，我们有。于是，算法终止。



图2.2 操作示例

基于算法2.2，算法2.3给出了快速排序的递归算法。该算法可以实现数组的任意连续片段的排序。算法第1行利用算法2.2对片段进行操作，操作后得到的片段进一步可以分成两个小片段和。其中，中的元素都比小，中的元素都比大。算法第2和3两行分别完成片段和的排序。调用算法2.3给整个数组排序，我们只需要设置参数和，即调用函数**QuickSort**()。

|  |
| --- |
| **算法2.3 QuickSort**() |
| **Input:** 数组，位置和 // 是起始位置和是结束位置 |
| 1: 置 // 调用算法2.2 |
| 2: **if** s **then**  3: QuickSort()  4: **end** **if**  5: **if** **then** |
| 6: QuickSort()  7: **end** **if** |
| **Output:** 更新后的数组 |

* 1. 改进的快速排序

算法2.2选取片段的起点元素为基准点。在实际应用中发现，当数组为近似有序状态时，快排算法的效率会大打折扣。产生这种现象的原因是以起点元素为基准点，得到的基准点位置大概率还是起点，这样就不能发挥快速排序算法分而治之的优势了。为了提高快排算法对近似有序数组的排序效率，可以采用片段的中心元素作为基准点。算法2.4给出了该设置下的片段分割算法。算法第3-12先做一轮左侧扫描，确保左侧元素都不大于基准。算法第13-28行与算法2.2一样，右侧扫描和左侧扫描交替进行。当时，算法将终止。

|  |
| --- |
| **算法2.4** |
| **Input:** 数组，位置和 // 是起始位置和是结束位置  1: 置, , // 是中心位置,是左侧位置和是右侧位置 |
| 2: 置 // 以中心位置元素为基准  3: **while** **do** // 先左侧扫描，看看中心位置前有没有元素比基准大  4: **if** **then**  5:  6: **else**  7: Break  8: **end** **if**  9: **end** **while**  10: **if** **then** // 中心位置前有元素比基准更小  11: 置和 // 和做交换  12: **end** **if**  13: **while** **do** |
| 14: **while** 且 **do** |
| 15: 置  16: **end** **while**  17: **if** **then**  18: 置和 // 和做交换，交换前 |
| 19:  20: **end** **if**  21: **while** 且 **do**  22: 置 |
| 23: **end** **while**  24: **if** **then**  25: 置和 // 和做交换，交换前 |
| 26:  27: **end** **if**  28: **end** **while** |
| **Output:** 更新后的数组和位置 |

图2.3给出了一个包含9个元素的数组的例子来展示了算法2.4。该算法以中心元素为基准点。如图2.3所示，算法2.4经过5次扫描和3次元素交换完成基准点的位置标定。如图2.3(a)所示，算法2.4初始化、和，有。算法2.4从最左侧开始扫描，当扫描到时，满足的条件，左侧扫描终止。然后，执行和的交换操作。操作之后的数组如图2.3(b)所示。算法2.4接着从最右侧扫描，当扫描到时，满足的条件，右侧扫描终止。然后，执行和的交换操作，并且设置。操作之后的数组如图2.3(c)所示。类似于前面的操作，分别右侧扫描和左侧扫描各做一次，将分别得到图2.3(d)和图2.3(e)中的结果。这两次扫描以后，有，算法终止。



图2.3 以中心元素为基点的操作示例

把算法2.4中第1行的替换为，我们就得到了一个改进的快速排序算法。

* 1. 堆排序
     1. 二叉树

我们有如下定义：

**定义2.1**（二叉树）：二叉树（binary tree）是指树中节点的度不大于2的有序树。

**定义2.2**（满二叉树）：一棵深度为且有个结点的二叉树称为满二叉树。

**定义2.3**（完全二叉树）：一棵深度为的有个结点的二叉树，对树中的结点按从上至下、从左到右的顺序进行编号，如果编号为（）的结点与满二叉树中编号为的结点在二叉树中的位置相同，则这棵二叉树称为完全二叉树。

**定义2.4**（叶子节点和非叶子节点）：二叉树上没有孩子节点的节点称为叶子节点，其它节点称为非叶子节点。

依据定义可知，完全二叉树的特点是：叶子节点只能出现在最下层和次下层，且最下层的叶子节点集中在树的左部。需要注意的是，满二叉树肯定是完全二叉树，而完全二叉树不一定是满二叉树。

* + 1. 堆

堆在逻辑上是一棵完全二叉树。它可以直接采用数组来表示。数组的任意位置可以表示堆的一个节点。节点的父节点序号就为，左右孩子节点的序号分别为和。图2.4给出了一个堆的例子。节点的数值，它的父节点序号就为，其数值。节点的左右孩子节点的序号分别为和，孩子节点的数值分别为和。



图2.4. 堆的示例

我们有如下定义：

**定义 2.5**（大顶堆）：堆的每个父节点的关键字都不小于其孩子节点。

**定义 2.6**（小顶堆）：堆的每个父节点的关键字都小大于其孩子节点。

由定义2.5和2.6可知，大顶堆的根节点具有最大的关键字，小顶堆的根节点具有最小的关键字。图2.5给出了一个大顶堆和小顶堆的例子。可以看出，图2.5(a)中的大顶堆的每个父节点的数值都不小于其孩子节点，图2.5(b)中的小顶堆的每个父节点的数值都不大于其孩子节点。需要注意的是，大顶堆的关键在于维持每个父节点的数值都不小于其孩子节点，并不需要确保深度为的节点一定要比深度为的节点关键字大。例如，图2.5(a)中有。



图2.5. 大顶堆和小顶堆的示例

* + 1. 排序算法

堆排序（从小到大排序）的基本思想是：将待排序序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。将其与末尾元素进行交换，此时末尾就为最大值。然后将剩余个元素重新构造成一个大顶堆，这样根节点会是个元素中的次小值。进一步将其与的元素互换，剩下个元素继续建大顶堆。如此反复执行，便能得到一个有序序列了。如果要降序排列，则类似地建立小顶堆。

算法2.4给出了如何维持给定父节点的关键字大于其孩子节点的方法。算法2.4的基本思想是，比对节点与其孩子节点的大小关系。如果有比节点数值大的孩子节点，则交换节点与较大的孩子节点的数值。接下来，算法进一步确认交换后的孩子节点的数值是否可以进一步下沉。需要注意的是，在执行算法2.4之前需要确保以节点的孩子节点为根节点的完全二叉树是一个大顶堆。

|  |
| --- |
| **算法2.4 AdjustDown**(, , ) // 节点数值下沉 |
| **Input:** 数组，位置，结束位置 // 只考虑数组中 |
| 1: 置 |
| 2: **while** **do** |
| 3: **if** 且 // 找两个子节点中大  4: 置  5: **end if**  6: **if** // 如果父节点比两个子节点都大，就不用动了  7: Break  8: **end if**  9: 置  10: 置和 // 和做交换，交换子节点和父节点的数值  11: 置 //把父节点设置为子节点  12: 置 //更新为对应的子节点  13: **end while** |
| **Output:** 更新后的数组 |

图2.6以数组为例给出了一个节点数值下沉的例子。由图2.6(a)，可以观察到，节点1的数值比其两个孩子节点3和4的数值都大，节点2的数值比其孩子节点5的数值大。我们以、和为输入，执行算法2.4。由于，算法2.4交换和。交换后的结果如图2.6(b)所示。此时，由于，算法2.4交换和。交换后的结果如图2.6(c)所示。节点没有孩子节点，算法2.4终止。



图2.6. 节点数值下沉的示例

利用算法2.4，算法2.5给出了建大顶堆的方法。该算法从第一个非叶子节点开始，按照节点编号的反序执行父节点数值下沉，直到执行到根节点为止。

|  |
| --- |
| **算法2.5 MakeHeap**(, ) //建大顶堆 |
| **Input:** 数组和长度 |
| 1: **for** to 0 **do** // 慢慢减小 |
| 2: **AdjustDown**() //调用算法2.4  3: **end for** |
| **Output:** 更新后的数组 |

图2.7以数组为例给出了一个建大顶堆的例子。如图2.7(a)所示，算法2.5从最后一个非叶节点开始执行节点数值下沉，需要交换和。图2.7(b)给出了下沉后的结果。接着对倒数第二个非叶节点执行节点数值下沉，需要交换和。图2.7(c-0)给出了下沉后的结果。最后，对根节点执行节点数值下沉。首先交换和，得到图2.7(c-1)中的结果。然后，交换和，得到图2.7(c-2)中的结果。此时，已经构建好大顶堆。



图2.7. 建大顶堆的示例

算法2.6给出了堆排序的完整算法。需要指出的是堆排序并不稳定，即堆排序结果并不是按照对象键值严格递增或严格递减的。得到不同排序结果的原因在于输入数组“不同”，即存在相同元素在不同输入数组内的位置不同。例如，输入数组时，得到的大根堆是；而输入数组时，得到的大根堆是。

|  |
| --- |
| **算法2.6 HeapSort**() |
| **Input:** 数组和长度 |
| 1: **MakeHeap**(**,** ) //调用算法2.5建堆 |
| 1: **for** to 0 **do** // 慢慢减小 |
| 9: 置  10: 置和 **//** 和做交换，把顶拿掉（把最大的拿掉）  2: **AdjustDown**(, , ) //调用算法2.4，剩余的数据重新形成大顶堆  13: **end for** |
| **Output:** 更新后的数组 |

* 1. 测试问题

利用Visual Studio C# 2015编程，在安装了Intel Core i7-7700 3.60-GHz CPU和32.00GB RAM的计算机上运行排序算法。

* + 1. 小规模数组排序

分别采用直接排序、冒泡排序、快速排序、堆排序对如下整数数组按照从小到大进行排序：

18 54 14 45 96 27 80 22 81 31 0 63 49 5 59 27 28 75 68 52 52 66 5 98 44 19 22 16 88 34 59 81 12 85 49 54 84 13 46 83 88 1 70 32 45 32 38 41 46 50

**参考答案：**

**0 1 5 5 12 13 14 16 18 19 22 22 27 27 28 31 32 32 34 38 41 44 45 45 46 46 49 49 50 52 52 54 54 59 59 63 66 68 70 75 80 81 81 83 84 85 88 88 96 98**

* + 1. 大规模数组排序

我们考虑长度为的数组，数组中的元素为之间的整数随机数。分别采用冒泡法、快速排序、改进快速排序、堆排序等方法对随机生成的数组进行排序。我们取数组长度从变化到。为了降低随机性的影响，随机生成10个数组，统计排序的平均时间，结果对比见表2.1。可以看出，随着数组规模的增大，冒泡排序所需的时间快速增加。而快速排序和堆排序所需的时间增加幅度相对较缓。此外，还可以看到，对完全随机的数组进行排序，快速排序和改进快速排序的速度几乎差不多。

表2.1：排序算法的效率对比

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数组规模 |  |  |  |  |
| 直接排序 | 0.0042 s | 0.4177 s | 41.3320 s | - |
| 冒泡排序 | 0.0042 s | 0.4471 s | 44.7343 s | - |
| 快速排序 | 0.0001 s | 0.0012 s | 0.0150 s | 0.1834 s |
| 改进快速排序 | 0.0001 s | 0.0014 s | 0.0155 s | 0.1794 s |
| 堆排序 | 0.0002 s | 0.0019 s | 0.0239 s | 0.2997 s |

为了进一步对比各种排序方法在效率方面的鲁棒性，我们随机扰动排好序的数组，分析数组的有序程度对算法效率的影响。我们对数组中的每个元素以概率进行扰动（即随机生成一个之间的整数代替原来的数字）。特别地，当时，我们将得到完全无序的数组。为了降低随机性的影响，我们重复10次，得到平均排序时间。选取数组长度，扰动概率从0.0001到0.5进行变化。表2.2给出了四种算法在不同扰动概率下，对扰动后的数组重新排序所需计算时间的对比。可以看出，快速排序的算法效率随着数组无序程度的增加而提升。特别地，当数组有序程度非常高时（例如和），快速排序算法的运行时间大幅增加。相反，随着数组无序程度的增加（概率增大），冒泡排序、改进快速排序、堆排序的效率是降低的。近似有序数组排序所需的时间降低的主要原因是，排序中的数值交换操作减少的缘故。此外，我们还可以观察到，相比于原始的快速排序，改进快速排序对于近似有序数组的排序效率非常高。综合表2.1和2.2中的测试结果，使用改进快速排序方法是比较好的选择。

表2.2：近似有序数组排序的效率对比

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 扰动概率 |  |  |  |  |  |
| 直接排序 | 22.2798 s | 22.3445 s | 23.8040 s | 25.6643 s | 31.8703 s |
| 冒泡排序 | 27.4325 s | 27.6766 s | 29.6560 s | 31.6901 s | 38.1876 s |
| 快速排序 | 0.4967 s | 0.0661 s | 0.0158 s | 0.0134 s | 0.0133 s |
| 改进快速排序 | 0.0069 s | 0.0077 s | 0.0097 s | 0.0120 s | 0.0134 s |
| 堆排序 | 0.0171 s | 0.0173 s | 0.0178 s | 0.0190 s | 0.0210 s |

# 元胞自动机

* 1. 概述

元胞自动机（Cellular Automaton，简称CA）是定义在一个由具有离散、有限状态的元胞组成的元胞空间上，按照一定的局部规则，在离散的时间维度上演化的动力学系统。在元胞自动机中，空间被一定形式的规则网络分割为许多单元。每一个单元被称为元胞（Cell），它只能在有限的离散状态集中取值。所有元胞的状态遵循相同的局部规则进行更新。大量的元胞通过简单的相互作用而构成动态系统的演化。不同于一般的动力学模型，元胞自动机不是由严格定义的物理方程或函数确定，而是用一系列模型构造的规则构成。凡是满足这些规则的模型都可以算作是元胞自动机模型。因此，元胞自动机是一类模型的总称，或者说是一个方法框架。其特点是时间、空间、状态都离散，每个变量只取有限多个状态，且其状态改变的规则在时间和空间上都是局部的。

元胞自动机由元胞、元胞空间、元胞状态、邻居、演化规则五部分构成：

* **元胞**：元胞自动机最基本的组成部分，分布在离散的一维，二维或多维欧几里得空间的晶格点上。
* **元胞空间**：元胞所分布在空间上的网格点的集合。
* **元胞状态**：元胞所处的状态。
* **邻居**：对元胞状态更新时只需要考虑临近的元胞的状态。
* **演化规则**：元胞根据当前状态及其邻居状况确定下一时刻该元胞的状态的动力学函数。



图3.1 三种常用的邻居。

如图3.1所示，每一个方格为一个元胞，所有方格构成的集合为元胞空间。元胞所处的状态通常依据研究问题的不同，会有不同的表示方法。例如在行人流研究中，可以用0和1分别表示元胞上面是否有行人；在自行车流研究中，可以用自然数字来表示元胞内的自行车数量。图3.1给出了三种常用的邻居。图3.1(a)-(c)所示黑色元胞分别有4、8、24个邻居（灰色元胞）。演化规则规定了黑色元胞的状态如何随灰色元胞的状态的变化而变化。

元胞自动机可用来研究很多一般现象。其中包括通信、信息传递（Communication）、计算（Compulation）、构造(Construction）、材料学（Grain Growth）、复制 (Reproduction）、竞争（Competition）与进化（Evolution]）等（Smith,1969; Perrier,1996）。同时。它为动力学系统理论中有关秩序(Ordering）、紊动(Turbulence）、混沌(Chaos）、非对称（Symmetry-Breaking）、分形（Fractality）等系统整体行为与复杂现象的研究提供了一个有效的模型工具 (Vichhac，1984; Bennett,1985）。元胞自动机自产生以来，被广泛地应用到社会、经济、交通、军事和科学研究的各个领域。本章主要介绍Wolfram的初等元胞自动机、单车道NaSch模型、双车道元胞自动机模型、城市路网BML模型、以及BML+NS耦合模型。

* 1. Wolfram的初等元胞自动机
     1. 简介

初等元胞自动机的提出者Stephen Wolfram，于1959年出生于英国，15岁时就发表了第一篇论文。后来他领导开发著名数学软件Mathematica。从1981年起，Wolfram开始致力于探索自然界中存在的复杂性的根源问题，他的一个关键思想就是用数值实验来研究元胞自动机。在1981年至1986年的短短几年之内，Wolfram发表了关于元胞自动机理论的一系列重要论文，一举奠定元胞自动机的理论基础。

Wolfram的初等元胞自动机定义在由个元胞组成的一维空间上。任意元胞有两种可能的状态0（白）和1（黑）。令表示元胞在时刻的状态。向量则表示整个元胞空间在时刻的状态，则表示整个元胞空间状态的演化过程，其中为演化时间步构成的集合，为最大演化次数。任意元胞有两个邻居，即元胞和。如图3.2所示，初等元胞自动机定义在包含9个元胞的一维空间上，每个元胞只有0（白）和1（黑）两种状态。元胞的邻居半径为1，即每个元胞有2个邻居元胞。例如元胞3的邻居为元胞2和4。



图3.2 Wolfram的初等元胞自动机。

* + 1. 演化规则

在任意时间步，每个元胞与自己的所有邻居都可以构成一种状态组合。例如，图3.2中元胞4加上邻居元胞3和5构成的状态组合为(1,0,0)，元胞7加上邻居元胞6和8构成的状态组合为(1,1,0)。对于边界元胞1和9，既可以采用开放边界，也可以采用周期性边界。如果采用开放边界，在元胞1的左侧和元胞9的右侧分别设置一个虚拟元胞，且虚拟元胞的状态始终设置为0。此时，元胞1及其邻居对应的状态组合为(0,1,0)；元胞9及其邻居对应的状态组合为(0,1,0)。如果采用周期性边界，元胞1的邻居为元胞9和2，对应的状态组合为(1,1,0)；元胞9的邻居为元胞8和1，对应的状态组合为(0,1,1)。

如图3.3所示，每个元胞（目标元胞）加上2个邻居，可以构成8种可能的状态组合。我们可以采用二进制对这8种可能的状态组合进行编码，即使用0到7来表示这8种可能的状态组合。目标元胞及其邻居所处的状态组合决定了该元胞下一个时间步时的状态。令表示第种状态组合下目标元胞在时间步时的状态，其中。依据定义，的取值只有0或1两种可能。于是，向量总共有28=256种可能的取值。向量的每一种取值代表了一种元胞自动机演化规则。因此，即Wolfram一维初等元胞自动机总共有256种规则。



图3.3 Wolfram的初等元胞自动机的输入和输出状态

对于任意给定的向量，我们可以采用下式来计算对应的Wolfram元胞自动机的规则编号：

|  |  |  |
| --- | --- | --- |
|  |  |  |

给定Wolfram元胞自动机的规则编号，我们可以采用算法3.1来得到对应的向量。我们还可以采用算法3.2来计算给定目标元胞的元胞状态组合的编号值。

|  |  |
| --- | --- |
| **算法3.1：**DecimalToBinary() //Wolfram元胞自动机的规则解码，十进制转化为二进制 | |
| **输入:** 规则编号 | |
| 1: | **for** to 7 **do** |
| 2: | // 表示R除以2的余数 |
| 3: | // 表示R除以2以后取整 |
| 4: | **end** **for** |
| **输出：** | |

|  |  |
| --- | --- |
| **算法3.2：**BinaryToDecimal (,) //二进制转化为十进制 | |
| **输入:** 元胞状态组合 | |
| 1: | **置** |
| 2: | **for** to 2 **do** |
| 2: | //计算元胞状态组合的编号 |
| 4: | **end** **for** |
| **输出：**元胞状态组合的编号 | |

* + 1. 仿真算法

给定规则编号，我们可以采用算法3.3来实现Wolfram元胞自动机的仿真。

|  |
| --- |
| **算法3.3：**Wolfram元胞自动机仿真 |
| **输入:** 规则编号，元胞总数，系统初始状态，最大演化次数 |
| 1: DecimalToBinary()。//对规则解码  2: **for** to **do**  3: **for** to **do**  4: =BinaryToDecimal (,)。//更新元胞状态组合的编号  5: =。//更新元胞状态  6: **end** **for**  7: **end** **for**  **输出：**整个元胞空间状态的演化过程。 |

* + 1. 仿真结果

我们设定，中间元胞（第500个元胞）的初始状态设为1，其它元胞的初始状态都设置为0。图3.4-3.6分别给出了26、30、110号规则的仿真结果。可以看出，有些规则可以演化出非常漂亮的图形。

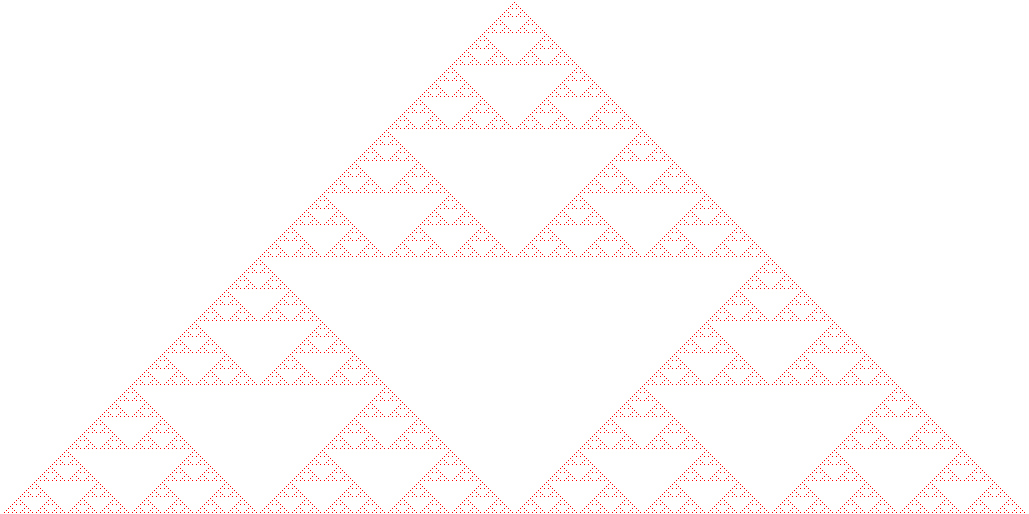


图3.4. 采用26号规则的仿真结果。

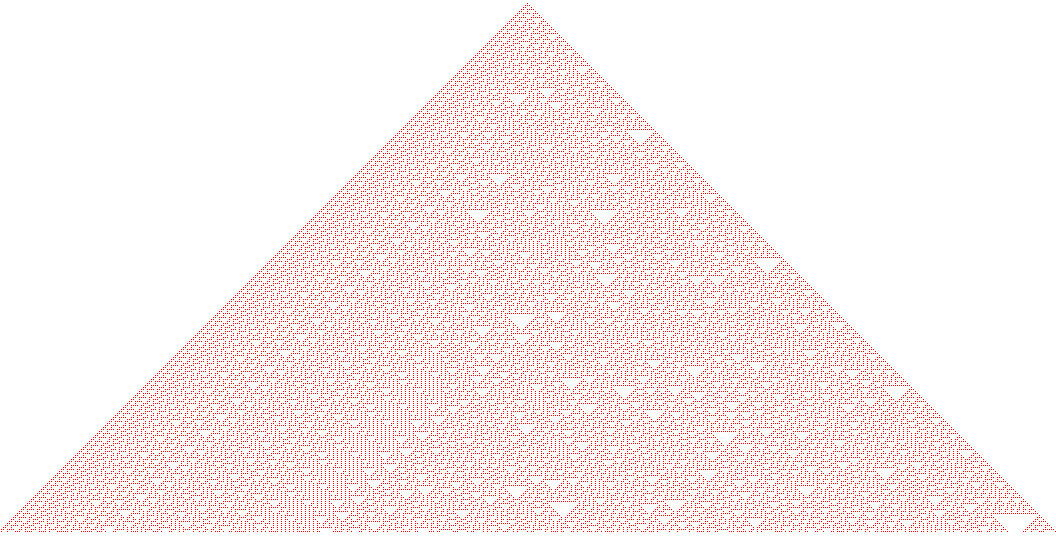


图3.5. 采用30号规则的仿真结果。

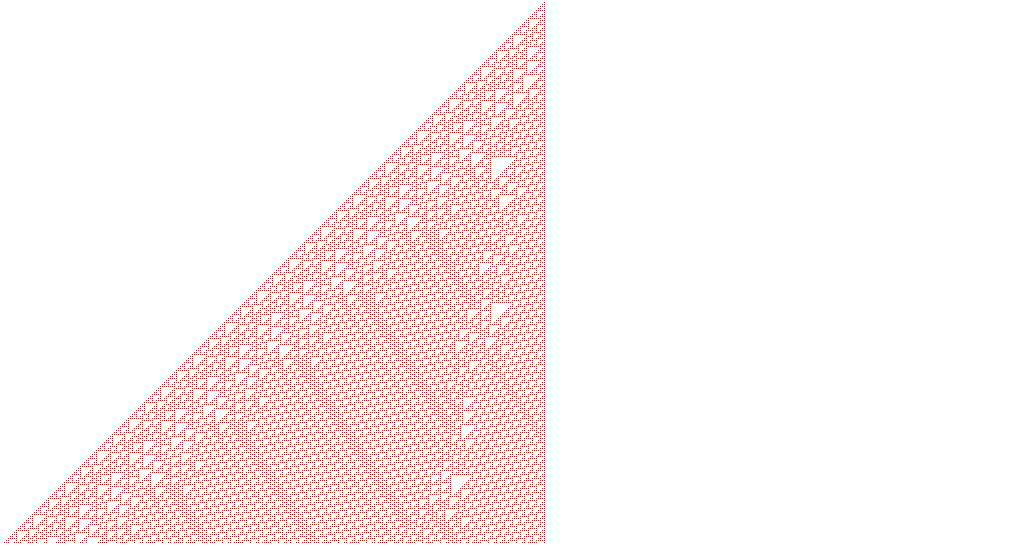


图3.6. 采用110号规则的仿真结果。

如图3.7所示，184号规则是最接近道路交通流的演化规则。基于该规则，我们设定，每个元胞的初始状态以0.3的概率设定为1，以0.7的概率设定为0。图3.8给出了仿真结果。

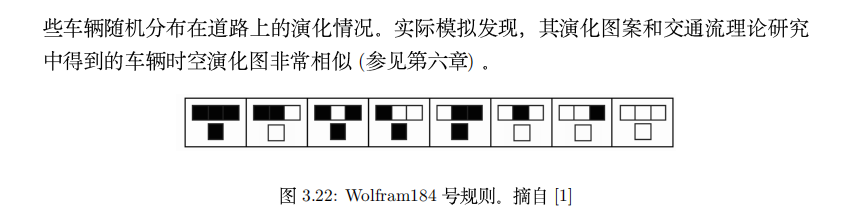


图3.7. 184号规则。

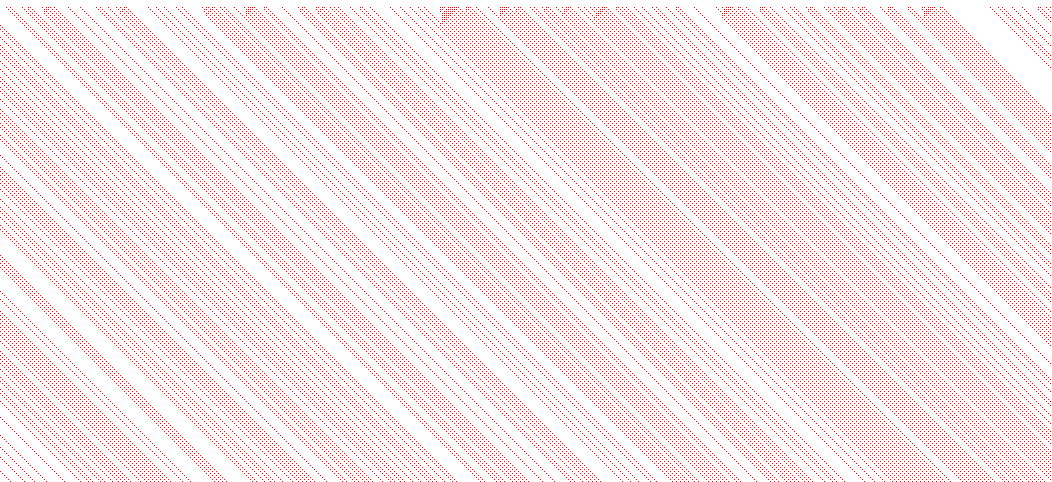


图3.8. 采用184号规则的仿真结果。

* 1. 单车道NaSch模型
     1. 简介

在20世纪90年代，元胞自动机模型被引入到交通领域，并在随后20多年成为了道路交通流领域中的一个重要研究热点。1992年，Nagel和Schreckenberg提出了著名的NaSch模型。该模型被用于模拟高速公路上单车道的交通流，可以再现交通拥堵形成、消散、幽灵塞车等现象。在NaSch模型中，时间、空间、车辆速度都被离散化。如图3.9所示，道路被离散为个格子（即元胞），任意时间步，每个元胞要么是空的，要么被一辆车占据。每辆车的速度可以是，其中为车辆的最大速度。NaSch模型采用周期性边界条件，即所有元胞围成一个环，所有车辆在环上面循环往复运动。



图3.9 单车道的离散。

* + 1. 演化规则

令为道路上的车辆数，表示系统中车辆构成的集合。和分别表示车辆在时刻所处的位置、速度、与前车之间的距离。向量表示系统在时刻的状态，则表示整个系统的演化过程，其中为演化时间步构成的集合，为最大演化次数。算法3.4给出了车辆状态的演化规则。算法的1-4行，用于执行车辆的加速、减速、随机慢化、运动（位置更新）等四个基本步骤。由于采用周期性边界，5-7行用于处理车辆驶出边界的情况。

|  |
| --- |
| **算法3.4：**UpdateVehicleState() **//**NaSch模型车辆状态更新 |
| **输入:** 车辆在时刻的状态和前车距离 |
| 1: 置 //加速，对应于现实中司机期望以最大速度行驶的特性  2: 置 //减速，驾驶员为了避免和前车发生碰撞而采取减速的措施  3: 以概率置 //随机慢化，各种不确定因素引起的车辆减速  4: 置 //运动，更新车辆在时刻所处的位置  5: **if** **then**  6: 。//更新车辆状态  7: **end if**  **输出：**车辆在时刻的状态 |

* + 1. 仿真算法

算法3.3给出了单车道NaSch模型的具体实现方法。算法的2-11行用于计算所有车辆在时刻与前车之间的距离。12-14行调用算法3.4更新所有车辆的状态。

|  |
| --- |
| **算法3.5：**NaSch模型仿真 |
| **输入:** 元胞总数，系统初始状态，最大演化次数 |
| 1: **for** to **do**  2: **for** to **do**  3: **if** **then**  4: //更新车辆与前车的距离  5: **else**  6: //车辆的前车是车辆  7: **end if**  8: **if** **then**  9: //修正周期性边界的影响  10: **end if**  11: **end** **for**  12: **for** to **do**  13: = UpdateVehicleState() //更新车辆状态  14: **end** **for**  15: **end** **for**  **输出：**整个元胞空间状态的演化过程 |

* + 1. 仿真结果

考虑一条具有100个元胞的道路，元胞长度为7.5m，单个时间步为1s，车辆最大速度。基于该设置，车辆的最大速度为135km/h，道路阻塞密度为133辆/km，随机概率取0.2。初始时刻让所有车辆静止排列在道路上，系统预热20000个时间步。

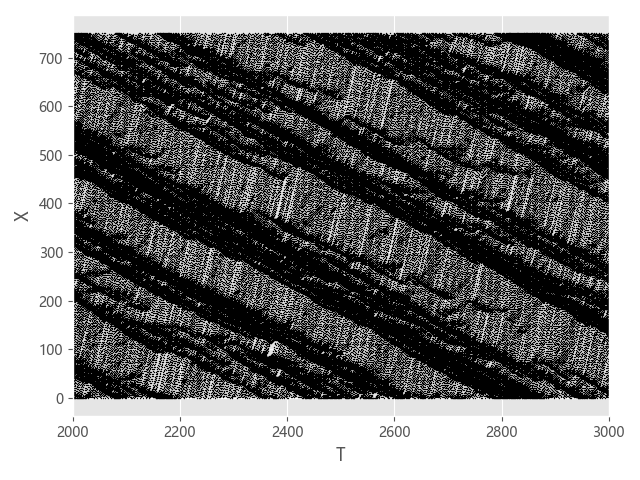
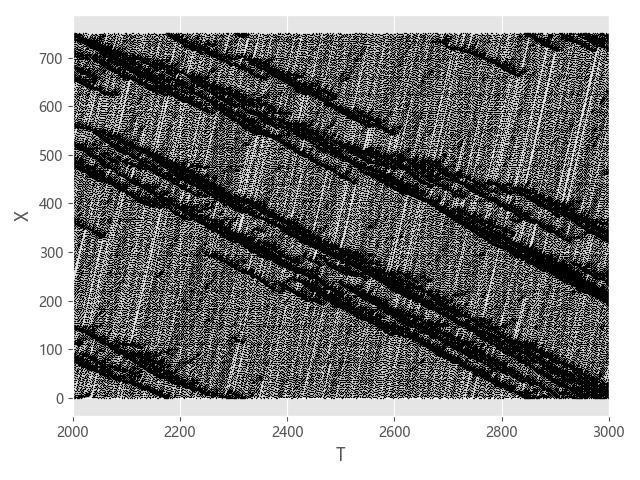
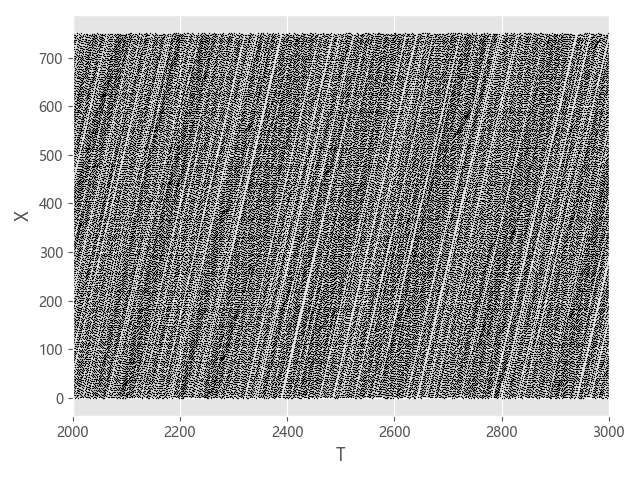


图3.10. NaSch模型仿真得到的时空图（道路密度分别为0.1，0.2，0.3）。

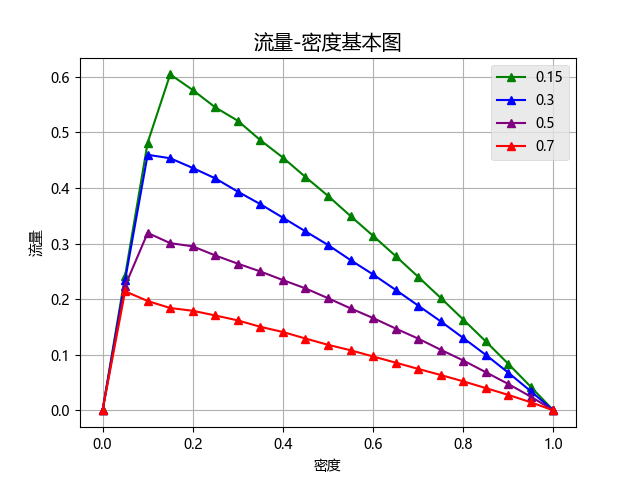


图3.11. NaSch模型仿真得到的流量-密度关系图（对比不同慢化概率）。

* 1. 双车道NaSch模型

双车道NaSch模型是对单车道NaSch模型的拓展。如图3.12所示，在双车道NaSch模型中，我们考虑道路上有两条平行的车道。每条车道都被离散为个元胞，任意时间步，每个元胞要么是空的，要么被一辆车占据。与NaSch模型一样，采用周期性边界条件，即所有元胞围成一个环形系统，所有车辆在环上面循环往复运动。令表示系统中车辆构成的集合，和分别表示时刻在两个车道上行驶的车辆集合。依据定义，有。任意辆车的速度可以是，其中为车辆的最大速度。令和分别表示车辆在时刻所处的位置、速度、与前车之间的距离，、、分别表示车辆在时刻所处的车道、与相邻车道后车之间的距离、与相邻车道前车之间的距离。向量表示系统在时刻的状态，则表示整个系统的演化过程。



图3.12. 双车道的离散。

双车道NaSch模型实施过程中一般是把每个时间步划分为两个子时间步：（1）车辆先按照换道规则进行换道，（2）然后在两条车道上的车辆分别按照单车道NaSch模型的规则进行状态更新。通常驾驶员在换道时是由某些动机来驱使的，而换道动机通常是由两部分组成的：(i)旁道上的行驶条件比本道要好和(ii)车辆在本道上无法按照期望速度行驶。在满足换道动机的同时，还必须满足安全条件才能够实现换道。

对双车道交通而言，换道规则可以是关于车道对称或非对称的。如果采用对称型换道规则，那么车辆的换道策略就与车辆换道的方向无关。同样，非对称换道也受到了人们的关注换道规则的非对称性通常有两种表现形式：在车流密度比较小时，车辆倾向于在左道上行驶，即左道为缺省道；右道上的车辆被禁止超车。这一切都可以通过调整元胞自动机模型的部分规则细节得以实现。这里我们只考虑对称换道规则。

在双车道NaSch模型中采用的换道动机和安全条件如下：

和

其中，是确保不会发生撞车的安全距离。表示车辆在本道上不能按期望的速度行驶，则表示旁道上的行驶条件要比本车道好。如果车辆同时满足换道动机和安全条件，则进行换道。

* + 1. 仿真算法

算法3.3给出了单车道NaSch模型的具体实现方法。为了便于计算车辆与前车以及相邻车道车辆的行驶间距，建议每条车道上的车辆按照位置顺序分别构建一个队列。算法的13-16行更新车辆与前车之间的距离，然后调用算法3.4更新车辆的状态。

|  |
| --- |
| **算法3.6：**双车道NaSch模型仿真 |
| **输入:** 元胞总数，系统初始状态，最大演化次数 |
| 1: **for** to **do**  2: **for** *l* to **do**  3: 置和 // 记录换道的车辆  4: **for each** **do**  5: 更新、  6: **if**满足换道动机和安全条件 **then** //换道  7: 置和  8: 置  9: **end** **if**  10: **end** **for**  11: **end** **for**  12: 置和 //更新车道上的车辆集合  13: **for** **do**  14: 更新  15: = UpdateVehicleState() //更新车辆状态  16: **end** **for**  17: **end** **for**  **输出：**整个元胞空间状态的演化过程 |

* + 1. 仿真结果

考虑一条具有100个元胞的双车道道路，元胞长度为7.5m，单个时间步为1s，快车的最大速度，慢车的最大速度为。慢车的比例为0.1，随机概率取0.2。初始时刻让所有车辆静止排列在道路上，系统预热20000个时间步。

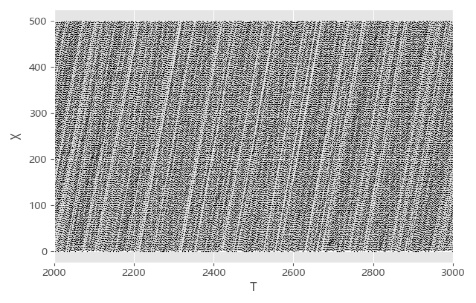
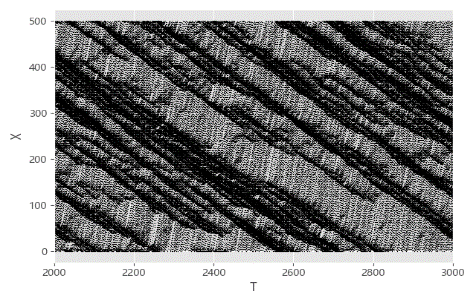
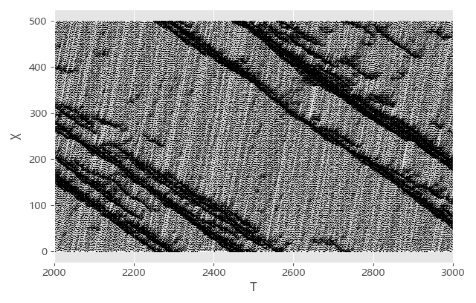
 

图3.13. 双车道NaSch模型仿真得到的单车道时空图（道路密度分别为0.1，0.2，0.3）。

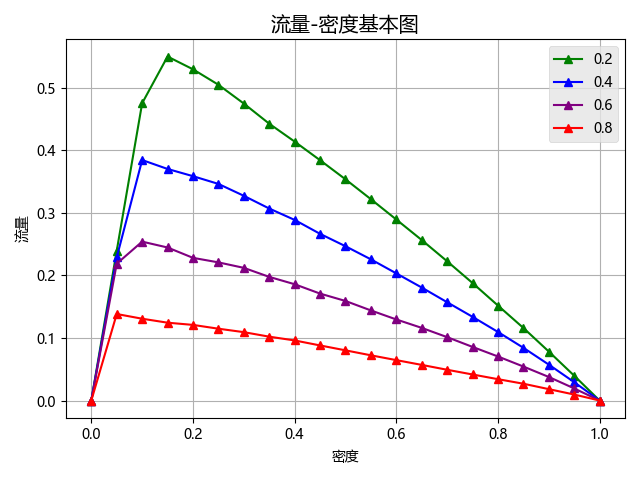
.

图3.14. 双车道NaSch模型仿真得到的单车道流量-密度关系图（（对比不同慢化概率））。

# 线性代数\*

* 1. 矩阵相乘

**定义 4.1** 矩阵与的乘积记为。令，其中是的第行元素与的第列对应元素乘积之和，即

|  |  |  |
| --- | --- | --- |
|  |  |  |

从定义可以看出，只有当左边矩阵的列数与右边矩阵的行数相等时，两个矩阵才能相乘。例如，设，，求及。我们有：

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

矩阵相乘的详细算法步骤如下：

|  |  |
| --- | --- |
| **算法4.1：**矩阵相乘 | |
| **输入:** 矩阵， | |
| 1: | **for do** |
| 2: | **for**  **do** |
| 3: | **置** |
| 4: | **for do** |
| 5: | 置 |
| 6: | **end for** |
| 7: | **end for** |
| 8: | **end for** |
| **输出：**。 | |

* 1. 矩阵求逆

**定义 4.2**（逆矩阵）**.** 令**为**阶单位矩阵。对于阶方阵，若有阶方阵，使得，则称方阵是可逆的，且把方阵称为方阵的逆矩阵。

如果方阵是可逆的，则的逆矩阵唯一。的逆矩阵记作，即有。是矩阵可逆的充分必要条件。若可逆，则有**。**两边取行列式，得。

**定理 4.2：**用初等矩阵左（右）乘矩阵**A，**其结果就是对矩阵做一次初等行（列）变换。

**推论 4.2：**，

**推论 4.3：**矩阵与等价的充分必要条件是存在有限个初等矩阵，使

**推论 4.4：**矩阵可逆的充分必要条件是它能表示成有限个初等矩阵的乘积。

综合以上两推论，矩阵与等价的充分必要条件也可叙述为：存在可逆矩阵**，**使得**。**

从推论4.4可知，可逆矩阵，即**.**记有。

由和表明当经行变换得到时**，E**在同样的行变换下得到了**P，**即。又由知**，**。

故

也可用初等变换法解矩阵方程，即

类似可导出

例：若，求**。**

**【分析】**用的方法，并参考算法4.2：

故

用初等行变换求逆矩阵的详细算法步骤如下：

|  |  |
| --- | --- |
| **算法4.2：**矩阵求逆 | |
| **输入:** n阶方阵，单位矩阵。即 | |
| 1: | 初始化当前行； | |
| 2: | **for**  **do** | |
| 3: | 寻找主元：找到当前行以后的第列绝对值最高的元素和其所在行 | |
| 4: | **if** 主元 **do** | |
| 5: | **if** 主元所在行不是第行 **do** | |
| 6: | 进行行交换;此时单位矩阵相应行也做同样的行交换 | |
| 7: | **end if** | |
| 8: | 将主元化为单位1 | |
| 9: | 利用初等行变换消去矩阵的第列上除去第行以外的各行元素；同时矩阵的相应行做出伴随A变化的行变化。 | |
| 10: | 当前行数增加1 | |
| 11: | **else** | |
| 12: | 则矩阵A不是满秩矩阵，不存在逆矩阵; | |
| 13: | **break;** | |
| 14: | **end if** | |
| 14: | **end for** | |
| **输出：A**。 | |

* 1. 计算矩阵的行列式
     1. 行列式的相关定义定理

**定义 4.3：**由个数排成行列的数表

作出表中位于不同行不同列的个数的乘积，并冠以符号，其中为的一个排列，这样的乘积共有项，它们的和

称为阶行列式，记作

简记为。称式为阶行列式的展开式或行列式的值。

简单地说，阶行列式等于数表中所有取自不同行不同列的个元素乘积的代数和，每项的符号为，其中为当行按标准排列后，对应列标排列的逆序数。

**定义 4.4：**在阶行列式中，划去元素所在的行和列中的元素，余下的元素按原来的位置次序构成的阶行列式称为元素的余子式，记作，并称为元素的代数余子式，记作，即

**定理4.3：**阶行列式等于它的任意行(列)的个元素与其对应的代数余子式乘积之和，即

或

例：若，计算。

**【分析】**按第一列展开计算：

解：

* + 1. 详细算法

用定理4.3计算矩阵的行列式详细算法步骤如下：

|  |  |
| --- | --- |
| **算法4.3：**计算矩阵的行列式(以下为函数 | |
| **输入:** n阶方阵 | |
| 1: | **if**  **do** |
| 2: | **return** |
| 3: | **else** |
| 4: | **置**; |
| 5: | **for** **do** |
| 6: | **置**； |
| 7: | 将元素所在行之后的每一行向前移一行 |
| 8: | **置** () // 提取元素的余子式 |
| 9： |  |
| 10: | **end for** |
| 11: | **return** |
| 12: | **end if** |
| **输出：**。 | |

* 1. 计算矩阵的秩

**定义 4.5：**在矩阵中，任取行列，位于这些行和列交叉处的个元素按原来的位置次序构成的阶行列式，称为矩阵的一个阶子式。

矩阵的阶子式共有个，阶方阵的阶子式只有一个，即为的行列式。

例如，

中有四个三阶子式

它们均等于零，而二阶子式，不为零。

**定义 4.6** 若矩阵有不为零的阶子式，且的所有阶子式全为零（如果有的话），称数为矩阵的秩，并规定零矩阵的秩为0。

根据行列式展开定理，当矩阵中所有阶子式全为零时，则中所有高于阶的子式（如果有的话）也全为零，因此，矩阵的秩就是中最高阶非零子式的阶数。显然

**定理 4.4** 阶梯形矩阵的秩等于中非零行的行数。

**定理 4.5** 矩阵的初等变换不改变矩阵的秩。

例：若 ，计算的秩**。**

**【分析】**应用定理4.5：矩阵的初等变换不改变矩阵的秩。

方法一：初等行变换化上三角矩阵

解：

故矩阵的秩为2。

方法二：采用算法4.4

**Step1:**初始化，令；

**Step2:**初等行变换

得，即矩阵的秩为2。

用定理4.5初等变换求矩阵秩详细算法步骤如下：

|  |  |
| --- | --- |
| **算法4.4：**矩阵求秩 | |
| **输入:** n阶方阵 | |
| 1: | 初始化当前行； |
| 2: | **for** **do** |
| 3: | 寻找主元：找到当前行以后的第列绝对值最高的元素和其所在行 |
| 4: | **if** 主元 **do** |
| 5： | **if** 主元所在行不是第行 **do** |
| 6: | 进行行交换 |
| 7: | **end if** |
| 8: | 利用初等行变换消去矩阵的第列上除去第行以外的各行元素。 |
| 9： | 当前行数增加1 |
| 10： | **end if** |
| 11： | **if**  **do** |
| 12： | **break** 即跳出**for**循环 |
| 13： | **end if** |
| 14: | **end for** |
| **输出：**。 | |

* 1. 平面魔方矩阵的生成

平面魔方矩阵将自然数从1到，排列成行列的**非奇异方阵**，使得每行、每列及两条主对角线上的个数的和都等于，这样的方阵称为阶幻方。当时，平面魔方矩阵可以分为三种情况来构造：为奇数、为4的倍数、为其它偶数（即的形式）。这三种情况得到的平面魔方矩阵分别被称为奇数魔方、双偶魔方和单偶魔方。下面详细介绍三种平面魔方的生成方法。

* + 1. 奇数魔方的生成

奇数魔方具有如下规律：

* 数字1位于方阵中的第一行中间一列；
* 数字（）所在行数比数字所在行数少1，若所在行数为1，则所在行数为；
* 数字（）所在列数比数字所在列数大1，若所在列数为，则所在列数为1；
* 如果数字是的倍数，则数字（）所在行数比数字所在行数大1，所在列数与数字相同。

依据以上规律，我们可以直接得到数字1所处的位置，然后利用数字所处的位置来不断更新数字的位置。算法4.5给出了奇数魔方的详细生成算法。需要注意的是，算法4.5中对于矩阵的编码是从开始的，而我们在编程中，对于矩阵矩阵的编码是从开始的。

|  |
| --- |
| **算法4.5：奇数魔方的生成算法** |
| **输入:** 奇数 |
| 1: 置方阵，,, |
| 2: **for** **do** |
| 3: **if** **do** // 和表示位置，表示要填充的数字 |
| 4: 置 |
| 5: **else**  **//** 执行和，如超出范围，则拉回来 |
| 6: 置和 |
| 7: 置和 |
| 8: 置 |
| 9: **end** **if** |
| 10: **end** **for** |
| **输出：方阵** |

* + 1. 双偶魔方的生成

的双偶魔方可以按照如下步骤来生成：

* 把数字1到按从上至下、从左到右顺序填入到的方阵中；
* 将整个的方阵按照的规模顺序划分为子方阵；
* 任意4×4子方阵中的两条对角线上的数字参照整个大方阵的中心作[**中心对称**](https://baike.baidu.com/item/%E4%B8%AD%E5%BF%83%E5%AF%B9%E7%A7%B0)**交换操作**（即位于的元素与位于的元素交换）。

基于以上步骤，算法4.6给出了双偶魔方的详细生成算法。依据第4行，可以得到初始化以后的矩阵位于的元素的取值为

式被用于第10行，对每个子方阵的对角线位置重新赋值。

|  |
| --- |
| **算法4.6：双偶魔方的生成算法** |
| **输入: 方阵的阶数** **//** 为子方阵的数目 |
| 1: 置方阵 |
| 2: **for** **do**  3: **for**  **do**  4: **//** 1到按从上到下、从左到右顺序填入中 |
| 5: **end** **for** |
| 6: **end** **for** |
| 7: **for** **do**  8: **for**  **do**  9: **if** **do** // 表示取余数，对角线  10: **//** 对每个子方阵的对角线位置重新赋值  11: **end** **if** |
| 12: **end** **for**  13: **end** **for** |

* + 1. 单偶魔方的生成

令和。的单偶魔方的生成步骤如下：

* 把大方阵分解为4个奇数阶子方阵，即；
* 使用算法4.5生成奇数魔方，计算；
* 元素变换：与在同列做对应交换(或)；
* 元素变换：与；与两对元素分别做对应交换。

基于以上步骤，算法4.7给出了单偶魔方的详细生成算法。

|  |
| --- |
| **算法4.7：单偶魔方的生成算法** |
| **输入: 方阵的阶数** |
| 1: 置，方阵, 方阵 **//**  2: 由算法4.5生成矩阵  3: 置  8: 置 |
| 9: **for** **do** | |
| 10: **for** **do** | |
| 11: **if** **do** **//** 按要求做相应的元素交换 | |
| 12: 交换与的值 | |
| 13: **end** **if** | |
| 14: **end** **for** | |
| 15: **end** **for**  16: 交换与的值,交换与的值。 | |
| **输出：方阵** |

* 1. 测试问题
     1. 矩阵相乘

有矩阵

和矩阵

求**。**

解：

* + 1. 矩阵的逆

求矩阵**A**=的逆**。**

解：为

* + 1. 魔方矩阵的生成和求逆

采用算法4.6、4.5和4.7我们分别可以生成如下魔方矩阵：

8阶魔方矩阵**A**=

9阶魔方矩阵**A**=

10阶魔方矩阵**A**=

其中，9阶魔方矩阵的逆矩阵=

图4.1.和4.2分别给出了不同规模的魔方矩阵的生成时间和魔方矩阵的求逆时间。可以看到，魔方矩阵的生成时间和魔方矩阵的求逆时间随着魔方规模的增大，都呈现出非线性的增长趋势。但是，魔方矩阵的生成比魔方的矩阵要快很多。当矩阵规模时，魔方矩阵都可以在1秒内生成。而当矩阵规模达到2000时，魔方矩阵求逆需要90秒以上的时间才能完成。这说明大规模矩阵求逆是比较耗时间的。



图4.1. 不同规模的魔方矩阵的生成时间



图4.2. 不同规模的魔方矩阵的求逆时间

* + 1. 矩阵的行列式

计算矩阵的行列式。

**A**=

解：**A**的行列式的值为-7241013504。

* + 1. 矩阵的秩

计算如下矩阵的秩：

解：。

# 基于Cplex的数学规划问题求解\*

* 1. 概述

CPLEX是由ILOG公司开发的，是解决困难的数学优化问题的软件包。CPLEX提供灵活的高性能优化程序，可以解决线性规划、二次方程规划（Quadratic Programming）、二次方程约束规划（Quadratically Constrained Programming）和混合整型规划（Mixed Integer Programming）问题。它可以用C、C++、JAVA、.NET 等多种计算机语言进行建模，同时拥有自己的优化编程语言，使得初学者很容易根据案例掌握其语法并使用。其中，线性规划（Linear programming，简称LP），是运筹学中研究较早、发展较快、应用广泛、方法较成熟的一个重要分支，它是辅助人们进行科学管理的一种数学方法，也是整数规划、非线性规划的基础。本章利用Cplex求解运输问题、指派问题、最大并发流量问题等三类经典的优化问题。

* 1. 运输问题

已知有个生产地，每个产地的产量为。有个销地，每个销地的销量为。从生产地运输产品到销售地的单位运输成本为。要求总运费最小。决策变量表示从产地运输到销地的运量。

当产销平衡时，此产销平衡的运输问题可以表述为如下数学模型：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，模型目标函数表示最小化总运输成本。第一个约束条件表示销量平衡；第二个约束条件表示产量平衡；第三个约束条件是非负约束。

实际问题中产销往往是不平衡的。例如，有一个销大于产的不平衡的运输问题，其中，每个销地的销量都有上下界。该产销不平衡运输问题可以描述为如下线性规划模型：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，和分别为销地j销量的下界和上界。与模型一样，只有第二个约束条件不一样，因为产量大于销量，所以约束条件是小于等于号，表明产量可能会有剩余。

* 1. 指派问题

已知有*n*个人需要完成*n*项任务，每个人只能完成一个任务，每个任务只能指派一个人去完成。要求完成所有任务花费的总时间最小。指派问题可以描述为如下线性整数规划问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，为0-1变量，如果，则指派第*i*个人去完成第*j*项任务；否则，不指派第*i*个人去完成第*j*项任务。模型的目标函数为最小化完成所有任务所需的总时间。第一和第二个约束条件表示为每个任务只能被一个人完成，每个人只能完成一个任务。第三个约束条件表示决策变量是0-1变量。

* 1. 最大并发流量问题
     1. 基于路段的模型

在一个具有多个起点和多个讫点的网络中，定义和分别为网络中的节点集合和路段集合，为网络中的目的地集合。为离开节点的路段集合，为进入节点的路段集合。最大并发流量问题（Maximum Current Flow Problem）可以描述为如下线性规划问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，为交通需求水平，为节点到节点的基准交通需求，为路段的通行能力,为路段上去往节点的交通流量，向量。

模型假设任意两个节点之间的交通需求随着其基准交通需求等比例变化（即比例为）。模型目标函数最大化交通需求水平，也就是最大化网络总的交通流量。第一个约束条件为节点的流量平衡约束，该约束条件考虑任意节点，进入该节点和该节点产生的去往目的地的交通流量等于离开该节点去往目的地的交通流量。第二约束条件表示路段通行能力限制。第三个约束条件为流量非负约束。

* + 1. 基于路径的模型及列生成算法\*

令为网络中的OD对集合，为OD对之间的路径集合，集合。最大并发流量问题可以描述为如下线性规划问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，为OD对之间的基准交通需求。为0-1变量，如果路段在路径上，则；否则，。为路径上的流量。向量**。**

线性规划问题只有在路径集给定的情况下才能够采用Cplex进行求解。然而，对于大规模网络而言，是没有办法能够事先穷举所有可能分配到流量的路径集。列生成方法很好地解决了该难题。

线性规划问题的对偶问题为：

|  |  |  |
| --- | --- | --- |
|  |  |  |

令为线性规划问题的最优解，和为线性规划问题的最优解和最优目标函数值。依据对偶定理，线性规划问题和具有相等的最优目标函数值，即有。假设存在一个OD对，有一条连接该OD对的路径使得下式成立：

|  |  |  |
| --- | --- | --- |
|  |  |  |

当我们把路径添加到路径集中以后，重新求解线性规划问题和。依据对偶定理，它们具有相同的最优目标函数值，设为。对于线性规划问题而言，把路径添加到路径集中等价于在模型中添加如下约束条件：

|  |  |  |
| --- | --- | --- |
|  |  |  |

线性规划问题在添加约束条件以后，式意味着原最优解因为违背约束条件而不再是该线性规划问题最优解。添加新约束使得可行域变小，进而导致线性规划问题的最优目标函数值变小，即。这意味着把路径添加到路径集中可以使得线性规划问题的最优目标函数值减少。

综上所述，可以形成列生成算法的基本步骤：（1）为每个OD对给定初始路径集；（2）利用Cplex求解线性规划问题，并可以得到前两组约束条件的拉格朗日乘子；（3）把网络中任意路段的权值设置为；（4）求得任意OD对之间的最短路径，如果条件成立，则把路径添加到OD对的路径集中；（5）重复执行步骤（2）-（4）直至没有新的路径可以添加到路径集中。

采用列生成算法求解最大并发流量问题的详细算法步骤如下：

|  |
| --- |
| **算法5.1：**列生成算法求解最大并发流量问题 |
| **输入:** 精度 |
| 1: 对任意路段，置 //初始最短路为经过路段最少的路径  2: 置为路段权值，对任意置  3: 求任意OD对之间的最短路径，并置  4: **while** 至少添加一条新路径 **do** **//**整个路径集有新路径生成  5: 利用Cplex求解线性规划问题得到和  6: 置为路段权值  7: **for each** **do**  8: 求OD对之间的最短路径  9: 如果成立，则置  10: **end for**  11: **end** **while** |
| **输出：** |

我们在判断条件中添加的目的是为了消除计算机的数值误差，避免重复添加已经存在路径集中的路径。对偶变量的理论值为非负。然而由于计算机数值误差的原因，Cplex可能会输出的值可能是负的。在第6行设置路段权重时，如果，则设置路段的权值为0。

* + 1. 等价的基于路径的模型\*

有如下线性规划问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

**定理5.1**. 令和分别为线性规划问题和的最优解，我们有。

**证明：**通过检查约束条件，我们很容易证明和分别为线性规划问题和的可行解。依据定义，我们有和。这意味着。■

定理5.1意味着我们可以通过求解线性规划问题来实现最大并发流量问题的求解。与线性规划问题相同，线性规划问题只有在路径集给定的情况下才能够采用Cplex进行求解。我们给出线性规划问题的对偶问题如下：

|  |  |  |
| --- | --- | --- |
|  |  |  |

令为线性规划问题的最优解，和为线性规划问题的最优解和最优目标函数值。依据对偶定理，线性规划问题和具有相等的最优目标函数值，即有。假设存在一个OD对，有一条连接该OD对的路径使得式成立，则把路径添加到路径集中以后，重新求解线性规划问题和。依据对偶定理，它们具有相同的最优目标函数值。类似于算法5.1，我们可以设计列生成算法求解线性规划问题，从而实现最大并发流量问题的求解。

* 1. 最小费用最大并发流量问题
     1. 基于路段的模型

最小费用最大并发流量问题（Minimum Cost Maximum Current Flow Problem）可以描述为如下线性规划问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，为路段的单位流量成本。为最高交通需求水平，需要通过求解最大并发流量问题来得到。在给定的情况下，可用调用Cplex直接求解最小费用问题。

* + 1. 基于路径的模型及列生成算法\*

最小费用最大并发流量问题可以描述为如下基于路径的线性规划模型：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，为路径上的单位流量成本。依据定义，我们有。

线性规划问题的对偶问题为：

|  |  |  |
| --- | --- | --- |
|  |  |  |

依据对偶定理，线性规划问题和具有相等的最优目标函数值。类似于算法5.1，我们可以设计列生成算法求解线性规划问题，从而实现最小费用最大并发流量问题的求解。

令为线性规划问题的最优解，为线性规划问题的最优解。假设存在一个OD对，有一条连接该OD对的路径使得下式成立：

|  |  |  |
| --- | --- | --- |
|  |  |  |

则把路径添加到路径集中以后，重新求解线性规划问题，直到找不到新的路径可以添加为止。

* 1. 测试问题
     1. 产销平衡运输问题

某公式经销甲产品。它下设三个加工厂。每日的产量分别是：为7吨，为4吨，为9吨。该公司把这些产品分别运往四个销售点。各销售点每日销量为：为3吨，为6吨，为5吨，为6吨。已知从各工厂到各销售点的单位产品的运价为表5.1表示。问该公司应如何调运产品，在满足各销售点的需要量的前提下，使总运费为最少。（来源：运筹学（清华大学出版社第四版）94页例1）

表5.1：产销平衡运输问题运费

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 加工厂 | 销地 | | | |
|  |  |  |  |
|  | 3 | 11 | 3 | 10 |
|  | 1 | 9 | 2 | 8 |
|  | 7 | 4 | 10 | 5 |

**标准答案：**表5.2给出了产销平衡运输问题的最优解，其对应的总运费为85元。

表5.2：产销平衡运输问题的最优解

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 加工厂 | 销地 | | | | 产量 |
|  |  |  |  |
|  |  |  | 5 | 2 | 7 |
|  | 3 |  |  | 1 | 4 |
|  |  | 6 |  | 3 | 9 |
| 销量 | 3 | 6 | 5 | 6 |  |

* + 1. 产销不平衡运输问题

设有三个化肥厂（A，B，C）供应四个地区（I，II，III，IV）的农用化肥。假定等量的化肥在这些地区使用效果相同。各化肥厂年产量，各地区需要量及从各化肥厂到各地区运送单位化肥的运价如表5.3所示。试求出总的运费最节省的化肥调拨方案。（来源：运筹学（清华大学出版社第四版）105页例2）

表5.3：产销不平衡运输问题的运费和产销量

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 化肥厂 | 需求地区 | | | | 产量 |
| I | II | III | IV |
| A | 16 | 13 | 22 | 17 | 50 |
| B | 14 | 13 | 19 | 15 | 60 |
| C | 19 | 20 | 23 | - | 50 |
| 最低需求 | 30 | 70 | 0 | 10 |  |
| 最高需求 | 50 | 70 | 30 | 不限 |  |

**标准答案：**表5.4给出了产销不平衡运输问题的最优解，其对应的总运费为2460。

表5.4：产销不平衡运输问题的最优解

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 化肥厂 | 需求地区 | | | | 产量 |
| I | II | III | IV |
| A |  | **50** |  |  | 50 |
| B |  | **20** |  | **40** | 60 |
| C | **50** | **0** |  |  | 50 |
| 最低需求 | 30 | 70 | 0 | 10 |  |
| 最高需求 | 50 | 70 | 30 | 不限 |  |

* + 1. 指派问题

有一份中文说明书，需译成英、日、德、俄四种文字。分别记作E、J、G、R。现有甲、乙、丙、丁四人。他们将中文说明书翻译成不同语种的说明书所需时间如表5.5所示。问应指派何人去完成何工作，使所需总时间最少？（来源：运筹学（清华大学出版社第四版）第146-147页例7）

表5.5. 指派问题算例的费用矩阵

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 人员 | 任务 | | | |
| E | J | G | R |
| 甲 | 2 | 15 | 13 | 4 |
| 乙 | 10 | 4 | 14 | 15 |
| 丙 | 9 | 14 | 16 | 13 |
| 丁 | 7 | 8 | 11 | 9 |

**标准答案：**表5.6给出了指派问题的最优解，完成所有任务所需的总时间为28小时。

表5.6. 指派问题的最优解

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 人员 | 任务 | | | |
| E | J | G | R |
| 甲 | 0 | 0 | 0 | 1 |
| 乙 | 0 | 1 | 0 | 0 |
| 丙 | 1 | 0 | 0 | 0 |
| 丁 | 0 | 0 | 1 | 0 |

* + 1. 最大并发流量问题

多OD网络最大流问题采用Sioux Falls网络和Chicago Sketch网络为测试网络（通过<http://www.bgu.ac.il/~bargera/tntp/>获取网络数据）。把网站上各网络给出的OD需求设置为基准交通需求。采用Cplex求解模型，可以直接得到最优的目标函数值分别为0.5233和0.4204。设置精度参数，基于模型和，采用列生成的方法也可以得到相同的结果。表5.7给出了不同的方法求解所需的CPU时间。可以看出基于路段的模型可以适用于小规模的网络，而基于路径的模型更加适用于大规模网络。

表5.7. 最大并发流量问题的求解效率

|  |  |  |
| --- | --- | --- |
| 网络 | Sioux Falls (s) | Chicago Sketch (s) |
| 基于路段的模型 | 0.24 | 128.03 |
| 基于路径的模型 | 0.30 | 32.06 |
| 基于路径的模型 | 0.28 | 2.83 |

* + 1. 最小费用最大并发流量问题

我们仍然采用Sioux Falls网络和Chicago Sketch网络来进行测试。把路段的自由流走行时间设为路段的单位流量成本，利用列生成方法求解基于路径的模型来得到网络的最高交通需求水平。进一步采用Cplex求解最小费用问题，可以分别得到两个网络的目标函数值（取最大流时网络的最小费用）为1832884.92和6765830.84。设置精度参数，基于模型，采用列生成的方法也可以得到相同的结果。表5.8给出了不同的方法求解最小费用最大并发流量问题所需的CPU时间（包含求解最大并发流量问题的时间）。可以看出基于路段的模型可以适用于小规模的网络，而基于路径的模型更加适用于大规模网络。

表5.8. 最小费用最大并发流量问题的求解效率对比

|  |  |  |
| --- | --- | --- |
| 网络 | Sioux Falls (s) | Chicago Sketch (s) |
| 基于路段的模型 | 0.47 | 217.46 |
| 基于路径的模型 | 0.45 | 21.78 |

# 最短路问题

* 1. 概述

最短路问题（Shortest Path Problem）是网络理论解决的典型问题之一，可用来解决管道铺设、线路安装、厂区布局和设备更新等实际问题。最短路问题就是：在赋权（即边权，如长度、成本、时间等）网络中，寻找两个节点之间总赋权之和最小的路径。依据网络是否有环和是否有负的赋权，最短路问题可以定义在不同的网络上面，并且求解算法也不尽相同。依据计算内容，最短路问题可以分为：（1）单点对单点最短路问题、（2）单源最短路问题（同时计算起始点到所有节点的最短路或者所有节点到终点的最短路）、（3）所有起终点对（All Pair）的最短路问题。

标号设置（Label-setting）算法和标号修正（Label-correcting）算法是应用比较广泛的两类求解单源最短路问题的算法。Label-setting与Label-correcting的区别在于：Label-setting逐步设定起始点至每个节点的最短路径长度值，一旦设定后就不再更改；而Label-correcting则逐步改进起始点至每个节点的最短路径长度值，直到起始点至所有节点的最短路径长度值不能改进为止。需要注意的是，应用Label-setting算法需满足边权非负的条件，而Label-correcting算法不受该限制。事实上，我们可以将Label-setting算法视作Label-correcting算法的一个特例。

以上两种算法均是求解单一起点或者单一终点的单源最短路问题。此外，还有求解所有起终点对的最短路算法。感兴趣的读者可以查阅Floyd–Warshall算法（Floyd, 1962），该算法适用的网络的边可以有正权、负权，但是不能存在负环；关于负环的内容可以参考Ahyja等（1993）。

* 1. 单点-单点最短路问题的数学模型

在一个具有多个起点和多个讫点的网络中，定义和分别为网络中节点的集合和弧的集合。令和分别为网络的节点数和弧的数量。令表示离开节点的弧的集合，表示进入节点的弧的集合，表示弧的权重。从起点到终点的单点对单点最短路问题可以描述为如下线性整数规划问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

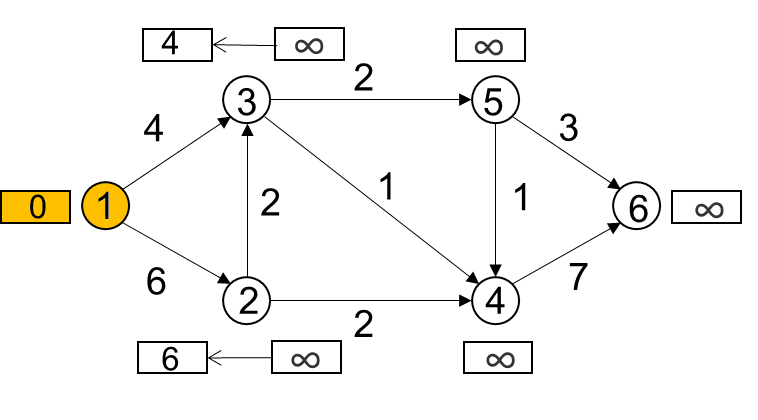
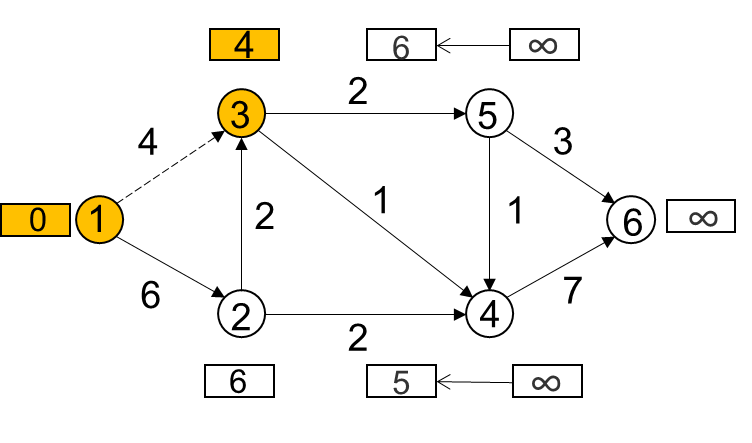
其中，为0-1决策变量。如果，则起点到终点的最短路经过弧；否则，起点到终点的最短路不经过弧。模型的第一个约束条件限制了起点离开的弧有且只有一条，到达终点的弧有且只有一条，因此该式规定了从起点到终点的一条完整路径。模型第二个约束条件中的为该路径经过的点集，*n*为总的节点数量，该式规定了任意节点在路径上不可被重复访问，该路径不存在环。当所有弧具有正的权重时，模型的第二个约束条件可以去掉。

* 1. 标号设置算法
     1. Dijkstra算法

本节采用Dijkstra（迪杰斯特拉）算法来说明Label-setting算法。该算法是由荷兰计算机科学家Dijkstra于1959年提出的。Dijkstra算法针对无负边权网络，基于广度优先搜索的思想，以起始点为中心向外层层扩展，直到扩展到终点为止。在Dijkstra算法的计算过程中，网络中存在两类节点：P标号节点和T标号节点。其中，P标号节点是已经确认得到最短路径的节点，P标号记录了起始节点到该节点的最短路径；T标号节点则是还没有确认得到最短路径的节点， T标号记录了起始节点到该节点已知的最短路径。Dijkstra算法的基本步骤是：（1）初始化所有节点为T标号节点，设置起始节点的T标号为0，其它节点的T标号为正无穷；（2）从所有T标号节点中选取T标号最小的节点设置为P标号节点，然后基于该P标号节点更新其相邻节点的T标号；（3）重复步骤（2）直至终点被设置为P标号节点。

算法6.1总结了从起始点到所有节点的单源最短路问题的Dijkstra算法。算法中，表示节点是P标号节点还是T标号节点。当时，节点是P标号节点；否则，节点是T标号节点。表示起点至节点的已知最短距离，表示在已知最短路上节点的前继节点（即通过哪个邻居节点到节点的）。是T标号节点构成的集合。第4和5行从所有T标号节点中选取T标号最小的节点设置为P标号节点，第6-10行则基于当前的P标号节点更新其相邻节点的T标号。图6.1给出了应用Dijkstra算法求解最短路的一个算例，展示了P标号节点和T标号节点的动态更新过程。

|  |
| --- |
| **算法6.1 Dijkstra** () // Dijkstra算法 |
| **输入：**起点  1: 对所有节点，置 //初始化所有节点为T标号节点，起点到其距离 |
| 2: 置，和 //起点没有前继节点（），且到其自身的最短距离为0  3: **for** **do** //次循环可以计算完 |
| 4: 置 //表示起点至节点的已知最短距离, 为T标号最小的点 |
| 5: 置 //把节点*i*设置为*P*标号 |
| 6: **for each** //为离开节点的弧的集合， |
| 7: **if** **then**  //为对应弧的权重，P标号节点不用考虑 |
| 8: 置和 //标记节点在最短路上的前继节点 |
| 9: **end if** |
| 10: **end for** |
| 11: **end for**  **输出：**最短路长度向量和前继节点向量 //前继节点向量用于提取最短路径 |

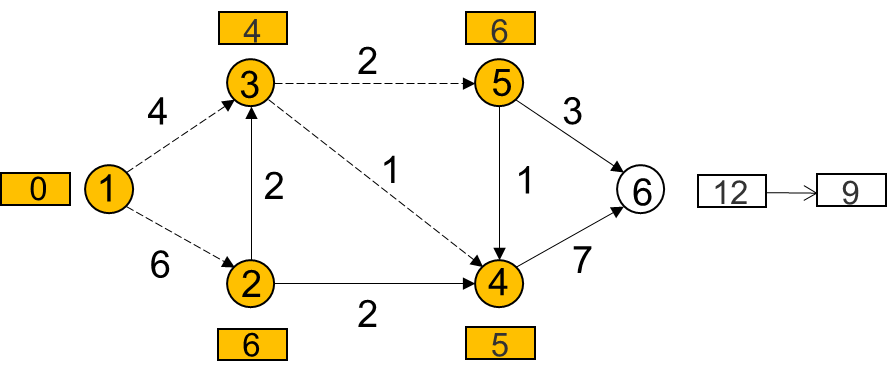
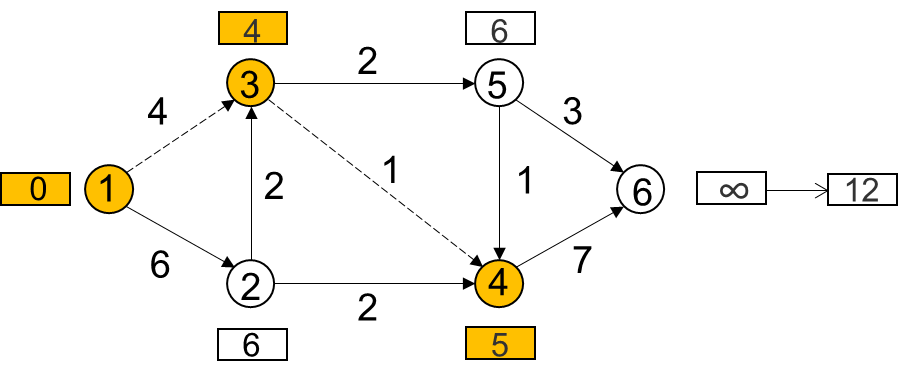
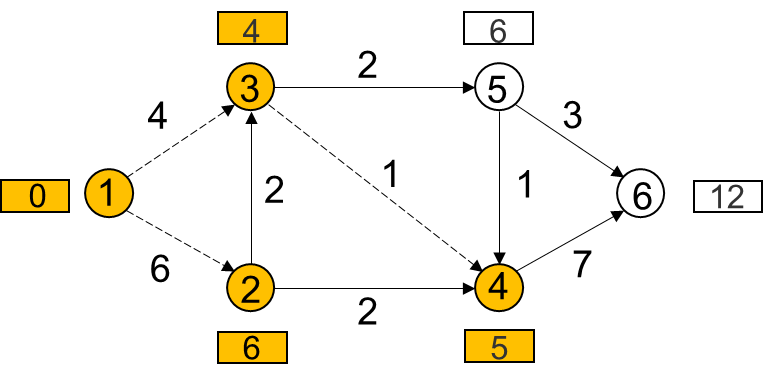
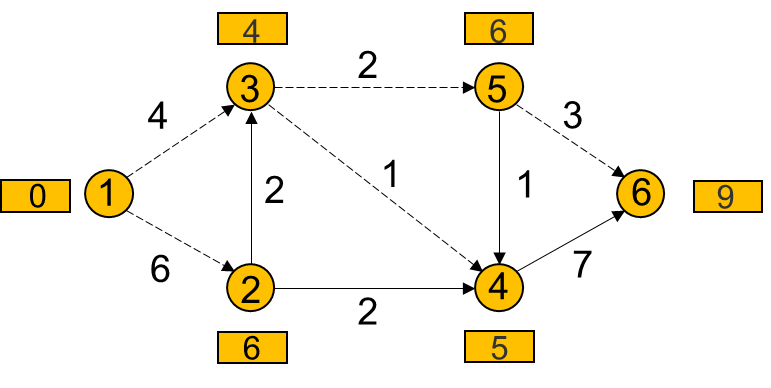
 

图6.1 Dijkstra算法示例。

在算法6.1的执行过程中，需要避免直接采用邻接矩阵的数据结构。这是因为在邻接矩阵的数据结构下，算法第6-10行的循环需要扫描网络中的所有节点。对于大规模网络而言，这种扫描方式是非常耗时的。我们可以采用表7.1中的数据结构来表示一个网络。

表7.1. 表示网络的数据结构

|  |  |  |
| --- | --- | --- |
| 数据类型 | 变量名称 | 变量说明 |
| int | m\_nNode | 节点数 |
| int | m\_nLink | 路段数量 |
| int[] | CumOutgoingLinkIndex | 每个节点（不含自己）累计离开的路段数量，维度为m\_nNode+1 |
| int[] | LinkOutNodeID | 路段的出节点，维度为m\_nLink |
| int[] | LinkInNodeID | 路段的入节点，维度为m\_nLink |
| double[] | LinkCost | 路段权重，维度为m\_nLink |

我们以图6.2中的网络来说明如何采用以上数据结构来对网络进行编码。该网络有6个节点，10条弧，每条弧上面标注了弧的权重。因此，我们有m\_nNode=6，m\_nLink=10。表7.1给出了CumOutgoingLinkIndex、LinkOutNodeID、LinkInNodeID、LinkCost等变量的详细取值。对于任意节点，[CumOutgoingLinkIndex[], CumOutgoingLinkIndex[])表示从该节点离开的路段集合。



图6.2. 示例网络。

表6.2. 示例网络的详细编码。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array index | CumOutgoingLinkIndex | LinkInNodeID | LinkOutNodeID | LinkCost |
| 0 | 0 | 0 | 1 | 5 |
| 1 | 2 | 0 | 2 | 3 |
| 2 | 5 | 1 | 2 | 1 |
| 3 | 7 | 1 | 3 | 2 |
| 4 | 8 | 1 | 4 | 4 |
| 5 | 10 | 2 | 3 | 1 |
| 6 | 10 | 2 | 4 | 4 |
| 7 | - | 3 | 5 | 6 |
| 8 | - | 4 | 3 | 3 |
| 9 | - | 4 | 5 | 2 |

* + 1. T标号排序的Dijkstra算法

在实际的编程计算过程中，算法7.1的第4行的计算是一个非常耗时间的计算步骤。这是因为计算需要对所有节点进行扫描比较。为了提高算法的计算效率，可以在算法执行过程中保持集合中的节点始终按照T标号从小到大的顺序排列。在T标号排序的情况下，算法第4行中的节点取中的第一节点即可。此外，在执行完算法第8行以后，还需要维护T标号排序。具体做法是：先把节点从中删除，然后采用二分法查找节点在中的插入位置，最后把节点插入到中去。

* + 1. T标号堆排序的Dijkstra算法

为了提高算法6.1的第4行寻找T标号最小的节点的效率，我们可以依据T标号大小构造T标号节点的小顶堆。于是，小顶堆的根节点就是T标号最小的节点。在算法执行过程中，当T标号节点被设置为P标号节点（算法6.1的第5行）时，小顶堆的根节点被拿掉了，此时需要把尾部T标号节点提到根节点来，并采用一个T标号下沉算法来维护小顶堆。当节点的T标号得到更新（算法6.1的第8行）时，也需要对当前的小顶堆进行维护。由于T标号的更新只会使得T标号变小。当T标号变得比其父节点的T标号还小时，需要对T标号进行上浮。

令为当前T标号节点构成的数组，为数组的长度（即T标号节点的数量）。令表示节点在数组中位置，依据定义，我们有和。定义向量。类似于算法2.4，算法6.2和6.3分别给出了一个T标号节点下沉算法和T标号节点上浮算法来维持给定节点的T标号小于其孩子节点。

|  |
| --- |
| **算法6.2 AdjustDown\_TNode**(, **,** , ) // T标号下沉 |
| **Input:** 数组，向量，位置，结束位置 //为父节点位置 |
| 1: 置 //为孩子节点位置 |
| 2: **while** **do** |
| 3: **if** 且 // 找两个孩子节点中T标号小的  4: 置  5: **end if**  6: **if** // 如果父节点比两个子节点都小，就不用动了  7: Break  8: **end if**  9: 置  10: 置和 // 和做交换，交换子节点和父节点的数值  11: 置和 // 更新T标号节点在T标号节点数组中的位置  12: 置 // 把父节点设置为子节点  13: 置 // 更新为对应的子节点  14: **end while** |
| **Output:** 更新后的数组和向量 |

|  |
| --- |
| **算法6.3 AdjustUp\_TNode**(, , ) // T标号上浮 |
| **Input:** 数组，向量，位置 // 孩子节点 |
| 1: 置 // 获取父节点 |
| 2: **while** **do** |
| 3: **if** // 如果父节点大与孩子节点，则交换；否则跳出循环  4: 置  5: 置和 // 和做交换，交换孩子节点和父节点的数值  6: 置和 // 更新T标号节点在T标号节点数组中的位置  7: 置 // 把孩子节点设置为父节点  8: 置 // 更新为对应的父节点  9: **else**  10: Break  11: **end if**  12: **end while** |
| **Output:** 更新后的数组和向量 |

算法6.4给出了T标号堆排序的Dijkstra算法来计算从起始点到所有节点的单源最短路。为了提高堆排序的效率，算法中的T标号节点只考虑已经搜索到节点。算法第4行提取小顶堆的顶点设置为当前的P标号节点。算法第8-10行把新探测到的节点加到堆的尾部。当节点的T标号得到更新时，算法第11行采用T标号节点上浮算法6.3来维护小顶堆。算法第14行把堆的顶点拿掉（拿掉前顶点是当前的P标号节点）。

|  |
| --- |
| **算法6.4 Dijkstra\_T\_Heap**() // T标号堆排序的Dijkstra算法 |
| **输入：**起点 |
| 1: 对所有节点，置和 //表示节点是否被探测到了  2: 置，，，，和 //初始时，起点是小顶堆唯一的节点  3: **for** **do**  4: 置 //提取小顶堆的顶点设置为*P*标号 |
| 5: **for each** |
| 6: **if** **then** |
| 7: 置和  8: **if** **then** //节点第一次探测到  9: 置，，和 //新探测到的点，添加到堆的尾部  10: **end if**  11: **AdjustUp\_TNode**(, , ) //节点更新了，调用算法6.3更新堆 |
| 12: **end if**  13: **end for**  14: 置，和 **//**把顶拿掉，补上最后一个元素，堆长减1  15: **AdjustDown\_TNode**(, **,** , ) //调用算法6.2，剩余的数据重新形成小顶堆 |
| 16: **end for**  **输出：**最短路长度向量和前继节点向量 |

* 1. 标号修正算法

算法6.5给出了广义的Label-correcting算法。该算法可以用来求解含负权边网络上的最短路问题。在算法执行过程中，当一个节点已知的最短路得到更新时，则其能探测到的节点的最短路也有可能被更新。这样的节点就成为了需要进一步检查的节点。Label-correcting算法采用一个队列来记录需要进一步检查的节点。算法第4行从队列中取出一个节点来检查其能探测到的节点的最短路是否能够得到更新。第8-10行表明如果某个节点的最短路得到更新，并且该节点不在队列中时，需要把节点添加到队列中。需要注意的是，中的节点是有顺序的，不同的算法的区别在于待检查的节点的插入位置不同。第9行中，节点有两种插入方法，可以得到两类Label-correcting算法：

* 单队列尾部Label-correcting算法：添加到队列的尾部。
* 单队列混合Label-correcting算法：如果节点是第一次探测到，则添加到队列的尾部；否则，添加到队列的头部。

在Label-correcting算法中，对于任意节点，如果不是真正的最短距离，则算法第4-9行的计算是没有实际意义的。这是因为节点找到真正的最短距离以后，第6-11行还需要执行一遍，以确认通过任意弧，节点的最短距离是否可以得到进一步的更新。在实际应用中方法二比方法一具有更高的计算效率。这是因为首次探测到的节点正好得到最短路的可能性相比于已经探测过的点要低。而方法二是优先检查非首次探测到的节点。

|  |
| --- |
| **算法6.5 LabelCorrecting** () // Label-correcting算法 |
| **输入：**起点  1: 对所有节点，置 //初始化起点到所有节点的距离为 |
| 2: 置，和 //初始时只有起点在队列里面  3: **while**  //队列为空的时候，算法终止 |
| 4: 选择一个节点，并置 //通常取中的第一个节点。 |
| 5: **for each** |
| 6: **if** **then** |
| 7: 置和 |
| 8: **if** **then** //得到更新的节点不在队列里面  9: 置 //不同的算法的区别在于插入队列的位置不同  10: **end if**  11: **end if** |
| 12: **end for** |
| 13: **end while**  **输出：**最短路长度向量和前继节点向量 |

基于算法6.5给出的是一个单队列Label-correcting算法，我们还可以得到一个双队列Label-correcting算法（见算法6.6）。在算法6.6中，采用了和两个队列。其中，中存放非首次探测到的待检查节点，存放首次探测到的待检查节点。算法第4-8行表明应优先从队列中选取待检查节点去做更新。第12-18行表明，如果某个节点的最短路得到更新，并且该节点不在队列或者中时，需要把节点添加到队列中。如果是首次探测到，则添加到队列的尾部；否则，添加到队列的尾部。双队列的规则优先检查队列中的节点。这是因为首次探测到的节点正好得到最短路的可能性相比于已经探测过的点要低。

|  |
| --- |
| **算法6.6 LabelCorrecting\_2Q** () //双队列Label-correcting算法 |
| **输入：**起点  1: 对所有节点，置和 |
| 2: 置，，，和 //放第一次探测到的节点  3: **while**  //两个队列都为空的时候，算法终止 |
| 4: **if** **then**  5: 选择一个节点，并置 //应用中，取中的首节点。  6: **else**  7: 选择一个节点，并置 //应用中，取中的首节点。  8: **end if** |
| 9: **for each** |
| 10: **if** **then** |
| 11: 置和 |
| 12: **if** **then** //得到更新的节点不在队列里面，则添加到队列里去  13: **if** **then** //首次探测到的节点放到的尾部  14: 置和  15: **else** //否则放到的尾部  16: 置  17: **end if**  18: **end if**  19: **end if** |
| 20: **end for** |
| 21: **end while**  **输出：**最短路长度向量和前继节点向量 |

* 1. 单点-单点最短路算法

考虑在网络中计算从起点到终点的最短路。我们可以分别基于Label-setting算法和Label-correcting算法来计算单点-单点的最短路。

* + 1. Dijkstra算法

我们只需要对Dijkstra算法稍加改造就可以有效地计算单点-单点最短路。在Dijkstra算法的执行过程中，一旦终点被设置为P标号节点，算法就可以终止了。此时，已经找到了起点到终点的最短路。

* + 1. 双向Dijkstra算法

双向Dijkstra算法分别从起点到终点出发各自使用单向Dijkstra算法，依次将距离起点最近的节点和距离终点最近的节点分别设为前向和后向P标号节点。双向Dijkstra算法实施的关键点在于：（i）如何更新起点到终点最短路长度的上界，（ii）如何终止算法。

在算法执行过程中，令和分别表示起点到节点以及节点到终点的已知最短路长度。令和分别表示前向P标号节点构成的集合和后向标号节点构成的集合。则可以构成起点到终点最短路长度的上界。基于此条件，当某个节点的前向（后向）T标号有更新时，我们可以先检查一下该节点是否是后向（前向）P标号节点。如果是的，可以尝试更新起点到终点的最短路长度的上界。我们有如下命题：

**命题6.1**：考虑一个只具有正权的网路，采用Dijkstra算法做双向搜索。令和分别表示当前前向和后向P标号节点。如果成立，则通过弧中继可以得到起点到终点的最短路。

**证明**：令为起点到终点最短路。依据定义，我们有和。我们考虑两种情况：（i）最短路上的所有节点都是当前前向P标号节点或者后向P标号节点，（ii）存在一个节点既不是当前的前向P标号节点，也不是当前的后向P标号节点。对于情况（i），存在，使得节点为路径上距离起点最远的前向P标号节点，并且有（即后向P标号节点）。于是，最短路径的长度。依据最短路定义，通过弧中继可以得到起点到终点的最短路。

对于情况（ii），最短路径上存在一个节点，既不是当前的前向P标号节点，也不是当前的后向P标号节点。令和分别为起点到节点以及节点到终点的最短路距离。依据定义，我们有和。最短路径的长度。这与为起点到终点最短路矛盾。因此，情形（ii）不成立。命题得证。■

命题6.1给出了双向Dijkstra算法的终止条件：，这时可以得到了起终点间的最短路。算法6.8给出了双向Dijkstra算法的详细计算过程。算法第6-13行基于起点做正向搜索，第14-21行基于终点做反向搜索，第22-24行给出了算法终止条件。当起点到终点最短路长度的上界得到更新时，我们需要把最优中继节点记录下来（第13和20行），以便取出最短路径。

|  |
| --- |
| **算法6.8 BiDirection\_Dijkstra** () // 双向Dijkstra算法 |
| **输入：**起点  1: 对所有节点，置和 //和分别表示前向搜索和后向搜索 |
| 2: 置，，，和  3: **for** **do** |
| 4: 置和 //双向分别搜索 |
| 5: 置，，和 //设置*P*标号 |
| 6: **for each** **do** //正向搜索 |
| 7: **if** **then** |
| 8: 置和 |
| 9: **if** 且 **then** //  10: 置和 //更新最优中继节点  11: **end if**  12: **end if**  13: **end for**  14: **for each** **do** //反向搜索 |
| 15: **if** **then**  16: 置和  17: **if** 且 **then** //  18: 置和 //更新最优中继节点  19: **end if**  20: **end if**  21: **end for**  22: **if** **then** //算法/终止条件  23: **Break**  24: **end if** |
| 25: **end for**  **输出：**最短路长度，最优中继节点，前继节点向量和后继节点向量 |

* + 1. 标号修正算法

我们也只需要对Label-correcting算法稍加改造就可以有效地计算单点-单点最短路。在Label-correcting算法的执行过程中，一旦终点被探测到，此时更新得到的就是起点到终点最短路长度的上界。当起点到任意节点已知最短路长度得到更新时，如果更新后的最短路长度超过上界，则该节点不需要加入待检查的队列里面。当待检查的队列为空时算法终止。算法6.7给出了基于单队列Label-correcting算法的单点-单点的最短路算法。该算法与算法6.5不同之处在于算法的第6行。这一行要求待更新的已知最短距离不能超过起点到终点最短路长度的上界。类似算法6.7，也可以构造基于双队列Label-correcting算法的单点-单点的最短路算法。

|  |
| --- |
| **算法6.7 One2OneLabelCorrecting** (,) //单点-单点 Label-correcting算法 |
| **输入：**起点  1: 对所有节点，置 //初始化起点到所有节点的距离为 |
| 2: 置，和  3: **while**  //队列为空的时候，算法终止 |
| 4: 选择一个节点，并置 |
| 5: **for each** |
| 6: **if** 且 **then** //待更新的已知最短距离不能超过上界 |
| 7: 置和  8: **if** **then** //终点节点不会加入到待检查的队列里面  9: 置  10: **end if**  11: **end if** |
| 12: **end for** |
| 13: **end while**  **输出：**最短路长度和前继节点向量 |

* + 1. 双向标号修正算法

双向Label-correcting算法分别从起点到终点出发各自使用单向Label-correcting算法。依次将距离起点最近的节点和距离终点最近的节点分别设为前向和后向P标号节点。双向Dijkstra算法实施的关键点在于：（i）如何更新起点到终点最短路长度的上界，（ii）如何终止算法。

命题6.1给出了双向Dijkstra算法的终止条件：，这时可以得到了起终点间的最短路。算法6.8给出了双向Dijkstra算法的详细计算过程。算法第6-13行基于起点做正向搜索，第14-21行基于终点做反向搜索，第22-24行给出了算法终止条件。当起点到终点最短路长度的上界得到更新时，我们需要把最优中继节点记录下来（第13和20行），以便取出最短路径。

|  |
| --- |
| **算法6.8 BiDirection\_LabelCorrecting** () // 双向Dijkstra算法 |
| **输入：**起点  1: 对所有节点，置和 //和分别表示前向搜索和后向搜索 |
| 2: 置，，，和  3: **for** **do** |
| 4: 置和 //双向分别搜索 |
| 5: 置，，和 //设置*P*标号 |
| 6: **for each** **do** //正向搜索 |
| 7: **if** **then** |
| 8: 置和 |
| 9: **if** 且 **then** //  10: 置和 //更新最优中继节点  11: **end if**  12: **end if**  13: **end for**  14: **for each** **do** //反向搜索 |
| 15: **if** **then**  16: 置和  17: **if** 且 **then** //  18: 置和 //更新最优中继节点  19: **end if**  20: **end if**  21: **end for**  22: **if** **then** //算法/终止条件  23: **Break**  24: **end if** |
| 25: **end for**  **输出：**最短路长度，最优中继节点，前继节点向量和后继节点向量 |

* 1. 有向无环网络上的最短路问题
     1. 拓扑排序

对一个有向无环图进行拓扑排序，是将中所有顶点排成一个序列，使得图中任意两个顶点，如果弧，则在序列中出现在之前。如果在序列中出现在之前，则不可能存在一条从到路径。通常，这样的序列称为满足拓扑次序（Topological Order）的序列，简称拓扑序列。需要注意的是拓扑序列并不一定是唯一的。

我们把进入节点的弧的个数称为该节点的出度（即集合中元素的个数），把离开节点的弧的个数称为该节点的出度（即集合中元素的个数）。算法6.7给出了一个拓扑排序算法。该算法的基本过程是：挑选网络中入度为0的节点，通过删除离开该节点的弧来降低对应节点的入度，重复该操作直到所有节点的入度都变成0为止。依据入度变成0的先后次序，对节点进行排序就得到了拓扑排序。算法第1-6行为初始化阶段，得到原始网络中各节点的入度，并把入度为0的节点记录下来。算法第10-14行降低节点的入度，并在降低过程中记录入度降到0的节点。应用拓扑排序算法6.7对网络节点进行拓扑排序，排序的复杂度为。

|  |
| --- |
| **算法6.7** TopologicalSort () // 拓扑排序算法 |
| 1: 置， //表示当前确定了拓扑序的节点个数 |
| 2: **for each** **do**  3: 置 //初始化节点的入度  4: **if** **then**  5: //把入度为0的节点加到队列中  6: **end if**  7: **while**  //队列为空的时候，算法终止 |
| 8: 选择一个节点，并置 //选取入度降到0的节点 |
| 9: 置，和 //节点的拓扑序号为，拓扑排序为的节点  10: **for each** |
| 11: //降低节点的入度  12: **if** **then** |
| 13: 置 //如果节点的入度降低到0，则添加节点到队列中 |
| 14: **end if** |
| 15: **end for** |
| 16: **end while**  **输出：**拓扑序向量和 |

图6.3给出了一个5个节点的有向无环图。表6.3展现了采用算法6.7进行拓扑排序的过程。依据6.3中选取中节点的次序，我们可以得到网络中节点的拓扑排序为：4, 3, 1, 6, 2, 5（即，，，，）。在第2步执行完入度降低后，我们也可以选取节点来执行下一步入度降低。基于该选择，我们将得到另外一个节点的拓扑序：4, 3, 6, 1, 2, 5。



图6.3一个有向无环图。

表6.3. 示例网络的拓扑排序过程。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 迭代次数 |  |  |  |  |  |  |  | 选取节点 |
| 0 | 1 | 2 | 1 | 0 | 3 | 2 |  | 4 |
| 1 | 1 | **1** | **0** | 0 | 3 | **1** |  | 3 |
| 2 | **0** | 1 | 0 | 0 | **2** | **0** |  | 1 |
| 3 | 0 | 1 | 0 | 0 | **1** | 0 |  | 6 |
| 4 | 0 | **0** | 0 | 0 | **1** | 0 |  | 2 |
| 5 | 0 | 0 | 0 | 0 | **0** | 0 |  | 5 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 |  | - |

* + 1. 拓扑排序下的最短路算法

考虑一个有向无环网络，利用算法6.7进行拓扑排序。令表示节点的拓扑序号，表示拓扑排序为的节点。对于有向无环网络，在Dijkstra算法框架下，计算任意给定节点的单源最短路，只需要从拓扑序为的节点（即起点）开始，拓扑序为的节点结束，按照拓扑排序，依次把节点设置为P标号节点。算法6.8给出了有向无环网络的最短路算法。该算法可以同时计算起点到所有节点的最短路，其时间复杂度为。算法第4行直接把当前拓扑序对应的节点设置为当前的P标号节点。这个求解最短路的方法特别适合在以最短路为子问题的迭代优化算法中。在这类问题中，提前将拓扑序存贮起来，这样每次迭代时调用最短路算法所需的时间仅为，为常数级的复杂度，效率非常高。

|  |
| --- |
| **算法6.8 Dijkstra\_TopologicalOrder** () //有向无环网络最短路算法 |
| **输入：**起点  1: 对所有节点，置 |
| 2: 置和  3: **for** **do** //依据拓扑序顺序更新即可 |
| 4: 置 //把设置为当前P标号节点 |
| 5: **for each** |
| 6: **if** **then** |
| 7: 置和 |
| 8: **end if** |
| 9: **end for** |
| 10: **end for**  **输出：**最短路长度向量和前继节点向量 |

* 1. 测试问题
     1. Label-setting算法和Label-correcting算法

采用Sioux Falls网络**（详见附录A，B）**，基于自由流走行时间，计算如下OD对间的最短路：1→20；7→13；2→24。表7.2给出了**参考答案。**

表7.2. Sioux Falls网络上的最短路径及其成本

|  |  |  |
| --- | --- | --- |
| OD对 | 最短路径 | 路径成本 |
| 1→20 | 1-2-6-8-7-18-20 | 22 |
| 7→13 | 7-18-20-21-24-13 | 19 |
| 2→24 | 2-1-3-12-13-24 | 21 |

* + 1. 有向无环网络Label-correcting算法

图6.4给出了一个包含8个节点、13条弧的无环有向网络。请给该网络节点的拓扑排序。参考的排序结果为：5, 1, 3, 6, 4, 7, 0, 2。



图6.4 拓扑排序测试网络

采用图6.4所示有向无环网络，路段权重如下，请给出节点5到所有点的最短路路径及其成本。表6.4给出了网络中各路段的成本。表6.5给出了参考答案。

表6.4. 有向无环网络的路段成本

|  |  |  |
| --- | --- | --- |
| **路段的入节点** | **路段的出节点** | **路段成本** |
| 4 | 5 | 0.35 |
| 5 | 4 | 0.35 |
| 4 | 7 | 0.37 |
| 5 | 7 | 0.28 |
| 7 | 5 | 0.28 |
| 5 | 1 | 0.32 |
| 0 | 4 | 0.38 |
| 0 | 2 | 0.26 |
| 7 | 3 | 0.39 |
| 1 | 3 | 0.29 |
| 2 | 7 | 0.34 |
| 6 | 2 | -1.2 |
| 3 | 6 | 0.52 |
| 6 | 0 | -1.4 |
| 6 | 4 | -1.25 |

表6.5. 有向无环网络上的最短路径及其成本

|  |  |  |
| --- | --- | --- |
| **目的地节点** | **最短路径** | **路径成本** |
| 0 | 5-1-3-6-0 | -0.27 |
| 1 | 5-1 | 0.32 |
| 2 | 5-1-3-6-2 | -0.07 |
| 3 | 5-1-3 | 0.61 |
| 4 | 5-1-3-6-4 | -0.12 |
| 5 | 5 | 0 |
| 6 | 5-1-3-6 | 1.13 |
| 7 | 5-1-3-6-4-7 | 0.25 |

* + 1. 最短路算法计算效率的对比

在Chicago Sketch网络（可以到网址<http://www.bgu.ac.il/~bargera/tntp/>下载网络数据）上，基于自由流走行时间，计算所有节点到所有节点的最短路（即以每个节点作为起点计算一次单源最短路问题）。

利用Visual Studio C# 2015编程，在安装了Intel Core i7-7700 3.60-GHz CPU和32.00GB RAM的计算机上运行，得到表6.6中的参考答案。表6.6中的结果表明，相比于面向对象的数据结构，纯数组的数据结构具有更高的代码执行效率。但是，需要注意的是，面向对象的数据结构具有更好的编程逻辑。

**表6.6.** Chicago Sketch网络上最短路计算效率的对比（C#）

|  |  |  |
| --- | --- | --- |
| 算法 | 面向对象的数据结构（s） | 纯数组的数据结构（s） |
| Label-setting算法（算法7.1） | 3.2157 | 2.7278 |
| T标号排序的Label-setting算法 | 0.2154 | 0.1811 |
| T标号堆排序的Label-setting算法 | 0.1902 | 0.1497 |
| 单队列尾部Label-correcting算法 | 0.1567 | 0.0830 |
| 单队列混合Label-correcting算法 | 0.1155 | 0.0644 |
| 双队列Label-correcting算法（算法7.3） | 0.1128 | 0.0608 |

利用Java编程，在安装了Intel Core(TM) i7-4790 CPU @ 3.60-GHz 和8GB RAM的计算机上运行，得到表6.7中的参考答案。表6.7中的结果同样表明，相比于面向对象的数据结构，纯数组的数据结构具有更高的代码执行效率。比较表6.6和表6.7中的结果，我们发现，与C#相比，Java具有更高的计算效率。可能的原因是，两个编程平台的运行机制不同。C#只调用单一线程进行运算，而Java则可以同时调用所有线程进行运算。

**表6.7.** Chicago Sketch网络上最短路计算效率的对比（Java）

|  |  |  |
| --- | --- | --- |
| 算法 | 面向对象的数据结构（s） | 纯数组的数据结构（s） |
| Label-setting算法（算法7.1） | 1.0538 | 0.5642 |
| T标号排序的Label-setting算法 | 0.1423 | 0.1588 |
| 单队列尾部Label-correcting算法 | 0.1101 | 0.03432 |
| 单队列混合Label-correcting算法 | 0.09405 | 0.0282 |
| 双队列Label-correcting算法（算法7.3） | 0.0899 | 0.03282 |

* 1. 参考文献

1. Ahyja, R. K., Orlin, J. B., Magnanti, T. L., 1993. Network flows: theory, algorithms, and applications. Prentice-Hall.
2. Floyd, R., 1962. Algorithm 97: Shortest path. Communications of the ACM, 5: 345.

# 旅行商问题

* 1. 概述

旅行商问题（Travel Salesman Problem，简称TSP）是运筹学领域经典的组合优化问题，其目的是在一个完全图中找到费用最小的哈密顿回路。其中,表示城市的集合，表示弧的集合。令和分别表示从城市连出和连入的城市集合。每条弧都有一个旅行成本。TSP在实际中具有非常广泛的应用，例如，机组调度、热轧调度、日程安排、自主移动机器人任务规划、印刷机调度等。由于TSP是一类具有二元决策变量的NP-hard问题，启发式算法常被用于解决这类问题，包括蚁群算法、遗传算法（Genetic Algorithm，简称GA）、局部搜索、神经网络、模拟退火、禁忌搜索（Tabu Search，简称TS）等。

TSP的数学模型可以表述如下：

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

其中，为0-1变量，表示路段是否在最优的旅行路径中，即推销员是否通过该路段。表示从城市到城市所需要的花费。令向量。目标函数表示总的旅行费用最小。约束条件和意味着最优路径上任意一个城市都只有一条边进入和一条边离开。约束条件保证推销员只能经过一个城市*i*到达城市*j，*约束条件保证推销员只能经过一个城市*j*到达城市*i*。约束条件表示变量是0-1变量。约束条件用于避免子回路。该约束条件有不同的数学表述形式。

Miller等（1960）最早提出了如下约束条件来代替子回路消除约束：

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

Desrochers和Laporte（1991）提出了如下约束条件来代替子回路消除约束：

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

当约束条件是起作用约束时，约束条件是最大平面定义。

* 1. 遗传算法

遗传算法是群体搜索启发式，它通过模拟自然进化过程搜索最优解。在遗传算法中，一组染色体构成一个种群。种群中的每个染色体表示求解问题的一个可行解。每个染色体的适应度值就是相应的可行解的目标函数值。算法首先生成一个初始的种群并评估种群中每个染色体的适应度值。在每一次迭代中，当代种群中的每个染色体分别以一定的交叉概率添加到交叉集合中和以一定的变异概率添加到变异集合中。该算法对交叉集合中的染色体两两配对，并对每对染色体随机挑选一个交叉算子进行交叉操作。值得一提的是如果交叉集合中染色体的数量是奇数，则随机挑选一个染色体删除。该算法对变异集合中的每个染色体随机挑选一个变异算子进行变异操作。根据新得到的染色体和当代种群，该算法利用锦标赛选择的方法来生成下一代种群。当达到最大迭代次数时，算法终止。遗传算法的步骤总结如下：

|  |
| --- |
| **算法8.1**：遗传算法 |
| 1: 生成个初始解（染色体）构成初始种群 |
| 2: 计算种群中每个染色体的适应度值，并令适应度值最大的染色体赋给最优解 |
| 3: 令 |
| 4: **while** **do** |
| 5: 令交叉集合，变异集合和子代染色体集合 |
| 6: 对种群中的每个染色体分别以交叉概率和变异概率添加到交叉集合和变异集合中  7: 对集合中的染色体两两配对并执行交叉算子，得到的新的染色体添加到集合中  8: 对集合中的每个染色体执行变异算子，得到的新的染色体添加到集合中  9: 计算集合中每个染色体的适应度值，并令适应度值最大的染色体为  10: **if** **then**  11: 令  12: **end if**  13: 利用锦标赛选择的方法从集合中挑选出个染色体，得到下一代种群  14: 令  15: **end while** |
| **输出:** 最优解 |

8.2.1解的编码和评估

我们采用城市的访问次序来编码一个解。图8.1给出了一个TSP的解的编码。网络中总共有8个城市。推销员访问城市的次序为。



图. 8.1. TSP解的编码

在TSP中，一个解的目标函数值由公式得到。它的适应度值。

8.2.2初始解的生成

我们随机生成一个城市的访问次序当作一个初始解。初始种群中的每一个染色体都由该方法生成。

8.2.3交叉算子

一个交叉算子被用于交叉两个染色体和并得到两个新的染色体和。在这小节中，我们详细介绍三种交叉算子。值得一提的是，这三种交叉算子被挑选的概率是相同的。

1. **部分映射交叉（**Partial-Mapped Crossover**）。**随机选择和中相同的一个区间，先整体交换和在该区间内的城市得到和，然后顺序找出中重复的城市，反序找出中重复的城市，为和中重复的城市建立映射表，最后根据映射表替换和中非交换区间重复的城市，分别得到两个子代染色体和。图8.2给出了一个部分映射交叉的例子。区间交换以后，中顺序重复的城市编号为1、2和9，中反序重复的城市编号为3、5和4。因此，我们可以构建映射关系：、和。



图8.2 部分映射交叉

1. **顺序交叉（**Order Crossover**）。**随机选择和中相同的一个区间。把和中不在该区间内的城市设置为0，得到和；删除中包含在中的城市得到，删除中包含在中的城市得到；用中的城市顺序替换中的城市0得到子代染色体，用中的城市顺序替换中的城市0得到子代染色体。图8.3给出了一个顺序交叉的例子。



图8.3. 顺序交叉

1. **基于位置交叉（**Position-based Crossover**）**。该操作与顺序交叉类似。随机选择和中一些相同的位置，每个位置以50%的概率入选。把和中不在这些位置的城市设置为0，得到和；删除中包含在中的城市得到，删除中包含在中的城市得到；用中的城市顺序替换中的城市0得到子代染色体，用中的城市顺序替换中的城市0得到子代染色体。图8.4给出了一个基于位置交叉的例子。



图8.4. 基于位置交叉

8.2.4变异算子

一个变异算子被用于变异一个染色体并得到新的染色体。在这小节中，我们详细介绍七种变异算子。值得一提的是，这七种变异算子被挑选的概率是相同的。

1. **随机交换变异。**在解向量中随机选择两个不同的位置和，并交换这两个位置的城市。如图8.5所示，和。



图8.5. 随机交换变异

1. **随机交换区间变异。**在解向量中随机选择两个不重叠的区间，把这两个区间内城市做为整体进行互换。图8.6给出了一个随机交换区间变异的例子。



图8.6. 随机交换区间变异

1. **随机插入变异。**在解向量中随机选择一个位置和插入位置，其中。把位于位置的城市插入到位置。如图8.7所示，城市5从位置7处插入到了位置3处。



图8.7. 随机插入变异

1. **随机插入区间变异。**在解向量中随机选择一个区间和一个插入位置，把区间内城市做为整体插入到位置。图8.8给出了一个随机插入区间变异的例子，这个例子里面。



图8.8. 随机插入区间变异

1. **随机区间反转变异。**在解向量中随机选择一个区间，反转该区间内城市的顺序。图8.9给出了一个随机区间反转变异的例子。



图8.9. 随机区间反转变异

1. **随机交换反转区间变异。**在解向量中随机选择两个不重叠的区间，先把两个区间做为整体进行互换，然后各自以50%的概率反转区间内城市的顺序。图8.10给出了一个随机交换反转区间变异的例子。



图8.10. 随机交换反转区间变异

1. **随机插入反转区间变异。**在解向量中随机选择一个区间和一个插入位置，先反转该区间内城市的顺序，再把反转后的区间整体插入到位置。图8.11给出了一个随机插入反转区间变异的例子，这个例子里面。



图8.11. 随机插入反转区间变异

8.2.5交叉概率和变异概率的自适应调整

在遗传算法中，适应度较差的个体，需要更多的交叉和变异。适应度较高的个体，为防止被破坏，应当降低其交叉和变异概率。为此，在当前种群中，我们根据每个染色体的适应度值自适应地调整它的交叉概率和变异概率。

染色体的交叉概率按照如下方法自适应调节：

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

其中，为种群中所有染色体的最大适应度值，为种群中所有染色体的平均适应度值和。

染色体的变异概率按照如下方法自适应调节：

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

其中，。

* 1. 禁忌搜索算法

禁忌搜索算法属于单点搜索启发式，它采用了邻域选优的搜索方法。为了能逃离局部最优解，算法可以接收劣解，也就是每一次迭代得到的新解不必一定优于当前解。但是，一旦接受了劣解，算法的解就可能陷入循环。为了避免循环，算法将当前解最近接受的一些变化（也就是边）放在禁忌表中，在随后的迭代中加以禁止。即只有不包含在禁忌表中的边的较好解（可能比当前解差）才被接受作为下一次迭代的当前解。值得一提的是，如果新解是禁忌的但比已知最优解还要好，那么它也会被接受。随着迭代的进行，禁忌表不断更新。经过一定迭代次数后，最早进入禁忌表的移动就会从禁忌表中解禁退出。

该算法主要包括三个部分：初始解的生成、邻域搜索、禁忌表的更新。邻域搜索的目的是通过利用邻域算子来扰动当前解并得到当前解的邻域解集合。与禁忌表相关的关键因素有两个：禁忌对象和禁忌长度。该算法把移动作为禁忌的对象。准确的来说就是把包含在新解中而不包含在当前解的边作为禁忌的对象。当一个边加入到禁忌表中，禁忌长度就是它将被禁止的迭代次数。禁忌表更新的目的是为了防止搜索过程中出现解的循环，避免陷入局部最优。该算法从一个初始解开始。在每一次迭代中，一个邻域算子被随机挑选来扰动当前解并得到它的邻域解集合。接着，从该邻域解集合中找出目标函数值最小的非禁忌的新解同时也找出目标函数值最小的禁忌的新解。然后，用非禁忌的最好的新解替换当前解。如果禁忌的最好的新解比非禁忌的最好的新解要好且比最优解也要好，则再用禁忌的最好的新解替换当前解。最后，更新禁忌表。当达到最大迭代次数时，该算法终止。禁忌搜索算法的总结如下：

|  |
| --- |
| **算法8.2**：TS( ) |
| 1: 随机生成一个城市的访问次序赋给初始解 |
| 2: 令当前解和最优解 |
| 3: 令禁忌表为空和 |
| 4: **while** **do** |
| 5: 随机生成一个整数 |
| 6: 对当前解执行邻域算子得到它的邻域解集合  7: 令和  8: **for** **each** **do**  9: **if** 不是禁忌的 **then**  10: **if** **then**  11: 令  12: 令  13: **end if**  14: **else** // 是禁忌的  15: **if** and **then**  16: 令  17: 令  18: **end if**  19: **end if**  20: **end for**  21: 令  22: 更新禁忌表  23: **if** **then**  24: 令  25: **end if**  26: 令  27: **end while** |
| **输出:** 最优解 |

为了提高禁忌搜索算法的精度，我们重复次算法8.2并挑选出目标函数值最小的解。算法8.3给出了详细的步骤流程。为了减小所需的CPU运行时间，我们采用并行计算的方法来执行这次算法。

|  |
| --- |
| **算法8.3**：加强的禁忌搜索算法 |
| 1: 令解和 |
| 2: **while** **do** |
| 3: 令解TS() // 算法8.2 |
| 4: **if** **then** |
| 5: 令 |
| 6: **end if**  7: 令  8: **end while** |
| **输出:** 解 |

8.3.1解的编码和评估及禁忌的定义

与遗传算法一样，我们采用城市的访问次序来编码一个解。一个解的目标函数值由公式得到。如果包含在新解中而不包含在当前解中的边都在禁忌表中，则解是禁忌的；否则，解不是禁忌的。

8.3.2初始解的生成

与遗传算法一样，我们随机生成一个城市的访问次序当作一个初始解。

8.3.3邻域搜索

我们可以采用两种邻域算子来扰动当前解：2-opt算子（）和3-opt算子（）。这两种邻域算子都是从当前解中删除一些弧，然后重新连接这些弧以得到一个新的解。

2-opt算子（）：该算子从当前解中随机挑选两个不同的位置和，。首先，该算子删除位置与位置对应的弧和位置与位置对应的弧。然后，把位置处的城市连接到位置处的城市和把位置处的城市连接到位置处的城市。本质上就是，该算子随机挑选一个区间进行逆序。图8.12给出了这个算子的一个例子，，。该算子先删除弧和弧，然后把城市3连接到城市8和把城市5连接到城市7。



图8.12. 2-opt算子

3-opt算子（）：该算子从当前解中随机挑选三个不同的位置，和，其中。首先，该算子删除三条弧：位置与位置对应的弧，位置与位置对应的弧和位置与位置对应的弧。然后，该算子从图8.13给出的7种重连方式中挑选出目标函数值最小的那一个当作新解。这7种重连方式分别是：

1. 位置处的城市连接到位置处的城市和位置处的城市连接到位置处的城市（即从位置到位置的区间逆序）；
2. 位置处的城市连接到位置处的城市和位置处的城市连接到位置处的城市（即从位置到位置的区间逆序）；
3. 位置处的城市连接到位置处的城市和位置处的城市连接到位置处的城市（可以理解为从位置到位置的区间逆序）；
4. 位置处的城市连接到位置处的城市，位置处的城市连接到位置处的城市和位置处的城市连接到位置处的城市（即从位置到位置的区间逆序和从位置到位置的区间逆序）；
5. 位置处的城市连接到位置处的城市，位置处的城市连接到位置处的城市和位置处的城市连接到位置处的城市（即从位置到位置的区间逆序和从位置到位置的区间逆序）；
6. 位置处的城市连接到位置处的城市，位置处的城市连接到位置处的城市和位置处的城市连接到位置处的城市（即从位置到位置的区间逆序和从位置到位置的区间逆序）；
7. 位置处的城市连接到位置处的城市，位置处的城市连接到位置处的城市和位置处的城市连接到位置处的城市（即从位置到位置的区间逆序，从位置到位置的区间逆序和从位置到位置的区间逆序）。



图8.13. 三条弧重连的七种方式

8.3.4禁忌表的更新

在每一次算法迭代中，包含在接受的新解而不包含在被替换前的解的边被添加到禁忌表中，这些边的禁忌次数设置为（也就是禁忌长度），禁忌表中其他边的禁忌次数减少1。当禁忌表中某些边的禁忌次数为0时，则把这些边从禁忌表中删除。

* 1. 测试问题

在这一节，我们采用六个TSP的基准算例来测试GA算法和TS算法的性能。这两种算法均采用C#语言编写，运行在一台配置为Intel(R) Core(TM) i9-10900K 3.70 GHz CPU and 32.0 GB RAM的电脑上。

8.5.1算例生成

我们采用六个TSP的基准算例：eil51，kroA100，kroA200，pr299，rd400和att532**（详见附录B）**。遗传算法的参数：种群规模，最大迭代次数。禁忌搜索算法的参数：禁忌长度，最大迭代次数，重复次数。

8.5.2两种算法性能的对比

利用算例eil51，我们测试GA算法和TS算法的性能。测试的方法主要分为两种：（1）在相同的迭代次数下比较两种算法的目标函数值的变化，（2）在相同的运行时间下比较两种算法的目标函数值的变化。这两种测试方法都是对算例eil51分别运行这两种算法20次记录各自的目标函数值的变化。

图8.14给出了这两种算法的目标函数值随着迭代次数的变化。从图中可以看出，TS算法的目标函数值在前250次迭代下降很明显，在那之后下降缓慢。GA算法的目标函数值在前4000次迭代下降很明显，在那之后下降缓慢。就迭代次数而言，TS算法比GA算法收敛速度更快而且最后收敛的目标函数值也更小。



图8.14. 两种算法的目标函数值随着迭代次数的变化

图8.15给出了这两种算法的目标函数值随着运行时间的变化。我们可以观察到TS算法和GA算法的目标函数值都在前1秒内下降很明显，在那之后都收敛了。其中，TS算法收敛的目标函数值要更小。



图8.15. 两种算法的目标函数值随着运行时间的变化

8.5.3两种算法在不同算例中的性能对比

对这六个基准算例中的每一个算例，我们运行GA算法和TS算法20次记录相应的平均目标函数值和平均运行时间。表8.1给出了这两种算法的性能对比。其中误差列表示相应算法与已知最优值之间的误差。我们可以观察GA算法和TS算法在算例eil51和kroA100中的误差都小于3%。此外，随着节点规模的增大，GA算法和TS算法的误差都在增大。但是TS算法的精度比GA算法的更好，它在最大规模算例（即att532）中的误差只有3.80%，而GA算法的误差达到17.08%。我们还可以观察到TS算法的运行时间在这六个基准算例中比GA算法的都要小，除了算例att532。此外，相比GA算法，TS算法的运行时间随着节点规模的增大而增加的幅度更大。

表8.1 CPLEX，GA算法和TS算法的表现对比

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算例 | 节点数 | 已知最优值 |  | GA算法 | | |  | TS算法 | | |  |
|  | 目标函数值 | 运行时间（秒） | 误差（%） |  | 目标函数值 | 运行时间（秒） | 误差（%） |  |
| eil51 | 51 | 428 |  | 436.75 | 11.93 | 2.00 |  | 437.85 | 0.41 | 2.25 |  |
| kroA100 | 100 | 21285 |  | 21890.84 | 49.96 | 2.77 |  | 21794.70 | 6.31 | 2.34 |  |
| kroA200 | 200 | 29368 |  | 31174.58 | 267.57 | 5.80 |  | 30396.54 | 102.42 | 3.38 |  |
| pr299 | 299 | 48191 |  | 52716.86 | 771.06 | 8.59 |  | 49719.17 | 513.75 | 3.07 |  |
| rd400 | 400 | 15281 |  | 17547.44 | 1710.65 | 12.92 |  | 15849.87 | 1694.33 | 3.59 |  |
| att532 | 532 | 86729 |  | 104593.53 | 3861.92 | 17.08 |  | 90155.99 | 5334.09 | 3.80 |  |

8.5参考文献

1. TSP问题库, 2007. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.
2. TSP问题库, 1995. <http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html>.
3. TSP问题已知最优解, 2020. <https://wenku.baidu.com/view/33b7739c76a20029bd642d81.html?from=search>.
4. Desrochers, M., & Laporte, G., 1991. Improvements and extensions to the Miller-Tucker-Zemlin subtour elimination constraints. Operations Research Letters, 10(1), 27-36.
5. Miller, C.E., Tucker, A.W., & Zemlin, R.A., 1960. Integer programming formulation of traveling salesman problems. Journal of the ACM, 7(4), 326-329.
6. Tsubakitani, S., & Evans, J. R., 1998. Optimizing tabu list size for the traveling salesman problem. Computers & Operations Research, 25(2), 91-97.

# 车辆路径问题\*

* 1. 概述

车辆路径问题（Vehicle Routing Problem, VRP）是交通运输管理研究中的一类经典优化问题。该问题通过选取恰当的车辆路径，加快对客户需求的响应速度，提高服务质量，增强客户对物流环节的满意度，降低服务商运作成本。车辆路径问题一般定义为：对一系列发货点和/或收货点，组织适当的行车路线，使车辆有序地服务它们，在满足一定的约束条件（如货物需求量、发送量、交发货时间、车辆容量限制、行驶里程限制、持续时间限制等）下，达到一定的目标(如路程最短、费用最小、时间最少、使用车辆最少等)。由此定义不难看出，TSP是VRP的一个特例。由于TSP是NP难题，因此，VRP也是NP难题。

根据约束条件的不同，VRP可以分为以下几类：带容量约束的VRP（Capacitated Vehicle Routing Problem，简称CVRP）、带时间窗的VRP（Vehicle Routing Problem with Time Windows，简称VRPTW）、多仓库的VRP（Multi-depot Vehicle Routing Problem，简称MDVRP）、带时间窗的多仓库的VRP（Multi-depot Vehicle Routing Problem with Time Windows，简称MDVRPTW）、多周期的VRP（Periodic Vehicle Routing Problem，简称PVRP）、带时间窗的多周期的VRP（Periodic Vehicle Routing Problem with Time Windows，简称PVRPTW）、多仓库和多周期的VRP（Multi-depot and Periodic Vehicle Routing Problem，简称MDPVRP）、分流配送的VRP（Split Delivery Vehicle Routing Problem，简称SDVRP）和带时间窗的分流配送的VRP（Split Delivery Vehicle Routing Problem with Time Windows，简称SDVRPTW）。

因为VRP是一类NP难题，精确算法只适用于求解小规模的算例，对于大规模算例可能都找不到可行解。而启发式算法能在合理时间内求得高质量的解。启发式算法通常能分为两大类。一类是群体搜索启发式，如遗传算法，蚁群算法，人工蜂群（ABC）算法等。群体搜索启发式的思想是基于多个解不断改善来得到更好的解。另一类是单点搜索启发式，如变邻域搜索（VNS）算法，自适应大邻域搜索（ALNS）算法，禁忌搜索算法等。与群体搜索启发式不同，单点搜索启发式的思想是基于单个解不断改善来得到更好的解。

* 1. 数学优化模型

CVRP是所有VRP分类中最基础的一类。CVRP可以定义在一个无向图上，其中是图中的点集合，是图中的弧集合。点表示图中的顾客。每一个顾客有一个非负的需求和非负的服务时间。点表示仓库的虚拟起点，点表示仓库的虚拟终点。令和分别表示所有车辆的集合和车辆总数，每辆车的容量都为。令和分别表示从节点连出和连入节点的节点集合。每一条弧都有一个非负的行驶时间。CVRP的决策变量定义如下：

|  |  |
| --- | --- |
|  | 0-1决策变量。如果弧被车辆经过则为1；否则为0。 |
|  | 非负实数决策变量。车辆开始服务节点的时刻。 |
|  | 非负整数决策变量。车辆服务完节点后，车上的顾客数量。 |

CVRP以最小化所有车辆的行驶时间为目标，考虑如下三类约束条件：（i）每辆车从仓库出发最后回到仓库；（ii）每个顾客只能被服务一次；（iii）每一辆车服务的顾客数量不能超过车辆的容量。CVRP的数学模型可以表述如下：

|  |  |  |
| --- | --- | --- |
|  |  |  |

约束条件如下：

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

其中是解向量，。是车辆的线路解向量，。

目标函数表示最小化所有车辆的行驶时间之和。约束条件表示每个顾客只能被服务一次。约束条件表示被依次服务的两个节点之间的车上顾客数量的关系。约束条件表示车上的顾客数量不能超过车辆的容量。值得一提的是，约束条件和可以消除子环。约束条件和分别表示从仓库出发的车辆数应小于车辆总数和回到仓库的车辆数应小于车辆总数。约束条件是流平衡约束。约束条件表示被依次服务的两个节点之间的开始服务时刻的关系。

* 1. 人工蜂群算法
     1. 算法概述

人工蜂群算法受启发于蜜蜂在蜂巢周围寻找蜜源的聪明行为。在该算法中，一个蜜源表示一个解，一个蜜源的花蜜量表示相应的解的目标函数值。蜜蜂群试图利用蜜源为蜂巢找到优质的花蜜。它们通过摇摆舞交流蜜源信息。

在人工蜂群算法中，蜜蜂的种类分为三种：采蜜蜂、跟随蜂和侦查蜂。采蜜蜂负责搜索蜜源和收集需要的信息。他们与跟随蜂分享信息。跟随蜂根据信息挑选已经搜索到的蜜源进行进一步的搜索。当一个采蜜蜂放弃搜索到的蜜源时，它就会变成一个侦查蜂并在蜂巢周围搜索一个新的蜜源。如果搜索到了新的蜜源，它就会从侦查蜂又变回采蜜蜂。值得一提的是，当在一定迭代次数内，蜜源的质量还不能得到改善，采蜜蜂才会放弃该蜜源。通常，采蜜蜂、跟随蜂和蜜源的数量是相同的。

人工蜂群算法是一个迭代算法。首先，它生成一组随机的解当作蜜源并给每一个蜜源分配采蜜蜂。在每一次迭代中，每一个采蜜蜂会随机利用一个邻域算子来生成一个新的蜜源并评估该蜜源的质量。如果新的蜜源的质量比原蜜源的质量要好，则用新蜜源替换原蜜源。在所有的采蜜蜂搜索完后，他们会把信息分享给跟随蜂。然后，每一个跟随蜂利用传统的轮盘赌选择的方法挑选一个蜜源。在那之后，每一个跟随蜂随机挑选一个邻域算子来生成新的蜜源加入到被挑选的蜜源的邻域解集中，并评估新的蜜源的质量。对每一个蜜源，如果它的邻域中最好的解比它的质量要好，则用它的邻域中最好的解替换它本身。当一个蜜源在一定迭代次数内不能得到改善，相应的采蜜蜂会变成侦查蜂，随机搜索一个新的蜜源来替换原蜜源。该算法在达到最大迭代次数时终止。

* + 1. 解的编码

我们采用了一个简易的解的编码方式。假设位顾客被辆车服务。CVRP的一个解是用一个长度为的向量表示。在向量中，从1到的个整数表示每一个顾客。个0表示每一辆车从仓库出发。两个相邻的0之间的顾客序列表示一个车辆的线路。图9.1给出了一个CVRP解的编码，其中。如图9.1所示，顾客4和1被同一辆车服务，该车辆先服务顾客4后服务顾客1。



图. 9.1. CVRP解的编码

* + 1. 解的评估

如果一条线路违背了容量约束，则称该线路是不可行的。如果一个解包含了不可行的线路，则称该解为不可行解。在算法的迭代过程中，我们允许不可行解的存在。令表示一个解的适应度值，其中表示解的容量约束的违背，是容量惩罚系数。在每一次迭代后，惩罚系数会自适应调整。其调整规则如下：在当前的所有蜜源中，如果违背容量约束的蜜源的数量超过了，则；否则，其中和为事先给定的参数。

* + 1. 初始解的生成

初始解的生成采用了并行构造线路的方法。首先，该方法给每条线路各随机安排一个顾客。接着，对剩下的未被服务的顾客构成的集合，该方法每次随机挑选一个顾客插入到当前解中最好的位置以使得当前解的目标函数值的增量最小。该过程一直重复直到所有未被服务的顾客都被插入到解中。初始种群中的每一个蜜源都由该方法生成。此外，算法9.1第29行中新蜜源也由该方法生成。

* + 1. 邻域算子

一个邻域算子被用于扰动当前解并得到一个新解。在这小节，我们详细介绍三种邻域算子。值得一提的是，在算法9.1第6和16行中，这三种邻域算子被选择的概率是相同的。

1. **随机交换两个节点**

这个算子随机挑选当前解中的两位置和，其中。然后互换这两个位置所在的顾客。如图9.2所示，位置，位置。



图. 9.2. 随机交换两个节点

1. **逆序一段子序列**

这个算子首先随机挑选当前解中一段子序列。子序列的长度是随机选取的。然后逆序该子序列。图9.3给出该算子的一个例子。如图9.3所示，子序列逆序成子序列。



图. 9.3. 逆序一段子序列

1. **交换两段逆序后的子序列**

这个算子首先随机挑选当前解中的两个子序列互换它们的位置。然后，以50%的概率逆序每个子序列。图9.4给出了该算子的一个例子。如图9.4所示，首先，子序列与子序列互换位置。然后，子序列逆序成子序列。



图. 9.4. 交换两段逆序后的子序列

* + 1. 轮盘赌选择的方法

在每一次算法迭代中，每个跟随蜂随机挑选一个蜜源。为了增大质量好的蜜源被挑选的概率，我们采用了传统的轮盘赌选择方法。每个蜜源被挑选的概率为，其中。

* + 1. 详细算法

求解CVRP的人工蜂群算法步骤总结如下：

|  |
| --- |
| **算法9.1**：人工蜂群算法 |
| 1: 随机生成一组初始蜜源，并给每一个蜜源分配一个采蜜蜂 // 为蜜源数量 |
| 2: 评估每一个蜜源的质量 |
| 3: 令，，和 |
| 4: **while** **do** // 为最大迭代次数 |
| 5: **for each** 蜜源 **do**  6: 随机挑选一个邻域算子来扰动蜜源以得到新蜜源 |
| 7: **if** **then**  8: 令和  9: **else**  10: 令  11: **end if**  12: **end for**  13: 令 // 是蜜源的邻域解集合  14: **for each** 跟随蜂 **do**  15: 利用基于蜜源质量的轮盘赌选择的方法挑选出一个蜜源  16: 随机挑选一个邻域算子来扰动蜜源以得到新蜜源  17: 令  18: **end for**  19: **for each** 蜜源 and **do**  20: 令蜜源的邻域解集中质量最好的解为  21: **if** **then**  22: 令和  23: **else**  24: 令  25: **end if**  26: **end for**  27: **for each** 蜜源 **do**  28: **if** **then** // 是事先给定的  29: 随机生成一个新蜜源替换原蜜源  30: **end if**  31: **end for**  32: 令当前所有蜜源中质量最好的并且是可行的蜜源为  33: **if** **then**  34: 令和  35: **end if**  36: 令  37: **end while** |
| **输出:** 最优解 |

* 1. 变邻域搜索算法
     1. 算法概述

变邻域搜索算法最早被Mladenovic和Hansen（1997）提出用于求解组合和全局优化问题。变邻域搜索算法源于在搜索过程中系统地改变当前解的邻域的思想。它能获得更好的局部最优解的基本理论是，一个邻域结构下的局部最优解与另一个邻域结构下的相去甚远。

该算法主要包括三个部分：初始可行解的生成，扰动和局部搜索。扰动的目的是通过利用一组邻域算子来避免当前解陷入局部最优。局部搜索的目的是在当前邻域中搜索更好的解。该算法从一个初始可行解开始。它利用一个邻域算子来扰动当前解到一个新的邻域，紧接着，利用局部搜索在当前邻域中寻找更好的解。该算法倾向于利用第一个邻域算子来扰动当前解。如果利用局部搜索得到的新解没有被接受，该算法将在下一次迭代中利用下一个邻域算子来扰动当前解；否则，在下一次迭代中第一个邻域算子将被使用。该算法采用最简单的接受准则，即如果新解比当前解要好，则用新解替换当前解；否则，不接受新解。当达到最大迭代次数时，算法终止。在变邻域搜索算法中，使用不同的邻域算子交替扰动当前解，有利于实现搜索深度和广度的良好平衡。

* + 1. 解的编码和评估

在变邻域算法中，我们采用另一种简易的解的编码方式。个向量被用于编码CVRP的一个解，这个向量的长度之和为。图9.5给出了一个CVRP的解的编码的例子，其中和。



图. 9.5. 另一种CVRP解的表示

与9.2节的人工蜂群算法不同，在变邻域算法中，我们要求解一直保持可行的。因此，一个解的好坏利用它的目标函数值来评估。质量越好的解，它的目标函数值越小。相反，质量越差的解，它的目标函数值越大。

* + 1. 初始可行解的生成

我们采用一种带有贪婪策略的并行构造线路的方法来生成初始可行解。该方法分为两个阶段：第一个阶段是给每一个车辆的空线路各分配一个顾客以得到一个可行解；第二个阶段是把剩余的顾客一个个插入到当前解中最好的位置以使得当前解的目标函数值的增量最小。伪代码9.1给出了该方法的具体流程。

|  |
| --- |
| **伪代码9.1**：INITIALSOLUTION\_GENERATION() |
| 1: 令一个顾客集合 |
| 2: 对顾客集合中的顾客按照其需求量大小进行降序排序得到集合 |
| 3: 令和  4: **for each** 车辆 **do** |
| 5: 把顾客插入车辆*k*的空线路中 |
| 6: 令 |
| 7: 令  8: **end for**  9: 令和  10: **while** **do**  11: **for each** 车辆 **do**  12: 找出顾客插入到线路*k*的最好位置以使得该线路的目标函数值的增量最小  13: **end for**  14: 把顾客插入到最优线路中以使得当前解的目标函数值的增量最小  15: 令 |
| 16: **end while** |

在伪代码9.1中，第2行表示对顾客集合按照顾客的需求量大小降序排序得到新的集合。第4-8行表示对每一个空线路各安排一个顾客。第9行表示把已被服务的顾客从集合中删除。第11-15行表示对当前顾客找出他/她插入到每一条线路中的最好位置同时得到该线路的目标函数值的增量。那之后，找出这些增量中最小的那一个同时得到其对应的线路。最后，把当前顾客插入到线路中的最好位置。值得一提的是，当一个顾客要插入到一条线路的位置处时，重新连接线路的两种方式中更好的那一种被采用，如图9.6所示。新顾客在插入到该线路位置处后，其之后的所有顾客有逆序和不逆序的两种方式。假设通过末尾逆序的方式得到的新线路的目标函数值更小，则此次该方式被采用。



图. 9.6. 初始可行解中的顾客的插入

* + 1. 扰动

我们设计了一种邻域算子来扰动当前解。值得一提的是，该邻域算子可以维持解的可行性。令Shaking()表示当前解执行次该邻域算子后得到新的可行解。该邻域算子的详细步骤介绍如下：首先随机挑选两个不同的车辆和，其中这两辆车的线路都至少服务了一个顾客。接着，该算子尝试互换这两条线路中各自的一个随机长度的子段以得到两条新的可行的线路。这两个子段都是插入在对方从线路中删除的位置。值得一提的是，这两条线路的任意一对子段都需要被检查以增大操作成功（即能得到两条新的可行线路）的可能性。一条线路的一个子段的定义是线路上一段连续服务的顾客的集合，集合中顾客的数量为该子段的长度。如果该算子操作不成功，另外两个不同的车辆从集合中随机选出。这个过程一直重复直到该算子成功找到一个新的可行的邻域解或者所有车辆的组合都被执行但没有找到一个新的可行的邻域解。图9.7给出了这个邻域算子的一个例子。车辆的线路中选取的子段1的长度为5，车辆的线路中选取的子段2的长度为3。执行完这个算子后，这两条线路都变成新的可行的线路。



图. 9.7. 邻域算子

* + 1. 局部搜索

这里介绍两种局部搜索算子用于改善扰动之后的解。一种称为单线路局部最优化（Single-route local optimisation）；另一种称为两条线路子段交换（Two route segment exchange）。前者执行线路内部的调整，后者执行线路之间的调整。注意，这两种局部搜索算子都能保证解的可行性。令LOCAL\_SEARCH()表示解在执行完局部搜索后得到新的可行解。这两个局部搜索算子的详细介绍如下。

1. **单线路局部最优化**

该算子尝试逆序一辆车的线路中所有的子段，并执行能该线路目标函数值下降最多的子段的逆序。这个过程一直重复直到不能改善这条线路的目标函数值。值得一提的是，尝试逆序的子段的长度范围为2或3。该算子的一个例子展示在图9.8中。被逆序的子段的长度为3。在执行单线路局部最优化后，车辆的线路变成一条新线路。



图. 9.8. 单线路局部最优化

1. **两条线路子段交换**

对于给定的一对车辆和，该算子尝试交换这两辆车的线路中各自的一个子段。这两个子段的长度都是从0变化到所在线路中顾客的数量。这两个子段都是插入在对方从线路中删除的位置。当找到第一对可以成功互换位置的两个子段且能使当前解的目标函数值降低，该算子终止。当所有可能被交换的子段都检查了，该算子也终止。很显然，这个算子包括把一条线路的一个子段插入到另一条线路的操作。当一个子段要被插入到另一条线路中时，四种可能的线路重连方式都需要被考虑，该算子会执行能使新线路目标函数值最低的那一种线路重连方式。图9.9给出了这四种线路重连方式。第一种方式为插入的子段不逆序，其末尾也不逆序。第二种方式为插入的子段逆序，但其末尾不逆序。第三种方式为插入的子段不逆序，但其末尾逆序。第四种方式为插入的子段逆序，其末尾也逆序。



图. 9.9. 四种线路重连方式

局部搜索的具体流程如下：首先，对每一个车辆的线路执行单线路局部最优化算子。然后从当前解中随机选出两条不同的线路执行两条线路子段交换算子。如果该算子操作失败（即不能改善当前解），则另外两条不同的线路从当前解中随机选出。如果该算子操作成功（即得到了两条新的可行线路，并改善了当前解），单线路局部最优化算子被用于进一步改善新得到的两条线路。当两条线路子段交换算子对当前解中任意两条线路都不能成功操作，则局部搜索停止。

* + 1. 详细算法

求解CVRP的变邻域搜索算法的总结如下：

|  |
| --- |
| **算法9.2**：VNS() |
| 1: 令初始解INITIALSOLUTION\_GENERATION() //见9.3.2小节 |
| 2: 令当前解和最优解 |
| 3: 令和 |
| 4: **while** **do** // 是最大迭代次数 |
| 5: 令Shaking() //见9.3.3小节 |
| 6: 令LOCAL\_SEARCH() //见9.3.4小节  7: **if** **then**  8: 令和  9: **else**  10: 令  11: **end if**  12: **if** **then**  13: 令  14: **end if**  15: **if** **then**  16: 令  17: **end if**  18: 令  19: **end while** |
| **输出:** 最优解 |

为了提高变邻域搜索算法的精度，我们重复次算法9.2并挑选出最好的解。详细的算法呈现在算法9.3中。为了减小所需的CPU运行时间，我们采用并行计算的方法来执行这次算法。

|  |
| --- |
| **算法9.3**：加强的变邻域搜索算法 |
| 1: 令和 |
| 2: **while** **do** |
| 3: 令VNS() // 算法9.2 |
| 4: **if** **then** |
| 5: 令 |
| 6: **end if**  7: 令  8: **end while** |
| **输出:** |

* 1. 自适应大邻域搜索算法
     1. 算法概述

局部搜索启发式算法通常建立在对解进行小规模扰动的邻域移动上，如从一条线路中删除一个顾客或者交换两条线路中的顾客。这种局部搜索启发式算法能在短时间内搜索大量的解，但是在每一次迭代中，一个解的改动都很小。当求解约束较紧的问题时，这种算法一般比较难使搜索从一个解空间跳转到另一个解空间。而大邻域搜索算法能较好的处理这种情况。该算法主要包括三个部分：初始可行解的生成，破坏算子和修复算子。在每一次迭代中，大邻域搜索算法会采用破坏算子从当前解中删除较多的顾客，紧接着采用修复算子把删除的顾客重新插回到当前解最优的位置处以得到更好的解。

这里，我们详细介绍自适应大邻域搜索算法，它是大邻域搜索算法的扩展。这两种算法不同的地方在于，大邻域搜索算法通常只有一个破坏算子和一个修复算子，而自适应大邻域搜索算法包括多个破坏算子和多个修复算子。该算法从一个初始可行解开始。在每一次迭代中，该算法采用轮盘赌选择方法挑选出一个破坏算子和一个修复算子。注意破坏算子和修复算子的挑选是相互独立的。通过破坏和修复当前解，该算法可以得到一个新的可行解。新解的接受是采用模拟退火的接受准则。每过一定的迭代次数，该算法会更新每个破坏算子和修复算子被选择的概率。当达到最大迭代次数时，算法终止。在自适应大邻域搜索算法中，自适应的调整每个算子被选择的概率使得表现好的算子更容易被选中，有利于算法得到更好的解。

* + 1. 解的编码和评估

与变邻域搜索算法一样，在自适应大邻域搜索算法中，我们要求解一直保持可行的。该算法的解的编码和评估与变邻域搜索算法的一致，故不再重复叙述。

* + 1. 初始可行解的生成

我们采用一种带有贪婪策略的顺序构造线路的方法来生成初始可行解。该方法的思想是每次构造一条可行线路直到所有车辆的线路构造完毕。在构造一条线路时，未被服务的顾客都是插入在该线路中使得目标函数值的增量最小的位置。伪代码9.2给出了该方法的具体流程。

|  |
| --- |
| **伪代码9.2**：INITIALSOLUTION\_GENERATION() |
| 1: 令一个未被服务的顾客集合 |
| 2: **for each** 车辆 **do**  3: 令顾客集合  4: **for each** 顾客 **do**  5: 找出顾客插入到线路中的最好位置以使得该线路的目标函数值的增量最小  6: **if** 顾客插入的最好位置存在 **then** //顾客能成功插入到线路中 |
| 7: 把顾客插入到该位置处并令 |
| 8: **end if** |
| 9: **end for** |
| 10: 令 //从顾客集合中删除已被服务的顾客 |
| 11: **end for** |

* + 1. 破坏算子

破坏算子都是以一定规则从当前解中删除一定数量的顾客。在每一次迭代中，删除顾客的数量是随机选择的，它的取值范围是。其中是事先给定的参数。不同的破坏算子挑选要删除的顾客的规则不同。这里，我们介绍三种常用的破坏算子：相似删除，随机删除和最坏删除。

我们首先介绍相似删除算子，它最早被Shaw（1997）提出。该算子的思想是从当前解中删除一些相似度比较接近的顾客。任意两个不同顾客和的相似度与这两个顾客之间的旅行时间和需求有关。相似度越小则表示顾客和顾客越相似。它的计算公式如下:

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中是旅行时间系数，是需求系数。

给定一个解和正整数，利用相似删除算子，我们可以从该解中删除个顾客得到新解。首先，该算子从当前解中随机挑选一个顾客，把他/她加入到要删除的顾客集合中并得到一个顾客集合。集合是由所有不包含在集合中的顾客构成的集合。然后，每次从集合中随机挑选一个顾客并对集合中的顾客按照其与顾客的相似度升序排序。接着从集合中以一定偏好随机挑选一个与顾客较为相似的顾客加入到集合中并把顾客从集合中剔除。当集合中的顾客数量达到时，从当前解中删除集合中的所有顾客。伪代码9.3给出了该算子的具体流程。

|  |
| --- |
| **伪代码9.3**：相似删除 |
| **输入：**一个解，正整数  1: 从当前解中随机挑选一个顾客 |
| 2: 令顾客集合和顾客集合 |
| 3: **while** **do** |
| 4: 从顾客集合中随机挑选出一个顾客 |
| 5: 对顾客集合中的顾客按照其与顾客的相似度升序排序 |
| 6: 随机生成一个小数  7: 令顾客为集合中索引序号为处的顾客 // 是一个给定的正整数  8: 令和  9: **end while**  10: 从当前解中删除集合中的所有顾客，得到新解 |
| **输出：**新解 |

随机删除算子比较简单，它的思想是每次从当前解中随机挑选一个顾客删除直到删除的顾客数量达到。最后，我们介绍最坏删除算子。该算子的思想是从当前解中删除一些成本较大的顾客。一个被服务的顾客的成本为把该顾客从所在线路删除后当前解的目标函数值的减少量。

给定一个解和正整数，利用最坏删除算子，我们可以从该解中删除个顾客得到新解。首先，该算子把所有顾客构成的集合赋给顾客集合，并对集合中的顾客按照其成本大小进行降序排序。然后，每次从集合中以一定偏好随机挑选一个成本较大的顾客加入到集合中并把顾客从集合中剔除。当集合中的顾客数量达到时，该算子从当前解中删除集合中的所有顾客。伪代码9.4给出最坏删除算子的具体流程。

|  |
| --- |
| **伪代码9.4**：最坏删除 |
| **输入：**一个解，正整数  1: 令顾客集合和顾客集合 |
| 2: 对集合中的顾客按照其成本大小进行降序排序 |
| 3: **while** **do** |
| 4: 随机生成一个小数  5: 令顾客为集合中索引序号为处的顾客 // 是一个给定的正整数  6: 令和  7: **end while**  8: 从当前解中删除集合中的所有顾客，得到新解 |
| **输出：**新解 |

* + 1. 修复算子

修复算子是把破坏算子删除的顾客以寻优的方式重新插回当前解中最好的位置。值得一提的是，修复算子并不能每次都成功地把所有被删除的顾客重新插回当前解中。如果在某一次迭代中，存在一个顾客没有被成功插回当前解中，则此次迭代新解退回到被破坏算子破坏之前的解。即在算法9.4第8行中，令。这里，我们介绍两种常用的修复算子：贪婪插入和后悔插入。

我们先详细介绍贪婪插入算子，它的思想是每次把最优顾客插入到当前解中最优位置直到所有被删除的顾客重新插回到当前解中。该算子的详细流程给出在伪代码9.5中。根据这个伪代码，首先，找出每个顾客插入解中最优位置并且得到相应的目标函数值的增量（第2-7行）。值得一提的是如果顾客不能成功插入到线路中，则。接着，找出使得解的目标函数值的增量最小的最优顾客（第8行）。如果顾客能成功插入解中，则把顾客插入到解的最优位置处并把他/她从集合中删除；否则，该算子终止（第9-14行）。

|  |
| --- |
| **伪代码9.5**：贪婪插入 |
| **输入：**一个解和顾客集合  1: **while** **do** |
| 2: **for each** 顾客 **do** |
| 3: **for each** 车辆 **do**  4: 计算顾客插入在线路的最好位置处，该线路目标函数值的增量  5: **end for** |
| 6: 令顾客插入在解最优位置处的目标函数值的增量  7: **end for**  8: 令最优顾客  9: **if** **then** //顾客能成功插入解中  10: 把顾客插入到解最优位置处  11: 令  12: **else** //顾客不能成功插入解中  13: **break**  14: **end if**  15: **end while** |
| **输出：**更新的解 |

与贪婪插入算子不同，后悔插入算子的思想是每次挑选后悔值最大的顾客插入到当前解中最优位置直到所有被删除的顾客重新插回到当前解中。一个未被服务的顾客的后悔值等于他/她插入在次优线路的最好位置的目标函数值的增量减去他/她插入在最优线路的最好位置的目标函数值的增量。该算子的详细流程给出在伪代码9.6中。根据这个伪代码，首先，找出每个顾客插入每条线路的最好位置并把相应的目标函数值的增量加入到增量集合中（第3-7行）。接着，对集合的增量按照其大小进行升序排序并得到顾客的后悔值（第8-9行）。最后，挑选出后悔值最大的顾客（第11行）。如果顾客能成功插入解中，则把顾客插入到解最优位置处并把他/她从集合中删除；否则，该算子终止（第12-17行）。

|  |
| --- |
| **伪代码9.6**：后悔插入 |
| **输入：**一个解和顾客集合  1: **while** **do** |
| 2: **for each** 顾客 **do** |
| 3: 令增量集合  4: **for each** 车辆 **do**  5: 计算顾客插入在线路的最好位置处，该线路目标函数值的增量  6: 令  7: **end for** |
| 8: 对集合的增量按照其大小进行升序排序，得到集合  9: 令顾客的后悔值  10: **end for**  11: 令后悔值最大的顾客  12: **if** **then** //顾客能成功插入解中  13: 把顾客插入到解最优位置处  14: 令  15: **else** //顾客不能成功插入解中  16: **break**  17: **end if**  18: **end while** |
| **输出：**更新的解 |

* + 1. 更新算子的得分和选择概率

在每一次迭代中，根据得到的新解的表现，我们会各给此次挑选出的破坏算子和修复算子一个分数。这主要分为三种情况：如果新解比目前的全局最优解要好，则挑选出的破坏算子和修复算子各得33分；如果新解比当前解要好但比目前的全局最优解要差，则挑选出的破坏算子和修复算子各得9分；如果新解比当前解要差但被接受了，则挑选出的破坏算子和修复算子各得13分。值得一提的是，第三种情况的得分比第二种情况的得分要大（即13>9），这是因为差的解被接受能增加解的多样性。

我们把算法的整个迭代过程分为一个个片段。每个片段都包含100次迭代。在每个片段开始时，所有的破坏算子和修复算子的得分都会设置为0。令表示破坏算子的集合。对所有的破坏算子，令表示破坏算子在片段的权重。在算法完成片段时，在子段中，破坏算子的权重的计算公式如下：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中为算子在片段中累计获得的分数，为算子在片段中累计被挑选的次数，为给定的参数。

值得一提的是所有算子在第一个片段的权重都设置为1。利用每个破坏算子的权重，我们可以计算破坏算子在片段的选择概率。修复算子的权重和概率的更新方式与破坏算子的一致，故我们不在详细介绍。

* + 1. 详细算法

求解CVRP的自适应大邻域搜索算法总结如下：

|  |
| --- |
| **算法9.4**：自适应大邻域搜索算法 |
| 1: 生成一个初始可行解INITIALSOLUTION\_GENERATION() // 见9.4.2小节  2: 令最优解和当前解 |
| 3: 计算初始温度并令当前温度 // 是温度参数  4: 令 |
| 5: **while** **do** // 是最大迭代次数 |
| 6: 基于每个破坏算子的被挑选的概率，利用轮盘赌选择方法挑选出一个破坏算子 |
| 7: 基于每个修复算子的被挑选的概率，利用轮盘赌选择方法挑选出一个修复算子 |
| 8: 利用挑选出的破坏算子和修复算子改变当前解以得到新可行解  9: 随机生成一个小数  10: **if** **then**  11: 令  12: **end if**  13: 更新挑选出的破坏算子和修复算子的得分  14: 令 // 是冷却率  15: **if** **then**  16: 令  17: **end if**  18: **if** **then** //每100次迭代更新一次算子的选择概率  19: 更新每个破坏算子的被挑选的概率  20: 更新每个修复算子的被挑选的概率  21: **end if**  22: 令  23: **end while** |
| **输出:** 最优解 |

值得一提的是，在自适应大邻域搜索算法中，我们采用了并行计算的方法来重复此算法并挑选出最好的解。这与算法9.3是类似的，故不在详细描述。

* 1. 测试问题

在这一节，我们采用5个CVRP的基准算例来测试三个算法的性能。这三个算法均采用C#语言编写，运行在一台配置为Intel(R) Core(TM) i9-10900K 3.70 GHz CPU and 32.0 GB RAM的电脑上。

* + 1. 算例生成

我们在CVRP问题库里面挑选了5个基准算例（XXX网址XXX）：vrpnc1，vrpnc2，vrpnc3，vrpnc4和vrpnc5**（详见附录C）**。人工蜂群算法中的参数设置为：蜜源数量，最大迭代次数，采蜜蜂抛弃蜜源参数，容量惩罚系数和惩罚系数的调整幅度参数。变邻域搜索算法中的参数：最大迭代次数和算法重复次数。自适应大邻域搜索算法中的参数：温度参数，最大迭代次数，算法重复次数，冷却率，旅行时间系数，需求系数，删除顾客参数，相似删除算子参数，最坏删除算子参数和权重参数。

* + 1. 三种算法的性能的对比

利用算例vrpnc1，我们测试ABC算法，VNS算法和ALNS算法的性能。测试的方法主要分为两种：（i）在相同的迭代次数下比较这三种算法的目标函数值的变化，（ii）在相同的运行时间下比较这三种算法的目标函数值的变化。这两种测试方法都是对算例vrpnc1分别运行这三种算法20次记录各自的目标函数值的变化。

图9.10给出了这三种算法的目标函数值随着迭代次数的变化。我们可以观察到ALNS算法和VNS算法的目标函数值在前500次迭代下降很明显，而在那之后这两个算法的目标函数值下降缓慢。ABC算法的目标函数值在2000-4000次迭代中下降很明显，而在4000次迭代之后下降缓慢。值得一提的是，ABC算法在前2000次迭代中没有目标函数值，这是因为该算法在前2000次迭代中没有找到可行解。我们还可以观察到在这三种算法中，ALNS算法的表现最好。它的收敛速度最快，而且收敛的目标函数值也是最小的。



图. 9.10. 三种算法的目标函数值随着迭代次数的变化

图9.11给出了这三种算法的目标函数值随着运行时间的变化。从图中可以看出，这三种算法的目标函数值在前1秒内下降都很明显，而在那之后三种算法的目标函数值下降缓慢。此外，我们还可以看出在相同的运行时间下，ALNS算法能得到质量更好的解。



图. 9.11. 三种算法的目标函数值随着运行时间的变化

* + 1. 三种算法在不同算例中的性能对比

对这5个基准算例中的每一个算例，我们运行ABC算法，VNS算法和ALNS算法20次记录相应的平均目标函数值和平均运行时间。表9.1给出了这三种算法的性能对比。其中，误差列表示相应的算法的目标函数值与已知最优解之间的误差。从表中可以看出，ALNS算法的目标函数值与已知最优解之间的误差在这5组基准算例中都是最小的。特别地，ALNS算法对算例vrpnc1的误差只有0.05%。此外，ALNS算法的运行时间是最短的，而VNS算法的运行时间是最长的。我们还可以看出随机节点规模的增大，三个算法的误差都在增大。

表9.1 CPLEX，ABC算法，VNS算法和ALNS算法的表现对比

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 算例名称 | 顾客数 | 车辆数 | 已知最优值 |  | ABC算法 | |  | | | VNS算法 | |  | | ALNS算法 | |  |
|  | 目标函数值 | 运行时间（秒） | 误差（%） |  | 目标函数值 | 运行时间（秒） | 误差（%） |  | 目标函数值 | 运行时间（秒） | 误差（%） |  |
| vrpnc1 | 50 | 5 | 524.61 |  | 530.69 | 20.84 | 1.15 |  | 543.33 | 31.74 | 3.45 |  | 524.90 | 3.59 | 0.05 |  |
| vrpnc2 | 75 | 12 | 835.26 |  | 872.66 | 48.98 | 4.29 |  | 871.70 | 93.82 | 4.18 |  | 836.86 | 9.46 | 0.19 |  |
| vrpnc3 | 100 | 10 | 826.14 |  | 865.98 | 75.57 | 4.60 |  | 863.06 | 342.36 | 4.28 |  | 829.27 | 18.42 | 0.38 |  |
| vrpnc4 | 150 | 14 | 1028.42 |  | 1112.72 | 167.02 | 7.58 |  | 1090.21 | 858.81 | 5.67 |  | 1048.89 | 58.89 | 1.95 |  |
| vrpnc5 | 199 | 20 | 1295.41 |  | 1419.80 | 290.06 | 8.76 |  | 1386.92 | 1225.81 | 6.60 |  | 1335.60 | 135.59 | 3.01 |  |

* 1. 参考文献

1. Szeto W.Y., Wu Y., Ho S.C. An artificial bee colony algorithm for the capacitated vehicle routing problem[J]. European Journal of Operational Research, 2011, 215: 126-135.
2. Mladenovic N., Hansen P. Variable neighborhood search[J]. Computers & Operations Research, 1997, 24(11): 1097-1100.
3. Fleszar K., Osman I.H., Hindi K.S. A variable neighbourhood search algorithm for the open vehicle routing problem[J]. European Journal of Operational Research, 2009, 195: 803-809.
4. Ropke S., Pisinge D. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows[J]. Transportation Science, 2006, 40(4): 455-472.
5. Shaw P. A new local search algorithm providing high quality solutions to vehicle routing problems[R]. Technical report, Department of Computer Science, University of Strathclyde, Scotland. 1997.

# 大规模整数规划问题\*

# 非线性规划

* 1. 概述

考虑如下一般形式数学规划问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，为决策变量构成的向量，为决策变量的数量。、、为的实值函数。为目标函数，为第个不等式约束，为第个等式约束，和分别为不等式约束和等式约束的数量。如果时，即不存在约束，则问题被称为无约束最优化问题；否则，问题被称为约束最优化问题。如果、、都是线性函数，则问题被称为线性规划问题；否则，问题(10.1)被称为非线性规划问题。

令为问题的可行域，则任意都是问题的**可行解**。对于给定的，如果存在的某个邻域，使得任意，都有，则称为问题的**局部最优解**。如果对任意都有，则称为问题的**全局最优解**。

* 1. 凸集、凸函数、凸规划

**定义10.1**（凸集）：给定集合，如果对任意都有：

|  |  |  |
| --- | --- | --- |
|  |  |  |

则称是凸集。

由上述定义可知，若一个集合是凸集，则连接该集合内任意两点的线段也必包含在此集合中。我们约定空集是凸集。

是定义在上的实值函数，记为。有如下定义：

**定义10.2**（凸函数）：设是一个非空凸集，，如果对任意都有：

|  |  |  |
| --- | --- | --- |
|  |  |  |

则称是上的凸函数。

**定义10.3**（严格凸函数）：设是一个非空凸集，，如果对任意都有：

|  |  |  |
| --- | --- | --- |
|  |  |  |

则称是上的严格凸函数。

图10.1给出了凸函数的几何含义。对于任意给定，连接任意两点的线段上的点的函数值，在这两点的加权函数值（即）的下方。如果是上的（严格）凸函数，则称是上的（严格）凹函数。



图10.1 凸函数的几何表示

函数二阶可微，其在点处的Hessian矩阵是如下的对称矩阵：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，是在点处的二阶偏导数。为描述方便，令为在处的海塞矩阵。

**定理10.1**：设是非空开凸集，二阶可微，则是上的凸函数的充要条件是：的Hessian矩阵在上半正定。

**定理10.2**：设是非空开凸集，二阶可微，如果的Hessian矩阵在上是正定的，则是上的严格凸函数。

**定义10.4**（凸规划）：若为凸集，在上是凸函数，则称如下数学规划问题

|  |  |  |
| --- | --- | --- |
|  |  |  |

为凸规划。

显然，线性规划是凸规划。凸规划问题有如下性质：

**定理10.3**：如果问题是凸规划，则其任意局部最优解都是全局最优解。

**定理10.4**：如果问题是凸规划，是严格凸函数，且，则该问题具有唯一的最优解。

* 1. 优化问题的最优性条件
     1. 无约束优化问题的最优性条件

考虑如下无约束优化问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，是定义在上的实值函数。

**定理10.5**：设在处可微，若是的局部极小点，则梯度，即

|  |  |  |
| --- | --- | --- |
|  |  |  |

**定理10.6**（二阶必要条件）：设在处二次可微，若是局部极小点，则梯度，且Hessian矩阵半正定。

**定理10.7**（二阶充分条件）：设在处二次可微，若梯度，且Hessian矩阵半正定，则是局部极小点。

**定理10.8**（充要条件）：设是定义在上的可微凸函数，，则是整体极小点的充要条件是。

* + 1. 约束优化问题的最优性条件

库恩-塔克尔条件（Kuhn-Tucker Condition，简称KT条件）是非线性规划领域中最重要的理论成果之一。它是判定约束非线性规划问题的某可行点为极小点的必要条件，也被称一阶必要条件。但一般来讲它并不是充分条件，因此满足这一条件的点并非一定就是极值点。对于凸规划来说，则是判别极小点的充分必要条件。KT条件的具体表述如下：

**定理10.9：**令为非线性规划问题的可行解，**。**满足如下条件：

（1）对任意，函数**在**处可微；

（2）对任意，函数在处连续可微；

（3）对任意和，向量与向量线性无关。

若是问题的局部最优解，则存在两组实数和使得

|  |  |  |
| --- | --- | --- |
|  |  |  |

* 1. 优化问题的一般迭代算法

利用定理10.9中给出的KT条件可以对非线性规划问题进行求解，但这种方法有很大的局限性：（i）实际的问题中，函数可能是不连续或者不可微的；（ii）需要解复杂的方程组，而方程组到目前仍没有有效的算法；（iii）实际的问题可能含有不等式约束，微分方法不易处理。因此，对于最优化模型通常采用迭代方法来求得它的最优解。迭代方法有两个关键的步骤：（i）寻找搜索方法（可行的下降方向）和（ii）确定搜索步长。有如下定义：

**定义10.5**（可行方向）：令是非线性规划的一个可行解，为处的某一方向。如果存在实数，使得对任意都有，则称方向是处的一个可行方向。

**定义10.6**（下降方向）：令是非线性规划的一个可行解，为处的某一方向。如果存在实数，使得任意都有，则称方向为处的一个下降方向。

**定义10.7**（可行下降方向）：令是非线性规划的一个可行解，如果方向既是处的可行方向，也是处的下降方向，则称为处的可行下降方向。

将目标函数在点处作一阶泰勒展开，则满足条件的方向必为点的下降方向。求解非线性规划问题，关键在于寻找合适的可行下降方向。我们有如下定理：

**定理10.10：**如果是非线性规划的一个局部极小点，则在处不存在可行下降方向。

线性规划问题可以采用如下一般迭代算法进行求解：

|  |
| --- |
| **算法10.1**  线性规划问题的一般迭代算法 |
| 输入：初始可行解 |
| 1: 置 |
| 2: **while** 不满足某种终止条件 **do** |
| 3: 选取在点处的搜索方向 |
| 4: 沿搜索方向寻求适当的步长  5: 更新迭代点，并置 |
| 6: **end while** |
| 输出：当前解 |

在上面的步骤中，选取搜索方向是最关键的一步，各种算法的区分，主要在于确定搜索方向的方法不同。

* 1. 一维搜索

单变量单峰函数求极值的过程就是一维搜索。它是非线性规划迭代算法遇到的共同问题，即在给定搜索方向的基础上确定步长，从而确定下一个迭代点。常用的一维搜索方法主要有斐波那契法、黄金分割法（0.618法）、插值法、二分法（求根法）等。

一维搜索问题就是求解如下单一变量的最优化问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中为一维搜索的搜索区间，在上只有唯一的局部极小点，即在上是一个单峰函数。

在实际应用中，函数不一定有显式的表达式。如果我们只能够计算得到的函数值，则可以采用黄金分割法来求解最优化问题；如果我们还能够计算得到的函数值，则可以采用二分法来求解最优化问题。

* + 1. 黄金分割法

令为在上的局部极小点。设有区间使得且。在区间内任意选取两点和。不失一般性，令。我们有如下结论：

**命题 10.1：**如果，则有；否则，有。

依据命题10.1，我们可以通过逐步缩小搜索区间的方法来求解问题的最优解。如图10.2.所示，当时，我们可以把区间舍弃掉（因为），把一维搜索的区间调整为。当.时，我们可以把区间舍弃掉（因为），把一维搜索的区间调整为。在黄金分割法中，我们设置：

|  |  |  |
| --- | --- | --- |
|  |  |  |



图10.2 黄金分割法原理图

黄金分割法的详细算法步骤如下：

|  |  |
| --- | --- |
| **算法10.2：**黄金分割法 | |
| **输入:** 精度 | |
| 1: | 置、、、、和。 |
| 2: | **while** **do** |
| 3: | **if** **do** |
| 4: | 置、、、和。 |
| 5: | **else** |
| 6: | 置、、、和。 |
| 7: | **end** **if** |
| 8: | **end** **while** |
| 9: | 。 |
| **输出：**。 | |

算法10.2中，为一维搜索的精度，和为当前的搜索区间，和为搜索区间的黄金分割点，和为黄金分割点的函数值。Line 9中，理论上可以取最终的搜索中的任意点。为了减少对目标函数值的评估次数，搜索结束以后，当时，我们可以取；否则，取。

* + 1. 二分法

令为在上的局部极小点。如果可导，则一维搜索问题就变成了方程求根，即求解，使得。

设有区间使得且。在区间内任意选取一点。我们有如下结论：

**命题 10.2：**如果，则有；否则，有。

依据命题10.2，我们可以通过逐步缩小搜索区间的方法来求解问题的最优解。当时（图10.3(a)），我们可以把区间舍弃掉（因为），把一维搜索的区间调整为。当时（图10.3(b)），我们可以把区间舍弃掉（因为），把一维搜索的区间调整为。在二分法中，我们设置：。



图10.3 二分法原理图

二分法的详细算法步骤如下：

|  |  |
| --- | --- |
| **算法10.3：**二分法 | |
| **输入:** 精度 | |
| 1: | 置、和。 |
| 2: | **while** **do** |
| 3: | **if** **do** |
| 4: | 置。 |
| 5: | **else** |
| 6: | 置。 |
| 7: | **end** **if** |
| 8: | **end** **while** |
| 9: | 。 |
| **输出：**。 | |

算法10.3中，为一维搜索的精度，和为当前的搜索区间，为搜索区间的中点。

* + 1. 非精确一维搜索

在实际计算过程中，选择最优步长的精确搜索方法往往计算量较大，特别是在当前迭代点与最优解距离较远时，搜索效率很低。此外，很多最优化算法的收敛速度本身并不依赖于精确的线性搜索过程。这促使部分研究选择放宽线性搜索的精确性要求，而去确保目标函数在每次迭代中具有满意的下降量，这类方法可以称为非精确的一维搜索方法或可接受的一维搜索方法，它可以在实际计算过程中大大节省计算量。通常来说，非精确一维搜索中步长的确定应在能够使得目标函数保证足够下降的同时，尽可能选择较大的步长取值，因为过小的步长可能会使得迭代过程在原地“徘徊”，进而影响收敛速度。

非精确一维搜索通常基于一定的准则来确定步长。常用的几种准则主要有：

（1）**最小化准则**。在每次迭代中，步长应满足如下条件：

|  |  |  |
| --- | --- | --- |
|  |  |  |

（2）**近似最小化准则**。在每次迭代中，步长应满足如下条件：

|  |  |  |
| --- | --- | --- |
|  |  |  |

（3）**Armijo准则**。设置标量,,,,其中，，。步长从集合中选取最大的元素使得如下条件成立：

|  |  |  |
| --- | --- | --- |
|  |  |  |

（4）**有限最小化准则**。设置标量,,其中，步长由下式定义：

|  |  |  |
| --- | --- | --- |
|  |  |  |

（5）**Goldstein准则**。定义为一固定参数，步长应满足如下条件：

|  |  |  |
| --- | --- | --- |
|  |  |  |

可以证明，如果有下界，存在一个关于的区间满足上式。

（6）**强Wolfe准则**。步长应同时满足如下两个条件：

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

其中和为参数。

（7）**Wolfe准则**。步长应同时满足式和如下条件：

|  |  |  |
| --- | --- | --- |
|  |  |  |

（8）**修改的Armijo准则**。设置标量,,,,,其中，，,,。步长从集合中选取最大的元素使得如下条件成立：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，可以解释为的Lipschitz常数。

可以证明，基于修改的Armijo准则计算得到的步长大于等于由Armijo准则得到的步长，且至少具有线性的收敛速率。下面，我们以修改的Armijo准则为例介绍步长的确定方法。

在准则（8）中，首先需要解决的问题便是Lipschitz常数的评估。在实际计算中无法提前预知，可以通过以下方法进行近似：

|  |  |  |
| --- | --- | --- |
|  |  |  |

或者

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中,为一个正整数,且。

步长需要从集合中选取并检验是否满足一维搜索准则。由于，集合中的元素是依次递减的，我们只需要从前到后依次检测此集合中的元素，并选择第一个满足式的元素设置为步长即可。

* + 1. 测试问题

采用黄金分割法和二分法求函数在区间[0, 10]上的极小点，要求缩短后的区间长度不大于原区间长度的1%。

**标准答案：**极小点为。

* 1. 无约束优化问题的求解算法

无约束优化问题的求解已被各国学者广泛研究并取得了丰富的研究成果，并提出了很多有效算法。一般而言，无约束优化问题的求解算法可以分为两类：（i）要求目标函数的导数信息，（ii）不需要目标函数的导数信息。第一类方法包括最速下降法、广义牛顿法、共轭方向法以及变尺度方法等；第二类方法通常指搜索法，包括Hooke-Jeeves直接搜索法和随机搜索法等。这里主要介绍最速下降法和牛顿法。

* + 1. 最速下降法

假设无约束优化问题的目标函数具有二阶连续偏导数。为当前解，最速下降法采用目标函数的负梯度方向作为下降方向：

|  |  |  |
| --- | --- | --- |
|  |  |  |

最速下降法采用如下方法确定搜索步长：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，为函数在处的海塞矩阵。

以上步长为精确步长。如采用该步长进行迭代，则两个相邻迭代点的梯度正交，即有

|  |  |  |
| --- | --- | --- |
|  |  |  |

依据无约束优化问题的最优性条件，最速下降法采用如下收敛准则：

|  |  |  |
| --- | --- | --- |
|  |  |  |

由于负梯度方向的最速下降性和正梯度方向的最速上升性。人们很容易认为梯度方向是最理想的搜索方向。必须指出点处的梯度方向，仅在点的一个小邻域内才具有最速下降的性质，而对于整个优化过程来说，那就是另一回事了。最速下降法的最初几步函数值变化可能会非常显著，但是迭代到最优点附近时，目标函数值的变化程度可能会非常小。因此，最速下降法经常与其他方法联合使用。在前期使用最速下降法，而在接近最优点时使用其他方法。

* + 1. 牛顿型算法

如果非线性目标函数具有二阶连续偏导，为了得到求解无约束问题收敛速度较快的算法，我们可以利用当前解的二阶泰勒展开式来近似目标函数，并用处指向近似二次函数的极小点方向作为搜索方向。依据该思路，在点取的二阶泰勒展开，我们有

|  |  |  |
| --- | --- | --- |
|  |  |  |

则其梯度为：

|  |  |  |
| --- | --- | --- |
|  |  |  |

依据无约束优化问题的最优性条件，这一近似函数的极小点应满足：

|  |  |  |
| --- | --- | --- |
|  |  |  |

从而有，即

|  |  |  |
| --- | --- | --- |
|  |  |  |

如果是二次函数，则其海塞矩阵为常数，式是精确的。在这种情况下，当海塞矩阵正定时，从任意一点出发，利用式只要一步即可求出的极小点。如果不是二次函数，式仅为一个近似表达式。此时，按式求得的极小点，只是的近似极小点。在这种情况下，选取下式作为搜索方向：

|  |  |  |
| --- | --- | --- |
|  |  |  |

选取固定的步长，我们就可以得到牛顿型迭代格式：

|  |  |  |
| --- | --- | --- |
|  |  |  |

按照以上方式求函数极小点的方法称为牛顿法，式所示的搜索方法称为牛顿方向。牛顿法收敛的速度很快，当的二阶导数及其海塞矩阵的逆矩阵便于计算时，这一方法非常有效。然而，在实际应用中，牛顿法还可能存在如下问题：（i）牛顿方向不一定是下降方向，经迭代，目标函数值可能上升，进而导致牛顿法可能不收敛；（ii）即使目标函数值下降，得到的也不一定是沿牛顿方向最好的点。

为了一定程度上克服以上问题，可以采用阻尼牛顿法，即增加沿牛顿方向的一维搜索。牛顿法和阻尼牛顿法具有如下共同的缺点：（i）海塞矩阵可能奇异（不可逆）；（ii）即时海塞矩阵非奇异，牛顿方向也不一定是下降方向。

为了克服以上缺陷，可以对牛顿法做进一步修正：构造一个对称正定矩阵，取代海塞矩阵，得到如下搜索方向：

|  |  |  |
| --- | --- | --- |
|  |  |  |

* + 1. BFGS方法

Broyden–Fletcher–Goldfarb–Shanno (BFGS)方法是一种拟牛顿法。该方法使用BFGS矩阵作为拟牛顿法中的对称正定迭代矩阵来近似目标函数的海塞矩阵。由于BFGS法对一维搜索的精度要求不高，并且由迭代产生的BFGS矩阵不易变为奇异矩阵，因而BFGS法相比其他拟牛顿法在计算中具有更好的数值稳定性。BFGS算法一般选取单位矩阵作为初始对称正定矩阵，而采用下式对矩阵进行更新：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，，。

把式代入到式计算搜索方向时，需要计算矩阵的逆。这在大规模问题的计算中是比较耗时的。BFGS方法可以采用如下方法更新矩阵的逆：

|  |  |  |
| --- | --- | --- |
|  |  |  |

于是，把式代入到式可以计算得到搜索方向。依据需要，可以采用固定搜索步长或者利用一维搜索确定搜索步长。

* + 1. 最小二乘法问题

最小二乘法是一种在误差估计、不确定度、系统辨识及预测、预报等数据处理诸多学科领域得到广泛应用的数学工具。它通过最小化误差的平方和寻找数据的最佳函数匹配。

选取一组个已知函数，其中，可以构造如下组合函数：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，向量为待定参数。

未知函数有个已知的观测点，其中。令表示残量，称为残差向量（residual vector）。最小二乘法就是通过确定参数向量使得残量的平方和最小来得到拟合函数的方法。换句话说，最小二乘法可以描述为如下非线性规划问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

非线性规划问题取极值的条件是目标函数的梯度为零。于是，我们有

|  |  |  |
| --- | --- | --- |
|  |  |  |

整理式，我们有

|  |  |  |
| --- | --- | --- |
|  |  |  |

通过求解线性方程组，我们可以得到非线性规划问题的最优解。特别地，当我们做线性拟合时，有、和。代入到线性方程组中，我们有

|  |  |  |
| --- | --- | --- |
|  |  |  |

等价地，线性方程组可以表示如下：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，，，，和。

通过求解线性方程组，可以得到线性拟合函数。当我们做二次多项式拟合时，有、、、。代入到线性方程组中，我们有

|  |  |  |
| --- | --- | --- |
|  |  |  |

等价地，线性方程组可以表示如下：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，，,和。

通过求解线性方程组，可以得到二次多项式拟合函数。在实际问题中，如何选取基函数()是一个复杂的问题。一般要根据问题的性质来决定。通常可以取的基函数有多项式、三角函数、指数函数、样条函数等。其中多项式函数使用最为广泛。

下面给出两个最小二乘法问题的例子：

1. 依据下表中弹簧拉伸的观察数据，试建立拉伸强度与拉伸倍数的线性回归方程。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 编号 | 拉伸倍数 | 强度（kg/mm2) | 编号 | 拉伸倍数 | 强度（kg/mm2) |
| 1 | 1.9 | 1.4 | 13 | 5.0 | 5.5 |
| 2 | 2.0 | 1.3 | 14 | 5.2 | 5.0 |
| 3 | 2.1 | 1.8 | 15 | 6.0 | 5.5 |
| 4 | 2.5 | 2.5 | 16 | 6.3 | 6.4 |
| 5 | 2.7 | 2.8 | 17 | 6.5 | 6.0 |
| 6 | 2.7 | 2.5 | 18 | 7.1 | 5.3 |
| 7 | 3.5 | 3.0 | 19 | 8.0 | 6.5 |
| 8 | 3.5 | 2.7 | 20 | 8.0 | 7.0 |
| 9 | 4.0 | 4.0 | 21 | 8.9 | 8.5 |
| 10 | 4.0 | 3.5 | 22 | 9.0 | 8.0 |
| 11 | 4.5 | 4.2 | 23 | 9.5 | 8.1 |
| 12 | 4.6 | 3.5 | 24 | 10.0 | 8.1 |

**参考答案**：把数据代入线性方程组，有

|  |  |  |
| --- | --- | --- |
|  |  |  |

求解以上线性方程组，有和。于是，我们可以得到线性回归方程：

|  |  |  |
| --- | --- | --- |
|  |  |  |

由图10.4可以看出，拉伸强度与拉伸倍数具有比较好的线性关系。



图10.4：拉伸强度与拉伸倍数的线性回归

（2）已知数据如下表所示，试用最小二乘法做二次多项式拟合。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0.0 | 0.2 | 0.5 | 0.7 | 0.85 | 1.0 |
|  | 1.000 | 1.221 | 1.649 | 2.014 | 2.340 | 2.718 |

**参考答案**：把数据代入线性方程组，有

|  |  |  |
| --- | --- | --- |
|  |  |  |

求解以上线性方程组，有、和。于是，我们可以得到二次多项式拟合函数：

|  |  |  |
| --- | --- | --- |
|  |  |  |

由图10.5可以看出，针对给出的数据，二次多项式拟合具有非常好的拟合效果。



图10.5：二次多项式拟合

* 1. 线性约束的非线性规划问题的求解方法

考虑如下只有线性约束的非线性规划问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，是维决策变量，是矩阵，是维列向量，是矩阵,是维列向量，目标函数在可行域上可微。

* + 1. Zoutendijk的可行方向法

我们有如下定理：

**定理10.11：**令是问题的一个可行解，假定，，其中，，非零向量是处的可行方向，则当且仅当，，是处的可行下降方向。

设是问题的可行点，依据问题中的不等式约束条件在处是否取等式，对矩阵和向量进行分解，即有使得和。可以采用如下线性规划问题得到处的可行下降方向：

|  |  |  |
| --- | --- | --- |
|  |  |  |

或者

|  |  |  |
| --- | --- | --- |
|  |  |  |

线性规划问题和中的最后一个约束条件可以确保问题具有有界的最优解。此外，由于我们的目的在于寻找可行的下降方向，所以只要知道的各分量的相对大小即可。令和分别为线性规划问题（或）的最优解和最优目标函数值。由于是可行解，且有目标函数值。因此，我们有。若，则，并且是一个可行的下降方向；否则，，并且是一个K-T点。

Zoutendijk的可行方向法通过求解如下一维搜索问题来确定搜索步长：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中

|  |  |  |
| --- | --- | --- |
|  |  |  |

，

需要注意的是：

1. 由于计算总有误差，所以一般来说，不可能准确成立，可取一个允许误差，若，计算即可结束。
2. 若可行点为内点，即，这时不必求解线性规划问题，可直接取。
3. 上述的*Zoutendijk*方法可能出现“锯齿现象”，使得收敛速度很慢，甚至不收敛到K-T点。1972年Wolfe曾举例说明上述的算法产生的点列可能不能收敛到该问题的K-T点。
   * 1. Frank-Wolfe算法

Frank-wolfe算法是Frank和Wolfe（1956）提出的求解线性约束下的非线性规划问题的一种算法。考虑如下优化问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，是优化问题的可行域，它由一组线性约束条件构成。

设目标函数在可行域上可微，点，目标函数在处的线性近似为：

|  |  |  |
| --- | --- | --- |
|  |  |  |

用上式右边的线性函数来近似代替原目标函数，则在的邻域有近似的线性规划问题如下：

|  |  |  |
| --- | --- | --- |
|  |  |  |

线性规划问题等价于如下线性规划问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

令为线性规划问题的最优解。由于是凸集，，，对于任意,我们都有。有如下定理：

**定理10.12：**设可微，。如果是近似线性规划问题的最优解，则

(1) 当时，是优化问题的K-T点；

(2) 当时，向量是在点处在可行域上的可行下降方向。

依据定理10.12，Frank-Wolfe算法采用作为搜索方向（即可行的下降方向）。基于该搜索方向，通过求解如下一维搜索问题来确定搜索步长：

|  |  |  |
| --- | --- | --- |
|  |  |  |

与*Zoutendijk*方法一样，一般采用作为Frank-Wolfe算法的终止条件。Frank-Wolfe算法的特点是通过对目标函数的近似线性化，把问题归为求解一系列辅助得线性规划问题。该方法简便，但收敛速度较慢。由于所求得各线性规划具有相同的约束条件，在有现成的求解线性规划程序的情况下，此方法也常被人们所采用。

* + 1. MSA算法

相继平均法（Method of Successive Averages，简称MSA）是另一种常被人们所采用的求解优化问题的方法。MSA算法与Frank-Wolfe算法使用相同的搜索方向，不同之处在于搜索步长的选取。MSA算法采用预先给定的步长，即第次迭代的步长。这一预先给定的步长序列满足如下条件:

|  |  |  |
| --- | --- | --- |
|  |  |  |

MSA算法的第次迭代为：

|  |  |  |
| --- | --- | --- |
|  |  |  |

展开上式，得

|  |  |  |
| --- | --- | --- |
|  |  |  |

可见是前面产生的个向量得平均值，这也是MSA算法名称的由来。与Frank-Wolfe算法相比较，这种方法的优点是：在每次迭代过程中，不需要通过求解线性搜索问题得到迭代步长，迭代步长是预先确定的。因而MSA算法计算简单，具有明显的实用价值。该方法的不足之处在于：由于MSA算法没有考虑迭代过程中当时的情况，因而可能由于步长不合适导致收敛速度比较慢。

* 1. 参考文献

[1] 郭耀煌等编著. 运筹学原理与方法. 成都：西南交通大学出版社，1994

[2] 钱颂迪等编著. 运筹学. 北京：清华大学出版社. 2012

# 用户平衡交通分配问题

* 1. 概述

交通分配（Traffic Assignment）就是将预测出来的OD（Origin-Destination）需求按照一定的规则符合实际地分配到网络中的各条道路上，并求出各条道路的交通流量。交通分配是四阶段交通规划方法中的最后一个阶段。

* 1. 符号及变量定义

在一个具有多个起点和多个讫点的交通网络中，定义和分别为网络中节点的集合和边的集合。设和分别为起点的集合和讫点的集合。我们采用如下符号及变量：

表11.1 符号及变量

|  |  |
| --- | --- |
| 集合 |  |
|  | OD对集合 |
|  | OD对间的路径集合 |
|  | 所有路径构成的集合，我们有 |
|  | 路径*p*上的所有路段构成的集合 |
| 参数 |  |
|  | 路段的通行能力 |
|  | 路段的自由流出行费用 |
|  | 0-1变量。如果路段在路径上，则；否则， |
|  | OD对间的交通需求 |
| 变量 |  |
|  | 路段的交通流量 |
| **x** | 向量**x** |
|  | 路径的交通流量 |
|  | 向量 |
|  | 向量 |
|  | 路段的出行费用 |
|  | 路径的出行费用 |
|  | 向量 |
|  | 路径走行时间向量关于路径流量向量的函数 |
|  | 向量 |

路径流量与OD需求之间的关系可以表述如下：

|  |  |  |
| --- | --- | --- |
|  |  |  |

路径流量与路段流量之间的关系可以表述如下：

|  |  |  |
| --- | --- | --- |
|  |  |  |

在用户平衡交通分配问题中，我们采用美国联邦公路局(BPR—Bureau of public road)开发

的阻抗函数，即BPR函数，来计算路段阻抗：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，和为模型参数，通常取和。

* 1. 数学模型
     1. Beckmann模型

用户平衡交通分配问题可以描述为如下数学规划问题（Beckmann等，1956）：

|  |  |  |
| --- | --- | --- |
|  |  |  |

数学规划问题是一个线性约束下的非线性规划问题。其中，目标函数是人工构造出来的，没有确切的物理含义。当阻抗函数是一个严格凸函数时，数学规划问题是一个凸规划问题，且路段流量具有唯一最优解。需要注意的是，路径流量通常不具有唯一性。

* + 1. 变分不等式模型

用户平衡交通分配问题可以描述为如下基于路径的变分不等式问题：即寻找，使得

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，

|  |  |  |
| --- | --- | --- |
|  |  |  |

Beckmann模型只能应用于描述具有对称的路段阻抗函数的UE交通分配问题，而变分不等式是描述UE交通分配问题的一般化方法。求解变分不等式问题得到的是路径流量，可以通过式进一步得到路段流量。

* 1. 间隙函数

在UE交通分配问题的求解过程中，我们采用如下间隙函数（Gap Function）来评价解的精度：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，是流量下OD对*w*之间的最小出行费用。当不是UE均衡解时，我们有。

* 1. Frank-wolfe算法

把模型中的第二个约束条件带入到目标函数中，我们可以等价地得到如下基于路径的UE交通分配模型：

|  |  |  |
| --- | --- | --- |
|  |  |  |

模型的目标函数的梯度为：

|  |  |  |
| --- | --- | --- |
|  |  |  |

令为模型的一个可行解（即一组可行的路径流量），为对应的路段流量构成的向量，为对应的路径走行时间构成的向量。依据定义，有

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |

利用目标函数在处的线性近似，我们可以构造如下线性规划问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

令为模型的最优解（即全有全无交通分配的路径流量），为对应的路段流量构成的向量（即全有全无交通分配的路段流量）。依据定理7.1，是UE交通分配问题在处的可行下降方向。

定义如下函数：

|  |  |  |
| --- | --- | --- |
|  |  |  |

依据式，我们可以得到如下一元方程：

|  |  |  |
| --- | --- | --- |
|  |  |  |

令是一元方程的根，则就是沿可行下降方向的最优步长。我们可以采用二分法来求解方程的根。

全有全无交通分配是UE交通分配算法中的重要步骤，实现全有全无交通分配的详细算法步骤如下：

|  |  |
| --- | --- |
| **算法11.1：**全有全无交通分配 | |
| **输入:** 路段流量 | |
| 1: | 置 |
| 2: | 依据路段流量更新所有路段的走行时间 |
| 3: | **for each**  **do** |
| 4: | 采用最短路算法，求到所有目的地的最短路径和该路径上的路段集合 |
| 5: | **for each**  **do** |
| 6: | **for each**  **do** |
| 7: | 置 |
| 8: | **end** **for** |
| 9: | **end** **for** |
| 10: | **end** **for** |
| **输出：**全有全无交通分配得到的路段流量 | |

在算法11.1中，第4行可以采用Lable-setting或者Label-correcting的方法同时把给定起点到所有目的地的最短路径都求出来。第6-8行把给定OD对(r, s)的交通需求全部加载到该OD对的最短路径上去。

采用Frank-wolfe算法求解UE交通分配问题的详细算法步骤如下：

|  |  |
| --- | --- |
| **算法11.2：**Frank-wolf算法求解UE交通分配问题 | |
| **输入:** 精度 | |
| 1: | 基于所有路段的自由流走行时间，利用算法11.1做全有全无交通分配，得到初始的路段流量，置迭代次数 |
| 2: | **while** **do** |
| 3: | 以为输入，利用算法11.1做全有全无交通分配，得到路段流量 |
| 4: | 置可行的下降方向 |
| 5: | 依据进行一维搜索得到最优步长 |
| 6: | 更新路段流量 |
| 7: | 置 |
| 8: | **end** **while** |
| **输出：** | |

可以采用式计算Line 2中的。单独计算间隙函数需要运行最短路算法来求OD对之间的最小出行费用。为了提升计算效率，的计算可以在全有全无交通分配过程中完成。第5行中的一维搜索可以采用算法10.3来实现。

* 1. 投影算法
     1. 投影的计算

**定理11.1** 向量是VI问题的解，当且仅当对任意的，都是如下不动点问题的解：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，表示在上的欧几里德投影。

采用投影算法求解基于路径的UE交通分配问题时，通常需要计算如下投影问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，为第次迭代的路径流量，为第次迭代的投影步长。

在UE交通分配问题中，可行域具有可分解的结构特征。我们有

其中

可行域可分解的结构特征使得投影问题可以依据OD进行分解，即只需要求解如下一系列的投影问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

* + 1. 线性投影

我们考虑如下投影问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，n为投影空间的维度，d为参数，，。

线性投影的详细算法步骤如下：

|  |  |
| --- | --- |
| **算法11.3：**线性投影算法 | |
| **输入:** 向量和参数 | |
| 1: | 对任意置，置迭代次数 |
| 2: | **while** 不成立 **do** |
| 3: | 计算得到集合 |
| 4: | **for each**  **do** |
| 5: | **If**  then |
| 6: |  |
| 7: | **else** |
| 8: |  |
| 9: | **end** **if** |
| 10: | **end** **for** |
| 11: | 置 |
| 12: | **end** **while** |
| **输出：**向量 | |

令n等于OD对w之间的路径数量，d等于OD对w之间的交通需求，，，则算法11.3可以应用于计算投影问题。

利用算法11.3可以计算投影问题，详细的计算步骤如下：

|  |  |
| --- | --- |
| **算法11.4：**投影问题的计算过程 | |
| **输入:** 向量 | |
| 1: | **for each** **do** |
| 2: | 利用算法11.3计算投影 |
| 3: | **end** **for** |
| **输出：**向量 | |

* + 1. 求解UE交通分配问题的投影算法

采用投影算法求解UE交通分配问题的详细步骤如下：

|  |  |
| --- | --- |
| **算法11.5：**求解UE交通分配问题的投影算法 | |
| **输入:** 投影步长、精度、路径集更新频率 | |
| 1: | 基于自由流走行时间，利用算法11.1做全有全无交通分配，得到初始的路径流量向量、路段流量、路段费用向量、和路径费用向量。采用全有全无交通分配中得到的各OD对的最短路径构成初始的路径集，置迭代次数。 |
| 2: | **while** **do** |
| 3: | **if** 对 取余等于0 then |
| 4: | 基于路段费用向量，求任意OD对的最短路径。如果，则，并置。 |
| 5: | **end if** |
| 6: | 利用算法11.4计算投影 |
| 9: | 利用路径流量，更新路段流量、路段费用向量、和路径费用向量。置。 |
| 10: | **end** **while** |
| **输出：** | |

算法11.5中，第2行中的间隙函数可以采用式来计算，第3-5行用于更新OD对之间的路径集。在本质上是一个不动点迭代算法。在算法11.5中，投影步长需要满足。其中，为函数的李普西兹常数。在实际应用中，函数的李普西兹常数很难得到。的取值太小会导致算法收敛速度慢，而的取值过大则可能导致算法不收敛。此外，投影算法收敛的充分条件为：路径阻抗函数连续且严格单调。

* + 1. 求解UE交通分配问题的双投影算法

求解UE交通分配问题的双投影算法可以详细表述如下：

|  |  |
| --- | --- |
| **算法11.6：**求解UE交通分配问题的双投影算法 | |
| **输入:** 投影步长、精度、参数和、路径集更新频率 | |
| 1: | 基于自由流走行时间，利用算法11.1做全有全无交通分配，得到初始的路径流量向量、路段流量、路段费用向量、和路径费用向量。采用全有全无交通分配中得到的各OD对的最短路径构成初始的路径集，置和迭代次数。 |
| 2: | **while** **do** |
| 3: | **if** 对 取余等于0 then |
| 4: | 基于路段费用向量，求任意OD对的最短路径。如果，则，并置。 |
| 5: | **end if** |
| 6: | 利用算法11.4计算投影 |
| 7: | **while** **do** |
| 8: |  |
| 9: | 利用算法11.4计算投影 |
| 10: | **end** **while** |
| 11: | 利用算法11.4计算投影 |
| 12: | 利用路径流量，更新路段流量向量、路段费用向量、和路径费用向量。置。 |
| 13: | **end** **while** |
| **输出：**路径流量向量 | |

算法11.6中，第7-10行用于确定合适的投影步长，第6和9行实质上是做了一个路径阻抗的预测，第11步利用确定好的投影步长和预测的路径阻抗做投影更新路径流量。算法11.6与算法11.5相比，增加了步长搜索，这使得算法对参数具有很好的适应性。此外，在理论上双投影算法具有更宽松的收敛条件，即要求路径阻抗函数连续且伪单调。

* 1. 系统最优交通分配
     1. 系统最优交通分配模型

系统最优原则假设出行者能接受统一的调度，以系统总阻抗最小为目标完成出行。系统最优交通分配问题可以表述为如下数学规划问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

以上模型的目标函数的Hassian矩阵是一个对角矩阵，对于路段行走时间函数是一个典型的凸的升函数。因此，目标函数的Hassian矩阵是一个正定矩阵，即目标函数是一个严格凸函数。约束集是凸集，目标函数是严格凸函数，故SO模型是一个严格凸规划，有唯一的路段流量解。

* + 1. UE模型与SO模型之间的相互转换

比较UE模型和SO模型，可以发现二者的目标函数不同，但约束条件完全一致。故二者之间可以进行转化。我们可以通过对走行时间函数的修正，来实现UE模型和SO模型的相互转化，并利用UE模型的求解算法来求解SO模型。

如果把UE模型中的路段阻抗函数设置为，则可以得到一个新的UE模型。其中，通常被称为路段*a*的边际出行成本，即路段*a*上增加一个单位交通量而引起网络总出行成本的变化量。依据下式

|  |  |  |
| --- | --- | --- |
|  |  |  |

可知新得到的SO模型等价于SO模型。

反之，如果把SO模型中的路段阻抗函数设置为，我们可以得到一个新的SO模型。依据下式

|  |  |  |
| --- | --- | --- |
|  |  |  |

可知新得到的SO模型等价于UE模型。

* 1. Greedy算法

Greedy算法是求解大规模UE交通分配问题（large-scale UE-TAP）常用的一种基于路径（path-based）的求解方法。算法首先将UE交通分配的数学模型分解成O-D子问题，通过将更多的计算成本投放在收敛性更差的O-D对上，进而有效提升计算效率和收敛性能。

* + 1. Greedy算法基本原理

首先，回顾经典UE交通问题的数学表达式，如下式(10.29)- (10.32)。

(10.29)

s.t.

(10.30)

(10.31)

(10.32)

经典UE交通问题的凸规划形式(10.29)- (10.32)是一个对于变量的可分离规划。即目标函数和所有的约束条件，都被能表述为单O-D变量函数的之和。通过拆分，能将复杂的可分离规划拆分为多个解法类似但更简单的子问题。如果算法在迭代过程中，能够把更多的计算成本投放在收敛性更差的子问题上，就能有效提升计算效率和收敛性能。

基于路径（path-based）的大规模UE求解算法，核心就在于对O-D对的分解，也称Gauss-Seidel分解。由于分解后的O-D子问题是连续求解的，因此我们只需要关注一个O-D子问题，将该O-D子问题表述如下：

(10.33)

s.t.

(10.34)

(10.35)

(10.36)

其中：表示仅由O-D对的需求形成的路段的流量，。

对O-D子问题的高效求解，是众多学者开发path-based算法的核心内容，也是掌握Greedy算法的要点。

* + 1. Greedy算法基本框架

一个求解基于路径UE交通分配问题的基本迭代框架如下：

|  |  |  |
| --- | --- | --- |
| **算法11.7：**求解基于路径UE交通分配问题的基本框架 | | |
| **输入:** 精度， | | |
| 1: | **初始化（lines 2-6）：** | |
| 2: | foreach O-D对: | |
| 3: | 求的最短路径，设路径的流量，并将添加到的路径集合。 | |
| 4: | foreach : | |
| 5: | 通过式(10.32)更新路段流量，更新路段成本和路段成本的导数。 | |
| 6: | 置迭代次数。 | |
| 7: | **主循环（lines 8-24）：** | |
| 8: | while true: | |
| 9: | **列更新（lines 10-13）：** | |
| 10: | foreach : | |
| 11: | 采用最短路算法，求到所有的最短路径,和上的路段集合。 | |
| 12: | foreach : | |
| 13: | if : 将添加到。 | |
| 14: | **自适应内循环（lines 15-22）：** | |
| 15: | 设，，。 | |
| 16: | while : | |
| 17: | if : | |
| 18: | 调整所有O-D对的路径流量，生成限制O-D对集合。（算法11.8） | |
| 19: | if : break。 | |
| 20: | 调整集合中O-D对的路径流量，生成新限制O-D对集合。（算法11.9） | |
| 21: | if : break。 | |
| 22: | else: ，，重复执行自适应内循环。 | |
| 23: | 计算，置迭代次数。 | |
| 24: | if : break。 | |
| 25: | else: ，重复执行主循环。 | |
| **输出：平衡时的路段流量。** | |  |

初始化为算法的第2-6行。采用全有全无方法给每个子问题生成初始路径，并更新路段流量、路径流量、路段阻抗、等基本变量。

主循环为算法的第8-24行。主要包括列更新和OD子问题的求解，是求解算法占用计算成本最高的部分。其中，列更新用于探索新的、阻抗（或成本）更小的路径。调整限制集合中O-D对的路径流量和单O-D对子问题的求解是算法的核心，将在后文中详细讲解。

自适应内循环为算法的第15-22行。对收敛性差的O-D对，增加该O-D对子问题的探索的次数，可以有效提升算法的收敛效率。

收敛性检验为算法的第23-25行。如果计算收敛精度达到了提前预设的精度要求，则迭代停止，否则重复执行主循环和自适应内循环。的计算公式如下：

(10.37)

其中，表示O-D对上的最短路径成本。

调整所有O-D对的路径流量，并生成限制O-D对集合的算法如下：

|  |  |  |
| --- | --- | --- |
| **算法11.8：**调整所有O-D对的路径流量 | | |
| 1: | 初始化空集合 | |
| 2: | foreach O-D对: | |
| 3: | 计算O-D对的。 | |
| 4: | if : | |
| 5: | 将添加到。 | |
| 6: | 用Greedy算法求解O-D 的单O-D子问题，并更新路段流量、路段成本和路段成本的导数。（算法11.10） | |
| **输出：集合。** | |  |

调整限制集合中O-D对的路径流量，生成新限制O-D对集合的算法如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **算法11.9：**调整限制集合中O-D对的路径流量 | | | |
| **输入：集合。** | |  | |
| 1: | 初始化空集合 | | |
| 2: | foreach O-D对: | | |
| 3: | 计算O-D对的。 | | |
| 4: | if : | | |
| 5: | 将添加到。 | | |
| 6: | 用Greedy算法求解O-D的单O-D子问题，并更新路段流量、路段成本和路段成本的导数。（算法11.10） | | |
| **输出：集合。** | | |  |

其中，O-D对的Gap函数的计算公式为：

(10.38)

* + 1. Greedy单O-D子问题求解方法

单O-D对子问题求解算法的设计，是基路径的UE求解算法中最重要的环节。子问题算法的求解效率不仅制约算法整体的计算时间，还对算法的收敛性有着重要的影响。

对式(10.33) 目标函数做二阶近似泰勒展开如下：

(10.39)

令：

(10.40)

(10.41)

移除所有的常数项，可将单O-D子问题的二次逼近形式表述为：

(10.42)

s.t.

(10.43)

(10.44)

写出以上规划的KKT条件（Karush-Kuhn-Tucker conditions），其中是关于O-D对的乘子。

(10.45)

(10.46)

可以将理解为O-D对之间的最小出行成本，二次逼近问题实际上是在当前解处采用以下线性函数逼近：

(10.47)

很容易计算目标函数对于路径流量的一阶和二阶偏导数：

(10.48)

(10.49)

引入常量来简化公式符号：

(10.50)

假设O-D对之间被使用的路径集是已知的，那么可将KKT条件式(10.45)和(10.46)改写如下：

(10.51)

(10.52)

O-D对之间的最小出行成本，和路径流量的计算公式如下：

(10.53)

(10.54)

用Greedy算法求解单O-D对子问题的具体算法如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **算法11.10：**Greedy算法求解单O-D对子问题 | | | |
| **输入：已知的当前路径流量解**。 | |  | |
| 1: | foreach ： | | |
| 2: | 分别根据式(10.48)和(10.49)计算和。 | | |
| 3: | 根据式(10.50)计算。 | | |
| 4: | 将所有的路径按照升序排列， | | |
| 5: | 令，，。 | | |
| 6: | 置，。 | | |
| 7: | while and ： | | |
| 8: | 置。 | | |
| 9: | 置。 | | |
| 10: | 置。 | | |
| 11: | 令。 | | |
| 12: | foreach : | | |
| 13: | 置。 | | |
| 14: | foreach : | | |
| 15: | 置。 | | |
| 16: | 令集合。 | | |
| 17: | foreach : | | |
| 18: | if : | | |
| 19: | foreach : | | |
| 20: | 根据更新路段流量。 | | |
| 21: | if : | | |
| 22: | 。 | | |
| 23: | 对，更新和。 | | |
| 24: | 令。 | | |
| **输出：新的路径流量解**。 | | |  |

算法的第1-15行为用Greedy算法求解O-D 的单O-D子问题；算法的16-23行为路段流量、成本、成本一阶导的更新；算法的第24行是删除流量为0的路径。

* 1. 灵敏度分析

下文介绍的灵敏度分析，是在静态UE交通分配问题达到平衡的条件下，分析平衡路段流量对路段费用函数和OD需求的导数。灵敏度分析的结果通过矩阵运算和求解一个线性方程组得出。被求解线性方程组的维度不超过路段数量。

* + 1. 灵敏度分析计算步骤

Shu Lu（2008）考虑路段费用函数和OD需求扰动，给出了UE均衡条件下，进行灵敏度分析的详细推导过程。灵敏度分析的具体过程如下：

**步骤1：判断O-D需求是否成立。**

将O-D需求的O-D对添加到O-D对集合，进行灵敏度分析。若存在的O-D对，将其从O-D对集合中删除。

**步骤2：计算UE平衡。**

获取UE平衡状态的路段流量，以及路径流量，路径成本等信息。用来表示OD-路径关系矩阵。若第条路径是第个O-D对之间的路径，那么，否则，。用来表示路段-路径关系，若第条路段在第条路径上，那么，否则，。

**步骤3：将路径划分成**, , , **四类。**

将所有路径按照路径流量和路径成本的大小，分成, , , 四类。四种路径的定义如下式(10.55)-(10.58)所示。

(10.55)

(10.56)

(10.57)

(10.58)

其中，和中的路径为未被使用的路径。未被使用且成本等于O-D间最小路径成本的路径被添加到集合；未被使用且成本大于O-D间最小路径成本的路径被添加到集合。

和为被使用的路径，即路径流量的路径。从O-D对被使用的路径之间，任意选择一条路径添加到集合，其余被使用的路径添加到集合。

按照路径集合, , , 的划分方法，将OD-路径关系矩阵和路段-路径关系矩阵，改写成分块矩阵的形式，如下式(10.58)。

(10.59)

**步骤4：计算矩阵A、B，以及矩阵列向量的基。**

矩阵A、B的计算公式如下(10.60)和(10.61)：

(10.60)

(10.61)

通常情况下，我们用Frank-Wolf算法、投影算法、Greedy算法等计算出的UE平衡解是不包含零流路径的，即和为空集。这种情况下，矩阵、和为空矩阵，也是空矩阵。

用初等变换方法计算矩阵列向量的基，记为。由于矩阵的行数等于路段数量，矩阵中每个列向量的维度也等于路段数量。因此，矩阵列向量的基不超过路段数量个，即矩阵的维度不超过路段数量。

**步骤5：检验灵敏度分析可行的条件。**

用来表示阻抗函数对路段流量的一阶导数，用来表示阻抗函数的扰动，即：

(10.62)

条件1.1（C1.1）：矩阵(10.63)是非奇异的，等价于非奇异。

(10.63)

条件1.2（C1.2）：矩阵(10.64)主余子式为正。

(10.64)

检验以下两个条件是否成立：（a）矩阵正定；（b）C1.1和C1.2同时成立。如果（a）和（b）其中任意一项成立，则平衡时的路段流量在局部区间内对的存在性、局部唯一性和李普希兹连续性成立，然后执行步骤6；如果条件（a）和（b）都不满足，则终止分析，无法得出任何关于平衡流量性质的结论。

**步骤6：检查子空间的性质。**

首先检查矩阵是否为空，如果是，执行步骤7；否则，尝试解决如下规划问题：

(10.65)

s.t.

(10.66)

(10.67)

如果规划问题是可解的，执行步骤7；否则，执行步骤8。

* + 1. 算例分析

给定一个如图11.1所示的样例网络，该样例网络包含4个节点、4条路段和两个O-D对。样例网络的关系矩阵如下：

(10.69)

(10.70)

其中，代表路径，代表路段，代表O-D对。

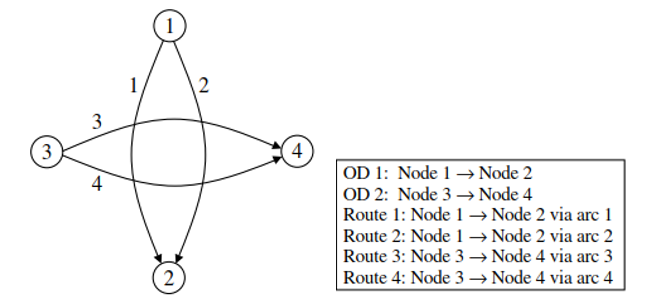


图11.1 样例网络

样例网络的路段成本函数如下：

(10.71)

给定参数，，接下来进行灵敏度分析。

**步骤1：**O-D需求，成立。

**步骤2：**在已知的条件下计算UE平衡。计算出平衡时的路段流量，路段成本。这里，平衡的路径流量结果是唯一的，即，路径成本，O-D最小成本为。

**步骤3：**将路径划分成, , , 四类。其中，，, 。注意和的划分是不唯一的，因为存在多种任选一条路径添加到集合的方式。划分路径后，可将关系矩阵改写成如下分块矩阵的形式：

(10.72)

其中，矩阵，，，为空。

**步骤4：**求解矩阵，和。

(10.73)

矩阵和为空，矩阵为空。因此矩阵。

矩阵的两个列向量是线性无关的，因此矩阵列向量的基。

(10.74)

**步骤5：**检验灵敏度分析是否可行的条件。

矩阵不是正定的，因此检验C1.1和C1.2是否同时成立。矩阵等价于，是非奇异的，因此C1.1成立。矩阵为空，条件C1.2成立。所以平衡路段流量解在局部区间内对的存在性、局部唯一性和李普希兹连续性成立。

**步骤6：**检查子空间的性质。矩阵为空，执行步骤7。

**步骤7：**计算灵敏度分析的梯度矩阵。

(10.75)

(10.76)

**步骤8：**方程在处不是半可微的，无需执行该步骤。

* 1. 测试问题
     1. 对称的UE交通分配问题（Frank-wolf算法和投影算法）

采用Sioux Falls网络为测试网络。有关网络结构、路段阻抗函数参数、OD需求等信息可以通过交通网络测试问题库（<http://www.bgu.ac.il/~bargera/tntp/>）获取。采用BPR函数作为路段阻抗函数。双投影算法的参数设置为：、、、。

有关对称的UE交通分配问题的求解算法的测试内容包括：

* 实现采用Frank-wolf算法、投影算法和双投影算法求解对称的UE交通分配问题，给出路段流量、目标函数、以及系统总阻抗
* 测试投影算法最优的投影步长
* 对比三种算法的求解效率
* 分析交通需求水平对算法效率的影响

图11.2-11.4分别给出了Frank-wolf算法、投影算法和双投影算法求解对称的UE交通分配问题时，算法的收敛过程。从图11.3可以看出，投影算法的收敛受到投影步长的影响。投影步长太小会导致算法收敛速度慢，而投影步长过大则可能导致算法不收敛。图11.5中的结果表明，双投影算法在求解效率和求解精度上都显著优于Frank-wolf算法。

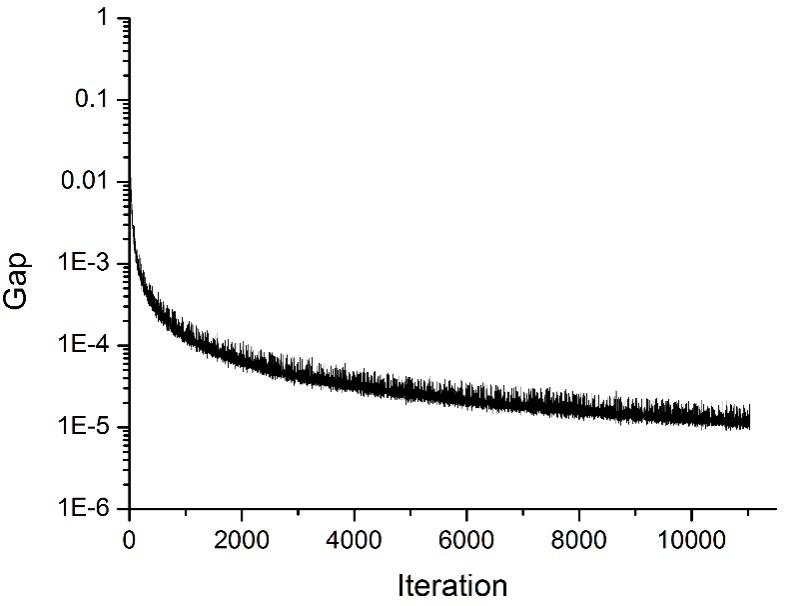


图11.2. Frank-wolf算法求解对称的UE交通分配问题的收敛过程

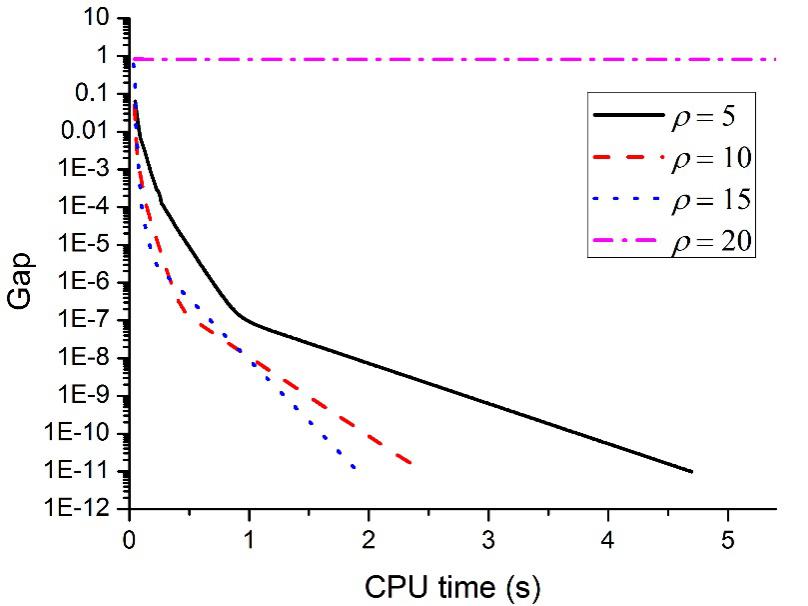


图11.3. 投影算法求解对称的UE交通分配问题的收敛过程

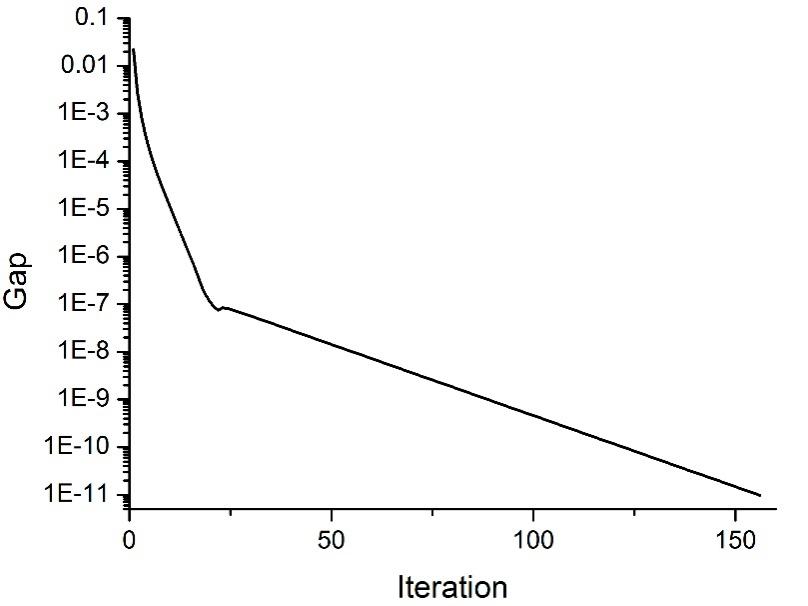


图11.4.双投影算法求解对称的UE交通分配问题的收敛过程

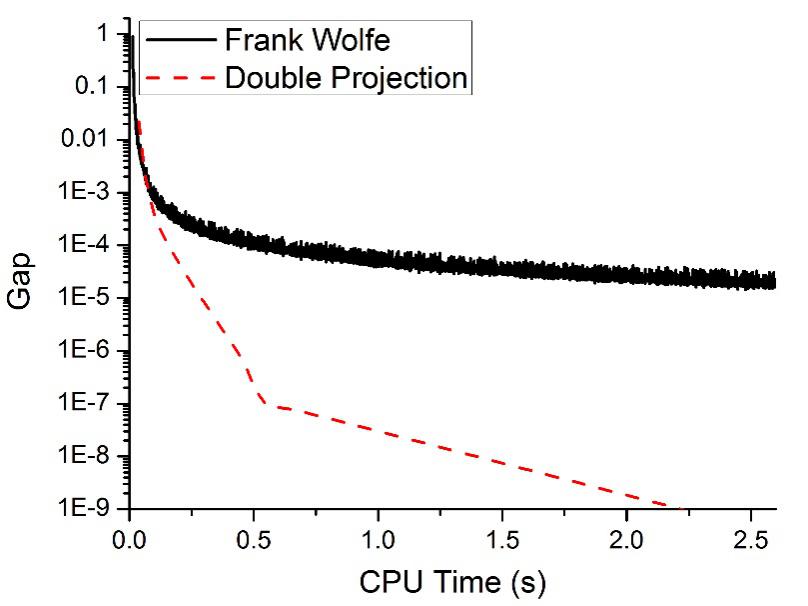


图11.5. Frank-wolf算法和双投影算法求解对称的UE交通分配问题的性能对比

* + 1. 非对称的UE交通分配问题（投影算法）

在非对称的UE交通分配问题中，采用如下路段阻抗函数（Panicucci等，2007）：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，是路段*a*的反向路段。其它的参数设置与对称的UE交通分配问题相同。

有关非对称的UE交通分配问题的求解算法的测试内容包括：

* 实现投影算法和双投影算法求解非对称的UE交通分配问题，给出最优的路径流量和系统总阻抗。
* 对比投影算法和双投影算法求解非对称的UE交通分配问题的收敛过程。

图11.6中的结果表明，投影算法的收敛受到投影步长的影响。投影步长太小会导致算法收敛速度慢，而投影步长过大则可能导致算法不收敛。当投影算法取合适的投影步长时，比双投影算法具有更好的收敛速度。

****

图11.6. 投影算法和双投影算法求解非对称的UE交通分配问题的性能对比

* + 1. SO交通分配问题

以Sioux Falls网络为测试网络，实现采用Frank-wolf算法、投影算法和双投影算法求解对称的SO交通分配问题，给出路段流量、目标函数、以及系统总阻抗。在编程实践中，只需要替换UE交通分配问题的阻抗函数即可实现SO交通分配问题的求解。图11.7-11.9分别给出了Frank-wolf算法、投影算法和双投影算法求解SO交通分配问题时，算法的收敛过程。相关结果求解与UE交通分配问题类似。

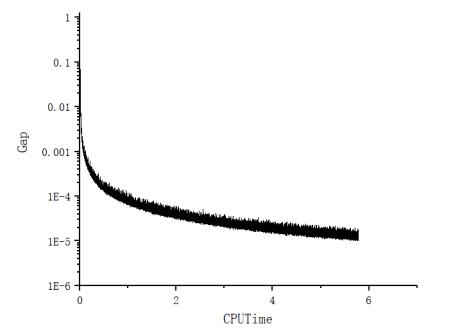


图11.7. Frank-wolf求解对称的SO交通分配问题

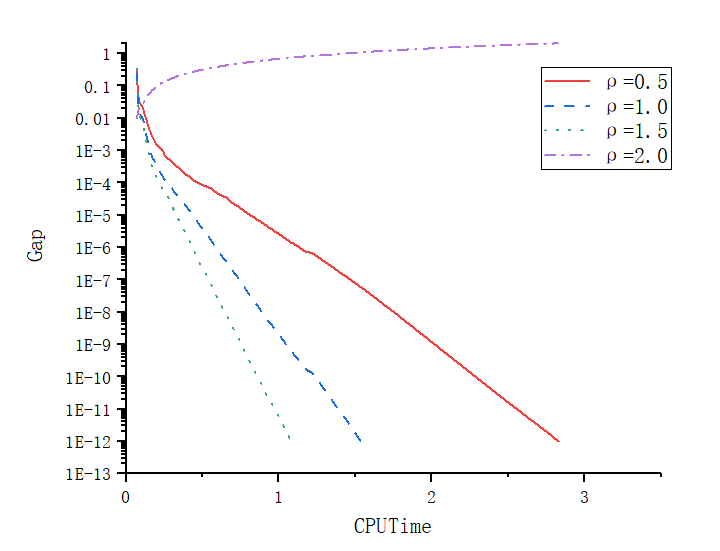


图11.8. 投影算法求解对称的SO交通分配问题

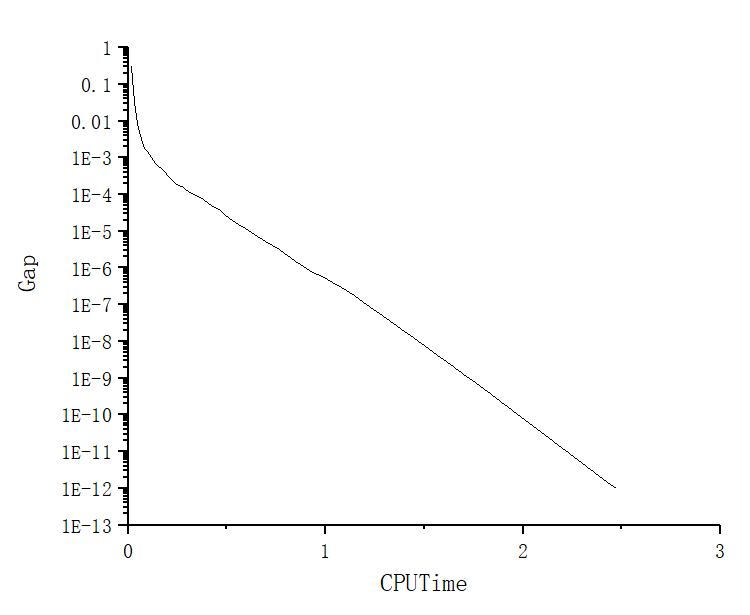


图11.9. 双投影算法求解对称的SO交通分配问题

* + 1. Greedy算法求解UE交通问题

采用Sioux Falls网络和Chicago Sketch网络为测试网络，采用BPR函数作为路段阻抗函数。测试Greedy算法的求解效率。

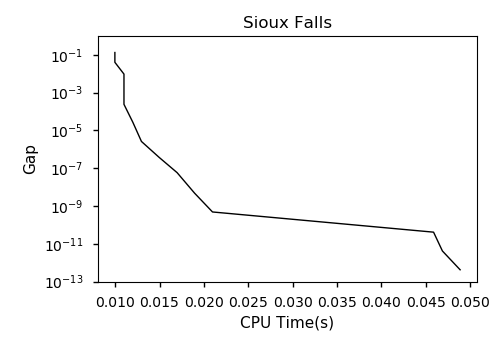


图11.10 Greedy算法求解Sioux Falls网络UE分配问题

表11.2 Greedy算法求解Sioux Falls网络UE分配问题

|  |  |  |
| --- | --- | --- |
| Iteration | Gap | CPU Time(s) |
| 1 | 0.131046 | 0.009998 |
| 2 | 0.039774 | 0.010971 |
| 3 | 0.009476 | 0.010971 |
| 4 | 0.001855 | 0.011999 |
| 5 | 0.000238 | 0.011999 |
| 6 | 2.69E-05 | 0.011999 |
| 7 | 2.60E-06 | 0.013963 |
| 8 | 3.74E-07 | 0.015958 |
| 9 | 5.82E-08 | 0.017985 |
| 10 | 4.77E-09 | 0.018951 |
| 11 | 4.85E-10 | 0.020971 |
| 12 | 4.15E-11 | 0.024933 |
| 13 | 4.19E-12 | 0.026928 |
| 14 | 4.25E-13 | 0.027925 |

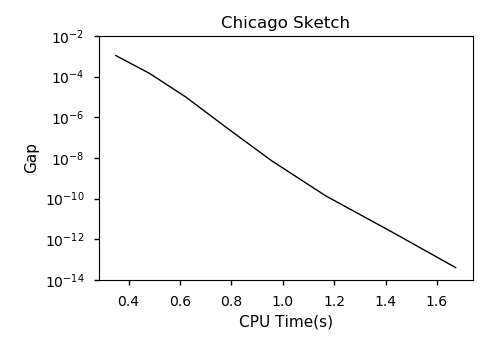


图11.11 Greedy算法求解Chicago Sketch网络UE分配问题

表11.3 Greedy算法求解Chicago Sketch网络UE分配问题

|  |  |  |
| --- | --- | --- |
| Iteration | Gap | CPU Time(s) |
| 1 | 0.001076 | 0.353084 |
| 2 | 0.000144 | 0.485701 |
| 3 | 9.85E-06 | 0.627323 |
| 4 | 2.77E-07 | 0.790884 |
| 5 | 7.25E-09 | 0.964421 |
| 6 | 1.35E-10 | 1.178849 |
| 7 | 2.42E-12 | 1.446164 |
| 8 | 4.09E-14 | 1.697461 |

用Greedy算法求解UE均衡的效率较高。对于包含24个节点，76条路段和528个O-D对的Sioux Falls网络，算法仅需0.05秒就能将Gap函数收敛到10-12。对于包含933个节点，2922条路段和93513个O-D对的Chicago Sketch网络，算法在1.5s左右将Gap函数收敛到10-12。

* + 1. 灵敏度分析

采用Sioux Falls网络为测试网络。有关网络结构、路段阻抗函数参数、OD需求等信息可以通过交通网络测试问题库（<http://www.bgu.ac.il/~bargera/tntp/>）获取。采用如下形式的阻抗函数：

(10.78)

有关灵敏度分析的测试内容包括：

1. 设置为无扰动情况下的O-D需求，设置为0，求解Sioux Falls网络的UE均衡，并用式(10.68)和(10.69)计算均衡状态下的灵敏度分析。
2. 进行需求扰动，将所有O-D对的需求均增大20，用上一问灵敏度分析的结果，预测新的路段流量。将路段流量预测结果与求解UE均衡问题得到路段流量进行对比。
3. 进行阻抗函数扰动，设，用灵敏度分析预测新的路段流量，并与求解UE均衡问题得到路段流量进行对比。

参考答案：

（1）计算Sioux Falls网络灵敏度分析。

**步骤1：**O-D需求成立。

**步骤2：**采用Frank-Wolfe算法、投影算法或Greedy算法计算UE平衡，得到路段流量，路段成本，路径流量、路径成本和O-D最小成本为。共得到770条用户最优路径，这里的路径流量解是不唯一的。

**步骤3：**将路径划分成, , , 四类。其中，，包含707条路径，包含179条路径。注意和的划分也是不唯一的。

**步骤4：**求解矩阵，和。其中，矩阵是一个76×179维的矩阵，矩阵为空，矩阵列向量的基是一个76×29维的矩阵，是一个29×29维的矩阵。

**步骤5：**检验灵敏度分析是否可行的条件。

检验C1.1和C1.2是否同时成立。矩阵等价于，是非奇异的，因此C1.1成立。矩阵为空，条件C1.2成立。所以平衡路段流量解在局部区间内对的存在性、局部唯一性和李普希兹连续性成立。

**步骤6：**检查子空间的性质。矩阵为空，执行步骤7。

**步骤7：**根据式(10.68)和(10.69)计算灵敏度分析的梯度矩阵。点击链接获取参考结果（链接：https://pan.baidu.com/s/16TPzwRFxV0exMwOm1jgo4A 提取码：ies0）。

**步骤8：**方程在处不是半可微的，无需执行该步骤。

（2）O-D需求扰动。

(10.79)

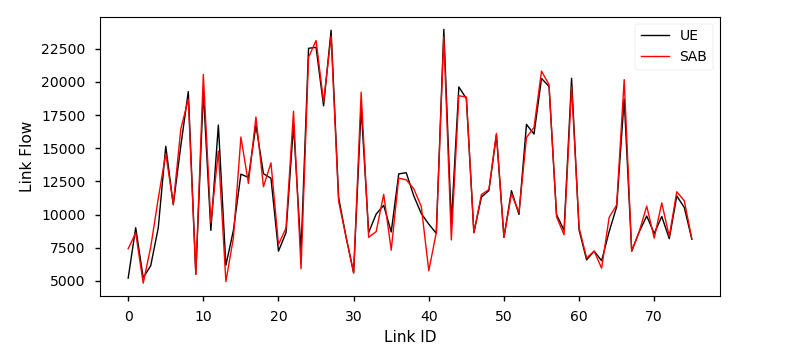


图11.12 SAB预测路段流量与UE均衡路段流量对比（需求扰动）

根据灵敏度分析的结果和式(10.79)预测新的路段流量。预测结果与求解UE均衡问题得到路段流量进行对比如图11.12所示。路段流量预测值与真实值的MAPE为6.28%，预测结果比较准确。

（3）阻抗函数扰动。

(10.80)

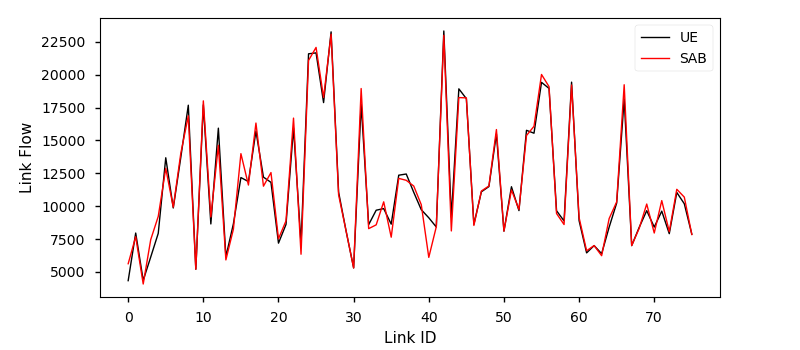


图11.13 SAB预测路段流量与UE均衡路段流量对比（阻抗函数扰动）

根据灵敏度分析的结果和式(10.80)预测新的路段流量。预测结果与求解UE均衡问题得到路段流量进行对比如图11.13所示。路段流量预测值与真实值的MAPE为4.72%，预测结果比较准确。

* 1. 参考文献

1. Frank, M., Wolfe, P. (1956). An algorithm for quadratic programming. Naval research logistics quarterly, 3(1-2), 95-110.
2. Lu, S. (2008). Sensitivity of static traffic user equilibria with perturbations in arc cost function and travel demand. Transportation Science, 42(1), 105-123.
3. Panicucci, B., Pappalardo, M., Passacantando, M., 2007. A path-based double projection method for solving the asymmetric traffic network equilibrium problem. Optimization Letters 1 (2), 171-185.
4. Sheffi, Y. (1985). Urban transportation network. Pretince Hall, 4.
5. Wang, Z., Zhang, K., et, al.(2021) An improved parallel block coordinate descent method for the distributed computing of traffic assignment problem. Transportmetrica A-Transport Science,
6. Xie, J., & Xie, C. (2016). New insights and improvements of using paired alternative segments for traffic assignment. Transportation Research Part B, 93, 406-424.
7. Xie, J., Nie, Y., & Liu, X. (2018). A Greedy Path-Based Algorithm for Traffic Assignment. Transportation Research Record, 0361198118774236.
8. 黄海军. (1994). 城市交通网络平衡分析: 理论与实践. 人民交通出版社.

# 基于Logit的随机用户平衡交通分配问题

* 1. 概述

UE交通分配假设所有的出行者能够精确地获取整个网络的出行信息，同时，还认为所有的出行者具有相同的计算能力，并且能够准确地估算出所有路径的出行费用。在这样的假设条件下，确定型的用户均衡问题能够方便地进行处理和求解。然而，它并不能准确地刻画实际的出行者的出行行为。实际中，出行者依据感知的出行费用去进行出行选择。因为他们对路段阻抗的估计是不准确的。因此，采用随机用户均衡问题能够更好地描述出行者在网络中的出行选择行为。

* 1. 符号及变量定义

在UE交通分配问题的符号和变量基础上，SUE交通分配还需要如下额外的符号及变量：

表12.1 符号及变量

|  |  |
| --- | --- |
| 参数 |  |
|  | 离散参数 |
| 变量 |  |
|  | 路径的选择概率 |
|  | 路径的理解阻抗 |

在SUE模型中，假设出行者不能够准确的估计出行阻抗，对实际的出行阻抗有一个理解误差。因此，出行者理解的路径出行费用包含两个部分：实际的路径出行费用和出行费用的理解误差。我们有

其中，是理解出行费用的随机项。

当理解出行费用的随机项服从独立同分布的Gumbel 分布时，我们可以得到多项式的Logit模型。此时，理解的路径出行费用是路径流量和路径出行费用的函数，具体如下：

其中，离散参数反映了出行者对于理解阻抗的聚合度量。

* 1. 数学模型
     1. 路径集的生成

一般而言，路径集的生成对于静态SUE问题是必须的。很多算法，如Dial（1971）提出的STOCH方法和*k*短路方法，可以用来生产随机交通分配所需的路径集。由于STOCH方法和它的推广DYNASTOCH方法不需要列举路径集，可以有效执行基于logit的随机网络加载，而被广泛用于随机交通分配中。Huang（1995）利用STOCH方法产生静态SUE模型所需的路径集。Ran和Boyce（1996）拓展了Dial（1971）提出的STOCH方法，提出了DYNASTOCH算法用于研究基于SDUO的路径选择问题。Dial（1971）的STOCH方法对有效路径的定义如下：

**定义 12.1（**Dial有效路径或者STOCH有效路径**）.** OD 对之间的有效路径仅包含使出行者离起点越来越远、离目的地越来越近的路段。

与之相对应的，Ran 和 Boyce（1996）的DYNASTOCH方法对有效路径的定义如下：

**定义 12.2（**DYNASTOCH有效路径**）.** OD 对之间的有效路径仅包含使出行者离目的地越来越近的路段。

Dial有效路径集（定义12.1）的提取可以基于每个OD对来实现，具体步骤是：

1. 计算起点到所有节点的最短路和所有节点到目的地的最短路；
2. 依据有效路径的定义确定OD对之间可以使用的路段集合，得到一个子网络；
3. 删除子网络中不与起点或者目的地联通的节点及其相连的弧，得到一个没环的子网络；
4. 对得到的子网络，依起点做拓扑排序；
5. 依据拓扑排序，顺序生成可行路径。

DYNASTOCH有效路径集（定义12.2）的提取可以基于每个目的地来实现，具体步骤是：

1. 计算所有节点到目的地的最短路；
2. 依据有效路径的定义确定到给定目的地可以使用的路段集合，得到一个子网络；
3. 删除子网络中不与目的地联通的节点及其相连的弧，得到一个没环的子网络；
4. 对得到的子网络，依目的地做**反向拓扑排序**；
5. 依据拓扑排序，反向顺序生成可行路径。

需要注意的是，Dial有效路径集包含在DYNASTOCH有效路径集中。也可以通过删除DYNASTOCH有效路径集中不满足Dial有效路径定义的路径，来得到Dial有效路径集。

基于以上两个假设得到的有效路径取决于对各条路段费用的评估，例如自由流走行时间。在进行SDUO交通分配之前需要提前把有效路径集固定下来，否则在迭代过程中改变路径集可能引起流量的波动 （Han，2003；Lim和Heydecker，2005）。

近似的k短路也是生成路径集的常用方法。这类方法都是针对单一OD来进行的。近似k短路方法主要有如下两种：

* **惩罚法**：该方法需要反复k次计算OD对之间的最短路，每次计算完最短路以后，都增大当前最短路上所有路段的阻抗。由于惩罚法执行过程中可能会得到重复的路径，在该过程中生成的路径如果不在路径集中，则添加到路径集中去。
* **路段消除法**：该方法也需要反复k次计算OD对之间的最短路，每次计算完最短路以后，把最短路上中间的一条路段的阻抗设置为无穷大。由于惩罚法执行过程中得到的路径是不重复的，该过程中生成的所有路径都添加到路径集中去。当OD对不连通时，提前终止计算。
  + 1. Fisk模型

SUE的均衡条件可以表述如下（Daganzo和Sheffi，1977）：在一个随机用户平衡的网络中，没有出行者可以通过单方面改变自己的路径选择来降低自己的理解出行阻抗。Fisk（1980）构造了如下非线性规划问题来描述基于Logit的SUE问题：

|  |  |  |
| --- | --- | --- |
|  |  |  |

数学规划问题是一个线性约束下的非线性规划问题，其中，目标函数是人工构造出来的，没有确切的物理含义。当阻抗函数是一个严格凸函数时，数学规划问题是一个凸规划问题，且路段流量和路径流量都具有唯一最优解。

* + 1. 无约束优化模型

SUE问题可以描述为如下无约束优化问题（Sheffi和Powell，1982）：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，是满意度函数，并有

对于基于Logit的SUE问题，满意度函数为

满意度函数关于路径出行费用的偏导等于路径选择概率，即有

对于基于Logit的SUE问题，把式代入式，我们有

依据定义，我们有

于是，我们可以得到无约束优化问题的目标函数的梯度为：

对于无约束优化问题，我们可以采用如下可行下降方向：

|  |  |  |
| --- | --- | --- |
|  |  |  |

* 1. 自适应平均法

采用自适应平均法求解SUE交通分配问题的详细算法步骤如下（Liu等，2009）：

|  |  |
| --- | --- |
| **算法12.1：**自适应平均法求解SUE交通分配问题 | |
| **输入:** 路径集，参数、和 | |
| 1: | 基于所有路段的自由流走行时间，计算所有路径的出行费用，利用计算路径选择概率，利用式计算路径流量，利用式计算得到初始的路段流量，置、和迭代次数 |
| 2: | **while** **do** |
| 3: | 依据路段流量计算路径阻抗，利用式计算路径选择概率，利用式计算路径流量 |
| 4: | 利用式计算可行的下降方向 |
|  | **if then** |
|  |  |
|  | **else** |
|  |  |
|  | **end if** |
| 5: | 确定步长 |
| 6: | 更新路段流量 |
| 7: | 置 |
| 8: | **end** **while** |
| **输出：** | |

自适应平均法不需要做一维搜索，可以克服相继平均法因为步长过小收敛慢的缺陷。

* 1. 测试问题
     1. 路径集的生成

采用Sioux Falls网络为测试网络。有关网络结构、路段阻抗函数参数、OD需求等信息可以通过交通网络测试问题库（<http://www.bgu.ac.il/~bargera/tntp/>）获取。基于自由流时间，生成有四类效路径集：（1）Dial有效路径、（2）DYNASTOCH有效路径、（3）基于惩罚法的近似k短路、（4）基于路段消除法的近似k短路。其中，k取值为5，惩罚法中的惩罚系数取1.2，路段消除法优先删除靠近起点的路段。

**参考答案：**所有OD对之间总共可以得到1280条Dial有效路径，1994条DYNASTOCH有效路径，基于惩罚法的近似k短路1879条，基于路段消除法的近似k短路3802条。

* + 1. SUE交通分配问题求解

采用Sioux Falls网络为测试网络。有关网络结构、路段阻抗函数参数、OD需求等信息可以通过交通网络测试问题库（<http://www.bgu.ac.il/~bargera/tntp/>）获取。采用BPR函数作为路段阻抗函数。离差参数，自适应平均法的参数取值为=1.5和=0.01。路径集采用Dail有效路径集和路段消除法得到的路径集取交集。

图12.1给出了SRAM和MSA两种算法求解SUE交通分配问题时，算法的收敛过程。从图12.1可以看出，SRAM具有非常快的收敛速度以及非常高的收敛精度。

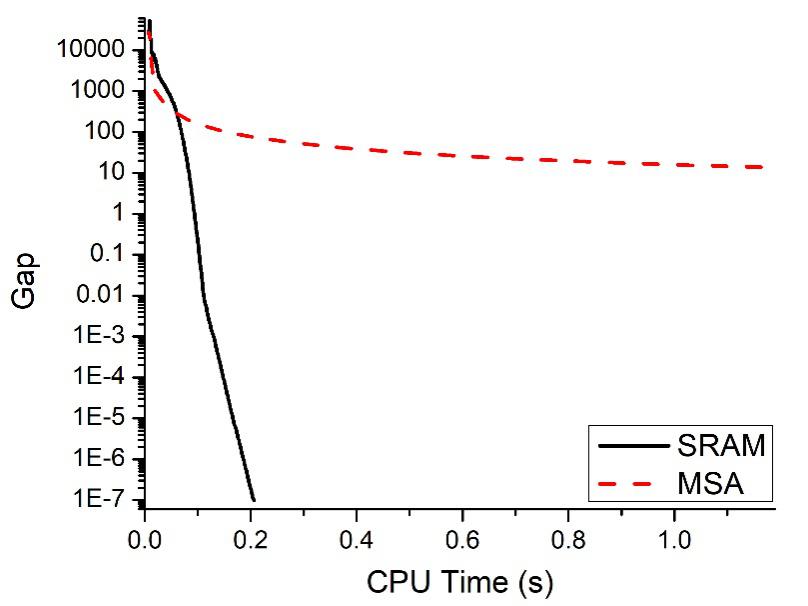


图12.1. SRMA和MSA求解SUE交通分配问题的收敛过程

* 1. 参考文献

1. Daganzo, C.F., Sheffi, Y., 1977. On stochastic models of traffic assignment. Transportation Science 11 (3), 253–274.
2. Dial, R.B., 1971. A probabilistic multipath traffic assignment model which obviates path enumeration. Transportation Research 5 (2), 83-111.
3. Fisk, C. (1980). Some developments in equilibrium traffic assignment. Transportation Research Part B 14(3), 243-255.
4. Han, S., 2003. Dynamic traffic modelling and dynamic stochastic user equilibrium assignment for general road networks. Transportation Research Part B, 37(3): 225-249
5. Huang, H.J., 1995. A combined algorithm for solving and calibrating the stochastic traffic assignment model. Journal of Operational Research Society 46 (8), 977-987.
6. Leblanc, L.J., 1975. An algorithm for the discrete network design problem. Transportation Science 9 (3), 183-199.
7. Lim, Y., Heydecker, B., 2005. Dynamic departure time and stochastic user equilibrium assignment. Transportation Research Part B, 39(2): 97-118
8. Liu, H., He, X., He, B., 2009. Method of successive weighted averages (MSWA) and self-regulated averaging schemes for solving stochastic user equilibrium problem. Networks and Spatial Economics 9 (4), 485-503.
9. Ran, B., Boyce, D.E., 1996. Modeling Dynamic Transportation Network: An Intelligent Transportation System Oriented Approach. Springer, Heidelberg.
10. Sheffi, Y., 1985. Urban Transportation Networks: Equilibrium Analysis with Mathematical Programming Methods. Prentice-Hall, Englewood Cliffs, New Jersey, USA.
11. Sheffi, Y., & Powell, W. B. (1982). An algorithm for the equilibrium assignment problem with random link times. Networks, 12(2), 191-207.

# 连续网络设计问题

* 1. 概述

一般地，城市交通网络设计问题（Network Design Problem, NDP）被表述为一个最优投资决策问题：在一定的投资约束条件下，考虑交通出行者行为选择情况的同时，改善某些路段或在交通网络中添加新的路段等，以使整个交通网络达到某种系统指标最优的目的(高自友等，2000)。通常NDP被分为两类问题：离散网络设计问题（在现有交通网络中添加新的路段）和连续网络设计问题（改进现有路段的供给能力）。还有些学者提出了混合网络设计问题，即同时考虑在网络中添加新的路段和改进现有路段，混合网络设计问题被认为更切合实际问题，不过其求解也更加复杂困难。

在城市交通系统中，政府部门对交通基础设施投入了大量资金和财政补贴，采用建设新的道路、改善已有路段、优化信号设置、更新交通工具等等措施以维护整个交通系统的正常运行，满足日益增长的交通需求；而公众则调节自己的出行行为以适应这些给定的交通设施。也就是说在政府部门为出行者提供交通基础设施之后，出行者根据具体的交通状况来决定是否出行，是乘公交车出行还是乘私家车出行，选择出租车还是骑自行车等。由于NDP决策过程中涉及到政府部门和公众的相互作用，是一个典型的双层决策问题，因此双层规划则成为描述交通网络设计问题的理想工具。

连续形式的上层决策变量表示如何改善现有路段的能力，它特别应用于信号优化设置和拓宽道路等问题中。事实上，国内外学者关于NDP的研究大部分都是集中于连续网络设计问题（Continuous NDP，CNDP），因为其决策变量是连的，可以用灵敏度分析等方法计算出平衡路段(径)流量对它的导数，相对比较容易设计算法，遗憾的是难以求解大型交通网络设计问题。

网络设计过程中，政府部门决定如何设计科学合理的规划方案，以使整个交通系统拥挤程度最小或社会经济效益最大，但是它不能控制出行者的出行选择行为，而出行者随着网络特性的改变及时调整自己的出行方式，以使自己的出行费用最小。这是一个典型的领导者—追随者对策问题，可以描述为一个双层规划问题，其数学模型一般形式如下：

其中，由下层问题求得：

这里和表示政府部门的目标函数和决策向量，表示政府部门决策向量的约束函数；和表示出行者的目标函数和决策向量，表示出行者决策向量的约束函数。

传统的NDP双层规划模型中，下层规划一般使用用户平衡配流模型。最常用的是固定需求的UE配流模型，也有学者使用基于logit分布的SUE配流模型。如果考虑到交通需求会随着网络的改善而变化，NDP也可以使用弹性需求的UE模型和SUE模型。本章以UE为例作为NDP问题的下层问题。

* 1. 连续网络设计问题的双层规划模型
     1. 符号及变量定义

在一个具有多个起点和多个讫点的交通网络中，定义和分别为网络中节点的集合和边的集合。设和分别为起点的集合和讫点的集合。令为网络需要改造的候选路段集合，为集合中的路段数量**。**在第12章的基础上，我们采用如下新增符号及变量：

表13.1 新增符号及变量定义

|  |  |
| --- | --- |
| 参数 |  |
|  | 路段的原始通行能力（CNDP）或设计通行能力（DNDP） |
|  | 路段的通行能力改造的上限 |
|  | 政府投资预算 |
|  | 路段建设或改造的费用 |
|  | 目标函数归一化系数 |
| 变量 |  |
|  | 路段改造后的通行能力 |
|  | 向量 |
|  | 路段的出行阻抗 |
|  | 路段的扩建费用函数 |

* + 1. 基于用户均衡的连续网络设计的数学模型

基于用户均衡的连续网络设计问题的具体模型表述如下:

上层问题

下层问题

|  |  |  |
| --- | --- | --- |
|  |  |  |

上层目标最小化系统总出行成本和总建设成本，是下层问题求解出的路段流量，是路段在通行能力为、流量为时的阻抗。保证候选路段改建后的通行能力不小于原始的通行能力。下层问题为标准的用户均衡模型（UE）。路段走行时间函数可以根据BPR函数设置如下：

|  |  |  |
| --- | --- | --- |
|  |  |  |

* + 1. 连续网络设计问题的求解算法概述

与DNDP相比，CNDP的研究受到了学者们更多的关注。事实上，现有的NDP研究文献中绝大部分都是研究UE平衡约束下的拓宽现有路段的决策问题。到目前为止，学者们设计了许多CNDP求解算法，不过其中大部分是启发式算法，这些启发式算法的不同之处在于线性近似上下层反应函数的方法的差异。归纳起来，这些算法可大致分为二类：迭代优化配流算法（Iterative Optimization Assignment Algorithm，IOA）、路段使用比例算法（Link Usage Proportion-Based algorithm，LUPB)和灵敏度分析法(Sensitivity Analysis-Based algorithm，SAB）。LUPB算法常用来求解某些OD需求作为上层决策变量的双层CNDP。这些问题包括匝道测量问题（Ramp Metering）、区域备用能力问题（Zone Reserve Capacity）和OD矩阵估计问题（OD Matrix Estimation)等。SAB算法利用了下层决策向量对上层决策变量的梯度信息，从理论上比IOA算法LUPB算法更精确。

* 1. 遗传算法

求解连续性网络设计问题的遗传算法的步骤与8.2节给出的遗传算法一样。下面详细介绍针对CNDP的遗传算法设计，包括解的编码、适应度计算、交叉算子、变异算子。

* + 1. 解的编码和适应度计算

我们直接采用实值向量对CNDP的可行解进行编码。图13.1示例了一个CNDP可行解的编码方案。在这个例子里面，我们令所有路段具有相同的最小通行能力和最大通行能力，分别为和。图13.1所示的解为。



图. 13.1. CNDP可行解的编码

把可行解代入到下层问题，通过求解一个UE交通分配问题得到均衡的路段流量。再把代入到上层问题的目标函数，得到目标函数值。由于CNDP的上层问题是一个最小化的优化问题，解的适应度可以定义为。

* + 1. 初始解的生成

初始化种群。随机选择一个路段进行道路扩建，在之间产生一个随机数；设置路段的通行能力，设置其它任意路段的通行能力为，得到一个新网络设计方案；把该网络设计方案添加到初始种群中去，即设置。重复该过程，直至生成*m*个初始解。

* + 1. 交叉操作算子

令和为两个被选择进行交叉的个体，如下三种交叉操作算子可以应用于求解CNDP的遗传算法：

1. 点交叉。随机选择一个交叉位置（路段），并交换网络设计方案和中路段的通行能力，得到两个新的网络设计方案和。图13.2给出了一个点交叉的例子。



图13.2. 点交叉

1. 随机交叉。对任意位置，以概率交换两个网络设计方案中路段的通行能力。所有位置都操作完成以后，得到两个新的网络设计方案和。图13.3给出了一个随机交叉的例子。



图13.3. 随机交叉

1. 区间交叉。随机选择一个区间，对区间内任意位置，交换两个网络设计方案和中路段的通行能力。区间内所有位置通行能力都交换完成以后，得到两个新的网络设计方案和。图13.4给出了一个区间交叉的例子。



图13.4. 区间交叉

以上三种交叉操作算子可以单独使用，也可以混合使用（例如每次随机选择使用其中的一个交叉操作算子）。

* + 1. 变异操作算子

令为被选择进行变异的个体，如下三种变异操作算子可以应用于求解CNDP的遗传算法：

1. 点变异。随机选择一个变异位置，并随机生成一个路段能力；变更路段的通行能力为，得到一个新的网络设计方案。图13.5给出了一个点变异的例子。



图13.5. 点变异

1. 随机变异。对任意位置，以概率确定路段的通行能力是否变化，如果变化，则在内随机设置其通行能力。所有位置都操作完成以后，得到一个新的网络设计方案。图13.6给出了一个随机变异的例子。



图13.6. 随机变异

1. 区间变异。随机选择一个区间，对区间内任意位置，随机生成一个路段能力，变更路段的通行能力为..。区间内所有位置的通行能力都重置以后，得到一个新的网络设计方案。图13.7给出了一个区间交叉的例子。



图13.7. 区间变异

以上三种变异操作算子可以单独使用，也可以混合使用（例如每次随机选择使用其中的一个变异操作算子）。

* 1. 基于响应面模型的求解算法
     1. 算法概述

大多数工程设计问题，需要模拟实验来评估采用不同设计参数时的目标函数和约束函数。 例如，为了找到最佳的机翼形状，常常针对不同的形状参数（长度，曲率，材料等）模拟机翼周围的气流。对于许多实际问题，单次模拟可能需要数分钟、数小时、甚至数天才能完成。 因此，类似设计优化、设计空间搜索、灵敏性分析和假设分析这种，需要数千、甚至数百万次模拟的任务，直接对原模型求解将是不可能的。

初始化

构造或更新响应面模型

构造候选点集合

候选点评分

挑选评分最小的候选点作为评估点，求解UE均衡问题并计算该点的网络总阻抗

更新最优解、步长等

判断停止条件

图 13.8 基于响应面模型的算法框架

针对这种目标性能与设计变量之间没有显示函数关系，且表现为多参数、高维数、强非线性以及对于目标函数的计算需要花费大量的时间的问题。如果在每次迭代都需要对目标函数进行多次计算，将会使优化过程变得非常耗时。基于此学者提出基于响应面模型的优化算法，这类算法的主要思想是利用显式的响应面模型（Response Surface Model）近似目标函数与设计变量之间隐式的复杂的函数关系，通过响应面模型函数值计算来近似原问题复杂的模拟和计算过程，从而大大降低优化过程中消耗在原问题模拟和计算过程中的时间。图13.8给出了基于响应面模型的算法框架。算法每次迭代在步长范围内随机选点，通过响应面模型评估这些点，以得到其中最有可能是最优解的点，然后将该点带入原问题进行模拟或者计算，更新构造响应面模型和目前的最优解，继续迭代，直到达到最大迭代次数时终止。该方法的优点是在每次迭代中只需对原问题的目标函数计算一次，因而算法的效率比GA更高。

* + 1. 响应面模型

响应面模型主要有多项式回归模型（Polynomial Regression，简称PR）、克里金（Kriging）模型、支持向量回归模型（Support Vector Regression，简称SVR）和径向基模型（Radial Basis Function，简称RBF）。RBF最初被用于处理地质数据的不规则地形轮廓，现有研究表明RBF很好的解决了高纬度和高阶非线性问题。RBF的精度介于PR和kriging之间但是RBF比kriging更容易建立。本章主要采用RBF模型。

给定一个包含*n*个解的解集，RBF模型可以写成如下形式：

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，为解集中的第*i*个解，为维度为的向量，为一个待确定的系数向量，是待确定的线性多项式函数，是欧氏距离。表示径向基函数，常用的函数形式有：线性函数、薄板样条函数（Thin Plate Spline Function）、立方函数、高斯函数等。基于最小二乘法的参数估计方法，RBF模型可以通过求解如下线性方程组来标定参数和：

其中，和

|  |  |  |
| --- | --- | --- |
|  |  |  |

则当且仅当，可以得到线性方程组的解如下([Powell, 1992](#_ENREF_1))：

* + 1. 初始评估点集的生成

采用算法13.1生成CNDP的初始评估点集。集合中包含个初始评估点（初始解），并且中初始解向量的最大线性无关组数量为。把集合中的评估点代入到下层UE交通分配，可以得到所有评估点的目标函数值向量。

|  |  |
| --- | --- |
| **算法13.1：**生成CNDP的初始评估点集 | |
| 1: | 置和。 |
| 2: | **for** **to** |
| 3: | 置 |
| 4: | 令为中的第个候选改造路段，随机生成一个路段能力 |
| 5: | 变更网络设计方案中路段的通行能力为 |
| 6: | 置 |
| 8: | **end** **for** |
| **输出：**。 | |

* + 1. 构造候选点集合

我们通过对当前最优解进行扰动来产生候选点。算法13.2的第3-9行给出了候选点的具体生成方法。通过重复执行算法13.2的第3-9行，我们可以生成候选点集合。通常候选点集合的规模取值在200~500之间。

|  |  |
| --- | --- |
| **算法13.2：**构造候选点集合 | |
| **输入:** 当前最优解，扰动概率，扰动步长，候选点集合的规模 | |
| 1： | 置 |
| 2： | **Repeat** |
| 3: | 置 |
| 4: | 任意候选路段以概率入选扰动路段集。 |
| 5: | 如果，则在中随机选择一个路段加入扰动路段集。 |
| 6: | **for each** **do** |
|  | 置为最优解中路段的通行能力 |
| 7: | 随机生成一个路段通行能力 |
| 8: | 变更网络设计方案中路段的通行能力为 |
| 9: | **end** **for** |
| 10： | 置 |
| 11： | **until** 中的候选点数量达到 |
| **输出：**候选点集合 | |

* + 1. 候选点评分

我们可以采用如下两个准则对解点集合中的候选点进行评分：

（1）预测值评分准则。依据RBF模型得到的所有候选点的预测值，可以给各候选点进行评分如下：

其中，和。

（2）距离评分准则。依据RBF模型得到的所有候选点的预测值，可以给各候选点进行评分如下：

其中，是候选点与当前样本点集中的点的最小距离，我们有，和。

结合预测值评分和距离评分，我们可以对候选点进行综合评分：

其中，为权重系数。在实际应用中，每次迭代可以取范围内的随机数，其中和分别为第*n*次迭代的最小权重和最大权重。通常取，可以取1。

* + 1. 基于响应面模型的CNDP求解算法

算法13.3给出了基于响应面模型的CNDP求解算法。算法第9行，对搜索步长做了调整。如果算法能够持续改进最优解，则设定小一点的搜索步长，增强局部搜索能力；否则，设定大一些的搜索步长，增强全局搜索能力。

|  |  |
| --- | --- |
| **算法13.3：**基于响应面模型的CNDP求解算法 | |
| **输入:** 最大迭代次数、最大连续成功次数、最大连续失败次数、初始步长。 | |
| 1: | 采用算法13.1生成CNDP的初始评估点集，并置迭代次数。 |
| 2: | 得到所有评估点的目标函数值向量和当前最优解 |
| 3: | **Repeat** |
| 4: | 基于评估点集，利用式更新代理模型的参数。 |
| 5: | 基于步长和当前最优解，采用算法13.2构造候选点集合。 |
| 6: | 采用式对集合中的候选点进行评分。 |
| 7: | 置为评分最好的候选点，并计算。 |
| 8: | 如果，则置、、连续成功计数器和连续失败计数器；否则，令和。 |
| 9 | 更新评估点集合。如果，则置和。如果，则置和。置𝑛=𝑛+1。 |
| 10: | **until** |
| **输出：** | |

基于响应面模型的求解算法更多的是提供了一个框架。代理模型的选取、候选点集合的构造、候选点的评分等都可以依据问题的特性进行选取或者设计。读者可以参考Regis（2011, 2014）、Regis和Shoemaker(2007, 2013)等，以加深对于基于响应面模型的CNDP求解算法的了解。

* 1. 测试问题

采用图13.9中的测试网络对本章中给出的GA和RBF有效性进行检验。该网络有6个节点、16条路段、2个OD对。如表13.2所示，我们设置了I和II两组OD需求。道路扩建成本函数设定为，其中为路段的单位能力扩建成本。表13.3给出了测试网络的路段相关参数，包括自由流走行时间、路段阻抗函数参数、最小通行能力、单位道路扩建成本。我们把所有路段的最大通行能力设置为+20，目标函数归一化系数。表13.4给出了不同文献的最优网络设计方案。

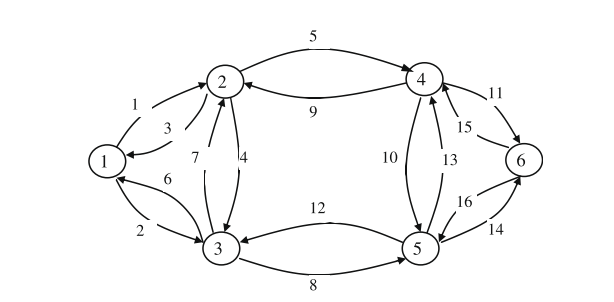


图 13.9测试网络

表13.2测试网络的网络需求

|  |  |  |
| --- | --- | --- |
| 情况 | OD需求1->6 | OD需求6->1 |
| I | 5 | 10 |
| II | 10 | 20 |

有关连续网络设计求解算法的测试内容包括：

1. 实现采用GA和RBF求解连续网络，并给出求解结果，包括：系统总阻抗，路段增加的通行能力。
2. 对比分析两种算法的求解效率：迭代次数和UE交通分配问题求解次数对目标函数值的影响。

表13.3测试网络的路段相关参数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 路段 |  |  |  |  |
| 1 | 1.0 | 10.0000 | 3.0 | 2.0 |
| 2 | 2.0 | 2.5000 | 10.0 | 3.0 |
| 3 | 3.0 | 1.0000 | 9.0 | 5.0 |
| 4 | 4.0 | 5.0000 | 4.0 | 4.0 |
| 5 | 5.0 | 10.0000 | 3.0 | 9.0 |
| 6 | 2.0 | 10.0000 | 2.0 | 1.0 |
| 7 | 1.0 | 10.0000 | 1.0 | 4.0 |
| 8 | 1.0 | 1.0000 | 10.0 | 3.0 |
| 9 | 2.0 | 4.0000 | 45.0 | 2.0 |
| 10 | 3.0 | 1.0000 | 3.0 | 5.0 |
| 11 | 9.0 | 0.2222 | 2.0 | 6.0 |
| 12 | 4.0 | 2.5000 | 6.0 | 8.0 |
| 13 | 4.0 | 6.2500 | 44.0 | 5.0 |
| 14 | 2.0 | 16.5000 | 20.0 | 3.0 |
| 15 | 5.0 | 1.0000 | 1.0 | 6.0 |
| 16 | 6.0 | 0.1667 | 4.5 | 1.0 |

GA和RBF参考的参数设定如下：

1. **GA**：种群规模取35；以等概率随机选择点交叉、随机交叉和区间交叉等算子的方法进行交叉操作，交叉概率取0.8；以等概率随机选择点变异、随机变异或区间变异等算子的方法进行变异操作，变异概率取0.2，随机交叉和随机变异的概率取0.625。
2. **RBF**：最大迭代次数、最大连续成功次数、最大连续失败次数、初始步长、扰动概率，候选点集合的规模。

由表13.5可以看出，当网络需求较小时，GA和RBF在解的精度上相差不大；但当网络需求变大以后，RBF在解的精度上明显优于GA。同时，由图13.10和图13.11可以看出，在求解连续网络设计问题时，RBF在收敛速度上显著优于GA。要得到相同的目标函数值，GA求解UE交通分配的次数远大于RBF。在小网络中，由于下层交通分配问题求解速度快，GA可能会具有优势；但随着网络规模的增大，RBF的优势越来越明显。

表13.4 文献给出的最优网络设计方案

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 情况I | | |  | 情况II | | |
| Wang and Lo(2010) | Friesz et al. (1992) | Meng et al. (2001) | | Wang and Lo(2010) | Friesz et al. (1992) | Meng et al. (2001) |
| Δy1 |  |  |  |  |  |  |  |
| Δy2 |  |  |  |  | 4.41 |  | 4.6153 |
| Δy3 |  |  | 0.0062 |  | 10 | 10.174 | 9.8804 |
| Δy4 |  |  |  |  |  |  |  |
| Δy5 |  |  |  |  |  |  |  |
| Δy6 | 4.41 | 3.1639 | 5.2631 |  | 7.42 | 5.7769 | 7.5995 |
| Δy7 |  |  | 0.0032 |  |  |  | 0.0016 |
| Δy8 |  |  |  |  | 0.54 |  | 0.6001 |
| Δy9 |  |  |  |  |  |  | 0.001 |
| Δy10 |  |  |  |  |  |  |  |
| Δy11 |  |  |  |  |  |  |  |
| Δy12 |  |  | 0.0064 |  |  |  | 0.113 |
| Δy13 |  |  |  |  |  |  |  |
| Δy14 |  |  |  |  | 1.18 |  | 1.3184 |
| Δy15 |  |  | 0.7171 |  |  |  | 2.7265 |
| Δy16 | 7.70 | 6.7240 | 6.7561 |  | 19.5 | 17.2786 | 17.5774 |
| 目标函数值 | 199.781 | 198.104 | 202.991 |  | 523.627 | 528.497 | 532.71 |
| 目标函数值\* | 199.794 | 201.337 | 202.997 |  | 522.795 | 533.331 | 532.691 |

注：“目标函数值\*”为UE交通分配问题取精度1.0E-12时评估的目标函数值。

表13.5 GA和RBF求解得到的最优网络设计方案

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 算法 | 遗传算法（GA） | | 响应面模型（RBF） | |
| 情况 | 情形Ⅰ | 情形Ⅱ | 情形Ⅰ | 情形Ⅱ |
| Δy1 |  |  |  |  |
| Δy2 |  | 4.874491 |  | 4.587526 |
| Δy3 |  | 9.950708 |  | 9.949012 |
| Δy4 |  |  |  |  |
| Δy5 |  |  |  |  |
| Δy6 | 5.231919 | 7.26395 | 5.1938 | 7.369820 |
| Δy7 |  |  |  |  |
| Δy8 |  | 0.199732 |  | 0.607871 |
| Δy9 |  |  |  |  |
| Δy10 |  |  |  |  |
| Δy11 |  |  |  |  |
| Δy12 |  |  |  |  |
| Δy13 |  |  |  |  |
| Δy14 |  | 1.283808 |  | 1.310392 |
| Δy15 |  |  |  |  |
| Δy16 | 7.890794 | 19.95188 | 7.5975 | 19.99772 |
| 目标函数值 | 199.6430 | 522.808062 | 199.6259 | 522.6471 |



图13.10 算法迭代次数对目标函数值的影响：(a)情况I和(b)情况II



图13.11 UE交通分配问题的求解次数对目标函数值的影响：(a)情况I和(b)情况II

* 1. 参考文献

1. 高自友，宋一凡，四兵峰.，2000. 城市交通连续平衡网络设计: 理 论 与方 法. 北京:中国铁道出版社.
2. Friesz, T.L., Cho, H.-j., Mehta, N.J., Tobin, R.L., Anandalingam, G., 1992. A Simulated Annealing Approach to the Network Design Problem with Variational Inequality Constraints. Transportation Science 26 (1), 18.
3. Meng, Q., Yang, H., Bell, M.G.H., 2001. An equivalent continuously differentiable model and a locally convergent algorithm for the continuous network design problem. Transportation Research Part B 35 (1), 83-105.
4. Regis, R.G., 2011. Stochastic radial basis function algorithms for large-scale optimization involving expensive black-box objective and constraint functions. Computers & Operations Research 38 (5), 837-853.
5. Regis, R.G., 2014. Constrained optimization by radial basis function interpolation for high-dimensional expensive black-box problems with infeasible initial points. Engineering Optimization 46 (2), 218-243.
6. Regis, R.G., Shoemaker, C.A., 2007. A stochastic radial basis function method for the global optimization of expensive functions. INFORMS Journal on Computing 19 (4), 497-509.
7. Regis, R.G., Shoemaker, C.A., 2013. Combining radial basis function surrogates and dynamic coordinate search in high-dimensional expensive black-box optimization. Engineering Optimization 45 (5), 529-555.
8. Wang, D.Z.W., Lo, H.K., 2010. Global optimum of the linearized network design problem with equilibrium flows. Transportation Research Part B 44 (4), 482-492.

# 离散网络设计问题

* 1. 离散网络设计问题

离散形式的上层决策变量用来表示如何选择要在网络中添加新的路段，它主要应用于新道路系统的设计问题中。LaBLance (1975)，Boyce和Janson(1980)，Chen和Alfa(1991)等先后研究了离散网络设计问题（Discrete NDP，DNDP），上层问题最小化系统总阻抗，下层问题为交通分配问题。

在一个具有多个起点和多个讫点的交通网络中，定义和分别为网络中节点的集合和边的集合。设和分别为起点的集合和讫点的集合。为了提升交通网络性能，需要在路网中某些路段新增一条车道。令表示候选的新增车道的路段集合。如果没有特别申明，本章使用符合和变量与第13章相同。

基于用户均衡的离散网络设计问题的数学模型具体表述如下：

上层问题

|  |  |  |
| --- | --- | --- |
|  |  |  |

下层问题

|  |  |  |
| --- | --- | --- |
|  |  |  |

其中，是上层问题的0-1决策变量，它表示候选路段是否添加一条车道,为目标函数量纲归一化系数。令向量。上层目标最小化系统总出行成本和总建设成本，是下层问题求解出的路段流量，是路段下新建决策为、流量为时的阻抗。下层问题为标准的用户均衡模型（UE）。在离散网络设计问题中，路段走行时间函数可以根据BPR函数设置如下：

其中，是路段*a*单条车道的通行能力。

离散网络设计问题一般被表述成带网络平衡约束的非线性整数规划模型，它被众多研究学者认为是极其难求解的问题之一。求解这类模型的常用方法有Bender分解法、分枝定界法和其他启发式算法。在早期的综述文献中，Magnanti和Wong(1984)概述了离散网络设计问题的一般建模方法，总结了以往的求解算法，并认为针对一些特殊的离散网络设计问题Lagrange松驰法或对偶上升法能非常有效地定界。一般来说，在离散优化文献中分枝定界法是研究得相当成熟的算法，用分枝定界法求解离散网络设计问题关键是如何处理此问题内在的非线性复杂性及上下层的合理定界等。Chen和Alfa(1991)用基于logit的随机增量配流方法研究了离散网络设计问题，并给出了求解小型和大型网络设计问题的计算经验。Gao等(2005)利用广义Benders分解法中的支撑函数思想来近似表示双层规划上下层反应函数，设计了一个全局收敛算法。

* 1. 遗传算法

求解离散网络设计问题的遗传算法的步骤与8.2节给出的遗传算法一样。下面详细介绍针对DNDP的遗传算法设计，包括解的编码、适应度计算、交叉算子、变异算子。

* + 1. 解的编码和适应度计算

我们直接采用0-1向量对DNDP的可行解进行编码。图14.1示例了一个DNDP可行解的编码方案。图14.1所示的解为。



图. 14.1. DNDP可行解的编码

把可行解代入到下层问题，通过求解一个UE交通分配问题得到均衡的路段流量。再把代入到上层问题的目标函数，得到目标函数值。由于DNDP的上层问题是一个最小化的优化问题，解的适应度可以定义为。

* + 1. 初始解的生成

初始化种群。对任意路段，以概率设置该路段为新建路段，得到一个新网络设计方案；把该网络设计方案添加到初始种群中去，即设置。重复该过程，直至生成*m*个初始解。

* + 1. 交叉操作算子

令和为两个被选择进行交叉的个体，和为交叉操作以后得到的两个新个体。如下三种交叉操作算子可以应用于求解DNDP的遗传算法：

1. **点交叉。**随机选择一个交叉位置（路段），并交换网络设计方案和中路段的新建决策，得到两个新的网络设计方案和。图14.2给出了一个点交叉的例子。



图14.2. 点交叉

1. **随机交叉。**对任意位置，以概率交换两个网络设计方案中路段的新建决策。所有位置都操作完成以后，得到两个新的网络设计方案和。图14.3给出了一个随机交叉的例子。



图14.3. 随机交叉

1. **区间交叉。**随机选择一个区间，对区间内任意位置，交换两个网络设计方案和中路段的新建决策。区间内所有位置是否新建的决策都交换完成以后，得到两个新的网络设计方案和。图14.4给出了一个区间交叉的例子。



图14.4. 区间交叉

以上三种交叉操作算子可以单独使用，也可以混合使用（例如每次随机选择使用其中的一个交叉操作算子）。

* + 1. 变异操作算子

令为被选择进行变异的个体，为变异以后得到的新个体。如下三种变异操作算子可以应用于求解DNDP的遗传算法：

1. 点变异。随机选择一个变异位置，把路段的新建决策设置为，得到一个新的网络设计方案。图14.5给出了一个点变异的例子。



图14.5. 点变异

1. 随机变异。对任意位置，以概率确定路段是否调整新建决策，如果变化，则把路段的新建决策设置为。所有位置都操作完成以后，得到一个新的网络设计方案。图14.6给出了一个随机变异的例子。



图14.6. 随机变异

1. 区间变异。随机选择一个区间，对区间内任意位置，把路段的新建决策设置为。可以得到一个新的网络设计方案。图14.7给出了一个区间变异的例子。



图14.7. 区间变异

* 1. 基于响应面模型的求解算法

基于响应面模型的DNDP求解算法的框架与基于响应面模型的CNDP求解算法相同。这里只介绍算法主要构成部分的设计。

* + 1. 初始评估点集的生成

采用算法14.1可以生成DNDP的初始评估点集。该集合中包含个初始评估点，并且这个解需要保证线性无关。把集合中的评估点代入到下层UE交通分配，可以得到所有评估点的目标函数值向量。

|  |  |
| --- | --- |
| **算法14.1：**生成DNDP的初始评估点集 | |
| 1: | 置和。 |
| 2: | **for** **to** |
| 3: | 置 |
| 4: | 令为中的第个候选新建路段 |
| 5: | 把网络设计方案中的路段设置为新建路段，即置 |
| 6: | 置 |
| 8: | **end** **for** |
| **输出：**。 | |

* + 1. 构造候选点集合

我们通过对当前最优解进行扰动来产生候选点。算法14.2的第3-9行给出了候选点的具体生成方法。通过重复执行算法14.2的第3-9行，我们可以生成候选点集合。通常候选点集合的规模取值在200~500之间。

|  |  |
| --- | --- |
| **算法14.2：**构造候选点集合 | |
| **输入:** 当前最优解，扰动概率，候选点集合的规模 | |
| 1： | 置 |
| 2： | **Repeat** |
| 3: | 置 |
| 4: | 任意候选路段以概率入选扰动路段集。 |
| 5: | 如果，则在中随机选择一个路段加入扰动路段集。 |
| 6: | **for each** **do** |
| 7: | 把网络设计方案中路段的新建决策设置为 |
| 8: | **end** **for** |
| 9： | 置 |
| 10： | **until** 中的候选点数量达到 |
| **输出：**候选点集合 | |

* + 1. 基于响应面模型的DNDP求解算法

我们只需要把13.4.2节模型中的向量替换为向量，就可以用于构造DNDP的响应面模型。算法14.3给出了基于响应面模型的DNDP求解算法。算法第9行，对扰动概率做了调整，其中和分别为最小扰动概率和最大扰动概率。如果算法能够持续改进最优解，则设定小一点的扰动，增强局部搜索能力；否则，设定大一些的扰动，增强全局搜索能力。

|  |  |
| --- | --- |
| **算法14.3：**基于响应面模型的DNDP求解算法 | |
| **输入:** 最大迭代次数、最大连续成功次数、最大连续失败次数、初始扰动概率。 | |
| 1: | 采用算法14.1生成DNDP的初始评估点集，并置迭代次数。 |
| 2: | 得到所有评估点的目标函数值向量和当前最优解 |
| 3: | **Repeat** |
| 4: | 基于评估点集，利用式更新代理模型的参数。 |
| 5: | 基于扰动概率和当前最优解，采用算法14.2构造候选点集合。 |
| 6: | 采用13.4.5节中的评分方法对集合中的候选点进行评分。 |
| 7: | 置为评分最优的候选点，并计算。 |
| 8: | 如果，则置、、连续成功计数器和连续失败计数器；否则，令和。 |
| 9 | 更新评估点集合。如果，则置和。如果，则置和。置𝑛=𝑛+1。 |
| 10: | **until** |
| **输出：** | |

* 1. 测试问题

采用Sioux Falls网络（见附录B）来测试GA和RBF的求解效果。在进行网络设计之前，网络上的总出行成本为7463473.7262。令该网络中所有的路段都为备选改造路段，新增一条车道的成本等于该路段的通行能力除以车道数，即。采用双投影算法（参考7.8节）（或Greedy算法）求解UE问题，求解精度取1.0E-4（或1.0E-12）。目标函数中的参数*φ*分别取值0.01、0.10和1.00进行测试。求解DNDP的算法参数设置如下：

1. 基于响应面模型的求解算法参数设置：，，，，，，。
2. 基于遗传算法求解参数设置：最大迭代次数取150，交叉概率取0.8，变异概率取0.2，种群个数设置为35，随机交叉和随机变异的概率取0.625。等概率随机采用轮盘赌选择、区间交叉和点变异。

有关离散网络设计求解算法的测试内容如下：

1. 采用遗传算法和响应面模型在不同归一化参数下求解离散网络，并给出求解结果，包括：系统总阻抗，增加车道的路段集合。
2. 对比分析两种算法的求解效率。
3. 对比分析不同归一化因子对目标函数值的影响。

由图14.8可以看出，GA的收敛速度略高于RBF。在给定的迭代次数下，随着归一化参数值的上升，上层目标函数值越来越大，表明扩建费用在求解网络设计模型时权重越大，总的系统成本则越大。由图14.9可以看出，在求解离散网络问题时，但要得到相同的目标函数值，GA求解UE交通分配的次数远大于RBF。在小网络中，由于下层交通分配问题求解速度快，GA可能会具有优势；但随着网络规模的增大，RBF的优势越来越明显。表14.1给出了不同的归一化参数下两个算法得到的最优解的目标函数值。可以看出，在给定的迭代次数下，GA得到的最优解略好于RBF。



图 14.8上层目标函数值随迭代次数的变化



图14.9上层目标函数值随UE交通分配问题求解次数的变化

表14.1 GA和RBF求解DNDP问题得到的目标函数值

|  |  |  |  |
| --- | --- | --- | --- |
| 算法 | GA | | |
| *φ* | 1.00 | 0.10 | 0.01 |
| 目标函数值 | 4151419.3343 | 3885002.6197 | 3852792.5578 |
| 算法 | RBF（150次迭代） | | |
| *φ* | 1.00 | 0.10 | 0.01 |
| 目标函数值 | 4179414.7823 | 3886996.2767 | 3853906.1605 |

* 1. 参考文献

1. Boyce, D.E., Janson, B.N., 1980. A discrete transportation network design problem with combined trip distribution and assignment. Transportation Research Part B 14 (1–2), 147-154.
2. Chen, M., Alfa, A.S., 1991. A network design algorithm using a stochastic incremental traffic assignment approach. Transportation Science 25 (3), 215-224.
3. Gao, Z., Wu, J., Sun, H., 2005. Solution algorithm for the bi-level discrete network design problem. Transportation Research Part B 39 (6), 479-495.
4. LeBlanc, L.J., 1975. An algorithm for the discrete network design problem. Transportation Science 9 (3), 183-199.
5. Magnanti, T.L., Wong, R.T., 1984. Network design and transportation planning: Models and algorithms. Transportation Science 18 (1), 1-55.

# 附 录

# 附录A：Sioux Falls 网络数据

A.1 Sioux Falls 网络

如图A.1所示，Sioux Falls网络由24个节点，76条路段，528个OD对构成（具体数值详见附录B）。路段自由流走行时间、路段通行能力、OD需求等数据也可以在交通网络测试问题库中下载，网址为<http://www.bgu.ac.il/~bargera/tntp/>。



图A.1. Sioux Falls网络

A.2 UE交通分配算法的数据结构

A.2.1 Frank-wolf算法

CNode //节点

|  |  |  |
| --- | --- | --- |
| 数据类型 | 变量名称 | 变量说明 |
| int | ID | 节点的编号，从零开始编号 |
| String | Name | 节点名称 |
| Point | Position | 节点坐标 |
| int | Origin\_ID | 节点对应的起点编号，-1表示不是起点 |
| List<int> | IncomingLink | 进入节点的路段编号集合 |
| List<int> | OutgoingLink | 离开节点的路段编号集合 |

CLink //起点

|  |  |  |
| --- | --- | --- |
| 数据类型 | 变量名称 | 变量说明 |
| int | ID | 路段的编号，从零开始编号 |
| CNode | pInNode | 路段的起节点 |
| CNode | pOutNode | 路段的终节点 |
| double | FreeFlowTravelTime | 自由流走行时间 |
| double | Capacity | 路段通行能力 |
| double | Alpha | BPR函数参数，一般取0.15 |
| double | Power | BPR函数参数，一般取4.0 |

COrigin //起点

|  |  |  |
| --- | --- | --- |
| 数据类型 | 变量名称 | 变量说明 |
| int | ID | 起点的编号，从零开始编号 |
| CNode | pOriginNode | 起点对应的节点 |
| List<int> | DestinationNode | OD对，只记录有需求的OD |
| List<double> | ODDemand | OD需求，只记录有需求的OD |

CNetwork //起点

|  |  |  |  |
| --- | --- | --- | --- |
| 数据类型 | 变量名称 | 变量说明 | |
| List<CNode> | m\_Node | 网络节点集合 | |
| List<CLink> | m\_Link | 网络路段集合 | |
| List<COrigin> | m\_Origin | 网络的起点集合 | |
| double[] | LinkFlow | 路段流量 | |
| double[] | LinkTravelTime | 路段走行时间 | |
| double | MaxUEGap | UE的最大误差，可以去1.0e-4 | |
| double | UEGap | UE的误差 | |
| double | CPUTime | CPU Time | |
| **public函数** |  | |  |
| void | ReadNode(string DataPath) | | 读取节点数据 |
| int | SearchNode(String nodeName) | | 依据节点名称查找节点的编号 |
| void | ReadLink(string DataPath) | | 读取路段数据 |
| void | ReadOrigin(string DataPath) | | 读取起点以及OD对数据 |
| void | NetworkInitialization() | | 网络初始化，各种关系的处理 |
| void | UpdateLinkTravelTime() | | 更新路段阻抗 |
| int[] | GetShortestPath(int orinode) | | 求解orinode到其它所有节点的最短路 |
| double[] | AllorNothingAssignment() | | 全有全无交通分配，可以同步计算UE误差 |
| void | UpdateLinkFlow(double[] AllNothingLinkFlow) | | 依据全有全无分配结果得到最优步长，并更新流量 |
| double | GetUEGap() | | 获取UE的误差 |
| double | UEFrankWolfe() | | UE交通分配的FrankWolfe求解，返回Gap |
| 辅助变量 |  | |  |
| double[] | ShortestPathCost | | 给定起点到其它所有节点到的最短距离 |
| int[] | ShortestPathParent | | 最短路上，节点的前继路段，-1表示没有前继路段 |
| Point | m\_CenterPosi | | 中心控制点 |
| double | m\_LatticeWidth | | 格子宽度 |
| **辅助函数** |  | |  |
| double[] | Copy(double[] x) | | 复制数组 |
| Point[] | GetArrowDirectionPoint(Point originPoint,Point destinatePoint,double angle,double length) | | 用于计算箭头的坐标，angle是偏离角度，length为箭头长度 |
| void | Output(double[] A,String DataPath) | | 把数组输出到文件里面 |
| void | DrawNetwork(PaintEventArgs e) | | 绘制网络 |
| void | SetCenterPoint(int x, int y) | | 设置绘图中心参照点 |
| void | ChangeCenterPoint(int x, int y) | | 改变中心参照点 |
| void | LatticeWidthZoom(int delta, int PX, int PY) | | 等比缩放 |

A.2.2 双投影算法

CNode //节点

|  |  |  |
| --- | --- | --- |
| 数据类型 | 变量名称 | 变量说明 |
| int | ID | 节点的编号，从零开始编号 |
| String | Name | 节点名称 |
| Point | Position | 节点坐标 |
| int | Origin\_ID | 节点对应的起点编号，-1表示不是起点 |
| List<int> | IncomingLink | 进入节点的路段编号集合 |
| List<int> | OutgoingLink | 离开节点的路段编号集合 |

CLink //起点

|  |  |  |
| --- | --- | --- |
| 数据类型 | 变量名称 | 变量说明 |
| int | ID | 路段的编号，从零开始编号 |
| CNode | pInNode | 路段的起节点 |
| CNode | pOutNode | 路段的终节点 |
| double | FreeFlowTravelTime | 自由流走行时间 |
| double | Capacity | 路段通行能力 |
| double | Alpha | BPR函数参数，一般取0.15 |
| double | Power | BPR函数参数，一般取4.0 |

COrigin //起点

|  |  |  |
| --- | --- | --- |
| 数据类型 | 变量名称 | 变量说明 |
| int | ID | 起点的编号，从零开始编号 |
| CNode | pOriginNode | 起点对应的节点 |
| List<int> | DestinationNode | OD对，只记录有需求的OD |
| List<double> | ODDemand | OD需求，只记录有需求的OD |
| List<int[]>[] | PathSet | OD对之间的路径集合 |
| List<double>[] | PathFlow | OD对之间的路径流量 |
| List<double>[] | PathCost | OD对之间的路径费用 |
| List<double>[] | CopiedPathFlow | OD对之间的路径流量 |
| List<double>[] | CopiedPathCost | OD对之间的路径费用 |

CNetwork //起点

|  |  |  |  |
| --- | --- | --- | --- |
| 数据类型 | 变量名称 | 变量说明 | |
| List<CNode> | m\_Node | 网络节点集合 | |
| List<CLink> | m\_Link | 网络路段集合 | |
| List<COrigin> | m\_Origin | 网络的起点集合 | |
| double[] | LinkFlow | 路段流量 | |
| double[] | LinkTravelTime | 路段走行时间 | |
| double | MaxUEGap | UE的最大误差，可以去1.0e-4 | |
| double | UEGap | UE的误差 | |
| double | CPUTime | CPU Time | |
| **public函数** |  | |  |
| void | ReadNode(string DataPath) | | 读取节点数据 |
| int | SearchNode(String nodeName) | | 依据节点名称查找节点的编号 |
| void | ReadLink(string DataPath) | | 读取路段数据 |
| void | ReadOrigin(string DataPath) | | 读取起点以及OD对数据 |
| void | NetworkInitialization() | | 网络初始化，各种关系的处理 |
| void | UpdateLinkTravelTime() | | 更新路段阻抗 |
| int[] | GetShortestPath(int orinode) | | 求解orinode到其它所有节点的最短路 |
| double | GetUEGap() | | 获取UE的误差 |
| void | DoubleProjectionInitilization() | | 更新路径集 |
| void | UpdatePathSet() | | 更新路径集 |
| void | UpdateLinkFlow() | | 依据路径流量更新路段流量 |
| void | UpdatePathCost() | | 更新路径费用 |
| double | EstimateLipschitzConstant() | | 估计Lipschitz常数 |
| void | CopyPathFlowAndCost() | | 暂存路径流量和费用 |
| void | CopyPathCost() | | 暂存路径费用 |
| void | CallBackPathFlowAndCost() | | 取回路径流量和费用 |
| void | CallBackPathCost() | | 取回路径费用 |
| double[] | ProjectionOntoSimplex(double[] coef, double aimsum) | | 线性投影 |
| void | PathFlowProjection(double alpha) | | 给定步长求投影 |
| void | UEDoubleProjection() | | UE交通分配的Double Projection方法求解 |
| 辅助变量 |  | |  |
| double[] | ShortestPathCost | | 给定起点到其它所有节点到的最短距离 |
| int[] | ShortestPathParent | | 最短路上，节点的前继路段，-1表示没有前继路段 |
| Point | m\_CenterPosi | | 中心控制点 |
| double | m\_LatticeWidth | | 格子宽度 |
| **辅助函数** |  | |  |
| double[] | Copy(double[] x) | | 复制数组 |
| Point[] | GetArrowDirectionPoint(Point originPoint,Point destinatePoint,double angle,double length) | | 用于计算箭头的坐标，angle是偏离角度，length为箭头长度 |
| void | Output(double[] A,String DataPath) | | 把数组输出到文件里面 |
| void | DrawNetwork(PaintEventArgs e) | | 绘制网络 |
| void | SetCenterPoint(int x, int y) | | 设置绘图中心参照点 |
| void | ChangeCenterPoint(int x, int y) | | 改变中心参照点 |
| void | LatticeWidthZoom(int delta, int PX, int PY) | | 等比缩放 |

# 附录B：旅行商问题网络数据

## B.1 eil51网络节点数据

**节点名称 x y**

1 37 52

2 49 49

3 52 64

4 20 26

5 40 30

6 21 47

7 17 63

8 31 62

9 52 33

10 51 21

11 42 41

12 31 32

13 5 25

14 12 42

15 36 16

16 52 41

17 27 23

18 17 33

19 13 13

20 57 58

21 62 42

22 42 57

23 16 57

24 8 52

25 7 38

26 27 68

27 30 48

28 43 67

29 58 48

30 58 27

31 37 69

32 38 46

33 46 10

34 61 33

35 62 63

36 63 69

37 32 22

38 45 35

39 59 15

40 5 6

41 10 17

42 21 10

43 5 64

44 30 15

45 39 10

46 32 39

47 25 32

48 25 55

49 48 28

50 56 37

51 30 40

## B.2 kroA100网络节点数据

**节点名称 x y**

1 1380 939

2 2848 96

3 3510 1671

4 457 334

5 3888 666

6 984 965

7 2721 1482

8 1286 525

9 2716 1432

10 738 1325

11 1251 1832

12 2728 1698

13 3815 169

14 3683 1533

15 1247 1945

16 123 862

17 1234 1946

18 252 1240

19 611 673

20 2576 1676

21 928 1700

22 53 857

23 1807 1711

24 274 1420

25 2574 946

26 178 24

27 2678 1825

28 1795 962

29 3384 1498

30 3520 1079

31 1256 61

32 1424 1728

33 3913 192

34 3085 1528

35 2573 1969

36 463 1670

37 3875 598

38 298 1513

39 3479 821

40 2542 236

41 3955 1743

42 1323 280

43 3447 1830

44 2936 337

45 1621 1830

46 3373 1646

47 1393 1368

48 3874 1318

49 938 955

50 3022 474

51 2482 1183

52 3854 923

53 376 825

54 2519 135

55 2945 1622

56 953 268

57 2628 1479

58 2097 981

59 890 1846

60 2139 1806

61 2421 1007

62 2290 1810

63 1115 1052

64 2588 302

65 327 265

66 241 341

67 1917 687

68 2991 792

69 2573 599

70 19 674

71 3911 1673

72 872 1559

73 2863 558

74 929 1766

75 839 620

76 3893 102

77 2178 1619

78 3822 899

79 378 1048

80 1178 100

81 2599 901

82 3416 143

83 2961 1605

84 611 1384

85 3113 885

86 2597 1830

87 2586 1286

88 161 906

89 1429 134

90 742 1025

91 1625 1651

92 1187 706

93 1787 1009

94 22 987

95 3640 43

96 3756 882

97 776 392

98 1724 1642

99 198 1810

100 3950 1558