Learning Notes

Create by

Chen Huanneng

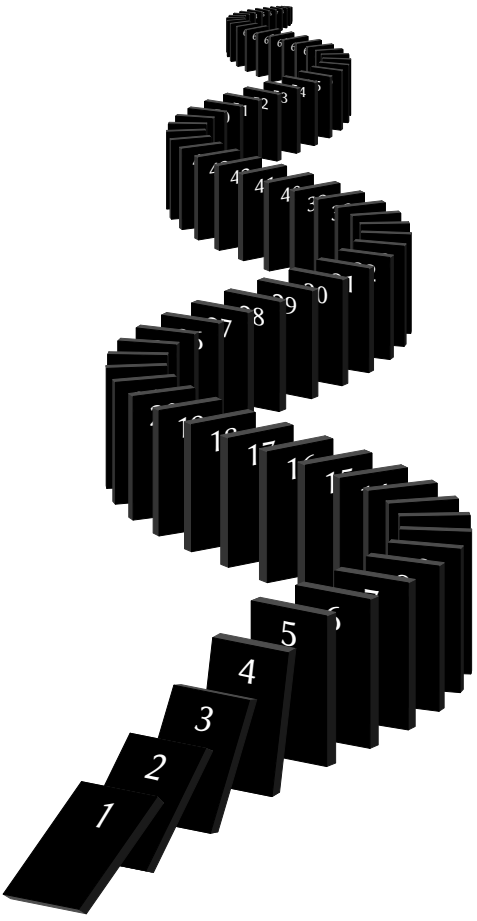
Update at
7 May 2025

Final review by
Abel

Learning Notes

Mira hacia el cielo, eres infinito
Romperás el capullo, volarás tan alto
Sigue avanzando, has llegado lejos

Mira Hacia El Cielo
G.E.M.

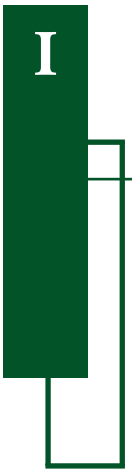


Chen Huaneng (Abel)
Xiamen University
huanengchen@foxmail.com

Contents

Preface

Part



Algorithm and Data Structure

1

Introduction to Algorithms and Data Structures

1.1 What is Algorithm

算法（Algorithm）是一个有限的、明确的、可计算的步骤序列，用于解决某类特定问题或执行特定计算任务[1]。算法通常由输入、输出和一系列操作组成。它们可以用自然语言、伪代码或编程语言来描述。算法具有以下的特性：

page 5

- 问题是明确的，包含清晰的输入和输出定义。
- 具有可行性，能够在有限步骤、时间和内存空间下完成。
- 各步骤都有确定的含义，在相同的输入和运行条件下，输出始终相同。

1.2 Analyzing Algorithms

分析算法通常指分析运行算法所需要耗费的资源，虽然有时候需要分析内存、通信带宽或者计算机硬件这类资源，但通常在算法分析中只考虑时间复杂度（Time Complexity）和空间复杂度（Space Complexity）。由于实际测试需要耗费大量的计算资源且难以统一衡量，因此通常采用理论估算的方法来评估算法的效率，这种估算方法被称为渐近复杂度分析（asymptotic complexity analysis），简称复杂度分析。复杂度分析能够体现算法运行所需的时间和空间资源与输入数据大小之间的关系，它描述了随着输入数据大小的增加，算法执行所需时间和空间的增长趋势（复杂度分析关注的不是运行时间或占用空间的具体值，而是时间或空间增长的“快慢”）。

1.2.1 Time Complexity

时间复杂度是一个定性描述算法运行时间的有关输入数据规模大小的函数，通常用基本操作数或步数来表示。基本操作是指在算法中执行的最小操作，例如加法、乘法、赋值等，通常假设每个基本操作的执行时间是相同且固定的（尽管在实际中执行加法和乘法的时间可能不同）。算法的时间复杂度可能因为相同大小的不同输入数据而不同，通常的做法是取最坏情况的时间复杂度来描述算法的性能。由于时间复杂度的函数通常难以精确计算，且对于小规模的输入数据来说，运行时间通常并不重要，因此通常关注当输入数据规模增大时算法的运行时间增长的趋势。因此，通常用渐近上界（asymptotic upper bound）[1]符号 O（Big O Notation）来表示算法的渐近时间复杂度（简称时间复杂

函数渐近上界：若存在正实数 c 和实数 n_0，使得对于所有的 $n > n_0$，均有 $T(n) \leq c \cdot f(n)$，则可以认为 $f(n)$ 给出了 $T(n)$ 的一个渐近上界，记为 $T(n) = O(f(n))$。

[1]时间复杂度分析本质上是计算“操作数量 $T(n)$”的渐近上界。

度）[2]，例如时间复杂度可以表示为 $O(n)$、$O(n^2)$、$O(\log n)$ 等。同样，除了渐近上界符号 O，还有渐近下界符号 Ω（对应最佳时间复杂度）和渐近紧确界（asymptotically tight bound）符号 Θ（对应平均时间复杂度）[3]等也可以用于描述时间复杂度[1]。

page 13-16, 25-28

在选择算法时，时间复杂度并非越低越好，一方面是需要考虑其他的因素，例如空间复杂度、实现难度、可读性等，另一方面是时间复杂度低并非代表算法的运行时间就一定低（因为时间复杂度只考虑了最高阶的项，且忽略了常数项），例如 $O(n^2)$ 的算法在 $n = 10$ 时运行时间可能比 $O(n \log n)$ 的算法还要短，但当 n 增大时，$O(n^2)$ 的算法就会变得很慢。

时间复杂度中的 $O(\log n)$ 不一定是以 2 为底的对数，这是因为 $O(\log_i n) = O(\log_i j) \cdot O(\log_j n)$，而 $\log_i j$ 是一个常数，因此可以忽略不计。实际上，$O(\log n)$ 是忽略底数的表达方式。

1.2.2 Space Complexity

空间复杂度是一个定性描述算法运行所需要的有关输入数据规模大小的存储空间的函数，这里的存储空间包括存储输入数据所需要的空间（input space）和存储算法运行所需要的空间（auxiliary space）。算法在运行过程中使用的内存空间主要包括以下几种：

- 输入空间：用于存储算法的输入数据。
- 暂存空间：用于存储算法在运行过程中的变量、对象、函数上下文等数据。
- 输出空间：用于存储算法的输出数据。

一般情况下，空间复杂度的统计范围是“暂存空间”加上“输出空间”。暂存空间可以进一步划分为三个部分：

- 暂存数据：用于保存算法运行过程中产生的各种常量、变量、对象等。
- 栈帧空间：用于保存调用函数的上下文数据。系统在每次调用函数时都会在栈顶部创建一个栈帧，函数返回后，栈帧空间会被释放。
- 指令空间：用于保存编译后的程序指令，在实际统计中通常忽略不计。

在分析一段程序的空间复杂度时，通常统计暂存数据、栈帧空间和输出数据三部分。和时间复杂度一样，空间复杂度也通常用渐近上界符号 O 来表示，例如空间复杂度可以表示为 $O(n)$、$O(n^\alpha)$、$O(n \log n)$、$O(2^n)$ 等。空间复杂度是考虑程序运行时占用内存的大小，而不是可执行文件（程序）的大小。与时间复杂度不同的是，通常空间复杂度只关注最差空间复杂度，因为内存空间是一项硬性要求，必须确保在所有输入数据下都有足够的内存空间预留。

1.3 What is Data Structures

数据结构（Data Structure）是组织和存储数据的方式，涵盖数据内容、数据之间关系和数据操作方法，它具有以下设计目标：

- 空间占用尽量少，以节省计算机内存。
- 数据操作尽可能快速，涵盖数据访问、添加、删除、修改等操作。

[2] 一般说快速排序的时间复杂度为 $O(n \log n)$ 是指快速排序在一般情况下的时间复杂度，而不是最坏情况的时间复杂度，这是业内的一个默认规定。

[3] 可能由于 O 符号过于朗朗上口，因此通常使用它来表示平均时间复杂度，但从严格意义上来讲，这种做法并不规范。因此遇到类似“平均时间复杂度 $O(n)$”的表述，直接理解为 $\Theta(n)$。

- 提供简洁的数据表示和逻辑信息，以便算法高效运行。

数据结构是计算机中组织和存储数据的方式，算法是在有限时间内解决特定问题的一组指令或操作步骤，算法与数据结构紧密相连，选择不同的数据结构会对同一个算法的性能产生很大的影响。

1.3.1 Classification of Data Structures

常见的数据结构包括数组、链表、栈、队列、哈希表、树、堆、图等，它们可以从“逻辑结构”和“物理结构”两个维度进行分类。

- 逻辑结构：描述了数据元素之间的逻辑关系，逻辑结构可以分为“线性”和“非线性”两大类。
  - 线性数据结构：数组、链表、栈、队列、哈希表，元素之间是一对一的顺序关系。
  - 非线性数据结构：树、堆、图、哈希表[4]。非线性数据结构还可以进一步划分为树形结构和网状结构，树形结构的元素之间是一对多的关系，比如树、堆、哈希表，而网状结构的元素之间是多对多的关系，比如图。
- 物理结构：反映了数据在计算机内存中的存储方式，可以分为连续空间存储（数组）和分散空间存储（链表）。物理结构从底层决定了数据的访问、更新、增删等操作方法，两种物理结构在时间效率和空间效率方面呈现出互补的特点。所有数据结构都是基于数组、链表或者二者的组合实现的[5]。

1.4 Iteration and Recursion

迭代（iteration）是一种重复执行某个任务的控制结构，在迭代中，程序会在满足一定的条件下重复执行某段代码，直到这个条件不再满足为止。迭代通常使用循环结构来实现，例如`for`、`while`等语句。

递归（recursion）是一种通过函数调用自身来解决问题的算法策略。迭代代表了一种“自下而上”地解决问题的思考范式，而递归则代表了一种“自上而下”地解决问题的思考范式（将问题分解为更小子问题的分治策略）。递归通常需要一个基准条件（base case）来终止递归调用，否则会导致无限递归。

递归函数每次调用自身时，系统都会为新开启的函数分配内存，以存储局部变量、调用地址和其他信息，这会导致两方面的结果：

- 函数的上下文数据都被存储在被称为“栈帧空间”的内存区域中，直至函数被返回后才会被释放。因此，递归通常比迭代更加耗费内存空间。（在实际中，编程语言允许的递归深度通常是有限的，过深的递归可能导致栈溢出错误。）
- 递归调用函数会产生额外的函数调用开销，例如参数传递、返回值处理等。因此递归通常比迭代的时间效率更低。

可以通过使用一个显式的栈来模拟调用栈的行为，从而将递归转化为迭代，但是这样转化可能会降低代码的可读性，并且在某些复杂问题中，模拟系统调用栈的行为可能会非常复杂，因此一般不值得做这样的转化。

[4] 由于哈希表的实现可以同时包含线性和非线性结构，因此线性结构和非线性结构中都有哈希表。

[5] 链表在初始化后，仍可以在程序运行过程中对其长度进行调整，因此也称为“动态数据结构”。数组在初始化后长度不可变，因此也成为“静态数据结构”。

1.4.1 Tail Recursion

如果函数在返回前的最后一步才进行递归调用，则该函数可以被编译器或解释器优化为迭代调用，使其在空间效率上与迭代相当，这种递归称为尾递归（tail recursion）。但是许多编译器或解释器并不支持尾递归优化，例如，`Python` 默认不支持尾递归优化，因此即使函数是尾递归形式，仍然可能会遇到栈溢出问题。

- 普通递归：当函数返回到上一层级的函数后，需要继续执行代码，因此系统需要保存上一层调用的上下文。
- 尾递归：递归调用是函数返回前的最后一个操作，这意味着函数返回到上一层级后，无需继续执行其他操作，因此系统无需保存上一层函数的上下文。

2

Character Strings

2.1 Substring Extraction and Pattern Matching

字符串中求子串和模式匹配是两个不同的概念：

- 求子串（Substring Extraction）：在给定字符串中找到的连续字符序列。子串可以是字符串的任意部分，包括单个字符、连续的一段字符或者整个字符串本身[1]。
- 模式匹配（Pattern Matching）：在一个字符串中查找特定模式（即一组字符）是否存在的过程。这个模式可以是单个字符、一组字符的序列，也可以包含通配符或正则表达式等模式匹配规则。

总的来说，子串是字符串中的一个连续片段，而模式匹配是在字符串中查找特定模式的过程。模式匹配可以涉及到更复杂的匹配规则，而不仅仅是简单的子串查找。

2.2 Pattern Matching Algorithms of String

字符串模式匹配算法主要用于在一个主文本字符串中查找一个或多个模式字符串出现的位置。

2.2.1 Naive String Search: Brute-Force

最简单也最直观的字符串匹配算法是暴力搜索（Brute-Force Search），也称为朴素字符串搜索（Naive String Search）。该算法的基本思想是从主字符串的每个位置开始，逐个字符地比较模式字符串和主字符串的子串，如果当前字符匹配则比较下一个字符，如果不匹配则回到主字符串的下一个位置继续比较，直到找到匹配或遍历完整个主字符串。时间复杂度为 $O(mn)$，其中 m 是模式字符串的长度，n 是主字符串的长度，空间复杂度为 $O(1)$。

2.2.2 Knuth-Morris-Pratt (KMP) Algorithm

首先讲解 KMP 算法的原理。KMP 算法的核心在于解决暴力求解时主串的指针回退造成的效率低下的问题，所以 KMP 算法的优势在于**用于和模式串对比的指向主串的指针不会回退**，这也是在匹配过程中，KMP 算法匹配的时间复杂度为 $O(n+m)$ 的原因，其中 n 为主串的长度，m 为模式串的长度（因为指向主串的指针不回退，所以无论最终是否找到与模式串相匹配的子串，指向主串的指针只会遍历一次主串）。

下面以一个例子来说明 KMP 算法的工作原理。现有主串为 ababcabcacbab，模式串为 abcac，此时要在主串中找到与模式串相匹配的第一个子串出现的位置（即该匹配的子串的起始位置）。

[1]空串是任何串的子串，串本身也是串的子串，所以计算串的子串数目时，要加上空串和串本身。

先由暴力解法讲起，在暴力匹配中，每次遇到不匹配的情况都需要将模式串从头开始遍历，即将主串上的指针向前移动一位后，将模式串的指针回退到模式串的起始位置开始和主串对比。

下面约定，主串为 S，主串中第 i 个字符记为 S_i，模式串为 P，模式串中第 j 个字符记为 P_j。

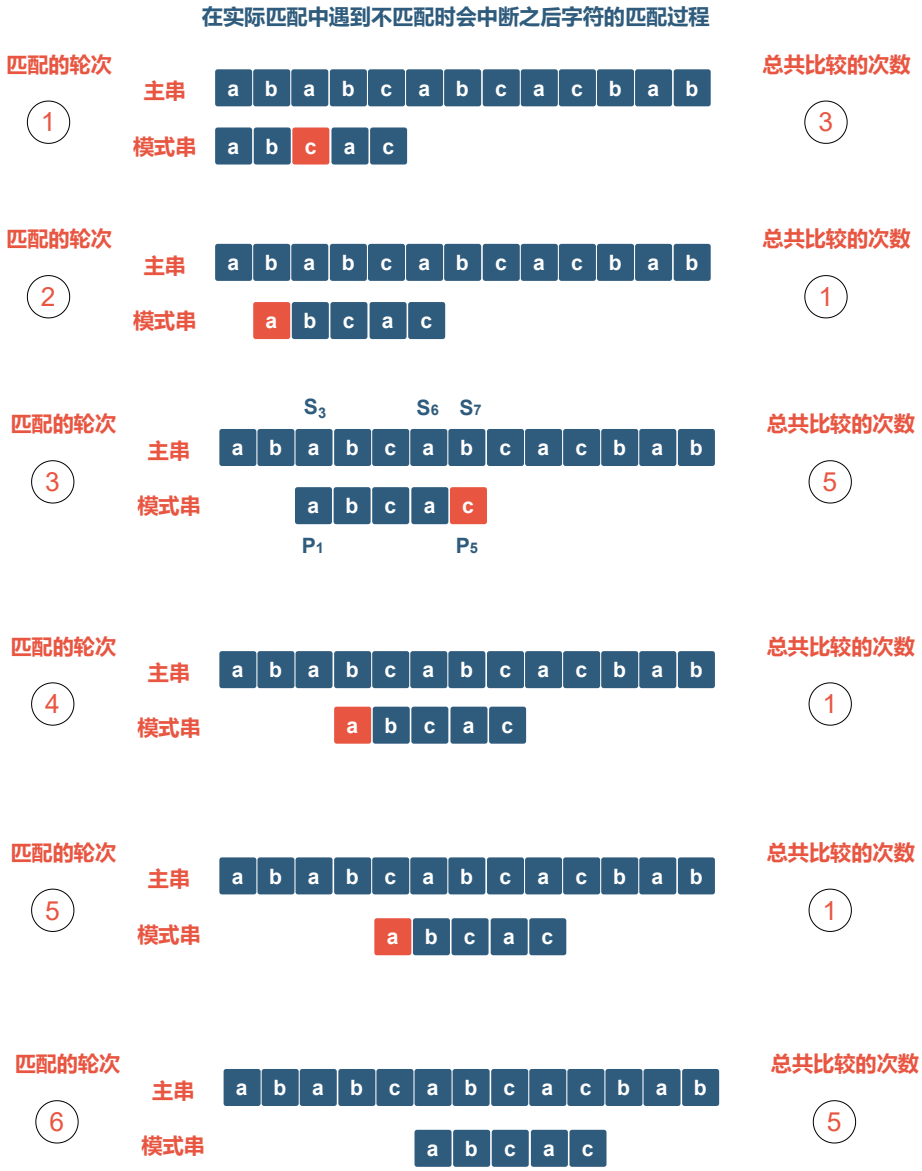


图 2-1: 暴力匹配的过程

通过观察可以知道，在第 3 轮比较的过程中，通过比较已经得知主串中的 $[S_3 \dots S_6]$ 的字符信息，因而可以知道 S_4, S_5 和 P_1 是不匹配的，因而第 4，5 轮的匹配是可以通过之前的已知信息来避免的，**而 KMP 算法的思想也正是通过之前匹配获得的主串和模式串的部分匹配信息来减少匹配次数进而提高算法效率**。接下来将分析如何运用该信息来减少匹配次数。

现在从更一般的情况来进一步分析，有主串 S 和模式串 P（模式串的长度 $\geq m+1$，主串的长度 $\geq$ 模式串的长度），现在有模式串的前 m 个字符和主串的某个子串相匹配，而第 $m+1$ 个字符不匹配，即 $P_{m+1} \neq S_{k+m}$。

现已知主串的某一子串 $[S_k, S_{k+1}, \dots, S_{k+m-1}]$ 和模式串的由 m 个字符组成的前缀 $[P_1, P_2, \dots, P_m]$ 相匹配，那么为了减少匹配的次数，我们思考能否在主串的匹配子串 $[S_k, S_{k+1}, \dots, S_{k+m-1}]$ 中找到这样一个子串（即主串的匹配子串的后缀），满足条件：

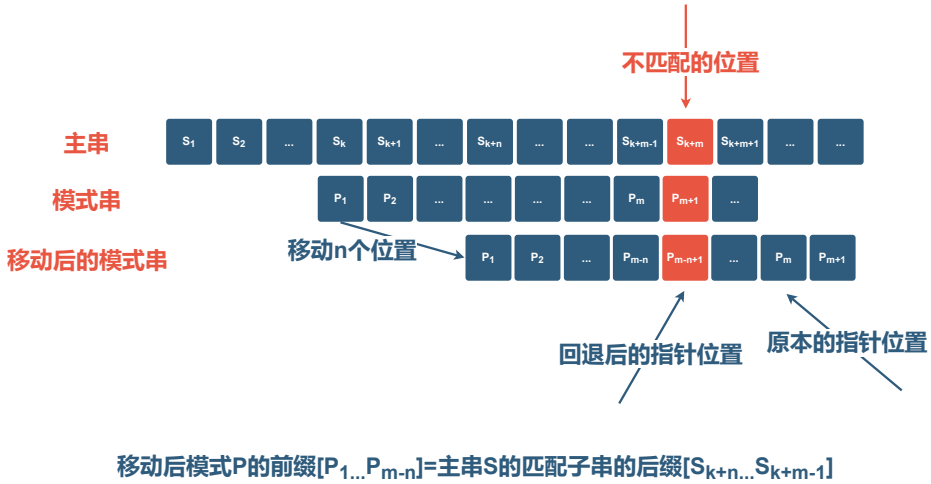


图 2-2: 更一般的匹配过程

$$[S_{k+n}, S_{k+n+1}, \ldots, S_{k+m-1}] = [P_1, P_2, \ldots, P_{m-n}] \tag{2-1}$$

并且使得 n 最小，即使得模式串的前进步数 n 最小（因为要找到主串中第一个匹配的子串，故要使得 n 最小），满足模式串的 $m-n$ 个字符的前缀和主串的匹配子串的 $m-n$ 个字符的后缀相匹配。而由于在移动 n 步之前我们有 $[S_k, S_{k+1}, \ldots, S_{k+m-1}] = [P_1, P_2, \ldots, P_m]$，所以实际上主串的匹配子串 $[S_{k+n}, S_{k+n+1}, \ldots, S_{k+m-1}]$ 可以看成是模式串 P 的一个后缀，又因要使 n 最小并且满足2-1，即使得主串的匹配子串的后缀 $[S_{k+n}, S_{k+n+1}, \ldots, S_{k+m-1}]$ 和模式串的前缀 $[P_1, P_2, \ldots, P_{m-n}]$ 的匹配长度最大，所以求使得 n 最小的问题转化为在模式串的某个长度为 m 的子串中求使得前缀和后缀匹配长度最大的问题。我们称字符串的前缀和后缀的最长相等前后缀长度为**部分匹配值（Partial Match）**，记为 PM。

仍然使用前面的例子，此时的模式串为 $P = [a, b, c, a, c]$。有如下的子串和部分匹配值[2]：

子串	长度	前缀	后缀	部分匹配值 (PM)
$[a]$	1	-	-	0
$[a, b]$	2	$\{a\}$	$\{b\}$	0
$[a, b, c]$	3	$\{a, ab\}$	$\{c, bc\}$	0
$[a, b, c, a]$	4	$\{a, ab, abc\}$	$\{a, ca, bca\}$	1
$[a, b, c, a, c]$	5	$\{a, ab, abc, abca\}$	$\{c, ac, cac, bcac\}$	0

表 2-1: 模式串的部分匹配值

由前面的论述可以知道，部分匹配值只和模式串的字符构成相关，而与主串的字符构成无关（因为求部分匹配值时，主串的匹配子串相当于模式串的一个子串，故求最大长度的前缀和后缀相等的子串时，相当于模式串自身的子串和自己求部分匹配值），所以当模式串确定时就可以确定模式串的部分匹配值，通过部分匹配值就可以得到模式串的最小移动步数 n（另外由于这样只需要知道模式串的构成时就可以求得部分匹配值而不需要知道主串的信息，故能够使得 KMP 算法的时间复杂度维持在 $O(n+m)$，其中 n 为主串的长度，m 为模式串的长度）。接下来说明 n 和部分匹配值的关系。

由前面的匹配过程，我们有 $m-n$ 为 $[P_1, P_2, \ldots, P_m]$ 模式串子串的部分匹配值，记 $[P_1, P_2, \ldots, P_m]$

[2]此处的前缀不包含最后一个字符，后缀不包含第一个字符，否则部分匹配值会和子串的长度一致，这样就失去了计算部分匹配值的意义。

的部分匹配值为 $next[m]$，$next$ 是一个数组，$next$ 名称表示和模式串移动到的下一个位置相关，m 表示模式串的最后一个匹配字符在模式串中的位置。

当模式串 $P = [P_1, P_2, \ldots, P_m, P_{m+1}, \ldots]$ 确定时，由于部分匹配字符串必定以 P_1 开始，所以当最后一个匹配字符 P_m 确定时（或者说当第一个不匹配字符 P_{m+1} 确定时），模式串的匹配子串 $[P_1, P_2, \ldots, P_m]$ 确定且唯一，所以部分匹配值也确定且唯一，即**最后一个不匹配字符在模式串中的位置唯一确定了部分匹配值**。用前面的例子说明如下：

在第 3 轮匹配时，由于 P_5 和 S_7 不匹配，所以有匹配的模式串子串 $[P_1, P_2, P_3, P_4] = [a, b, c, a]$。其部分匹配值为 1，所以移动的步数 $n = m - next[m] = 4 - next[4] = 4 - 1 = 3$，即跳过第 4，5 轮的匹配，直接从第 6 轮匹配继续开始判断。

由前面的推导，我们有一个重要结论：**模式串移动的步数 n = 部分匹配的长度 m - 和不匹配的字符位置相关的部分匹配值 $next[m]$**。

现在的最后一个问题在于已知模式串的情况下如何快速得到模式串的所有部分匹配值。如果使用暴力穷举的方法，效率明显过于低下。下面将会介绍如何快速根据模式串算出所有的部分匹配值，即 $next$ 数组。

另外，因为在实际的代码实现中，改变的不是模式串的位置（这样叙述是为了方便理解），而是改变指向对比当前字符的指针的位置。所以在实际代码实现中，修改的不是模式串，而是改变指向模式串当前字符的指针的位置。**部分匹配值 $next$ 数组也可以理解为下一次进行比较时可以跳过比较的字符数**，这是因为部分匹配值相当于主串的匹配子串的后缀和模式串的前缀相等的最大长度，所以可以直接跳过这些字符进行下一次比较。

写成伪代码则是`j = next[j]`，即指向模式串当前字符（当前指针位置 $m+1$）的指针 j 将直接跳过模式串的匹配前缀的长度 $next[j]$ 个字符继续进行比较。

接下来将介绍如何快速计算给定模式串的 $next$ 数组，计算模式串的 $next$ 数组是一个递推的过程：

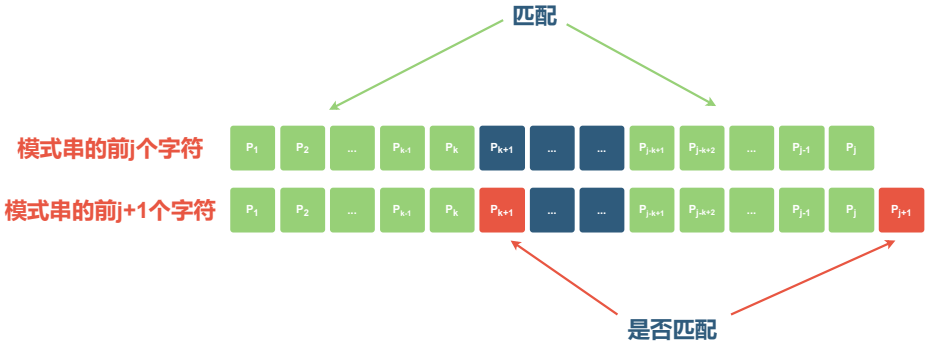


图 2-3: 计算 $next[j+1]$ 的过程

1. 当 $j = 1$ 时，即取模式串的第一个字符时，此时显然没有前后缀存在，故 $next[1] = 0$。
2. 接下来描述当已知前 j 个字符的 PM 值时，求前 $j+1$ 个字符的 PM 值的过程。当 $j > 1$ 时，设 $next[j] = k\,(k < j)$，则 k 应该满足条件 $[P_1, P_2, \ldots, P_k] = [P_{j-k+1}, \ldots, P_j]$ 且不存在 $k' > k$ 也满足该条件，如图2-3所示。那么应该如何求 $next[j+1]$，此时有两种情况：
   - 若 $P_{k+1} = P_{j+1}$，则表明在模式串中有 $[P_1, P_2, \ldots, P_{k+1}] = [P_{j-k+1}, \ldots, P_{j+1}]$ 成立，所以 $next[j+1] = k + 1 = next[j] + 1$。
   - 若 $P_{k+1} \neq P_{j+1}$，则表明在模式串中有 $[P_1, P_2, \ldots, P_{k+1}] \neq [P_{j-k+1}, \ldots, P_{j+1}]$。此时寻

找模式串的最长相等的前后缀难道要使用暴力求解吗，其实我们可以先尝试能否利用之前已知的信息来减少匹配的次数。由于 $[P_1, P_2, \ldots, P_k] = [P_{j-k+1}, \ldots, P_j]$ 且不存在 $k' > k$ 也满足该条件，所以最长相等的前后缀应该比 $k+1$ 小（可以用反证法证明），那么此时只需要找比 $k+1$ 小的最长相等的前后缀即可，即找到 h 满足 $[P_1, P_2, \ldots, P_h, P_{h+1}] = [P_{j-h+1}, \ldots, P_j, P_{j+1}]$ 且不存在 $h' > h$ 满足该条件，如图2-4所示。由于 $[P_1, P_2, \ldots, P_k] = [P_{j-k+1}, \ldots, P_j]$，因此在寻找满足条件的 h 时，可以先找到满足条件 $[P_1, P_2 \ldots, P_h] = [P_{j-h+1}, \ldots, P_j]$ 的 h，然后再判断 P_{h+1} 和 P_{j+1} 是否相等，如果相等则 $next[j+1] = h+1$，否则继续寻找满足条件的 h，直到找到满足条件的 h 或者 $h = 0$ 为止。而因为 $[P_1, P_2, \ldots, P_k] = [P_{j-k+1}, \ldots, P_j]$，所以寻找满足条件 $[P_1, P_2 \ldots, P_h] = [P_{j-h+1}, \ldots, P_j]$ 的 h 相当于在 $[P_1, P_2, \ldots, P_k]$ 中寻找最长相等的前后缀，而因为 $[P_1, P_2, \ldots, P_k]$ 的部分匹配值已经计算出来了，所以可以直接使用 $next[k]$ 来寻找满足条件的 h，即 $h = next[k]$，此时再判断 P_{h+1} 和 P_{j+1} 是否相等（即判断 $P_{next[k]+1}$ 和 P_{j+1} 是否相等），如果相等则 $next[j+1] = h+1 = next[k]+1$，否则继续寻找满足条件的 h，即 $h = next[next[k]]$，直到找到满足条件的 h 或者 $h = 0$ 为止。

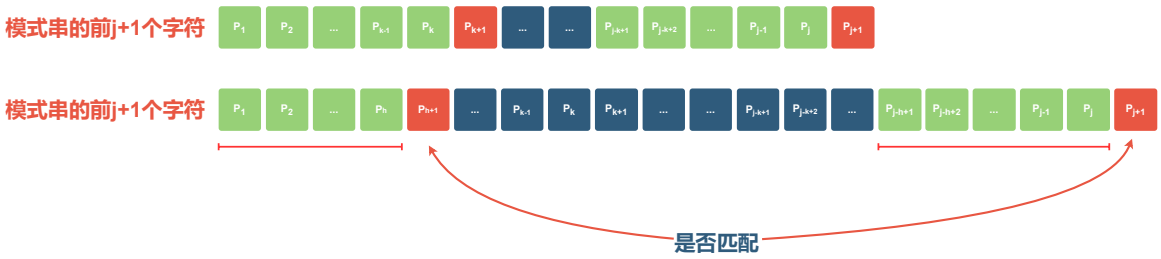


图 2-4: 当 $P_{k+1} \neq P_{j+1}$ 时，寻找满足条件的 h 的过程

由上述的递推过程中可以看出，KMP 算法的时间复杂度为 $O(m+n)$，空间复杂度为 $O(m)$，其中匹配的时间复杂度为 $O(n)$，因为主串的指针不回退，而计算模式串的部分匹配值的时间复杂度为 $O(m)$，这是因为在计算模式串的部分匹配值时，并没有使用暴力匹配，而是利用先前已经得到的部分匹配值的信息来减少匹配次数，其中 m 为模式串的长度，n 为主串的长度。虽然暴力匹配的时间复杂度为 $O(mn)$，但在实际应用中，暴力匹配的实际复杂度接近 $O(m+n)$，这是因为在大部分文本和模式串中（比如在一本小说中搜索 goodbye 单词），模式串的第一个字符就和主串不匹配，只有在某些情况下（比如在二进制文本中寻找 0101001），KMP 算法才比暴力匹配快得多（出现了很多部分匹配的情况）。

现在来考虑当模式串中出现了重复字符时，如何改进 $next$ 数组为 $nextval$ 数组提高效率。以下列例子说明：

主串	a	a	a	b	a	a	a	b
模式串	a	a	a	a	b			
j	1	2	3	4	5	6	7	8
$next[j]$	0	1	2	3	0			
$nextval[j]$	0	0	0	0	4			

表 2-2: 改进 $next$ 数组为 $nextval$ 数组的例子

当 $S_4 = b$ 和 $P_4 = a$ 进行比较时，由于不匹配，所以会进行 $next[4] = 3$ 的跳转，此时模式串的指针会跳到 $P_3 = a$，而此时 $P_3 = a$ 和 $S_4 = b$ 仍然不匹配，所以会持续跳转，直到 $P_1 = a$ 和 $S_4 = b$ 进行比较，此时仍然不匹配，才会递增主串的指针，继续进行比较。而由于 P_4 和 P_3, P_2, P_1 都是相同的字符，所以在比较时，主串的 S_4 和模式串的 P_4 不匹配时，就没有必要继续比较模式串的 P_3, P_2, P_1，因

为它们都是相同的字符，即不应该使指针跳转 $next[j]$，故修正 $next[j]$ 的值继续递归为 $next[next[j]]$ 直到两者不相同为止，此时的数组命名为 $nextval$ 数组，而与主串的匹配算法不变。

用 C++ 语言实现 KMP 算法的代码如下[3]：

Code 2.1: KMP Algorithm

```cpp
#include <iostream>
#include <vector>
#include <string>

using namespace std;

void getNext(int *next, const string &p) {
    int m = p.size(), j = 0; // j is the length of the previous longest prefix suffix
    next[0] = 0; // The first character has no proper prefix or suffix
    for (int i = 1; i < m; ++i) { // Start from the second character
        // Check if the j > 0 because we need to index to the previous next value
        while (j > 0 && p[i] != p[j]) { // Backtrack if there is a mismatch
            j = next[j - 1]; // j - 1 because next is 0-indexed
        }
        if (p[i] == p[j]) { // If characters match, increment j, namely the length of the current prefix
            ++j;
        }
        next[i] = j; // Set the next value for the current character
    }
}

int kmp(const string &s, const string &p) {
    int n = s.size(), m = p.size();
    if (n < m) { // Check if the pattern is longer than the text
        cout << "Pattern is longer than text." << endl;
        return -1;
    }
    if (m == 0) { // Check if the pattern is empty
        cout << "Empty pattern." << endl;
        return -1;
    }
    vector<int> next(m);
    getNext(&next[0], p); // Initialize the next array for the pattern p
    for (int i = 0, j = 0; i < n; ++i) { // i is the index in the main string s and j is the index in the pattern p
        while (j > 0 && s[i] != p[j]) { // Mismatch after j matches
            j = next[j - 1]; // Use the next array to skip unnecessary comparisons, minus 1 because next is 0-indexed
        }
        if (s[i] == p[j]) { // Match found and increment j to check next character
            ++j;
        }
        if (j == m) { // If j equals the length of the pattern, the first occurrence is found
            cout << "Pattern found at index: " << i + 1 - m << endl;
            return 0; // Return after finding the first occurrence
        }
    }
    return -1; // If no match is found, return -1
```

[3]相关的题目参考LeetCode 28. 找出字符串中第一个匹配项的下标。

Code 2.1 (continued)

```
}

int main() {
    string s, p;
    cin >> s >> p;
    cout << "The main string is: " << s << endl;
    cout << "The pattern string is: " << p << endl;
    int result = kmp(s, p);
    if (result == -1) {
        cout << "Pattern not found." << endl;
    } else {
        cout << "Pattern found successfully." << endl;
    }
    return 0;
}
```

Part

II

C Plus Plus

3

Input and Output

3.1 Standard Input and Output Streams

常用的输入输出的库是`iostream`，流（stream）是 C++ 中处理输入输出的基本概念，流是从输入输出设备中读取或写入数据的一连串字节，术语“流”源于字节是随着时间的推移而顺序流动产生或消耗的。

写入数据到流中称为输出（output），从流中读取数据称为输入（input）。对于输出来说，操作符`<<`称为输出操作符（output operator），用于将数据写入流中；对于输入来说，操作符`>>`称为输入操作符（input operator），用于从流中读取数据。输出操作符的左边必须是一个输出流对象（`ostream` object），右边必须是一个要输出的值或表达式，输出操作符会将右边的值写入左边的输出流中，输出操作符的返回值是左边的输出流对象，这样可以实现连续的输出操作，输入操作符则类似。比如下面的代码片段中，输出操作符`<<`将字符串`"Hello world!"`写入到标准输出流`std::cout`中，并且返回了`std::cout`，这和第二行的代码是等价的，同样和第三行和第四行的代码也是等价的。

```
    std::cout << "Hello world!" << std::endl;
    (std::cout << "Hello world!") << std::endl;
    std::cout << "Hello world!";
    std::cout << std::endl;
```

`endl`是一个叫做“流操纵符”（stream manipulator）的特殊值，它会终止当前行并刷新（flush）与当前输出设备相关的缓冲区（buffer）。刷新缓冲区的目的是确保所有输出都被写入到输出设备中，特别是在输出设备是终端或文件时。刷新缓冲区可以确保输出的内容立即可见或可用[1]。

当使用`istream`或`ostream`作为条件时，效果是测试流的状态。如果流的状态是有效的（valid），也即流没有发生错误（例如，读取或写入失败）或到达流的末尾（end-of-file, EOF），则条件为真（true）。例如下面代码中的`std::cin >> value`会从标准输入流`std::cin`中读取一个整数值并将其存储在变量`value`中，如果读取成功，则`std::cin`的状态是有效的，因此条件为真，循环继续执行；如果读取失败（例如输入不是一个整数），则`std::cin`的状态变为无效，循环结束。

```
#include <iostream>
int main() {
    int sum = 0, value = 0;
    while (std::cin >> value) {
        sum += value;
    }
    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

[1]输出经常在进行调试时使用，在这种情况下，刷新缓冲区是必要的，否则当程序出现错误时，输出仍旧停留在缓冲区中，导致调试信息出现在错误发生之后，导致调试信息指向了错误发生之后的代码。

4

Variables and Types

4.1 Types

4.1.1 Signed and Unsigned Types

C++ 中有两种类型，有符号类型（signed type）和无符号类型（unsigned type），有符号类型可以表示正数、负数和零，而无符号类型只能表示大于或等于零的数。在决定使用有符号类型还是无符号类型时，以下的建议可以帮助你做出选择：

- 如果你知道变量只会存储非负数（例如计数器、索引等），可以使用无符号类型，这样可以增加可表示的最大值。
- 使用`int`来进行整数运算，因为`short`太小，并且在实际中，`long`经常和`int`的大小相同，如果数据的值超过了`int`规定的最小字节数，则可以使用`long long`。
- 不要使用 plain `char`类型或`bool`类型来进行数值运算，而是只在存储字符或布尔值的时候使用。使用`char`类型来进行计算的时候可能会导致意外的结果，这是因为`char`类型在某些机器上是 signed 的，而在其他机器上是 unsigned 的，这取决于编译器的实现。如果需要使用小整数（tiny integer），需要显式地使用`signed char`或者`unsigned char`。
- 使用`double`来进行浮点数运算，因为`float`的精度通常不足以满足大多数应用的需求，并且相对于`float`来说，`double`的计算消耗可以忽略不计，实际上在某些机器中双精度浮点数的计算要快于单精度浮点数的计算。使用`long double`获得的更高精度通常是没有必要的，并且需要消耗大量的运行时（run-time）资源。

4.1.2 Type Conversion

类型转换（type conversion）会自动发生在使用某种类型用于需要另一种类型的上下文中。具体的转换后的结果取决于类型允许的值的范围和类型的大小：

- 当将一个非布尔值类型的算术类型赋值给布尔值类型时，非零值会转换为 true，零值会转换为 false。
- 当将一个布尔值类型赋值给非布尔值类型时，true 会转换为 1，false 会转换为 0。
- 当将一个浮点数类型赋值给整数类型时，会截断小数部分，保留整数部分。
- 当将一个整数类型赋值给浮点数类型时，会将整数转换为浮点数，保留小数部分为 0，如果整数的值超出了浮点数类型的表示范围，则会导致精度丢失。

- 当将一个超出类型表示范围的值赋值给一个无符号类型时，赋予的值会取该无符号类型能表示的最大值的模（modulo）的余数。例如，如果将-1 赋值给一个 8-bit 的`unsigned char`，则结果为 255，因为 8-bit 的无符号整数类型的表示范围是 0 到 255，即对 256 取模，-1 对 256 取模的余数是 255。
- 当将一个超出类型表示范围的值赋值给一个有符号类型时，结果是未定义的（undefined），程序可能看起来正常运行、可能崩溃或者产生垃圾值。

当表达式中涉及无符号类型和有符号类型的运算时，C++ 会将有符号类型转换为无符号类型，这可能导致意外的结果。例如，如果将一个负数与一个无符号整数相加，结果可能是一个非常大的正数，因为负数被转换为无符号整数后，其值变成了一个很大的正数。

4.1.3 Literals

字面量（literal）是指在代码中直接写出的值，C++ 支持多种类型的字面量，包括整数、浮点数、字符、字符串等。由单引号括起来的字符是字符字面量，由双引号括起来的字符串是字符串字面量，字符串字面量是由一个字符数组表示的，字符数组的最后一个元素是空字符（`'\0'`），表示字符串的结束。相邻的两个字符串字面量（只被空格、制表符或换行符分隔）会被连接成一个字符串字面量。

```
std::cout << "a really, really long string literal "
             "that spans two lines." << std::endl;
```

4.2 Variables

变量（variable）是一个存储数据的命名位置，可以在程序中使用变量来存储和操作数据。变量的类型决定了它可以存储的数据类型和大小以及可以对变量进行的操作。C++ 中的变量必须先声明后使用，声明变量时需要指定变量的类型和名称。

4.2.1 Initialization and Assignment

变量的初始化（initialization）和赋值（assignment）是两个不同的概念。初始化是在变量声明时给变量赋予一个初始值，而赋值是在变量已经声明后，给变量重新赋值。

列表初始化（list initialization）是 C++11 引入的一种初始化方式，它使用花括号（`{}`）来初始化变量。列表初始化可以用于所有类型的变量，包括内置类型和用户定义的类型。列表初始化的好处是可以防止窄化转换（narrowing conversion），即从一个较大的类型转换为一个较小的类型时可能导致数据丢失。

```
long double ld = 3.1415926536;
int a{ld}, b = {ld};
int c(ld), d = ld;
```

在上面的代码中，变量`a`和`b`使用列表初始化，而变量`c`和`d`使用普通的初始化方式。由于`ld`是一个长双精度浮点数（long double），而`int`是一个整数类型，列表初始化会防止窄化转换，因此`a`和`b`初始化会发生错误，而`c`和`d`会被初始化为 3。

变量如果没有进行初始化，变量会进行默认初始化（default initialization），被赋予的值取决于变量的位置和类型。如果内置类型（built-in）的变量在任何函数体外声明，则会被初始化为 0，如果在函数体内声明，则不会被初始化，变量的值是未定义的。因此为了避免使用未初始化的变量导致未定义行为，建议在声明内置类型的变量时总是进行初始化。

4.2.2 Declaration and Definition

在 C++ 中，变量的声明（declaration）和定义（definition）是两个不同的概念[1]。声明是告诉编译器变量的类型和名称，而定义是为变量分配内存空间并有可能初始化变量。为了获得一个声明但不定义变量，可以使用`extern`关键字，`extern`关键字用于声明一个变量在其他文件中定义，而不是在当前文件中定义。

```
extern int i; // declares but dose not define i
int j; // declares and defines j
```

任何带有显式初始化的声明都是定义，同样可以给带有关键字`extern`的变量进行初始化，但是这样相当于覆盖了关键字`extern`的含义，实际上是定义了一个变量而不是声明一个变量。

```
extern double pi = 3.1416; // definition
```

但是在函数内部如果给带有`extern`关键字的变量进行初始化，则会导致编译错误。

声明和定义的设计是为了在多个文件中共享变量，通过在一个文件中定义变量并在其他文件中声明变量，可以实现跨文件的变量共享。这样可以避免在每个文件中都定义相同的变量，从而减少代码重复和内存浪费。

4.2.3 Identifiers

标识符（identifier）是 C++ 中用于命名变量、函数、类等的名称。标识符由字母、数字和下划线组成，但不能以数字开头。标识符是区分大小写的，这意味着`myVariable`和`myvariable`是两个不同的标识符。

标识符的命名可以遵循一些约定，以提高代码的可读性和可维护性。以下是一些常见的命名约定：

- 标识符应该具有描述性，能够清楚地表达其含义。
- 变量名通常使用小写字母。
- 类名通常使用大写字母开头的驼峰命名法（CamelCase）。
- 具有多个单词的标识符可以用下划线分隔（`my_variable`）或使用驼峰命名法（`myVariable`）。

4.3 Compound Types

4.3.1 References

引用（reference）定义了一个已经存在的对象的别名（alias）。当定义一个引用时，不是复制初始化的值，而是绑定（bind）到一个已经存在的对象上。引用一旦绑定到一个对象，就不能再绑定到其他对象，因此引用必须在声明时进行初始化。

```
int ival = 1024;
int &refVal = ival; // refVal refers to (is another name for) ival
int &refVal2; // error: a reference must be initialized
refVal = 2; // assigns 2 to the object to which refVal refers, i.e., ival
int ii = refVal; // same as ii = ival
int &refVal3 = refVal; // refVal3 is bound to the object to which refVal is bound, i.e.,
↪  to ival
```

[1] 变量只能被定义一次，但是可以被声明多次。

当引用被创建之后，对引用的任何操作都等同于对原始对象的操作。由于引用不是一个独立的对象，而是原始对象的别名，它没有自己的存储空间，编译后通常是指针的语法糖[2]，因此无法定义一个指向引用的引用（reference to a reference）。

```
int& &refVal4 = refVal; // error: a reference cannot be bound to another reference
```

可以在单次定义中定义多个引用，每个引用的标识符是由前缀`&`分隔的。

```
int i = 1024, i2 = 2048; // i and i2 are both ints
int &r = i, r2 = i2; // r is a reference bound to i; r2 is an int
int i3 = 1024, &ri = i3; // i3 is an int; ri is a reference bound to i3
int &r3 = i3, &r4 = i2; // both r3 and r4 are references
```

引用的类型和被引用的原始对象的类型必须完全相同，并且不能将引用绑定到字面量（literal）或更一般的表达式结果上[3]。

```
int &refVal5 = 10; // error: initializer must be an object
double dval = 3.14;
int &refVal6 = dval; // error: initializer must be an int object
```

4.3.2 Pointers

指针（pointer）是一个对象，它存储了另一个对象的地址。指针和引用都是间接访问对象的方式。不同于其他的内置类型，定义在块作用域（block scope）中的指针变量不会被自动初始化，因此在使用指针之前必须先对其进行初始化，否则会导致未定义行为（undefined behavior）。

```
int *ip1, *ip2; //both ip1 and ip2 are pointers to int
double dp, *dp2; // dp2 is a pointer to double; dp is a double
```

通过取地址操作符（address-of operator, `&`）可以获取一个对象的地址，并将其赋值给一个指针变量。

```
int ival = 42;
int *p = &ival; // p holds the address of ival; p is a pointer to ival
```

由于引用不是对象，因此不能获取引用的地址，所以不能定义一个指向引用的指针。同样，指针的类型和被指向的对象的类型必须完全相同[4]，这是因为指针的类型决定了指针解引用（dereference）时的行为，即如何访问指针所指向的对象。

存储在指针中的地址可以是以下几种类型之一：

1. 指向一个对象的地址。
2. 指向一个对象末尾之后的地址（past the end of an object），这通常用于表示一个数组的末尾。
3. 空指针（null pointer），表示指针不指向任何对象。
4. 非法的指针，除了以上三种情况之外的指针。

虽然第二和第三种情况是合法的，但是它们不指向任何有效的对象，因此在解引用这些指针时会导致未定义行为。

空指针（null pointer）是一个特殊的指针，它不指向任何对象。有以下几种方式可以定义一个空指针：

```
int *p1 = nullptr; // equivalent to int *p1 = 0;
int *p2 = 0; // directly initializes p2 from the literal constant 0
// must #include cstdlib
```

[2] 语法糖（syntactic sugar）是指计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更加方便使用。

[3] 例外的情况参见4.4.1

[4] 例外的情况参见4.4.2

```
int *p3 = NULL; // NULL is defined in cstdlib, equivalent to 0
int zero = 0;
int *p4 = zero; // error: cannot assign an int to a pointer
```

最推荐使用的初始化指针的方法是使用`nullptr`，这是 C++11 标准引入的，`nullptr`是一个可以被转换成任何其他指针类型的特殊类型的字面量。尽管指针可以通过赋值 0 来表示空指针，但是将整数变量赋值给指针是非法的，尽管这个整数变量的值恰好是 0。最好在创建指针时指向需要被指针指向的对象，或者赋值为空指针，这样能避免一些不必要的运行时错误。

和算术表达式可以用于条件判断类似，指针也可以用于条件判断，任何非零指针表示`true`。给定两个合法的同类型指针可以进行比较，当这两个指针指向同一个地址时则判断为这两个指针相等。由于这些操作都用到了指针的值，因此用于条件判断或者比较的指针必须是合法的指针，如果指针是非法的，则条件判断或比较的结果是未定义的。

两个指针都为空指针、指向同一个对象或者都指向同一个对象之后的地址。

`void*`是一个特殊的指针类型，它可以指向任何类型的对象，但是不能直接解引用（dereference）或进行算术运算。同其他类型的指针类似，`void*`指针也存储了一个地址，但是存储在该地址的对象的类型是未知的。

```
double obj = 3.14, *pd = &obj;
// ok: void* can hold the address value of any data pointer type
void *pv = &obj; // obj can be an object of any type
pv = pd; // pv can hold a pointer to any type
```

但是对于`void*`指针只能进行有限的操作：将它和其他指针进行比较、将它传递给函数、将它作为函数的返回值以及将它赋值给另外一个`void*`指针。不能对`void*`指针进行解引用或算术运算，因为编译器不知道存储在该地址的对象的类型，而对象的类型决定了可以对该对象进行的操作。

一般来说，我们将`void*`指针来将内存作为内存处理（deal with memory as memory），而不是用于访问存储在内存中的对象。

4.3.3 Compound Type Declarations

C++ 中的变量定义可以包含一个基类型（base type）和一系列的声明符号（declarators），声明符号可以是指针、引用或数组等。在同一个定义中可以将不同的声明符号与相应的变量以不同的方式和基类型关联起来。因此，在一个定义中可以定义多个不同类型的变量。

```
// i is an int; p is a pointer to int; r is a reference to int
int i = 1024, *p = &i, &r = i;
int* p1, p2; // p1 is a pointer to int; p2 is an int
int *p3, *p4; // both p3 and p4 are pointers to int
```

由于存在`int* p;`这样的定义方式，因此容易混淆基类型和声明符号的关系，基类型是`int`，而声明符号`*`用于修饰变量`p`，表示`p`是一个指向整数的指针，而不是一个整数。因此推荐将声明符号紧贴变量名，以提高代码的可读性。

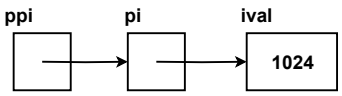


图 4-1: 指向指针的指针

声明符号的理解方式是从右到左的，即声明符号修饰的是它右边的变量名。比如指向指针的指针（pointers to pointers）是指一个指针变量存储了另一个指针的地址。如图4-1所示，`pi`是指向整数的指针，而`ppi`是指向指针的指针。

```
int ival = 1024;
int *pi = &ival; // pi points to an int
int **ppi = &pi; // ppi points to a pointer to an int
```

由于引用不是一个对象，因此不能定义指向引用的指针，但是由于指针是一个对象，因此可以定义指向指针的引用。

```
int i = 42;
int *p; // p is a pointer to int
int *&r = p; // r is a reference to the pointer p
r = &i; // r refers to a pointer; assigning &i to r makes p point to i
*r = 0; // dereferencing r yields i, the object to which p points; changes i to 0
```

同样理解指向指针的引用也是从右到左，即最接近变量名的声明符号`&`修饰变量名`r`，表示`r`是一个引用，剩余的声明符号说明了该引用指向的是一个指针类型，最后基类型说明该变量是一个指向指向整数类型的指针的引用（`r` is a reference to a pointer to an `int`）。

4.4 `const` Qualifier

可以通过`const`限定符（qualifier）来定义一个常量（constant），常量是一个在程序运行时不能被修改的值。使用`const`限定符可以防止意外修改变量的值，从而提高代码的安全性和可读性。由于创建常量之后不能进行修改，因此常量必须在声明时进行初始化，同样初始化器（initializer）可以是任意复杂的表达式。

```
const int bufSize = 512; // input buffer size
bufSize = 512; // error: attempt to write to const object
const int i = get_size(); // ok: initialized at run time
const int j = 42; // ok: initialized at compile time
const int k; // error: k is uninitialized const
```

由于对象的类型决定了可以对对象进行的操作集合，因此带有`const`限定符的对象和不带`const`限定符的对象可以进行的操作是不同的。其中一个限制是不能对带有`const`限定符的对象进行修改操作。

带有`const`限定符的对象在编译时经常会被优化为字面量（literal），即编译器会将变量名替换成其相应的值。为了替换变量名，编译器需要知道变量的初始化值，当一个程序被分为多个文件时，每个使用到`const`限定符修饰的变量的文件都需要知道该变量的初始化值，因此需要在每个文件中都进行定义。为了支持这种情况，并且防止多次定义同名变量，`const`限定符修饰的变量默认被定义为属于文件的（defined as local to the file），因此在多个不同的文件中使用同名的`const`限定符修饰的变量就像在不同文件中定义了不同的变量。

如果需要在多个文件中共享同一个`const`限定符修饰的变量，即在一个文件中定义该变量并在其他文件中进行声明并使用该变量，则需要在定义该变量的文件中使用`extern`关键字进行定义，并在需要使用该变量的其他文件中声明时也使用`extern`关键字。

```
// file_1.cc defines and initializes a const that is accessible to other files
extern const int bufSize = fcn();
// file_1.h
extern const int bufSize; // same bufSize as defined in file_1.cc
```

4.4.1 *References to `const`*

对于指向`const`限定符修饰的对象的引用，不能对引用的对象进行修改[5]。

[5] `const` reference is a reference to `const`：由于引用不是一个对象，因此不能用限定符`const`修饰引用，由于引用无法修改指向其他的对象，因此从某种程度来说引用本身就是`const`的。

```
const int ci = 1024;
const int &r1 = ci; // ok: both reference and underlying object are const
r1 = 42; // error: r1 is a reference to const
int &r2 = ci; // error: nonconst reference to a const object
```

由于ci不能修改，因此同样也不能用引用来间接修改ci的值。由于const限定符修饰的对象不能被修改，因此不能将const限定符修饰的对象绑定到一个非const限定符修饰的引用上。

在引用中，引用指向的对象的类型和引用本身的类型必须完全相同，而在const修饰的引用中是一个例外，即可以将任何可以经过类型转换变成引用的类型的表达式绑定到const限定符修饰的引用上（a reference to const）。特别是，可以将一个const限定符修饰的引用指向一个非const限定符修饰的对象、一个字面量或者更一般的表达式结果上。

```
int i = 42;
const int &r1 = i; // we can bind a const int& to a plain int object
const int &r2 = 42; // ok: r2 is a reference to a literal
const int &r3 = r1 * 2; // ok: r3 is a reference to an expression result
int &r4 = r1 * 2; // error: r4 is a plain, nonconst reference
```

理解在初始化时，const限定符修饰的引用可以绑定到一个非const限定符修饰的对象、字面量或表达式结果上，是因为编译器会将这些对象经过类型转换后变成const限定符修饰的一个临时（temporary）对象[6]，然后将其绑定到const限定符修饰的引用上。这种类型转换是安全的，因为const限定符修饰的对象不能被修改，因此不会影响原始对象的值。

```
double dval = 3.14;
const int &ri = dval;
```

在上面的代码中，dval是一个双精度浮点数，而ri是一个const限定符修饰的整数引用。编译器会将dval转换为一个临时的const int对象，然后将其绑定到ri上。由于ri是一个const限定符修饰的引用，因此不能对其进行修改。

```
const int temp = dval; // create a temporary const int from the double
const int &ri = temp; // bind ri to that temporary
```

因此下面的代码输出的两次结果中ri都是3，因为ri绑定到的临时对象的值是3。同时也说明，尽管不能通过const限定符修饰的引用来修改原始对象的值，但是可以通过修改原始对象或通过其他方式来修改原始对象的值（如果原始对象是非const限定符修饰的对象）。

```
double dval = 3.14;
const int &ri = dval;
double &ri2 = dval; // ri2 is a reference to dval, not to the temporary
std::cout << "Value of ri: " << ri << std::endl;
std::cout << "Value of ri2: " << ri2 << std::endl;
dval = 1.71; // Changing dval
std::cout << "Updated value of ri: " << ri << std::endl; // ri remains unchanged
std::cout << "Value of ri2: " << ri2 << std::endl; // ri2 reflects the change in dval
```

4.4.2 *Pointers and const*

和引用一样，被const限定符修饰的指针不能修改指向的对象的值，但是能够指向非const限定符修饰的对象，即指针的类型和被指向的对象的类型可以不完全相同。

```
const double pi = 3.14; // pi is const; its value may not be changed
double *ptr = &pi; // error: ptr is a plain pointer
const double *cptr = &pi; // ok: cptr may point to a double that is const
```

[6]当需要存储计算表达式的值时，编译器就会创建一个无名对象（unnamed object），即临时对象。C++ 程序员通常将 temporary object 简称为 temporary。

```
    *cptr = 42; // error: cannot assign to *cptr
    double dval = 3.14; // dval is a double; its value can be changed
    cptr = &dval; // ok: but can't change dval through cptr
```

不同于引用，由于指针是一个对象，因此可以定义由`const`限定符修饰的指针，即指针本身是`const`的，不能修改指针的值，即不能修改指针指向的对象[7]。和其他类型一样，`const`限定符修饰的指针必须在声明时进行初始化，并且在初始化之后该指针指向的对象不能被修改。通过在`*`后面添加`const`限定符来表明指针本身是`const`的，而不是指向的对象是`const`的。

```
    int errNumb = 0;
    int *const curErr = &errNumb; // curErr will always point to errNumb
    const double pi = 3.14159;
    const double *const pip = &pi; // pip is a const pointer to a const object
```

`curErr`是一个`const`限定符修饰的指针，指向一个整数类型的对象`errNumb`，因此不能修改`curErr`指向的对象，但是可以修改`errNumb`的值。因为`pip`是一个`const`限定符修饰的指针并且它指向的对象是`const double`类型的，因此不能修改`pip`指向的对象，也不能通过指针`pip`修改`pi`的值。

4.4.3 *Top-Level `const` and Low-Level `const`*

一般地，将修饰对象本身的`const`限定符称为顶层`const`（top-level `const`），它可以出现在任何对象类型中，包括内置类型、类类型和指针类型等。而底层`const`（low-level `const`）则出现在像指针和引用的复合类型（compound types）的基类型中。指针不像其他类型，它可以同时有顶层`const`和底层`const`。

```
    int i = 0;
    int *const p1 = &i; // we can't change the value of p1; const is top-level
    const int ci = 42; // we cannot change ci; const is top-level
    const int *p2 = &ci; // we can change p2; const is low-level
    const int *const p3 = p2; // right-most const is top-level; left-most const is low-level
    const int &r = ci; // const in reference types is always top-level
```

顶层`const`和底层`const`在复制对象时比较重要，当复制对象时，顶层`const`会被忽略，而底层`const`则永远不会被忽略。当复制对象时，两个对象必须有相同的底层`const`或者它们之间的底层`const`可以进行转换。一般而言，可以将非`const`限定符修饰的类型赋值给`const`限定符修饰的类型，但是不能将`const`限定符修饰的类型赋值给非`const`限定符修饰的类型。

```
    i = ci; // ok: copying the value of ci; top-level const in ci is ignored
    p2 = p3; // ok: pointed-to type matches; top-level const in p3 is ignored
    int *p = p3; // error: p3 has a low-level const but p doesn't
    p2 = p3; // ok: p2 has the same low-level const qualification as p3
    p2 = &i; // ok: we can convert int* to const int*
    int &r = ci; // error: can't bind an ordinary int& to a const int object
    const int &r2 = i; //ok: can bind const int& to plain int
```

[7] 至于能否修改指针指向的对象的值则取决于指针指向的对象是否有`const`修饰符修饰。

Part

III

Traveling Salesman Problem

5

Symmetric Traveling Salesman Problem

5.1 Problem Definition

旅行商问题（Traveling Salesman Problem, TSP）是组合优化领域的经典问题之一，其核心目标是给定城市列表和每对城市之间的距离，求恰好访问每个城市一次并返回起始城市的最短可能路线。该问题于 1930 年正式提出，是优化中研究最深入的问题之一，被用作许多优化方法的基准。自从该问题被正式提出以来，一直是运筹学、计算机科学和物流管理等领域的研究热点，尽管该问题在计算上很困难，但许多启发式方法和精确算法是已知的[2-3]。

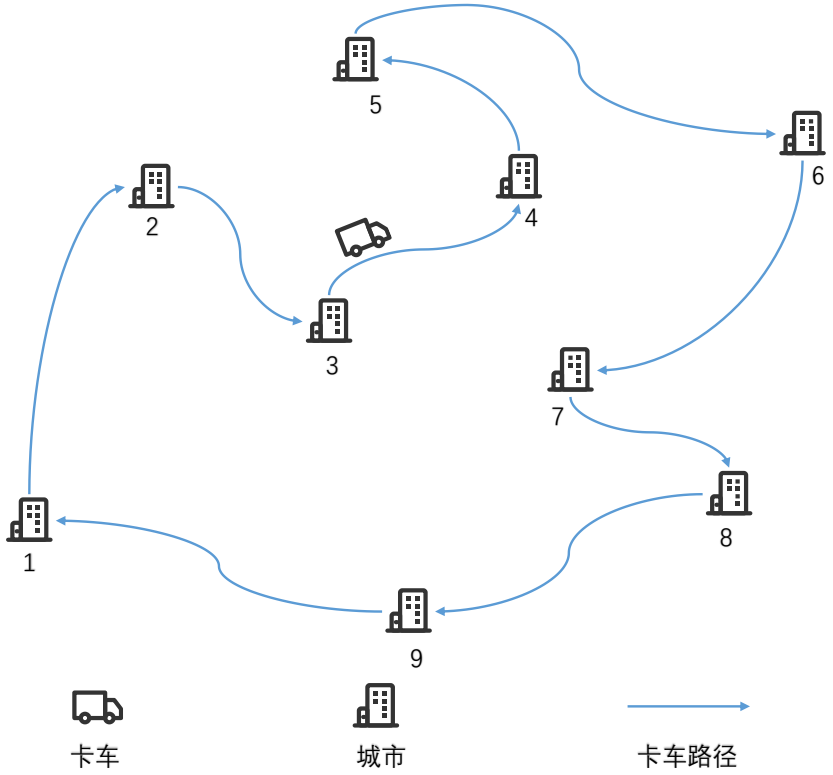


图 5-1: TSP 示意图

TSP 可以表述为整数线性规划模型[4]：假设共有 N 个城市，每个城市的编号为 $1, \cdots, N$，从城市 i

到城市 j 的旅行成本（距离）为 $c_{ij} > 0$。旅行商的目标是从任意一个城市出发访问完所有的城市，每个城市只能访问一次，最后回到最初的城市，目标是找到一条依次访问所有城市且访问城市不重复的最短路线。TSP 中的决策变量为 $x_{ij} = \begin{cases} 1, & \text{存在从城市 } i \text{ 到城市 } j \text{ 的路径} \\ 0, & \text{其他} \end{cases}$，城市节点集合表示为 $V(|V| = N)$。由于可能存在子回路，所以在构建 TSP 模型时需要消除会产生子回路的情况，这里采用 Miller-Tucker-Zemlin (MTZ) 约束进行子回路的消除[5]，引入连续变量 $u_i(\forall i \in V, u_i \geq 0)$，其取值可以为任何非负实数（实数集合表示为 R）。这里用 u_i 表示编号为 i 的城市的访问次序，比如当 $u_i = 5$ 时表示编号为 1 的城市是从出发点开始，第 5 个被访问到的点。因此，TSP 的数学模型可以表示为 MILP 5.1。

Model 5.1: TSP MILP

$$\min \quad \sum_{i \in V} \sum_{j \in V, i \neq j} c_{ij} x_{ij} \tag{5-1}$$

$$\text{s.t.} \quad \sum_{i \in V} x_{ij} = 1, \qquad \forall j \in V, i \neq j \tag{5-2}$$

$$\sum_{j \in V} x_{ij} = 1, \qquad \forall i \in V, i \neq j \tag{5-3}$$

$$u_i - u_j + N x_{ij} \leq N - 1, \qquad \forall i, j \in V; i \neq j \tag{5-4}$$

$$u_i \geq 0, \qquad u_i \in R \tag{5-5}$$

$$x_{ij} \in \{0, 1\}, \qquad i, j \in V; i \neq j \tag{5-6}$$

目标函数5-1表示最小化访问所有城市的成本（距离），约束5-2和5-3保证每个城市节点的入度和出度为 1，即每个城市只进入一次和出去一次，保证了每个城市只访问一次，不会被重复访问，约束5-4消除子回路，约束5-5和5-6表示变量的取值范围。

5.2 Solution Methods

5.2.1 Exact Methods

Concorde是一个求解 TSP 的精确算法，由ANSI C编写。在Concorde Download页面可以下载到最新版本的 Concorde。下载后通过在命令行输入命令5.1进行解压，解压后会得到一个名为 `concorde` 的文件夹，编译的过程参考Ubuntu（Linux）安装 concorde 过程或者参考Installing Concorde。

Code 5.1: Concorde unzip

```
gunzip co031219.tgz
tar xvf co031219.tar
```

Concorde 需要 linear programming solver，常用的有QSOpt和 IBM 的CPLEX，由于 Concorde 自从 2003 年之后就没有更新过，因此当前 CPLEX 的版本已经不再合适，因此选用 QSOpt。QSOpt 的路径假设为`path=/home/kaiyouhu/qsopt`，用命令5.2下载 QSOpt 的头文件，

Code 5.2: QSOpt download

```
mkdir qsopt
cd qsopt
wget -O qsopt.a http://www.math.uwaterloo.ca/~bico/qsopt/beta/codes/PIC/qsopt.PIC.a # WSL 上的 Ubuntu 测试有效
```

Code 5.2 (continued)

```
wget http://www.math.uwaterloo.ca/~bico/qsopt/beta/codes/linux64/qsopt.h
wget http://www.math.uwaterloo.ca/~bico/qsopt/beta/codes/linux64/qsopt
```

然后在 `concorde` 文件夹下使用命令5.3编译 Concorde，编译完成后会在当前目录下生成一个名为 `TSP` 的文件。

Code 5.3: Concorde make

```
CFLAGS="-fPIC -O2 -g" ./configure --with-qsopt=/home/kaiyouhu/qsopt # 如果 QSOpt 的路径不同记得修改
# continue
make
cd TSP
./concorde -s 99 -k 100 # -s 99 表示使用随机种子 99，-k 100 表示使用 100 个城市的 TSP 实例进行测试
```

在 `Python` 中使用 `Concorde` 可以参考pyconcorde，该项目是 `Concorde` 的 `Python` 接口，需要注意的是这个 `Python` 接口不支持 `Windows` 系统，但是支持 `WSL`，具体的安装细节参考文档说明。

5.2.2 *Heuristic Methods*

LKH是解决 TSP 的Lin-Kernighan启发式算法的有效实现，基于 ANSI C 编写。在此之外，LKH 还有一个改进的LKH-3版本，支持多种约束条件 (Asymmetric capacitated vehicle routing problem, Multiple traveling salesmen problem, Sequential ordering problem...) 的 TSP 求解。在LKH页面可以下载到最新版本的 LKH 实现代码。下载后通过在命令行输入命令5.4进行解压，解压后会得到一个名为 `LKH-2.0.10` 的文件夹，然后在 `Linux` 或者 `WSL` 下输入命令`make`编译 LKH 代码，编译完成后会在当前目录下生成一个名为 `LKH` 的文件。

Code 5.4: LKH unzip and make

```
tar xvfz LKH-2.0.10.tgz
cd LKH-2.0.10
make
```

在 `Python` 中使用 `LKH` 可以参考LKH_TSP，该项目是 `LKH` 的 `Python` 接口，需要注意的是这个 `Python` 接口不支持 `Windows` 系统，但是可以在 `WSL` 中运行。

Part

IV

Vehicle Routing Problem

车辆路径规划问题（Vehicle Routing Problem, VRP）是物流配送领域中的核心优化问题之一，由 George Dantzig 和 John Ramser 于 1959 年首次提出[6]。其目的是为一组具有容量限制的车辆设计最优配送路线，使得所有客户需求被满足且总运输成本（如距离、时间或费用）最小化。当车辆容量足够大时，VRP 退化为 TSP，即当车辆容量足够时，所有货物都可以在一次行驶中全部配送，只需要经过一次配送中心。大多数情况下 VRP 的车辆容量总是小于需要配送的所有货物重量的总和，所以与旅行商问题（TSP）不同，VRP 需要同时考虑多车辆协同、客户需求分配、车辆容量限制等复杂约束，因此更具现实意义和研究挑战性。

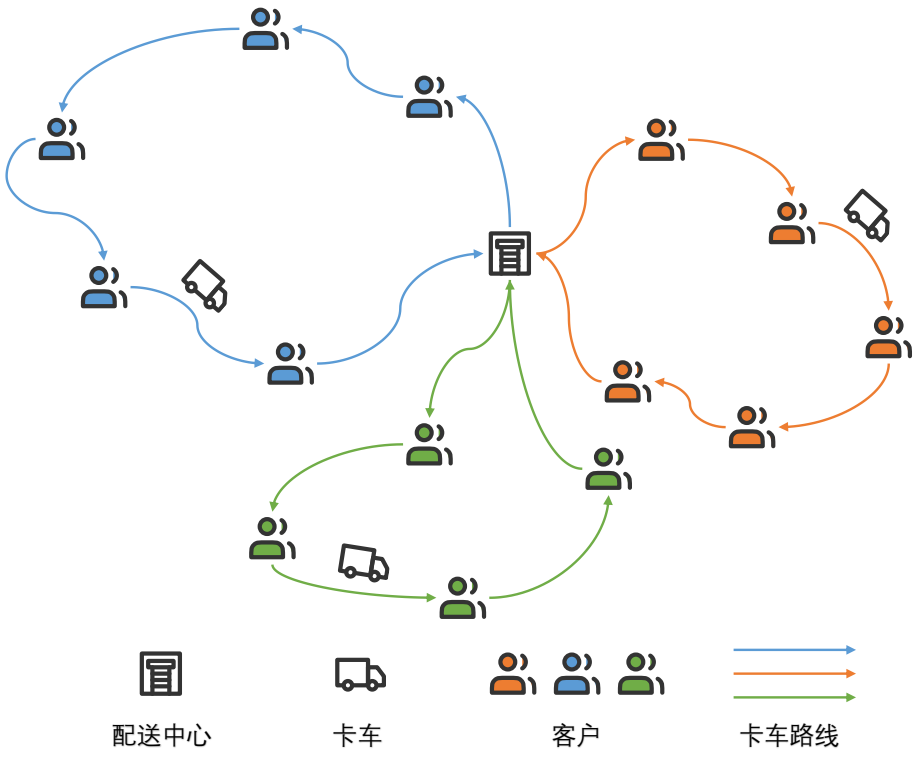


图 5-2: VRP 示意图

VRP 可以表述为整数规划问题[7]：假设存在一个配送中心（仓库）和若干顾客点，顾客点需求为 $q_i(i = 1, 2, \cdots, n)$，车辆载重上限为 Q，每辆车从仓库出发并最终返回仓库。定义决策变量 $x_{ijk} = \begin{cases} 1, & \text{车辆 } k \text{ 从 } i \text{ 行驶到 } j \\ 0, & \text{其他} \end{cases}$，节点集合 $V = \{0, 1, 2, \cdots, n\}$，其中 0 表示配送中心，$S = \{1, 2, \cdots, n\}$ 表示顾客节点，车辆集合 $K = \{1, 2, \cdots, m\}$，c_{ij} 表示从点 i 到点 j 的行驶成本（距离或时间），同样在 VRP 中为了消除子回路，引入辅助变量 u_i 表示车辆访问顾客点 i 时的累计载重量。因此，VRP 的数学模型可以表示为 MILP 5.2。

Model 5.2: MILP VRP

$$\min \quad \sum_{k \in K} \sum_{i \in V} \sum_{j \in V, j \neq i} c_{ij} x_{ijk} \tag{5-7}$$

$$\text{s.t.} \quad \sum_{j \in S} x_{0jk} = 1, \qquad \forall k \in K \tag{5-8}$$

Model 5.2 (continued)

$$\sum_{i \in S} x_{i0k} = 1, \qquad \forall k \in K \tag{5-9}$$

$$\sum_{k \in K} \sum_{i \in V, i \neq j} x_{ijk} = 1, \qquad \forall j \in S \tag{5-10}$$

$$\sum_{i \in V, i \neq j} x_{ijk} = \sum_{i \in V, i \neq j} x_{jik}, \qquad \forall j \in V, k \in K \tag{5-11}$$

$$u_j \geq u_i + q_j - Q(1 - x_{ijk}), \qquad \forall i, j \in S, k \in K \tag{5-12}$$

$$q_j \leq u_j \leq Q, \qquad \forall j \in S \tag{5-13}$$

$$x_{ijk} \in \{0, 1\}, \qquad \forall i, j \in V, k \in K \tag{5-14}$$

目标函数5-7表示最小化所有车辆的总行驶成本，约束5-8和5-9确保每辆车从配送中心出发并最终返回，约束5-10保证每个顾客点只被访问一次，约束5-11保证了流量守恒，即保证了路径的连续性，约束5-12和5-13通过 MTZ 方法消除子回路并满足车辆的容量限制，约束5-14对变量进行限制。

Part

V

TDRP: Truck and Drone collaborative Routing Problem

6

Traveling Salesman Problem with Drone

6.1 Flying Sidekick Traveling Salesman Problem

Flying Sidekick Traveling Salesman Problem (FSTSP) 由 Murray(2015) 等[8]提出。FSTSP 描述：假定有 c 个顾客需要服务，这个服务可以由无人机或者卡车来提供，但是有些顾客由于某些限制（比如包裹重量超过无人机的载重限制）只能由卡车提供服务。卡车和无人机必须从单一仓库出发，并且返回该仓库一次，即不能重复访问该仓库，无人机和卡车可以同时或者分别离开（返回）仓库。在无人机起飞前需要卡车司机装载包裹和更换电池的时间 s_L，降落时需要给无人机装卸货物和充电的时间 s_R。每次无人机的一次服务称为一次 sortie，一次 sortie 分为三个节点 $\langle i, j, k \rangle$，起飞点 i 可以是仓库也可以是顾客点，中间节点 j 是需要服务的顾客节点，节点 k 可以是仓库也可以是卡车所在的顾客节点，无人机在整个运输过程中可以进行多次 sortie 服务多个顾客，但是一次 sortie 的时间必须在无人机的续航时间内。FSTSP 的目标是最小化服务所有顾客并且返回仓库（无人机和卡车都返回）的时间。

关键假设如下：

- 无人机每次 sortie 的过程中只能服务一个顾客节点，但是在这期间卡车可以服务多个顾客节点。
- 无人机被假定为匀速飞行，如果无人机比卡车提前到达会合点则无人机不能在中途停下休息以节省电量。
- 无人机可以在降落点重新发射，但是无人机不能返回上一次的发射点。
- 无人机和卡车的会合点必须在顾客节点，而不能在中间的任何位置会合，并且卡车不会重新访问已经服务过的顾客节点来回收无人机。
- 无人机和卡车都不能访问除了仓库以外的非顾客节点（即只考虑简化过后的实际情况），并且无人机和卡车都不能重新访问已经服务过的顾客节点。
- 如果无人机返回仓库则不能再次进行服务，这是基于大多顾客节点都离仓库较远（大于无人机的续航里程）的假设，在无人机可以直接起飞服务顾客节点的假设下，PDSTSP 会更加适合。

FSTSP 数学模型的符号含义如表6-1所示。

FSTSP 数学模型可以表示为 MILP 6.1。

表 6-1: FSTSP 模型符号及含义

符号	含义
0	起点仓库
$c+1$	终点仓库（和起点仓库相同，只是为了建模方便的另一个记号）
$C=\{1,2,\cdots,c\}$	全部客户集合
$C'\subseteq C$	无人机可访问的客户集合
$N_0=\{0,1,2,\cdots,c\}$	流出节点集合
$N_+=\{1,2,\cdots,c+1\}$	流入节点集合
$N=\{0,1,2,\cdots,c,c+1\}$	全部节点集合
$\langle i,j,k\rangle\in P, i\in N_0, j\in\{C':j\neq i\}$, $k\in\{N_+:k\neq i,k\neq j,\tau'_{ij}+\tau'_{jk}\leq e\}$	无人机飞行路径集合
$\tau'_{ij}/\tau_{ij}, i\in N_0, j\in N_+, i\neq j, \tau_{0,c+1}\equiv 0$[a]	弧 $\langle i,j\rangle$ 的飞行/行驶时间成本
s_L/s_R	无人机发射/回收耗时
e	无人机续航时长，以单位时间来衡量
$x_{ij}\in\{0,1\}, i\in N_0, j\in N_+, i\neq j$	卡车路由决策变量
$y_{ijk}\in\{0,1\}, i\in N_0, j\in C, k\in\{N_+:\langle i,j,k\rangle\in P\}$	无人机路由决策变量
$t'_i/t_i\geq 0, i\in N_+, t'_0=t_0=0$	无人机/卡车有效到达时间戳辅助变量
$p_{ij}\in\{0,1\}$[b], $p_{0j}=1,\forall j\in C$	卡车访问次序先后辅助变量（为了确保无人机连续的 sortie 和卡车访问的顺序一致[c]）
$1\leq u_i\leq c+2, i\in N_+$	卡车破子圈辅助变量（和标准 TSP 的 MTZ 形式破子圈辅助变量类似[d]）

[a] 出于完备性的考虑，当只有一个顾客节点的时候，这个顾客将由无人机从仓库直接起飞进行服务。

[b] 当卡车访问顾客节点 $j\in\{C:j\neq i\}$ 时，顾客节点 $i\in C$ 已经在之前的某个时间点被卡车访问过了，则 $p_{ij}=1$。

[c] 当顾客节点 i 或者 j 仅被无人机服务时，p_{ij} 的取值就不重要。

[d] u_i 表示顾客点 i 在卡车访问的路径中的次序，比如 $u_5=1$ 表示顾客点 $i=5$ 是卡车访问路径中的第 1 个节点，但是不同于 TSP，在 FSTSP 中需要通过约束将无人机服务的顾客点 i 排除在外。

Model 6.1: FSTSP MILP

$$\min \quad t_{c+1} \tag{6-1}$$

$$\text{s.t.} \quad \sum_{\substack{i\in N_0\\ i\neq j}} x_{ij} + \sum_{\substack{i\in N_0\\ i\neq j}} \sum_{\substack{k\in N_+\\ \langle i,j,k\rangle\in P}} y_{ijk} = 1, \quad \forall j \in C \tag{6-2}$$

$$\sum_{j\in N_+} x_{0j} = 1 \tag{6-3}$$

$$\sum_{i\in N_0} x_{i,c+1} = 1 \tag{6-4}$$

$$u_i - u_j + 1 \leq (c+2)(1-x_{ij}), \quad \forall i \in C, j \in \{N_+ : j \neq i\} \tag{6-5}$$

$$\sum_{\substack{i\in N_0\\ i\neq j}} x_{ij} = \sum_{\substack{k\in N_+\\ k\neq j}} x_{jk}, \quad \forall j \in C \tag{6-6}$$

$$\sum_{\substack{j\in C\\ j\neq i}} \sum_{\substack{k\in N_+\\ \langle i,j,k\rangle\in P}} y_{ijk} \leq 1, \quad \forall i \in N_0 \tag{6-7}$$

$$\sum_{\substack{i\in N_0\\ i\neq k}} \sum_{\substack{j\in C\\ \langle i,j,k\rangle\in P}} y_{ijk} \leq 1, \quad \forall k \in N_+ \tag{6-8}$$

$$2y_{ijk} \leq \sum_{\substack{h\in N_0\\ h\neq i}} x_{hi} + \sum_{\substack{l\in C\\ l\neq k}} x_{lk}, \quad \forall i \in C, j \in \{C : j \neq i\}, k \in \{N_+ : \langle i,j,k\rangle \in P\} \tag{6-9}$$

$$y_{0jk} \leq \sum_{\substack{h\in N_0\\ h\neq k}} x_{hk}, \quad \forall j \in C, k \in \{N_+ : \langle 0,j,k\rangle \in P\} \tag{6-10}$$

$$u_k - u_i \geq 1 - (c+2)\left(1 - \sum_{\substack{j\in C\\ \langle i,j,k\rangle\in P}} y_{ijk}\right), \quad \forall i \in C, k \in \{N_+ : k \neq i\} \tag{6-11}$$

$$t'_i \geq t_i - M\left(1 - \sum_{\substack{j\in C\\ j\neq i}} \sum_{\substack{k\in N_+\\ \langle i,j,k\rangle\in P}} y_{ijk}\right), \quad \forall i \in C \tag{6-12}$$

$$t'_i \leq t_i + M\left(1 - \sum_{\substack{j\in C\\ j\neq i}} \sum_{\substack{k\in N_+\\ \langle i,j,k\rangle\in P}} y_{ijk}\right), \quad \forall i \in C \tag{6-13}$$

$$t'_k \geq t_k - M\left(1 - \sum_{\substack{i\in N_0\\ i\neq k}} \sum_{\substack{j\in C\\ \langle i,j,k\rangle\in P}} y_{ijk}\right), \quad \forall k \in N_+ \tag{6-14}$$

$$t'_k \leq t_k + M\left(1 - \sum_{\substack{i\in N_0\\ i\neq k}} \sum_{\substack{j\in C\\ \langle i,j,k\rangle\in P}} y_{ijk}\right), \quad \forall k \in N_+ \tag{6-15}$$

Model 6.1 (continued)

$$t_k \geq t_h + \tau_{hk} + s_L \left(\sum_{\substack{l \in C \\ l \neq k}} \sum_{\substack{m \in N_+ \\ \langle k,l,m \rangle \in P}} y_{klm} \right) + s_R \left(\sum_{\substack{i \in N_0 \\ i \neq k}} \sum_{\substack{j \in C \\ \langle i,j,k \rangle \in P}} y_{ijk} \right) - M(1 - x_{hk}),$$

$$\forall h \in N_0, k \in \{N_+ : k \neq h\}$$

(6-16)

$$t'_j \geq t'_i + \tau'_{ij} - M \left(1 - \sum_{\substack{k \in N_+ \\ \langle i,j,k \rangle \in P}} y_{ijk} \right), \quad \forall j \in C', i \in \{N_0 : i \neq j\} \tag{6-17}$$

$$t'_k \geq t'_j + \tau'_{jk} + s_R - M \left(1 - \sum_{\substack{i \in N_0 \\ \langle i,j,k \rangle \in P}} y_{ijk} \right), \quad \forall j \in C', k \in \{N_+ : k \neq j\} \tag{6-18}$$

$$t'_k - (t'_j - \tau'_{ij}) \leq e + M(1 - y_{ijk}), \quad \forall k \in N_+, j \in \{C : j \neq k\}, i \in \{N_0 : \langle i,j,k \rangle \in P\}$$

(6-19)

$$u_i - u_j \geq 1 - (c+2)p_{ij}, \quad \forall i \in C, j \in \{C : j \neq i\} \tag{6-20}$$

$$u_i - u_j \leq -1 + (c+2)(1 - p_{ij}), \quad \forall i \in C, j \in \{C : j \neq i\} \tag{6-21}$$

$$p_{ij} + p_{ji} = 1, \quad \forall i \in C, j \in \{C : j \neq i\} \tag{6-22}$$

$$t'_l \geq t'_k - M \left(3 - \sum_{\substack{j \in C \\ \langle i,j,k \rangle \in P \\ j \neq l}} y_{ijk} - \sum_{\substack{m \in C \\ m \neq i \\ m \neq k \\ m \neq l}} \sum_{\substack{n \in N_+ \\ \langle l,m,n \rangle \in P \\ n \neq i \\ n \neq k}} y_{lmn} - p_{il} \right)$$

(6-23)

$$\forall i \in N_0, k \in \{N_+ : k \neq i\}, l \in \{C : l \neq i, l \neq k\}$$

$$t_0 = 0 \tag{6-24}$$

$$t'_0 = 0 \tag{6-25}$$

$$p_{0j} = 1, \quad \forall j \in C \tag{6-26}$$

$$x_{ij} \in \{0,1\}, \quad \forall i \in N_0, j \in \{N_+ : j \neq i\} \tag{6-27}$$

$$y_{ijk} \in \{0,1\}, \quad \forall i \in N_0, j \in \{C : j \neq i\}, k \in \{N_+ : \langle i,j,k \rangle \in P\} \tag{6-28}$$

$$1 \leq u_i \leq c+2, \quad \forall i \in N_+ \tag{6-29}$$

$$t_i \geq 0, \quad \forall i \in N \tag{6-30}$$

$$t'_i \geq 0, \quad \forall i \in N \tag{6-31}$$

$$p_{ij} \in \{0,1\}, \quad \forall i \in N_0, j \in \{C : j \neq i\} \tag{6-32}$$

目标函数6-1追求最小化卡车到达终点仓库 $c+1$ 的有效时间 t_{c+1}，通过约束 6-14 和 6-15 来对齐无人机和卡车最晚到达终点仓库的时间，所以目标函数相当于 $\min\{\max\{t_{c+1}, t'_{c+1}\}\}$。

参考知乎解读文章，可以将约束条件可以分为四类：

- 客户有关的约束：约束 6-2 要求对于任何一位顾客 j，必须且只能被卡车（或无人机）服务一次。
- 卡车有关的约束：

  - 卡车流平衡约束：约束 6-3 要求卡车必须从起点仓库流出一次，约束 6-4 要求卡车必须从终点仓库流入一次，约束 6-6 要求卡车在中间节点满足流入和流出相等的流平衡约束。

  - 卡车破子圈约束：约束 6-5 是 MTZ 形式的破子圈约束[1]，去除了子圈存在的可能，这里 M 取到了 $u_i - u_j + 1$ 的上界 $c + 2$，u_i 可以理解为点 i 的访问次序，比如 $u_1 = 5$ 可以理解为点 1 是从出发点开始，第五个被访问到的点。

- 无人机有关的约束：

  - 无人机发射、回收节点流约束：约束 6-7 表示无人机可以从非终点仓库流出，约束 6-8 表示无人机可以从非起点仓库流入。

  - 无人机访问、回收节点时间戳约束：约束 6-17 表示无人机访问顾客的时间戳应该符合时间逻辑，即不早于起飞时间戳 t'_i+ 前往服务顾客点的飞行时长 τ'_{ij}[2]，约束 6-18 表示无人机回到卡车的时间戳应该符合时间逻辑，即不早于访问顾客点的 t'_j+ 返回卡车的飞行时长 τ'_{jk}+ 回收无人机用时 s_R[3]。

  - 无人机电量续航约束：约束 6-19 表示无人机的飞行时间不能超过其续航时间，即到达会合点 t'_k 的有效时间 − 无人机从节点 i 的起飞时间 $(t'_j - \tau'_{ij})$（不直接使用 t'_i 是因为 t'_i 不是起飞的时间戳而是无人机到达 i 点的时间戳）要在无人机的续航时间 e 之内。

  - 无人机飞行次序约束：约束 6-23 要求无人机对于任意两条路径 $\langle i, j, k\rangle$ 和 $\langle l, m, n\rangle$ 而言，如果无人机先从节点 i 起飞之后的某个时间才从节点 l（$p_{il} = 1$）起飞，则无人机必须先完成上一次飞行才能继续下一次飞行（$t'_l \geq t'_k$），并且任意两条路径之间无交叉。

- 无人机和卡车同步有关的约束：

  - 无人机发射、回收点卡车访问约束：约束 6-9 要求对于非起点发射的无人机（$\forall i \in C$），卡车必须经过无人机的起飞点 i 和降落点 k，约束 6-10 要求对于从起点仓库起飞的无人机来说，卡车必须经过无人机的降落点。

  - 无人机访问顾客时卡车访问次序约束：约束 6-11 要求卡车必须先访问无人机的起飞点再访问无人机的降落点。

  - 无人机发射点时间戳约束：约束 6-12 和 6-13 为无人机发射点的有效时间约束，要求无人机在发射节点的有效时间等于卡车在该点的有效时间，共同实现了卡车和无人机在发射节点时间上的对齐。

  - 无人机回收点时间戳约束：约束 6-14 和 6-15 为无人机回收点的有效时间约束，要求无人机在回收节点的有效时间等于卡车在该点的有效时间，共同实现了卡车和无人机在回收节点时间上的对齐。

  - 卡车访问顾客节点时间戳约束：约束 6-16 要求卡车访问当前顾客点 k 时必须要先将需要起飞的无人机 s_L 发射或者需要降落的无人机 s_R 回收，并且要大于到达顾客点 h 的有效时间戳 t_h+ 路径 $\langle h, k\rangle$ 所花费的时间 τ_{hk}[4]。

- 辅助变量和决策变量：

[1]关于破子圈约束可以参考TSP 中两种不同消除子环路的方法和浅谈旅行商问题（TSP）的七种整数规划模型

[2]起飞时间 s_L 没有被包含进来的原因是，当无人机从顾客点 i 起飞时，约束 6-12 和 6-13 会使得 $t'_i = t_i$，然后约束 6-14 和 6-15 会将起飞的时间 s_L 包含在飞行到顾客点 j 的时间内。

[3]这里降落的时间 s_R 必须被包含进来的原因是，卡车可能比无人机更快到达会合点 k（原文中举出了一个例子以助于理解[8]）。

[4]这里假设卡车从 $h \in N_0$ 行驶到 $k \in N_+$。

- 卡车访问次序约束：约束6-20，6-21和6-22决定了卡车访问次序 p_{ij} 取值的合理性，u_i 和 p_{ij} 主要用于约束被卡车访问的节点之间的次序，对于被无人机服务的顾客点 i 或者 j 来说，u_i 和 p_{ij} 的取值并不重要。
- 辅助变量及决策变量的初始值和取值范围：约束6-24和6-25给定了卡车和无人机有效时间的初始值，约束6-26规定了起点仓库的访问次序一定在其他所有顾客节点之前，约束6-27和6-28给定了决策变量的取值范围，约束6-29规定了卡车破子圈辅助变量的取值范围，约束6-30和6-31规定了卡车和无人机的有效时间必须是非负实数，约束6-32给定了卡车访问次序辅助变量的取值范围。

在约束 equations (6-12) to (6-19) 中，$M \geq \max\{t_{c+1}, t'_{c+1}\}$ 取一个非常大的数，需要大于等于最后到达终点仓库的卡车（无人机）的有效时间。由于无法事先确定最小可接受的 M 值，因此一种方法是用 nearest neighbor heuristic 来计算一个访问所有顾客节点并返回仓库的时间上限。算法的大致过程：初始化 $M = 0$，然后从仓库开始构建卡车路径（$i = 0$），找到最近的还未访问过的顾客节点 j，让 $M \leftarrow M + \tau_{ij}$，更新 $i = j$ 然后重复这个过程，即不断添加距离最近的未访问过的顾客节点直到所有的顾客都被访问一遍，最终让 $M \leftarrow M + \tau_{i,c+1}$，即让卡车返回仓库。

6.1.1 *Flying Sidekick Traveling Salesman Problem with Multiple Drops*

Gonzalez-R 等 (2020)[9] 最早提出了类型为 Flying Sidekick Traveling Salesman Problem with Multiple Drops (FSTSP-MD) 的 MILP 数学模型，在文章中，作者称之为 truck-drone team logistic (TDTL)。该问题聚焦于 FSTSP 问题的延伸，即考虑无人机可以在单次起飞降落的过程（在 Murray 等 (2015)[8] 中被称为 sortie）服务多个顾客节点的情况，但是不同于 FSTSP，TDTL 的起始节点和终止节点不是同一个节点。

关键假设如下：

- 不考虑卡车的最大航行里程，即卡车在整个运输过程中不需要补充燃料。
- 无人机和卡车只能在顾客节点会合，而不能够在任意的中间路程中进行会合。无人机和卡车在会合点通过更换电池使无人机充满电。无人机和卡车在会合点需要进行时间对齐，即如果无人机先到达会合点则需要等待卡车到达，反之亦然。
- 对于无人机更换电池的次数没有限制，无人机更换电池的服务时间相对于总体的时间而言可以忽略不计，因此不考虑更换电池的时间。由于无人机在每次和卡车会合时是通过更换电池来达到充电的目的，所以每次无人机在和卡车会合后起飞时，无人机的电池是满的。
- 只要无人机有充足的电量，无人机就会在服务过程中访问多个顾客直到下一次和卡车在顾客节点会合。
- 当无人机搭载在卡车上进行访问顾客节点时，无人机不消耗电量。
- 节点之间的距离用欧几里得距离来衡量，无人机和卡车都以恒定的速度行驶。
- 为了简化模型，假定无人机的电量消耗是线性的，即无人机的电量不会随着起飞降落或者携带的货物重量而进行非线性的变化。
- 每个顾客必须被无人机或者卡车服务一次。

TDTL 数学模型的符号含义如表6-3所示。

TDTL 数学模型可以表示为 MILP 6.2。

表 6-3: TDTL 模型符号及含义

符号	含义
$G = \{N, A\}$	定义了要访问的节点集合和连接它们的有向链接集合的图。
N	图 G 的节点集合。
$o \in N$	任务的起始节点。
$e \in N$	任务的结束节点。
A	图 G 中的有向链接集合。
$\delta^+(i)$	可以从节点 $i \in N$ 通过链接 A 到达的节点集合。
$\delta^-(i)$	可以通过链接 A 到达节点 $i \in N$ 的节点集合。
Q	无人机电池的最大充电水平（以时间单位表示），即电池更换时的电量。
t_{ij}^T	链接 $(i,j) \in A$ 上卡车的行驶时间。
t_{ij}^D	链接 $(i,j) \in A$ 上无人机的行驶时间。
M	一个足够大的常数。
u_{ij}	二元变量，如果链接 $(i,j) \in A$ 被卡车遍历，则等于 1。
v_{ij}	二元变量，如果链接 $(i,j) \in A$ 被无人机遍历，则等于 1。
s_i	连续变量，衡量节点 $i \in N$ 被卡车或无人机服务后的最早出发时间。
b_i^-	连续变量，衡量无人机到达节点 $i \in N$ 时的电池电量。
b_i^+	连续变量，衡量无人机离开节点 $i \in N$ 时的电池电量。

Model 6.2: TDTL MILP

$$\min \quad s_e \tag{6-33}$$

$$\text{s.t.} \quad \sum_{j \in \delta^-(i)} u_{ji} \le 1, \quad \forall i \in N \setminus \{o, e\} \tag{6-34}$$

$$\sum_{j \in \delta^+(i)} u_{ij} - \sum_{j \in \delta^-(i)} u_{ji} = 0, \quad \forall i \in N \setminus \{o, e\} \tag{6-35}$$

$$\sum_{j \in \delta^+(o)} u_{oj} = 1 \tag{6-36}$$

$$\sum_{i \in \delta^-(e)} u_{ie} = 1 \tag{6-37}$$

$$\sum_{j \in \delta^-(i)} v_{ji} \le 1, \quad \forall i \in N \setminus \{o, e\} \tag{6-38}$$

$$\sum_{j \in \delta^+(i)} v_{ij} - \sum_{j \in \delta^-(i)} v_{ji} = 0, \quad \forall i \in N \setminus \{o, e\} \tag{6-39}$$

$$\sum_{j \in \delta^+(o)} v_{oj} = 1 \tag{6-40}$$

$$\sum_{i \in \delta^-(e)} v_{ie} = 1 \tag{6-41}$$

$$\sum_{i \in \delta^-(j)} u_{ij} + \sum_{i \in \delta^-(j)} v_{ij} \ge 1, \quad \forall j \in N \setminus \{o\} \tag{6-42}$$

$$s_j \ge s_i + t_{ij}^T \cdot u_{ij} - M \cdot (1 - u_{ij}), \quad \forall (i,j) \in A \tag{6-43}$$

$$s_j \ge s_i + t_{ij}^D \cdot v_{ij} - M \cdot (1 - v_{ij} + u_{ij}), \quad \forall (i,j) \in A \tag{6-44}$$

Model 6.2 (continued)

$$s_o = 0 \tag{6-45}$$

$$b_j^- \le Q + M \cdot (2 - v_{ij} - u_{ij}), \quad \forall (i,j) \in A \tag{6-46}$$

$$b_j^- \ge Q - M \cdot (2 - v_{ij} - u_{ij}), \quad \forall (i,j) \in A \tag{6-47}$$

$$b_j^+ \le Q + M \cdot (2 - v_{ij} - u_{ij}), \quad \forall (i,j) \in A \tag{6-48}$$

$$b_j^+ \ge Q - M \cdot (2 - v_{ij} - u_{ij}), \quad \forall (i,j) \in A \tag{6-49}$$

$$b_j^- \le b_i^+ - t_{ij}^D + M \cdot \left(1 - v_{ij} + u_{ij} + \sum_{k \ne i} u_{kj}\right), \quad \forall (i,j) \in A \tag{6-50}$$

$$b_j^- \ge b_i^+ - t_{ij}^D - M \cdot \left(1 - v_{ij} + u_{ij} + \sum_{k \ne i} u_{kj}\right), \quad \forall (i,j) \in A \tag{6-51}$$

$$b_j^+ \le b_j^- + M \cdot \left(1 - v_{ij} + u_{ij} + \sum_{k \ne i} u_{kj}\right), \quad (i,j) \in A \tag{6-52}$$

$$b_j^+ \ge b_j^- - M \cdot \left(1 - v_{ij} + u_{ij} + \sum_{k \ne i} u_{kj}\right), \quad (i,j) \in A \tag{6-53}$$

$$b_j^- \le b_j^+ - t_{ij}^D + M \cdot \left(1 - v_{ij} + u_{ij} + 1 - \sum_{k \ne i} u_{kj}\right), \quad (i,j) \in A \tag{6-54}$$

$$b_j^- \ge b_j^+ - t_{ij}^D - M \cdot \left(1 - v_{ij} + u_{ij} + 1 - \sum_{k \ne i} u_{kj}\right), \quad (i,j) \in A \tag{6-55}$$

$$b_j^+ \le Q + M \cdot \left(1 - v_{ij} + u_{ij} + 1 - \sum_{k \ne i} u_{kj}\right), \quad (i,j) \in A \tag{6-56}$$

$$b_j^+ \ge Q - M \cdot \left(1 - v_{ij} + u_{ij} + 1 - \sum_{k \ne i} u_{kj}\right), \quad (i,j) \in A \tag{6-57}$$

$$b_0^+ = Q \tag{6-58}$$

目标函数6-33追求最小化服务完所有顾客并到达终止节点的时间 s_e。

- 卡车相关的路径约束： 约束 6-34 限制每个顾客节点最多只能被卡车服务一次；约束 6-35 要求卡车在顾客节点满足流入和流出相等的流平衡；约束 6-36 和 6-37 分别要求卡车只能在起始节点流出一次，只能在终止节点流入一次。
- 无人机相关的路径约束： 约束 6-38 限制每个顾客节点最多只能被无人机服务一次；约束 6-39 要求无人机在顾客节点满足流入和流出相等的流平衡；约束 6-40 和 6-41 分别要求无人机只能在起始节点流出一次，只能在终止节点流入一次。
- 节点访问约束： 约束 6-42 要求每个节点都必须被卡车或无人机（或者两者一起）访问过一次。
- 同步相关的约束： 约束 6-43 和 6-44 计算了每个节点的离开时间；约束 6-45 规定了离开起始节点的时间。
- 无人机电量相关约束： 约束 6-58 表示无人机离开起始节点 o 时的电量为 Q。与无人机电量相关的约束根据无人机的状态不同可以分为三种类型，如图6-1所示。

  a) 卡车搭载无人机：约束 equations (6-46) to (6-49) 表示当无人机和卡车同时访问节点 j 时，

即卡车搭载无人机进行服务，无人机在到达节点 j 和离开节点 j 时的状态都是充满电的状态（$b_j^- = b_j^+ = Q$），即卡车搭载无人机访问时无人机不消耗电量。

b) 无人机单独服务节点：约束 equations (6-50) to (6-53) 表示当无人机单独访问节点 j 时，无人机在到达节点 j 的电量应该是无人机在离开节点 i 的电量减去路径 (i,j) 所消耗的电量（$b_j^- = b_i^+ - T_{ij}^D$），当无人机离开节点 j 时的电量应该等于无人机到达节点 j 的电量（$b_j^+ = b_j^-$），因为假设无人机在降落和起飞过程中不消耗电量。

c) 无人机与卡车会合：约束 equations (6-54) to (6-57) 表示当无人机与卡车在节点 j 会合时，无人机到达节点 j 时的电量应该是 $b_j^- = b_i^+ - T_{ij}^D$，当无人机与卡车会合时，无人机会在卡车上通过替换电池来将电量充满，因此当无人机离开节点 j 时的电量应该是 $b_j^+ = Q$。

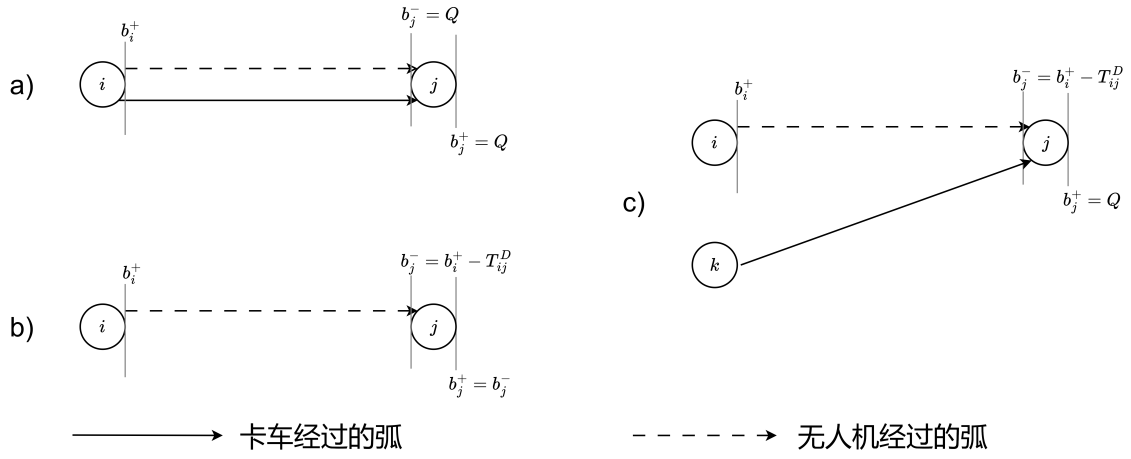


图 6-1: TDTL 模型无人机电量约束类型示意图

Liu 等 (2021)[10] 同样提出了类型为 FSTSP-MD 的 MILP 数学模型。在文章中作者称之为 Two-Echelon Routing Problem for the Truck and Drone (2E-RP-T&D)。但是不同于 Gonzalez-R 等 (2020)[9] 提出的 TDTL 模型，该问题考虑了不同重量的包裹对于无人机飞行消耗能源的影响。和 FSTSP 及 TDTL 一样，同样假设一辆卡车搭载一架无人机进行包裹递送，卡车的载重和续航没有进行限制，即卡车能够运送所有的包裹。该问题同样假设无人机和卡车在顾客点（或仓库）会合，但是该问题要求卡车必须先于无人机到达会合点（出于安全考虑）。2E-RP-T&D 的目标是最小化无人机和卡车的总成本。

Windras Mara 等 (2022)[11] 在 Gonzalez-R 等 (2020)[9] 的基础上提出了 multi-drop FSTS，但是不同于 Gonzalez-R 等 (2020)[9] 的假设，该模型考虑了起飞和降落的服务时间。

multi-drop FSTSP 数学模型的符号含义如表6-5所示。

表 6-5: multi-drop FSTSP 数学模型符号及含义

符号	含义
n	顾客的数量
k	操作的索引
$G=(V,A)$	定义了要访问的节点集合 V 和连接它们的弧集合 A 的图
V	图 G 的节点集合,包括一个仓库节点 $\{0\}$ 和 n 个客户节点 $N=\{1,2,\ldots,n\}$
A	图 G 中的弧集合，连接节点对 (i,j)，其中 $i,j\in V$
$Vd\subseteq V$	可以由无人机服务的节点子集
N	客户节点集合，即 $N=\{1,2,\ldots,n\}$
$K=[OP_1,\cdots,OP_k]$	所有可行操作的集合
C	卡车路径的非负节点数组
D	无人机路径的非负节点数组
$S\subseteq V$	节点的子集
$OP_k=\{\mathscr{T}_k,\mathscr{D}_k\}$	第 k 次递送的顺序，也被称为 operation，其中 $\mathscr{T}_k$ 是卡车在 OP_k 时必须访问的节点序列，$\mathscr{D}_k$ 是无人机在 OP_k 时必须访问的节点序列
$K(v)$	包含了节点 v 的所有可行操作集合
$K^-(v)$	包含了以节点 v 为起始点或发射点的所有可行操作集合
$K^+(v)$	包含了以节点 v 为终止点或会合点的所有可行操作集合
$K^{\mathscr{T}}(v)$	包含了以卡车访问节点 v 的所有可行操作集合
$K^{\mathscr{D}}(v)$	包含了以无人机访问节点 v 的所有可行操作集合
c_n	顾客的需求
$d^{\mathscr{T}}_{i\to j}$	卡车从节点 i 行驶到 j 的距离
$d^{\mathscr{D}}_{i\to j}$	无人机从节点 i 行驶到 j 的距离
$\tau^{\mathscr{T}}_{i\to j}$	弧 $(i,j)\in A$ 上卡车的行驶时间
$\tau^{\mathscr{D}}_{i\to j}$	弧 $(i,j)\in A$ 上无人机的行驶时间
s^L	无人机的启动时间
s^R	无人机的回收时间
ϵ	无人机的续航时间
$w^{\mathscr{T}}_r$	卡车在节点 j 的等待时间
$w^{\mathscr{D}}_r$	无人机在节点 j 的等待时间
$t(\mathscr{T}_k)$	卡车在 OP_k 中完成其对应任务所需的时间，计算公式为6-60
$t(\mathscr{D}_k)$	无人机在 OP_k 中完成其对应任务所需的时间，计算公式为6-61
$t_k=t(OP_k)$	OP_k 的完成时间 (OP_k 的成本函数)，计算公式为6-59
x_k	二元决策变量，如果选择操作 k ，则等于 1，否则等于 0
Y_v	二元辅助决策变量，如果节点 v 被选为至少一个操作的起始节点，则等于 1，否则等于 0

Definition 6.1: time for operations

$$t(OP_k) = \begin{cases} \max\{t(\mathscr{T}_k), t(\mathscr{D}_k)\}, & \text{当}OP_k\text{包含至少一个无人机访问的节点时} \\ t(\mathscr{T}_k), & \text{其他} \end{cases} \tag{6-59}$$

$$t(\mathscr{T}_k) = \begin{cases} s^L + \tau^{\mathscr{T}}_{i\to C_1} + \sum_{i=1}^{|C|-1} \tau^{\mathscr{T}}_{C_i\to C_{i+1}} + \tau^{\mathscr{T}}_{C_{|C|}\to j} + s^R, & \text{当}D \neq \varnothing\text{且}|C| > 1 \\ s^L + \tau^{\mathscr{T}}_{i\to C_1} + \tau^{\mathscr{T}}_{C_1\to j} + s^R, & \text{当}D \neq \varnothing\text{且}|C| = 1 \\ \tau^{\mathscr{T}}_{i\to j}, & \text{其他} \end{cases} \tag{6-60}$$

$$t(\mathscr{D}_k) = \begin{cases} s^L + \tau^{\mathscr{D}}_{i\to D_1} + \sum_{i=1}^{|D|-1} \tau^{\mathscr{D}}_{D_i\to D_{i+1}} + \tau^{\mathscr{D}}_{D_{|D|}\to j} + s^R, & \text{当}|D| > 1 \\ s^L + \tau^{\mathscr{D}}_{i\to D_1} + \tau^{\mathscr{D}}_{D_1\to j} + s^R, & \text{其他} \end{cases} \tag{6-61}$$

multi-drop FSTSP 数学模型可以表示为 MILP 6.3。

Model 6.3: multi-drop FSTSP MILP

$$\min \quad M = \sum_{k\in K} t_k x_k \tag{6-62}$$

$$\text{s.t.} \quad \sum_{k\in K(v)} x_k \geq 1, \quad \forall v \in V \setminus \{0\} \tag{6-63}$$

$$\sum_{k\in K^{\mathscr{D}}(v)} x_k \leq 1, \quad \forall v \in Vd \setminus \{0\} \tag{6-64}$$

$$\sum_{k\in K^{\mathscr{T}}(v)} x_k \leq 2\left(1 - \sum_{k\in K^{\mathscr{D}}(v)} x_k\right), \quad \forall v \in Vd \setminus \{0\} \tag{6-65}$$

$$\sum_{k\in K^{+}(v)} x_k \leq nY_v, \quad \forall v \in V \tag{6-66}$$

$$\sum_{k\in K^{+}(v)} x_k = \sum_{k\in K^{-}(v)} x_k, \quad \forall v \in V \tag{6-67}$$

$$\sum_{k\in K^{+}(S)} x_k \geq Y_v, \quad \forall S \subset V \setminus \{0\}, v \in S \tag{6-68}$$

$$\sum_{k\in K^{+}(0)} x_k \geq 1 \tag{6-69}$$

$$\tau^{\mathscr{D}}_{i\to D_1} + \sum_{i=1}^{|D|-1} \tau^{\mathscr{D}}_{D_i\to D_{i+1}} + \tau^{\mathscr{D}}_{|D|\to j} + w^{\mathscr{D}}_r \leq \epsilon \tag{6-70}$$

$$Y_0 = 1 \tag{6-71}$$

$$x_k \in \{0,1\}, \quad \forall k \in K \tag{6-72}$$

$$Y_v \in \{0,1\}, \quad \forall v \in V \tag{6-73}$$

目标函数6-62旨在最小化总路径的完成时间。

- 顾客相关的约束：约束 6-63 确保所有的节点都被至少访问过一次；约束 6-64 确保所有可以被无人机服务的顾客点最多被无人机服务一次；约束 6-65 确保被无人机服务过的节点不会被卡车服务，反之亦然，系数 2 是由于当无人机没有服务过节点 v 时，节点 v 可以作为卡车某个操作的终点及另一个操作的起点（即在节点 v 最多在两个操作中存在，一次作为操作的起点，一次作为操作的终点）；约束 6-66 表示如果节点 v 是某个操作的终止节点，那么节点 v 一定要是某

个操作的起始节点。

- Hamilton Graph约束：约束 6-67 确保以节点 v 为终止点的操作接下来的操作会以节点 v 为起始点（流量平衡约束）；约束 6-68 确保任何非空的客户子集 S 必须至少有一个操作从外部进入，即如果节点 v 被选为某个操作的起始节点 ($Y_v = 1$)，则必须存在至少一个操作进入包含 v 的子集 S，避免子回路的产生（消除子回路约束）；约束 6-69 确保终止点为仓库的操作只有一次，这是因为如果终止点为仓库的操作大于 1 次，由于约束 6-65 限制了卡车和无人机服务节点的次数，并且约束 6-63 确保每个节点都被访问过至少一次，而目标函数是最小化时间成本，如果多次以仓库为终止点的操作存在，则会增加成本，因此终止节点为仓库的操作大于 1 次的情况会被过滤掉。
- 无人机相关的约束：约束 6-70 要求无人机在一次 sortie 过程中消耗的电量不能超过其续航限制。
- 决策变量和辅助决策变量及初始值：约束6-71通过确保仓库被作为起点访问使得仓库作为路径的起始点；约束6-72和6-73定义了决策变量 x_k 和辅助决策变量 Y_v 的取值范围。

Windras Mara 等 (2022)[11] 用 Adaptive Large Neighborhood Search (ALNS) 来解决该问题，文章中使用到的 ALNS 算法符号及含义如6-7所示。

Definition 6.2: acceptance criterion

$$\Pr = \begin{cases} \exp\left(\frac{-(M'-M)}{BZxT_c}\right), & \text{for } M' - M > 0 \\ 1, & \text{for } M' - M \leq 0 \end{cases} \tag{6-74}$$

$$\Pr(c^-) = \frac{\omega_c^-}{\sum_{q \in Q} \omega_q^-}, \quad \forall c \in Q \tag{6-75}$$

$$\Pr(c^+) = \frac{\omega_c^+}{\sum_{r \in R} \omega_r^+}, \quad \forall c \in R \tag{6-76}$$

$$\omega_{i+1}^- = \lambda(\omega_i^-) + (1-\lambda)(\eta_i) \tag{6-77}$$

$$\eta_i = \begin{cases} \Omega_1, & \text{if } M_i \leq M^* \\ \Omega_2, & \text{if } M_i \leq M_{i-1} \\ \Omega_3, & \text{if } M_i > M_{i-1} \wedge U(0,1) < \Pr \\ \Omega_4, & \text{if } M_i > M_{i-1} \end{cases} \tag{6-78}$$

$$f_c = f_1 - (f_1 - f_0)\left(\frac{i_c}{I}\right), \quad \forall c \in \{1, \cdots, I\} \tag{6-79}$$

$$ND_c = n(f_c), \quad \forall c \in \{1, \cdots, I\} \tag{6-80}$$

(6-81)

6.2 Parallel Drone Scheduling Traveling Salesman Problem

Parallel Drone Scheduling Traveling Salesman Problem (PDSTSP) 同样由 Murray 等 (2015)[8] 提出。PDSTSP 适用于大量的顾客节点在无人机直接从仓库起飞的续航里程范围内。PDSTSP 描述：一辆卡车和一群（单个或者多个都可以）完全相同的无人机从单个仓库出发分别服务顾客，每个顾客只能被服务一次，卡车遵循 TSP 路径服务，无人机直接从仓库起飞服务顾客，不同于 FSTSP，PDSTSP 中的无人机不需

表 6-7: ALNS 算法使用到的符号及含义

符号	含义
s^*	最佳可行解
s'	新的可行解
M	目标函数值
M'	新的目标函数值
M^*	最优目标函数值
Q	destroy operators 集合
R	repair operators 集合
ω^-	destroy operatpr 权重
ω^+	repair operator 权重
η_I	destroy/repair operators 在当前迭代 I 中的表现
Pr	选择的概率，计算公式为6-74
T_c	当前的温度
T_1	初始的温度
T_0	最终的温度
α	冷却系数
BZ	Boltzmann 常数
$\Omega_1, \Omega_2, \Omega_3$	权重调整参数
λ	衰减参数
f_1	初始的 destroy ratio
f_0	最终的 destroy ratio
I_T	在某一温度下的迭代次数
P	概率
f_c	递减线性函数
ND_c	被 destroyed 的顾客节点数量，由6-79和6-80决定
AR_s	Archive of solutions for archive repair
RD	Destroy operator - Random removal
WTD	Destroy operator - Worst truck distance removal
WDD	Destroy operator - Worst drone distance removal
BD	Destroy operator - Block destroy removal
RR	Repair operator - Random repair
AR	Repair operator - Archive repair
LDR-1	Repair operator - Least distance repair $d_{i-1\to i}$
LDR-2	Repair operator - Least distance repair $d_{i\to i+1}$
HR	Repair operator - Hill climbing

要和卡车进行会合。PDSTSP 的目标是最小化最终到达仓库的卡车（无人机）的时间。

PDSTSP 数学模型的符号含义如表6-9，基本上沿用了 FSTSP 的符号。

表 6-9: PDSTSP 模型符号及含义

符号	含义
0	起点仓库
$c+1$	终点仓库
$C=\{1,2,\cdots,c\}$	全部客户集合
$C' \subseteq C$	可以接受无人机访问的客户集合[a]
$C'' \subseteq C'$	在无人机的航行范围内可以接受无人机服务的顾客集合[b]
$N_0=\{0,1,2,\cdots,c\}$	流出节点集合
$N_+=\{1,2,\cdots,c+1\}$	流入节点集合
$N=\{0,1,2,\cdots,c,c+1\}$	全部节点集合
$v \in V$	无人机集合
$\tau'_{i,j}/\tau_{i,j}$	弧 $\langle i,j\rangle$ 的飞行/行驶时间成本
$1 \le \hat{u}_i \le c+2$	卡车破子圈辅助变量
$\hat{y}_{i,v} \in \{0,1\}, i \in C'', v \in V$	无人机访问决策变量
$\hat{x}_{i,j} \in \{0,1\}, i \in N_0, j \in \{N_+ : j \neq i\}$	卡车路由决策变量

[a] 指包裹重量没有超过无人机的载重限制，不需要顾客签收，顾客的位置允许无人机起降等限制。
[b] 当 $\tau'_{0,i}+\tau'_{i,c+1} \le e$ 时，顾客 $i \in C'$ 属于集合 C''。

PDSTSP 数学模型可以表示为 MILP 6.4。

Model 6.4: PDSTSP MILP

$$\min \quad z \tag{6-82}$$

$$\text{s.t.} \quad z \ge \sum_{i\in N_0}\sum_{\substack{j\in N_+\\ j\neq i}} \tau_{i,j}\hat{x}_{i,j} \tag{6-83}$$

$$z \ge \sum_{i\in C''} (\tau'_{0,i}+\tau'_{i,c+1})\hat{y}_{i,v}, \quad \forall v \in V \tag{6-84}$$

$$\sum_{\substack{i\in N_0\\ i\neq j}} \hat{x}_{i,j} + \sum_{\substack{v\in V\\ j\in C''}} \hat{y}_{j,v} = 1, \quad \forall j \in C \tag{6-85}$$

$$\sum_{j\in N_+} \hat{x}_{0,j} = 1 \tag{6-86}$$

$$\sum_{i\in N_0} \hat{x}_{i,c+1} = 1 \tag{6-87}$$

$$\sum_{\substack{i\in N_0\\ i\neq j}} \hat{x}_{i,j} = \sum_{\substack{k\in N_+\\ k\neq j}} \hat{x}_{j,k}, \quad \forall j \in C \tag{6-88}$$

$$\hat{u}_i - \hat{u}_j + 1 \le (c+2)(1-\hat{x}_{i,j}), \quad \forall i \in C, j \in \{N_+ : j \neq i\} \tag{6-89}$$

$$1 \le \hat{u}_i \le c+2, \quad \forall i \in N_+ \tag{6-90}$$

$$\hat{x}_{i,j} \in \{0,1\}, \quad \forall i \in N_0, j \in \{N_+ : j \neq i\} \tag{6-91}$$

Model 6.4 (continued)

$$\hat{y}_{i,v} \in \{0,1\}, \quad \forall i \in C'', v \in V \tag{6-92}$$

目标函数6-82追求最小化完工时间 z，即无人机和卡车最晚到达终点仓库的时间，通过约束6-83和6-84分别限制卡车和无人机最晚到达终点仓库的时间来实现；约束6-85确保了每个顾客能且只能被服务一次，服务可以由无人机或者卡车提供；约束6-86和6-87要求卡车必须从起点仓库 0 出发并返回终点仓库 $c+1$ 一次，约束6-88要求卡车在中间的顾客节点满足流入和流出相等的流约束；约束6-89是 MTZ 形式的破子圈约束；约束6-90，6-91和6-92给出了决策变量和辅助决策变量的取值范围。

Ham(2018)[12] 在 PDSTSP 的基础上提出了 multi-truck, multi-drone, and multi-depot scheduling problem constrained by time-window, drop-pickup synchronization, and multi-visit, with the objective to minimize the maximum completion time over all tasks (PDSTSP^{+DP})，不同于 PDSTSP，该问题考虑了无人机递送和收回货物的情况，即当无人机递送货物到顾客返回时，可以飞行到另一个需要退货的顾客节点进行回收货物以使无人机满载，从而满足退货的需求。另外，该问题还考虑了服务顾客的时间窗问题，即顾客需求有时间限制，并且可以在不同的时间段为同一个顾客服务。

7

Traveling Salesman Problem with multiple Drones

8

multiple Traveling Salesman Problem with Drones

Ventresca 等 (2019)[13] 提出了 multiple Traveling Salesman Problem with Drones (mTSPD) 的 mixed integer programming (MIP) 公式，目标函数是最小化所有卡车和无人机返回仓库的最终时间。该文章没有考虑无人机的降落和起飞等服务时间，一个创新点在于无人机降落时不需要降落在原来搭载该无人机的卡车上，而是可以选择降落在附近的其他卡车上。

9

Vehicle Routing Problem with Drones

Wang 等 (2019)[14] 提出了 Vehicle Routing Problem with Drones (VRPD)，目标是最小化无人机和卡车的总物流成本，在这个问题中，每辆卡车可以搭载 N 架无人机，因此是一个 $M:N$ 的多对多 VRP。在这个问题中，无人机能够和卡车一起行驶，当卡车停在服务中心（service hub）时，如果顾客货物需求和距离都在无人机的载重和续航范围内，则无人机可以从服务中心（或仓库）起飞进行服务，文章中限制了无人机不能在顾客节点进行降落，而是采用类似空投的方式进行服务（parachute airdrop），但是无人机可以和不同的卡车会合一起行驶。针对 VRPD，作者提出了 arc-based mixed integer programming model，并且采用 branch-and-price 算法进行解决。

Part

VI

TSDD: Truck Support Drone Delivery

Liu 等 (2017)[15] 提出了 two echelon cooperated ground vehicle and its carried unmanned aerial vehicle routing problem (2E-GU-RP)。该问题主要研究如何合理安排无人机及其路线和车辆路线来最小化总服务时间，即无人机和车辆从仓库出发服务所有顾客后返回仓库的总时间，无人机可以在一次起飞降落中服务多个顾客节点，但是所有的顾客节点都无法被车辆直接进行服务。车辆在无人机进行服务顾客的过程中充当临时仓库和充电站的角色。

Part

VII

Datasets for Traveling Salesman Problem with Drone

10

Datasets used in FSTSP-MD

Traveling Salesman Problem with Drones (TSP-D) 是经典的 TSP 问题的拓展，它在 TSP 问题的基础上增加了无人机。无人机可以和车辆一起工作，或者自主起飞服务。根据无人机单次起飞降落过程中服务的顾客点数量的不同可以将问题分为单次起飞服务单个顾客点的和单次起飞服务多个顾客点。同样对无人机和车辆的会合点也有限制，即无人机只能在顾客节点或者仓库节点会合，因此会产生无人机和车辆之间互相等待的时间。接下来的部分将会介绍一些用于 TSP-D 问题的数据集。

10.1 TSP-D Instances by Bouman et al. (2018)

TSP-D-Instances仓库包含了用于 TSP-D 问题的二维数据集，即只有仓库和顾客节点的横纵坐标。在数据集中以符号 `/*` 开始，以符号 `*/`结束的行是注释行，在读取数据时需要忽略。该数据集包含了 Agatz 等 (2018)[16] 和 Bouman 等 (2018)[17] 所用的数据集。相关的解决代码（`Java` 实现）可以在Drones-TSP仓库中找到。在这个数据集中，两点之间的距离是欧几里得距离（Euclidean distance），即两点之间的距离是两点之间的直线距离。有关数据集字段说明可以参考数据集的注释和数据集仓库的说明。该数据集分类如下（在所有的情况中，生成的第一个节点位置被选作仓库节点）：

- `uniform`：每个节点的 x 和 y 坐标都是从取值范围为 $\{0, 1, \ldots, 100\}$ 的独立均匀分布中随机生成的。
- `singlecenter`：对于每个位置，首先均匀地从区间 $[0, 2\pi]$ 中抽取一个角度 α，然后从一个均值为 0，标准差为 50 的正态分布中抽取一个距离 r，坐标 $(x, y) = (r \cdot \cos\alpha, r \cdot \sin\alpha)$，用这种方法生成的节点位置更有可能集中在中心点 $(0, 0)$ 附近，比 `uniform` 的数据集更能模拟圆形城市中心的情况。
- `doublecenter`：生成方式和 `singlecenter` 类似，但在生成每个位置后，有 50% 的概率将其沿 x 轴平移 200 个距离单位，这种方法生成的节点位置更有可能集中在两个中心点 $(0, 0)$ 和 $(200, 0)$ 附近，模拟了一个具有两个中心的城市的情况。
- `restricted`：在原有限制的基础上增加了一些额外的限制。
  - `maxradius`：增加了无人机不能飞行超过一定半径的限制。
  - `novisit`：增加了无人机不能访问的顾客节点，比如以`-novisit-20-rep_2.txt` 为后缀的文件表示有 20% 的顾客节点被随机选中用于表示无法被无人机访问的顾客节点，由于对于同一个数据来说，不同次数的随机生成会影响选中的顾客节点，因此 `rep_2` 表示第二次随机生成的数据。具体的不能被无人机访问的顾客节点由字段 `#NOVISIT` 表示，例如数据文件中的 `#NOVISIT 1` 表示第一个顾客节点不能被无人机访问，也即生成的节点数据中的第二行数据（因为默认生成的第一行数据是仓库节点）。

10.2 TSPDroneLIB by Bogyrbayeva et al. (2023)

TSPDroneLIB仓库包含了用于 TSP-D 问题和 FSTSP 问题的数据集和相关的链接。该仓库提到了10.1的数据集，另外包括了 Bogyrbayeva 等 (2023)[18] 使用的数据集。相关的算法可以在TSPDrone.jl仓库中找到。有关数据集的字段说明可以在TSPDroneLIB/data/Bogyrbayeva/description.md中找到。该数据集分类如下：

- **Random**：包含了三个不同节点数量大小的数据集，分别为 $n = 20, 50, 100$，每个数据集包含了 100 个算例，每行表示包含横纵坐标的一个算例，遵循格式 $x_1, y_1, x_2, y_2, \ldots, x_n, y_n$，即每行的每两个数一组组成一个节点的横纵坐标，同时 x_1, y_1 表示仓库节点的横纵坐标。该数据集的生成方法为：在 $[1100] \times [1100]$ 的范围内，从均匀分布中随机抽取每个节点的横纵坐标，唯一的例外是仓库节点，其位置分布在 $[0, 1] \times [0, 1]$ 范围内，这意味着仓库总是位于角落。这种数据集的生成方法和 Agatz 等 (2018)[16] 使用的数据集生成方法是一致的。
- **Amsterdam**：该数据集的数据格式和 **Random** 数据集相同，共有四种不同大小的算例，分别为 $n = 10, 20, 50, 100$。该数据集基于 Haider 等 (2019)[19] 研究中使用的电动汽车 (electric vehicle, EV) 停车位置数据集。这些位置反映了潜在顾客的位置，因为电动汽车通常停放在城市街道的路边充电器旁。为了适应 Bogyrbayeva 等 (2023)[18] 的研究，该文章从整个 **Amsterdam** 数据集中随机选取仓库和客户节点来创建不同的问题算例。

10.2.1 DPS algorithm and DRL algorithm implemented in TSPDrone.jl

TSPDrone.jl仓库如前所述，解决了单车辆配备单架无人机的 TSP-D 问题，实现了 Bogyrbayeva 等 (2023)[18] 提出的 Divide-Partition-and-Search (DPS) 算法和 Deep Reinforcement Learning (DRL) 算法，其中 DPS 算法是基于 Agatz 等 (2018)[16] 的 TSP-ep-all 算法和 Poikonen 等 (2019)[20] 的 divide-and-conquer 启发式算法开发的。

DPS 算法是 Bogyrbayeva 等 (2023)[18] 提出的一种基于分治策略的启发式算法，用于解决 TSP-D 问题。其核心思想是将大规模问题分解为较小的子问题，并通过组合子问题的解来获得全局解。DPS 算法的主要步骤如下：

1. Divide：将全部节点划分为多个子组，每个子组包含固定数量（由参数 g 控制）的节点。例如，$g = 10$ 表示每个子组有 10 个节点。
2. Partition & Search：在每个子组内，采用 TSP-ep-all 算法[16] 进行分区。TSP-ep-all 算法通过如下步骤优化子组内的路径：
   (a) 生成初始 TSP 路径：使用 Concorde TSP Solver 得到卡车单独服务的初始路径。
   (b) 动态规划分区：将路径中的节点划分为由卡车和无人机协同服务的子集，以最小化总时间。
   (c) 局部搜索优化：进一步调整分区以提升解的质量。
3. 合并与全局优化：将所有子组的解合并为完整的路径，并进行全局优化（如调整子组边界或路径顺序）以消除局部最优的局限性。

在 DPS 算法中，较大的子组 g（如 $g = 25$）会提升解的质量，但增加计算时间；较小的子组 g（如 $g = 10$）则相反。当 $g = N$（总节点数）时，DPS 退化为直接应用 TSP-ep-all 算法，不再划分子组。

TSPDrone.jl 用**Julia**[1]实现，初始 TSP 路径由 Concorde TSP Solver 生成，分区过程基于动态规划和局部搜索，使用该仓库 DPS 算法的步骤如下：

[1]学习 **Julia** 的课程参考 MIT 的Introduction to Computational Thinking。

1. 安装Julia，在命令行中输入`julia`进入 **Julia** 环境，输入命令安装必要的依赖：

```
] add https://github.com/chkwon/TSPDrone.jl
```

2. 使用 DPS 算法需要提供顾客节点的 x 和 y 坐标，仓库的 (x, y) 坐标需要是第一个元素，参数 **truck_cost_factor** 和参数 **drone_cost_factor** 分别代表卡车和无人机的成本因子，会乘以从横纵坐标中计算出来的欧氏距离来得到卡车和无人机的行驶成本。

Code 10.1: DPS algorithm usage example (a)

```julia
using TSPDrone
n = 10
x = rand(n); y = rand(n);
truck_cost_factor = 1.0
drone_cost_factor = 0.5
result = solve_tspd(x, y, truck_cost_factor, drone_cost_factor)
@show result.total_cost;
@show result.truck_route;
@show result.drone_route;
```

如果正常运行，会输出如下结果（根据随机数生成的结果可能会有所不同），其中节点 11 作为终止节点表示仓库节点（即终止节点的代号会在总的节点数量上 +1）：

```
result.total_cost = 1.6022013835206805
result.truck_route = [1, 4, 5, 2, 8, 6, 11]
result.drone_route = [1, 9, 4, 10, 5, 7, 8, 3, 11]
```

3. 或者也可以直接提供卡车和无人机的成本矩阵（即原本根据欧氏距离矩阵乘以成本因子得到的矩阵），同样，仓库节点被标记为节点 1：

Code 10.2: DPS algorithm usage example (b)

```julia
using TSPDrone
n = 10
dist_mtx = rand(n, n)
dist_mtx = dist_mtx + dist_mtx' # symmetric distance only
truck_cost_mtx = dist_mtx .* 1.0
drone_cost_mtx = truck_cost_mtx .* 0.5
result = solve_tspd(truck_cost_mtx, drone_cost_mtx)
@assert size(truck_cost_mtx) == size(drone_cost_mtx) == (n, n)
```

4. 使用命令`print_summary(result)`可以输出结果总结：

```
julia> print_summary(result)
Operation #1:
  - Truck        = 0.17988883875173492 : [1, 3]
  - Drone        = 0.11900891950265155 : [1, 4, 3]
  - Length       = 0.17988883875173492
Operation #2:
  - Truck        = 0.4784476248243221 : [3, 9]
  - Drone        = 0.27587675362585756 : [3, 7, 9]
  - Length       = 0.4784476248243221
Operation #3:
  - Truck        = 0.445749847855226 : [9, 6]
  - Drone        = 0.48831605249544785 : [9, 10, 6]
  - Length       = 0.48831605249544785
Operation #4:
  - Truck        = 0.9269158918021541 : [6, 5, 8, 11]
  - Drone        = 0.8714473929102112 : [6, 2, 11]
```

```
  - Length       = 0.9269158918021541
Total Cost = 2.073568407873659
```

5. 函数`solve_tspd`的可选参数包括：

Code 10.3: DPS algorithm optional keyword arguments

```
n_groups::Int = 1,
method::String = "TSP-ep-all",
flying_range::Float64 = MAX_DRONE_RANGE,
time_limit::Float64 = MAX_TIME_LIMIT
```

- `n_groups`：用于分治法的子组数量。例如，如果 $n = 100$ 且 `n_groups = 4`，则每组将有 25 个节点，然后将方法 `method` 应用于每个组。
- `method`：可以是以下的几种方法之一，`TSP-ep` 及其衍生方法（`TSP-ep-1p`、`TSP-ep-2p`、`TSP-ep-2opt` 和 `TSP-ep-all`）是基于 route-first, cluster-second 框架的启发式算法，由 Agatz 等 (2018)[16] 提出，用于解决 TSP-D：
  - `TSP-ep` (Exact Partition)：使用 TSP 求解器（如 Concorde）生成最优 TSP 路径，然后以初始 TSP 路径为基础，通过精确划分算法（动态规划，时间复杂度为 $O(n^3)$）将 TSP 路径分割为卡车和无人机的协同路径。
  - `TSP-ep-1p`：在 `TSP-ep` 的基础上，引入单点移动领域搜索 (One-Point Move)，通过调整单个节点的位置优化路径。即先对初始路径进行 Exact Partition，然后遍历每个节点，尝试将其移动到路径中的其他位置，计算目标函数改进，然后接受最大的移动，迭代直至无法进一步优化。
  - `TSP-ep-2p`：在 `TSP-ep` 的基础上，引入两点交换领域搜索 (Two-Point Swap)，通过交换两个节点的位置优化路径。即先对初始路径进行 Exact Partition，然后遍历所有节点对，尝试交换两者的位置，计算目标函数改进，然后接受最大的交换，迭代直至无法进一步优化。
  - `TSP-ep-2opt`：在 `TSP-ep` 的基础上，引入 2-opt 领域搜索，通过反转路径中的子段优化路径。即先对初始路径进行 Exact Partition，然后遍历所有可能的路径子段，尝试反转子段并重新计算总时间，接受改进最大的反转操作，迭代直至无法进一步优化。
  - `TSP-ep-all`：在 `TSP-ep` 的基础上，综合应用所有领域搜索策略（`1p`、`2p`、`2opt`），通过多策略组合优化路径。即先对初始路径进行 Exact Partition，然后在每轮迭代中，尝试所有领域操作（One-Point Move, Two-Point Swap, 2-opt），选择改进最大的操作，迭代直至无法进一步优化。（表现最佳，但运行时间较长 $O(n^5)$）
- `flying_range`：无人机的飞行范围，默认值为 `Inf`。飞行范围与无人机成本矩阵中的值进行比较，即 `drone_cost_mtx` 或欧氏距离乘以 `drone_cost_factor`。
- `time_limit`：算法运行的总时间限制，以秒为单位。对于每个组，时间限制平均分配。例如，如果 `time_limit = 3600.0` 且 `n_groups = 5`，则每组的时间限制为 $3600/5 = 720$ 秒。

10.3 TSPLIB by Reinelt (1991)

TSPLIB是一个用于和 TSP 问题相关的数据集，包含了 Symmetric Traveling Salesman Problem (STSP)、Asymmetric Traveling Salesman Problem (ATSP)、Hamiltonian Cycle Problem (HCP)、Sequential Ordering

Problem (SOP)、Capacitated Vehicle Routing Problem (CVRP)的数据集，可以在tsp95.pdf中查看有关数据集的完整说明文档。除了 HCP 以外，其他的问题都是定义在完全图上，且所有的距离都是以整数表示的。每个文件都包括说明部分和数据部分，说明部分包含了有关文件的格式和内容的信息。

部分 STSP 数据集如表10-1所示。

表 10-1: Symmetric Traveling Salesman Problem Examples

数据名称	城市数量	距离计算方式	最优值
a280	280	EUC_2D	2579
berlin52	52	EUC_2D	7542
bier127	127	EUC_2D	118282
ch130	130	EUC_2D	6110
ch150	150	EUC_2D	6528
d198	198	EUC_2D	15780
d493	493	EUC_2D	35002
d657	657	EUC_2D	48912
eil51	51	EUC_2D	426
eil76	76	EUC_2D	538
eil101	101	EUC_2D	629
fl417	417	EUC_2D	11861
gil262	262	EUC_2D	2378
kroA100	100	EUC_2D	21282
kroB100	100	EUC_2D	22141
kroC100	100	EUC_2D	20749
kroD100	100	EUC_2D	21294
kroE100	100	EUC_2D	22068
kroA150	150	EUC_2D	26524
kroB150	150	EUC_2D	26130
kroA200	200	EUC_2D	29368
kroB200	200	EUC_2D	29437
lin105	105	EUC_2D	14379
lin318	318	EUC_2D	42029
linhp318	318	EUC_2D	41345
p654	654	EUC_2D	34643
pcb442	442	EUC_2D	50778
pr76	76	EUC_2D	108159
pr107	107	EUC_2D	44303
pr124	124	EUC_2D	59030
pr136	136	EUC_2D	96772
pr144	144	EUC_2D	58537
pr152	152	EUC_2D	73682
pr226	226	EUC_2D	80369
pr264	264	EUC_2D	49135
pr299	299	EUC_2D	48191
pr439	439	EUC_2D	107217
rat99	99	EUC_2D	1211

表 10-1(续)

数据名称	城市数量	距离计算方式	最优值
rat195	195	EUC_2D	2323
rat575	575	EUC_2D	6773
rat783	783	EUC_2D	8806
rd100	100	EUC_2D	7910
rd400	400	EUC_2D	15281
st70	70	EUC_2D	675
ts225	225	EUC_2D	126643
tsp225	225	EUC_2D	3919
u159	159	EUC_2D	42080
u574	574	EUC_2D	36905
u724	724	EUC_2D	41910

10.4 Amazon Delivery Dataset by Kaggle

Amazon Delivery Dataset是一个 Amazon 公司最后一公里物流运营情况的数据集，包含了超过 43632 次配送的多城市数据，数据字段包括订单详情、配送人员、天气、交通情况、配送仓库和配送地点的经纬度等信息。要将数据集转换为可以用于 TSP-D 问题的数据集，需要将数据集中的经纬度转换为欧几里得距离，即两点之间的直线距离，当然在此之前需要对原始数据集进行一些数据的预处理工作，详细的代码在附录A.1中。

Part

VIII

Appendix

A

Source Code

A.1 Amazon delivery data preprocessing code

关于已知两点经纬度计算两点之间距离的方法，这里使用了Haversine formula[1]，但是要注意这个公式只是一个近似值，即假设地球是一个球体，而实际上地球是一个椭球体，不过对于不是精确到亚米级别的应用来说，这个公式的精度是足够的，误差在 0.5% 以内。如果需要更精确的方法可以参考Vincenty's formulae和Geographical distance。根据经纬度判断这个点是否在陆地的方法可以参考 Python 的库global-land-mask。

Code A.1: Amazon delivery data preprocessing code to filter data

```python
import pandas as pd
from math import radians, sin, cos, sqrt, atan2
from global_land_mask import globe

# Haversine 公式计算距离
def haversine(lat1, lon1, lat2, lon2):
    R = 6371.393 # 地球半径近似值
    lat1_rad, lon1_rad = radians(lat1), radians(lon1)
    lat2_rad, lon2_rad = radians(lat2), radians(lon2)
    dlon = lon2_rad - lon1_rad
    dlat = lat2_rad - lat1_rad
    a = sin(dlat/2)**2 + cos(lat1_rad)*cos(lat2_rad)*sin(dlon/2)**2
    return R * 2 * atan2(sqrt(a), sqrt(1-a))

# 读取数据
df = pd.read_csv('amazon_delivery.csv')

# 步骤 1: 筛选顾客节点 >=10 的仓库
valid_warehouses = df.groupby(['Store_Latitude', 'Store_Longitude']).filter(lambda x: len(x) >= 10)

# 步骤 2: 剔除顾客-仓库经纬度差 >=1 的订单
valid_warehouses = valid_warehouses[
    (abs(valid_warehouses['Store_Latitude'] - valid_warehouses['Drop_Latitude']) < 1) &
    (abs(valid_warehouses['Store_Longitude'] - valid_warehouses['Drop_Longitude']) < 1)
]

# 步骤 3: 计算距离并筛选 <=50 公里的订单
```

[1]在代码中使用的不是 arcsin 而是 arctan，这是因为当 sin 值接近 1 时，直接使用 arcsin 可能导致精度问题，而 arctan 通过显式分离分子分母，可以使得计算更加稳定。arcsin 和 arctan 之间的转换可以参考：实用反三角函数运算公式。

Code A.1 (continued)

```
valid_warehouses['Distance'] = valid_warehouses.apply(
    lambda row: haversine(row['Store_Latitude'], row['Store_Longitude'],
                          row['Drop_Latitude'], row['Drop_Longitude']),
    axis=1
)
valid_warehouses = valid_warehouses[valid_warehouses['Distance'] <= 50]

# 步骤 4: 去重同一仓库下的重复顾客
valid_warehouses = valid_warehouses.drop_duplicates(
    subset=['Store_Latitude', 'Store_Longitude', 'Drop_Latitude', 'Drop_Longitude']
)

# 步骤 5: 检查仓库是否在陆地
# 提取唯一仓库坐标
warehouse_coords = valid_warehouses[['Store_Latitude',
↪  'Store_Longitude']].drop_duplicates()

# 使用 global_land_mask 检查陆地
warehouse_coords['Is_Land'] = warehouse_coords.apply(
    lambda row: globe.is_land(row['Store_Latitude'], row['Store_Longitude']),
    axis=1
)

# 合并陆地标记到原始数据
valid_warehouses = valid_warehouses.merge(
    warehouse_coords[['Store_Latitude', 'Store_Longitude', 'Is_Land']],
    on=['Store_Latitude', 'Store_Longitude'],
    how='left'
)

# 步骤 6: 剔除位于海里的仓库数据
final_data = valid_warehouses[valid_warehouses['Is_Land']]

# 输出结果
final_data.to_excel('amazon_delivery_filtered_data.xlsx', index=False)
warehouse_stats = final_data.groupby(['Store_Latitude',
↪  'Store_Longitude']).size().reset_index(name='Count')
warehouse_stats.sort_values(by='Count',
↪  ascending=False).to_excel('warehouse_stats.xlsx', index=False)
```

首先删除不需要的数据列。接着通过将前面得到的数据导入到Google Maps中，可以看到仓库数据大致可以聚类成 22 个簇，因此聚类时设置聚类数量为 22。然后将同一聚类的仓库节点和配送节点合并到同一个 `Excel` 文件中，因此总共会生成 22 个不同聚类的 `Excel` 文件。

Code A.2: Amazon delivery data preprocessing code to sort data

```
import pandas as pd
import numpy as np
from sklearn.cluster import KMeans
import os

# 1. 读取 Excel 文件并提取需要的列
df = pd.read_excel("amazon_delivery_filtered_data.xlsx")
filtered_df = df[["Store_Latitude", "Store_Longitude", "Drop_Latitude",
↪  "Drop_Longitude"]]

# 2. 提取唯一的仓库坐标用于聚类
```

Code A.2 (continued)

```
store_coords = filtered_df[["Store_Latitude", "Store_Longitude"]].drop_duplicates()

# 3. 使用 KMeans 进行聚类（已知聚类数 =22）
kmeans = KMeans(n_clusters=22, random_state=42, n_init=10)
store_coords["Cluster"] = kmeans.fit_predict(store_coords[["Store_Latitude",
↪  "Store_Longitude"]])

# 4. 将聚类标签合并回原始数据
merged_df = filtered_df.merge(
    store_coords,
    how="left",
    on=["Store_Latitude", "Store_Longitude"]
)

# 5. 创建保存结果的文件夹
output_dir = "clustered_nodes"
os.makedirs(output_dir, exist_ok=True)

# 6. 按聚类分组处理数据
for cluster_id in range(22):
    # 提取当前聚类的数据
    cluster_data = merged_df[merged_df["Cluster"] == cluster_id]

    # 分离仓库节点和顾客节点
    store_nodes = cluster_data[["Store_Latitude",
    ↪  "Store_Longitude"]].drop_duplicates()
    customer_nodes = cluster_data[["Drop_Latitude",
    ↪  "Drop_Longitude"]].drop_duplicates()

    # 生成唯一 ID
    store_nodes["ID"] = ["store_" + str(i) for i in range(len(store_nodes))]
    customer_nodes["ID"] = ["customer_" + str(i) for i in range(len(customer_nodes))]

    # 重命名列以匹配
    store_nodes.rename(columns={"Store_Latitude": "latitude", "Store_Longitude":
    ↪  "longitude"}, inplace=True)
    customer_nodes.rename(columns={"Drop_Latitude": "latitude", "Drop_Longitude":
    ↪  "longitude"}, inplace=True)

    # 合并节点并整理列顺序
    combined_nodes = pd.concat([store_nodes, customer_nodes], ignore_index=True)
    combined_nodes = combined_nodes[["ID", "latitude", "longitude"]]

    # 输出每个聚类的节点数量
    total_nodes = len(store_nodes) + len(customer_nodes)
    print(f"聚类 {cluster_id} 有 {total_nodes} 个节点（仓库节点：{len(store_nodes)}，顾客
    ↪  节点：{len(customer_nodes)}）")

    # 保存到 Excel 文件
    output_path = os.path.join(output_dir, f"cluster_{cluster_id}.xlsx")
    combined_nodes.to_excel(output_path, index=False)
```

B

Appendix B

Epilogue

References

[1] CORMEN T H, LEISERSON C E, RIVEST R L, et al. Introduction to algorithms[M]. Third edition ed. Cambridge, Massachusetts London, England: MIT Press, 2009.

[2] OENCAN T, ALTINEL I K, LAPORTE G. A comparative analysis of several asymmetric traveling salesman problem formulations[J/OL]. Computers & Operations Research, 2009, 36(3): 637-654. DOI: 10.1016/j.cor.2007.11.008.

[3] ROBERTI R, TOTH P. Models and algorithms for the asymmetric traveling salesman problem: an experimental comparison[J/OL]. Euro Journal on Transportation & Logistics, 2012, 1(1-2): 113-133. DOI: 10.1007/s13676-012-0010-0.

[4] PAPADIMITRIOU C H, STEIGLITZ K. Combinatorial optimization: algorithms and complexity[M]. Dover edition ed. Mineola, NY: Dover Publications, 1998: 308-309.

[5] MILLER C E, TUCKER A W, ZEMLIN R A. Integer programming formulation of traveling salesman problems[J/OL]. Journal of the Acm, 1960, 7(4): 326-329. DOI: 10.1145/321043.321046.

[6] DANTZIG G B, RAMSER J H. The truck dispatching problem[J/OL]. Management Science, 1959, 6(1): 80-91. DOI: 10.1287/mnsc.6.1.80.

[7] TOTH P, VIGO D. Vehicle routing: Problems, methods, and applications, second edition[J/OL]. Society for Industrial and Applied Mathematics, 2014: 4-6. DOI: 10.1137/1.9781611973594.fm.

[8] MURRAY C C, CHU A G. The flying sidekick traveling salesman problem: Optimization of drone-assisted parcel delivery[J/OL]. Transportation Research Part C: Emerging Technologies, 2015, 54: 86-109. DOI: 10.1016/j.trc.2015.03.005.

[9] GONZALEZ-R P L, CANCA D, ANDRADE-PINEDA J L, et al. Truck-drone team logistics: A heuristic approach to multi-drop route planning[J/OL]. Transportation Research Part C: Emerging Technologies, 2020, 114: 657-680. DOI: 10.1016/j.trc.2020.02.030.

[10] LIU Y, LIU Z, SHI J, et al. Two-echelon routing problem for parcel delivery by cooperated truck and drone[J/OL]. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 2021, 51(12): 7450-7465. DOI: 10.1109/TSMC.2020.2968839.

[11] WINDRAS MARA S T, RIFAI A P, SOPHA B M. An adaptive large neighborhood search heuristic for the flying sidekick traveling salesman problem with multiple drops[J/OL]. Expert Systems with Applications, 2022, 205: 117647. DOI: 10.1016/j.eswa.2022.117647.

[12] HAM A M. Integrated scheduling of m-truck, m-drone, and m-depot constrained by time-window, drop-pickup, and m-visit using constraint programming[J/OL]. Transportation Research Part C: Emerging Technologies, 2018, 91: 1-14. DOI: 10.1016/j.trc.2018.03.025.

[13] KITJACHAROENCHAI P, VENTRESCA M, MOSHREF-JAVADI M, et al. Multiple traveling salesman problem with drones: Mathematical model and heuristic approach[J/OL]. Computers & Industrial Engineering, 2019, 129: 14-30. DOI: 10.1016/j.cie.2019.01.020.

[14] WANG Z, SHEU J B. Vehicle routing problem with drones[J/OL]. Transportation Research Part B: Methodological, 2019, 122: 350-364. DOI: 10.1016/j.trb.2019.03.005.

[15] LUO Z, LIU Z, SHI J. A two-echelon cooperated routing problem for a ground vehicle and its carried unmanned aerial vehicle[J/OL]. Sensors, 2017, 17(5): 1144. DOI: 10.3390/s17051144.

[16] AGATZ N, BOUMAN P, SCHMIDT M. Optimization approaches for the traveling salesman problem with drone[J/OL]. Transportation Science, 2018, 52(4): 965-981. DOI: 10.1287/trsc.2017.0791.

[17] BOUMAN P, AGATZ N, SCHMIDT M. Dynamic programming approaches for the traveling salesman problem with drone[J/OL]. Networks, 2018, 72(4): 528-542. DOI: 10.1002/net.21864.

[18] BOGYRBAYEVA A, YOON T, KO H, et al. A deep reinforcement learning approach for solving the traveling salesman problem with drone[J/OL]. Transportation Research Part C: Emerging Technologies, 2023, 148: 103981. DOI: 10.1016/j.trc.2022.103981.

[19] HAIDER Z, CHARKHGARD H, KIM S W, et al. Optimizing the relocation operations of free-floating electric vehicle sharing systems[J/OL]. SSRN Electronic Journal, 2019. DOI: 10.2139/ssrn.3480725.

[20] POIKONEN S, GOLDEN B, WASIL E A. A branch-and-bound approach to the traveling salesman problem with a drone[J/OL]. INFORMS Journal on Computing, 2019, 31(2): 335-346. DOI: 10.1287/ijoc.2018.0826.

Index