北京郵電大學

物流系统综合课程设计

题目：基于循环取货（Milk-run）方式的零部件配送

姓　　名：李想、薛潇、陈茜颖
苏凤、李明妍

学　　院：自动化学院

专　　业：物流工程

项目小组：第四组

指导教师：刘玉坤、雷全胜

2019 年 9 月

目录

1. 项目简介

1.1 方案背景

安吉物流是全球业务规模最大的汽车物流服务供应商，共有员工 17,000 人，拥有船务、铁路、公路等 10 家专业化的轿车运输公司以及 50 家仓库配送中心，仓库总面积超过 440 万平方米，年运输和吞吐量超过 570 万辆商品车，并且全部实现联网运营。其下属业务之一的零部件物流是其重要的业务组成部分。

汽车零部件物流是各个环节必须衔接得十分流畅的高技术物流行业，是国际物流业公认的最复杂、最具专业性的物流领域，特别是零部件的入厂物流更体现出极高的专业性和复杂性。近年来，循环取货的配送模式（Milk-run）在安吉零部件入厂物流业务方面得到了广泛的应用和发展，给汽车制造业供应链管理带来重大流程革命及变动。供应链的管理和优化，可为企业带来巨大的效益。在企业内部，通过采用现代化手段，建立完善的物流网络体系，使各企业更加适应新的市场环境。在企业外部，通过对供应链的协调管理，以供应商为中心，以网络管理为核心，利用现代科技手段，准确及时的获取信息，迅速沟通零部件供应商和整车生产商，并依靠供应链的整体优势，共享信息资源，发挥供应链的整体优势提升企业核心竞争力。

由此看来，将循环取货的运作模式应用于零部件入厂物流是非常具有优势的。但是当下理论与实际的差距仍然很大，该模式能否成功落地并为企业带来持续的效益仍然有待思考。公司引入循环取货的运作模式的目的在于如何对零部件供应商现有的运输网络加以优化，使之既能够尽量满足生产波动的需求，同时又能将运营成本控制在一定范围内。但是目前在应用的过程中发现物流运作的成本仍然偏高，在总的物流费用成本中运输的成本达到了 44%。

随着供应商规模的不断扩大，路径设计成为了一个非常棘手的问题。另外，如何在有限车辆的前提下尽量提高装载率是循环取货模式具备成本控制的特点主要瓶颈。但是由于汽车的零部件众多，一家供应商提供的零部件产品也是多种类的，零部件的包装尺寸更是不尽相同，使得装载率很难得到保证。因此运输卡

车在其 Milk-run 路径上实施循环取货时，在哪些供应商中对哪些零部件进行装箱操作能够满足运输卡车的配载实现最大装载率的要求，同时在该条 Milk-run 路径中配置多少辆运输卡车（可循环使用）能够在保证主机厂的生产需求订单的前提下，能够有效控制运输成本。另外，在实际操作中还存在这样的问题：主机厂的生产订单每天的需求是不同的，对零部件的需求总是在一定的范围内产生波动，因此如何对运输车辆进行动态调度对于节约运输成本就显得非常重要。

1.2 方案目标

综合分析基于循环取货(milk-run)方式的零部件配送案例后，我们发现在循环取货模式应用之初，由于业务量较小，供应商也比较少，而且分布也比较集中，因此循环取货的模式取得了比较好的效果。然而随着汽车生产厂家的生产量急剧增长，公司的业务量也在急剧的攀升，使得当供应商的规模不断扩大后，运输车辆的路径设计就变成了一个棘手的问题。为解决安吉汽车物流运输车辆的路径设计问题，我们需要对以下问题进行全面考虑与分析：

（1）将各个供应商的出货地址、严格的出货窗口设置纳入考虑范围，设计一个适用于业务量较大、供应商位置分布广泛的路径规划方案。

（2）考虑装载率、库存成本、运输成本以及缺货风险对已设计的路径规划方案进行优化，使优化后的方案能更满足企业的需求与战略发展。

（3）建立一个科学、客观的评价体系，来对我们的优化方案进行分析和评价。

（4）设计关于异常配送过程的应急优化方案。如何在异常运行流程中，在保障生产顺利进行的前提下，科学的设计应急运输方案，进行合理的决策以控制运营成本是一个值得研究的问题。只有考虑充分且方案成熟的应急准备，才能够最大程度地在各种意外层出不穷的情况下保障生产的顺利进行。

1.3 基本流程

具体的技术路线图如图 1-1 所示：

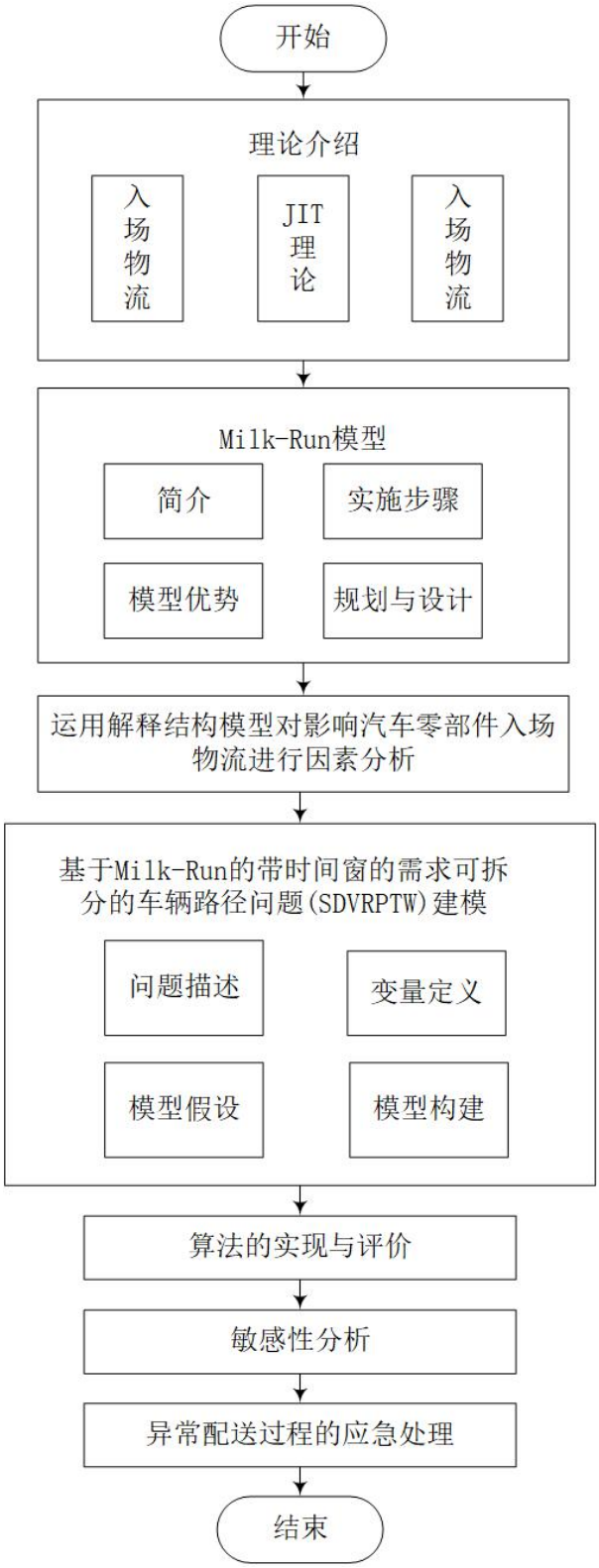

开始
理论介绍
入场物流
JIT理论
入场物流
Milk-Run模型
简介
实施步骤
模型优势
规划与设计
运用解释结构模型对影响汽车零部件入场物流进行因素分析
基于Milk-Run的带时间窗的需求可拆分的车辆路径问题(SDVRPTW)建模
问题描述
变量定义
模型假设
模型构建
算法的实现与评价
敏感性分析
异常配送过程的应急处理
结束

图 1-1 技术路线图

2 相关理论介绍

2.1 入场物流

从汽车零部件采购到整车销售的过程中，汽车物流包括了零部件入场物流、厂内生产物流、整车销售物流、售后逆向物流四个主要部分。零部件入场物流又指供应物流，是指零部件从供应商仓库到汽车厂仓库的之间的所有物流活动，包括物料的采购、包装、运输以及库存管理和控制等一系列增值活动。合理的入场物流能使汽车的制造装配按工序顺利进行，缩短产品的生产周期，同时降低零部件库存，节约物流成本，满足订单需求，为企业带来更大的利润。而一旦出现供应不及时的情况，汽车厂将会出现停线危险。入场物流属于整个整个汽车物流供应链的源头，管理好了入场物流可以提高企业的敏捷性，提升供应链的增值水平，更好的协调产销矛盾，所以入场物流对于整个汽车物流供应链来说是十分重要的。

汽车具有成千上万中零部件，如何把他们运送到汽车厂进行装配，保证流水线的平稳有序生产就是我们所需考虑的入场物流如何组织的问题。目前各大汽车厂的入场物流模式主要分为四类：

1) 供应商自主送货到汽车厂仓库，由汽车厂负责配送至生产线上。

供应商把自己所生产的零部件送至汽车厂的仓库，由汽车厂负责把零部件拆包转移至工位器具，并送至相应工位。以零部件到达汽车厂为界限，之前的物料破损由供应商自己负责，到达仓库之后产品出现人为的损坏是汽车厂的责任。供应商每月需支付给汽车厂相应的仓储费用以及管理费用（拆包费和配送费用）。

2) 供应商自主送货到汽车厂仓库，并负责配送到工位上。

供应商不仅要送货到线边仓库,而且还要安排叉车工人负责配送到流水线上。供应商同样需要租用汽车厂仓库的一定面积区域用来存储。这种方式的供应商-般距离汽车厂距离较近，每天基本定时定量送货，出现产品问题，由供应商自己负责。每月支付给汽车厂一定的线边仓面积费用，而汽车厂则需支付配送费用(采购费用与物流剥离的情况下)。

3) 供应商的零部件集货至零部件集散中心，再配送至线边仓和工位。

集货到零部件集散中心(以下简称集散中心)可分为循环取货和供应商自主送货两种。循环取货则一般由三方物流公司或者汽车厂自己车队负责运输。集散中

心的管理可由汽车厂负责或者招标指定的三方物流负责。从集散中心配送至汽车厂时也可分为汽车厂自己车队运输和三方物流配送两种方式。根据企业发展各自的核心竞争力趋势，一.般是由三方物流公司进行管理配送。集散中心的选址是循环取货关注的重点，关系到取货路径和频数的规划。循环取货(又称为 Milkrun 模式)是目前比较流行的方式，具有较复杂的费用分摊方式和库存策略。

4) 供应商送货至第三方物流公司，由三方物流公司负责配送至生产线边仓及工位。

供应商自己送货至汽车厂指定的三方物流公司,此之前的物权归供应商所有。到达三方物流仓库之后，根据不同企业也有所不同，有的归三方物流负责，但大部分企业实施的还是供应商的所有权。三方物流每天负责把供应商的货物配送至汽车厂线边仓，再由三方物流公司人员配送上线。在此种情况下，供应商在三方物流处租用仓库，三方物流则在汽车厂的线边仓租用供临时摆放的仓储面积。物流费用则是三方物流收取供应商的仓储费，配送费用和管理费用，汽车厂收取三方物流的线边仓仓储费用。汽车厂本身仓储面积紧张时，较适用此种方式。

2.2 准时化理论（JIT）

入场物流是汽车行业实施 JIT 中的重要一环。准时化理论为丰田公司提出，其本质在于通过“有计划的消除浪费和持续改善生产率”来降低成本，减少库存，提高质量。JIT 理论提倡的是准时，对生产、配送都要求很高的时效性，以便于能够及时暴露出生产过剩和其他设备，人员方面的浪费，从而进行改善。JIT 采用看板管理的方法，包括生产看板和传送看板。看板是用来发布指示、领取指示或作为目视化管理的工具。以最终用户的需求为生产起点，即由看板传递下道工序向上道工序的需求信息。看板的内容一般包括看板号，零件编码，工位器具容量(零件数量)，供应商编码，订单号等信息。各生产环节通过看板进行传递，从市场订单拉动车间生产,再到供应商供货形成一体化服务,从而实现拉动式生产。

而对于库存管理，准时化理论(JIT)一直提倡零库存，即努力减少在制品库存和相关的运输成本，从而提高企业的投资回报率，实现一个流生产，“需要一件,生产一件”，使库存降低到最低。库存越低，物料在生产系统中的流动就越快，就能更好的响应顾客的需求，向顾客提供所需的产品。这样既能减少成本，

又能提高企业的竞争优势。

2.3 精益生产

精益生产指的是依靠大规模生产制造技术来降低成本,减少生产时间,提高质量,同时更好地响应市场需求。JIT 理论和精益生产实质上是一致的，区别在于观察问题的角度和考虑问题的范围不同。JIT 是在企业内部持续改善的思想，而精益思想则从外部关注的顾客开始。从理解顾客的需求，获得客户订单并反馈给客户，是精益思想的起源。通过分析所有产品生产的活动来识别客户价值,然后从客户角度来优化整个生产流程。

精益生产研究的主要内容包括工厂的现场管理、新产品设计开发、与顾客的关系、与供应商的关系等方面，从而使企业成为一个精益企业。在目前应用精益思想的研究中,发现精益生产注重提升企业系统的稳定性可以从一系列数据中得以证明:精益生产让生产时间减少 90%，库存减少 90%，使生产效率提高 60%,减少市场缺陷 50%，不合格废品率降低 50%，而安全指数提高了 50%。可以看出精益生产是当前工业界最佳的生产组织体系和方式之一。

3.Milk-Run 模式

3.1Milk-Run 简介

Milk-Run，中文翻译为“牛奶式取货”、“循环取 / 送货”、“集货配送”等，是一种优化的物流系统。该思想起源于英国北部农场的牛奶运输方式，配送牛奶的工人驾驶 卡车，按照计划路线依次将牛奶送至各家门口，并回收前一天的空奶瓶。后来该方法 被借鉴发展成为用同一运输车辆到多个供应商或客户处进行取送货的操作模式。

Milk-Run 是一种典型的 JIT 运输，是一种精益供应链的管理方法。对于汽车制造企业而言，启动 Milk-Run 的实质就是企业对于 JIT 生产方式的追求，企业希望通过实施 Milk-Run 来保证零部件供应的及时性、准时性，而不至于造成缺料停线。在此基础上，实现运输、库存成本的节约。

对于一个汽车主机厂，Milk-Run 取货路线可能有上百条。运输配送车队一般由 3PL 企业安排运输车辆从汽车厂的配送中心或 RDC 出发，按照设定好的车辆路径和订单信息依次从各个供应商处或集拼中心处取货，经过一个循环，最终回 到主机厂的配送中心或 RDC，实现供应商和仓库之间的 Milk-Run 取货，适用于小批量、多频次的近中程距离的物料零部件配送。采用 Milk-Run 取货模式，与企业建立合作的 3PL 公司能够有效地为企业服务实现 JIT 供应，降低整个运输物流的成本，提高物料零部件配送效率，增强企业的市场竞争力。

3.2Milk-Run 实施步骤

根据前文对 milk-run 取货模式的描述，其具体实行步骤如下图 3-1 所示：

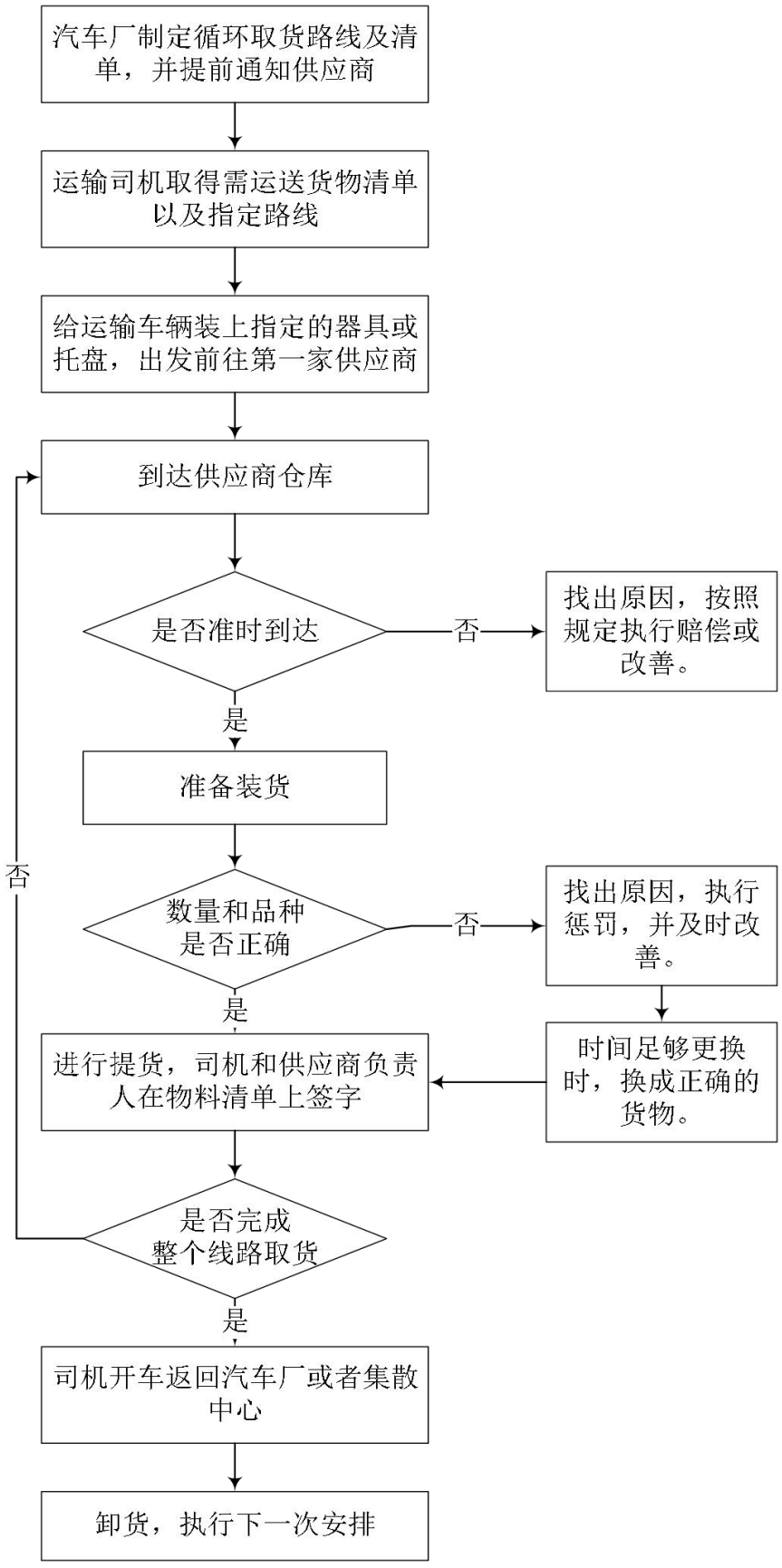


图 3-1Milk-Run 的作业流程图

（1）由 S 企业制定 milk-run 取货的 BOM 清单、设定的 milk-run 取货 路径图和标准包装工装(周转箱、托盘等)，并提前告知每条 milk-run 取货线路上的供应商需要取货韵零部件货量和件数以及 milk-run 取货车辆预计到达 的时间，让相关供应商做好准备。将 BOM 清单交给对应的司机进行取货。

（2）司机准备物料零部件包装，按照设定好发车的时间和 milk-run 取货 路线行驶至第一家供应商处。

（3）若司机准时到达第一家供应商，则按照 BOM 清单取货；若司机到达时间提前或滞后，则按照相关规定进行验证改进。

（4）若司机收货时核对物料零部件完好，数量正确，则实施取货；若出现差错，则找出原因进行沟通，供应商出现问题则对其进行处罚，主机厂 BOM 清单出错则按照正确的数量取货。

（5）司机取货完成后和供应商处的负责人在 BOM 清单上签字，行驶至下 一家供应商处取货。

（6）反复执行(3)(4)(5)步骤，直到 milk-run 取货车辆基本满载，milk-run 取货完成之后司机返回主机厂或 RDC。

（7）仓库人员使用叉车等进行卸货分拣，司机准备执行下一次 milk-run 取货任务。

3.3 实施 Milk-Run 模型的优点

实施 Milk-Run 模型有一下几个特点：

（1）基本实现 JIT 供应，显著降低库存。

（2）在 Milk-Run 模式下，汽车制造企业不需要建立大量的安全库存来保证生产的连续性，将安全库存缩小至几小时，个别零件或许可以实现“零库存”，有效降低了库存成本。

（3）对于供应商来说，只需根据订单数量及其交付时间按需安排生产即可，有效降低了库存成本，甚至实现“零库存”。

（4）提高车辆运输效率，降低运输成本。

（5）Milk-Run 模式是对总装厂到各个零件供应商之间的车辆行驶路线进行了组合优化，减少了运输总里程，降低了运输成本。

（6）优化整个供应链的运作。

（7）Milk-Run 模式要求供应链上各参与方的有效合作，保证各环节紧密的紧密衔接，实现准时准确的供货。对各参与方和信息技术等的高要求从整体上提高了整个供应链的服务水平。

3.4Milk-Run 的规划与设计

3.4.1 前期设计需求数据

（1）供应商数据。包括供应商出货仓库的详细地址及场地情况、配送量等。零件数据。包括零件清单（BOM 表）；零件包装情况；零件性质；料箱料架数据等。

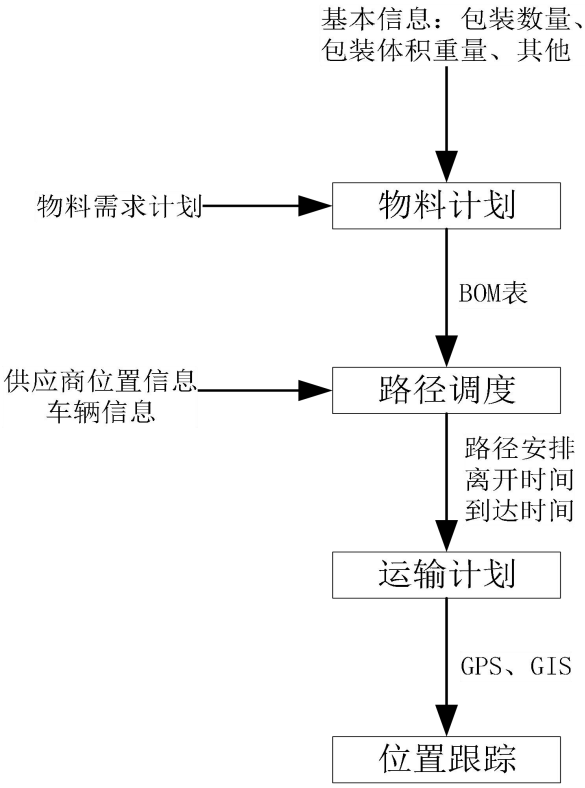
（2）总装厂数据。包括供应商代号；客户订单、生产计划；收货场地情况等。

（3）运作数据。包括单位库存成本；车辆载重、平均形式速度；装卸货操作时间等。

此例中，通过查阅资料我们假定库存成本为：0.042 元/平方米·时；运输成本：5.2 元/公里

3.4.2Milk-Run 工作流程

整体工作流程如图 3-2 所示：



3-2Milk-Run 整体工作流程图

3.5Milk-Run 的评估标准

采用 Milk-Run 模式，需要考虑一下几个方面：

（1）安全性原则。保证零部件质量是运输的首要原则。

（2）准确性原则。要求做到及时准确送货，不错送、不漏送。

（3）经济性原则。保证总成本最优。

4. 影响汽车零部件入厂物流的因素分析

在分析现有入厂物流现状和问题的基础上，对影响入厂物流的因素进行总结，发现主要存在库存成本、运输规划、物流人员、包装、信息共享、供应商服务水平、流程与规范、交通条件、供应链协调、生产计划、数据收集 12 方面的问题。

对影响入厂物流模式的各因素使用 ISM 实用化方法分析。通过两两比较，直观地确定各要素之间的二元关系，并在两两要素交汇处的方格内用符号 V、A 和

X 加以标识。根据各影响要素间二元关系的传递性，逻辑推断出递推的二元关系，并用加括号的标识符表示，得到的要素关系分析图如表 4.1 所示：

表 4.1 要素关系分析图

A	A	A	(A)	A	(A)	A		(V)	X	1.库存成本
(A)	A	A	A	A	A	A	A		2.运输规划	
	V	V						3.物流人员		
			A	A			4.包装			
X	(V)	V		V	V	5.信息共享				
(A)	V		A		6.供应商服务水平					
(A)	V	(V)		7.流程与规范						
	V		8.交通条件							
A	V	9.数据收集								
A	10.生产计划									
11.供应链协调										

在此基础上，加入反映自身到达关系的单位矩阵，建立起系统要素的可达矩阵，如图 4.2 所示。

$$M = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \end{array} \begin{array}{c} \begin{array}{ccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

图 4.2 可达矩阵

对可达矩阵的缩减矩阵进行层次化处理，再根据层次结构建立多级递阶有向图，最终得出解释结构模型，如图 4.3、4.4、4.5 所示。

M（L）=

	2	4	10	6	9	3	8	7	11
2	1	0	0	0	0	0	0	0	0
4	1	1	0	0	0	0	0	0	0
10	1	0	1	0	0	0	0	0	0
6	1	0	1	1	0	0	0	0	0
9	1	0	1	0	1	0	0	0	0
3	0	0	1	0	1	1	0	0	0
8	1	1	1	1	0	0	1	0	0
7	1	1	1	0	1	0	0	1	0
11	1	0	1	1	1	0	0	1	1

图 4.3 具有层次结构的缩减矩阵

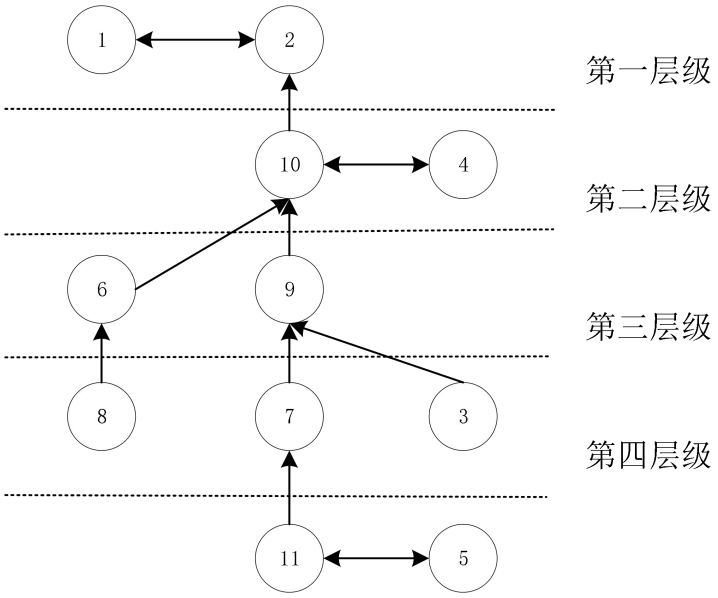


图 4.4 多级递阶有向图

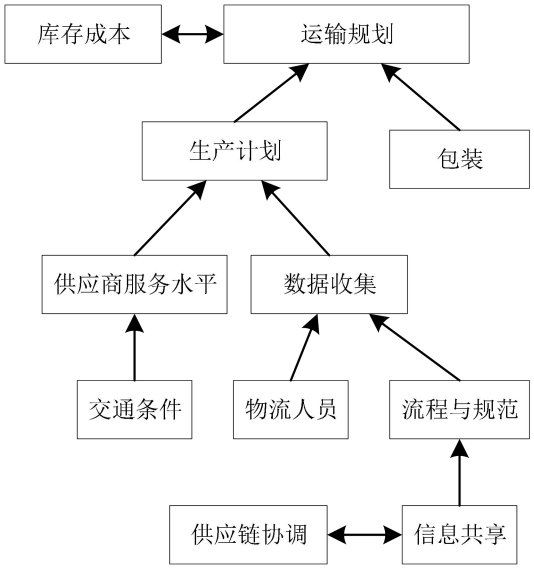


图 4.5 解释结构模型

在图 4 中，可以看出影响入厂物流模式因素，从长期性和根本上来说，取决于这个汽车生产行业的整个供应链协调与信息资源的共享程度，从短期性和直接性来说，与汽车零部件的库存成本、取货运输规划、整装生产计划和运输包装形式等直接相关，而供应商服务水平、数据收集、交通条件、物流人员素质、流程与规范直接影响以上四个要素。这充分说明，要想提高入厂物流水平，应该尽量降低库成本，通过优化运输路径的方式，采用合理的包装提高装载率，同时也要制定相关的流程与规范准则，加强物流人员素质教育培训，让整个供应链收集到的需求数据精准无误，降低牛鞭效应，资源得到优化配置的同时大幅减少物流成本费用。

5.基于 Milk-Run 的 SDVRPTW 的问题建模

通过对 Milk Run 的规划研究分析，易知车辆路径问题是 Milk-Run 成功运作的最关键问题。因此，如何安排适当规模的车辆按照事先设计的送货路线，在满足货物配送量、车辆装载限制、时间窗约束等约束条件下，实现多目标的优化结果，包括综合成本最小、行驶路程最短、车辆数目最少、车辆利用率高等，这也是 Milk Run 实施的重点。所以，接下来这一部分主要是介绍车辆路径相关问题，并以此为基础，构建适当的 Milk Run 循环送货路径优化模型，并设计合适的算法，

以实现模型目标的最优。

5.1 车辆路径问题（VRP）描述

车辆路径问题（Vehicle Routing Problem，简称 VRP）。其基本的过程为：在一个存在供求关系的系统中，有一定数量的客户需求点，且每个客户都有不同数量的货物需求，由一个配送中心为这些需求点提供货物，安排适当的车辆行驶路线，在满足一定约束条件（如货物需求、车辆载重和收发货时间等）下，使客户的需求得到满足，并使目标函数（比如成本最小、距离最短、车辆数目最少等）得到优化。下图 5-1 为简单的 VRP 配送示意图：

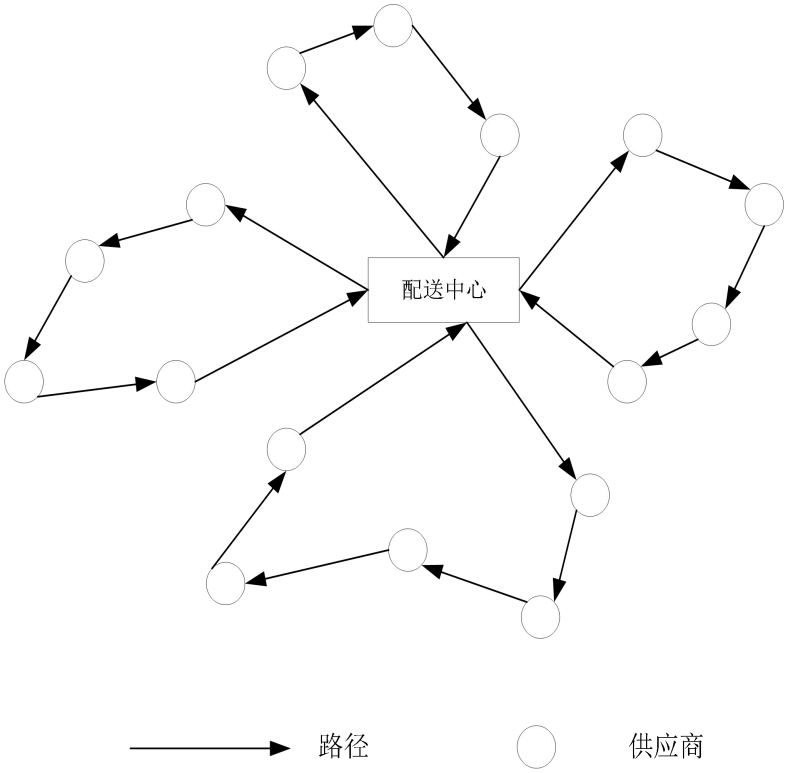


图 5-1 VRP 配送示意图

一般的 VRP 是以总费用最少为目标，寻求满足一下 3 个约束条件的解：

（1）每辆车以同一配送中心为起点和终点。

（2）每个需求点只能由一辆车进行服务。

（3）每条路的总需求不能超过一辆车的总装载量。

VRP 的数学模型如下：

$$\min f = \sum_{i \in N_0} \sum_{j \in N_0} \sum_{k \in V} C_{ij} x_{ijk} \quad (5\text{-}1)$$

$$s.t. \sum_{i} q_i y_{ki} \le Q(k = 1,2,...,k) \quad (5\text{-}2)$$

$$\sum_{\mathrm{k}} y_{ki} = 1, (i = 1,2,...,n) \quad (5\text{-}3)$$

$$\sum_{\mathrm{k}} y_{k0} = k \quad (5\text{-}4)$$

$$\sum_{\mathrm{i}} x_{ijk} = y_{kj}, (j = 1,2,...,n) \quad (5\text{-}5)$$

$$\sum_{\mathrm{j}} x_{ijk} = y_{ki}, (i = 1,2,...,n) \quad (5\text{-}6)$$

$$x_{ijk} \in \{0,1\} (i, j = 0,1,...n; k = 1,2,...k) \quad (5\text{-}7)$$

$$\mathrm{y}_{ki} \in \{0,1\} (i = 0,1,..., n; k = 1,2,...k) \quad (5\text{-}8)$$

$$\mathrm{y}_{ki} = \begin{cases} 1, \text{由车辆k完成点i的任务} \\ 0 \end{cases} (\mathrm{i} = 0,1,..., \ \mathrm{n}; k = 1,2,..., k)$$

$$\mathrm{x}_{\mathrm{ijk}} = \begin{cases} 1, \text{车辆k从点}i\text{行驶到}j\text{完成} \\ 0 \end{cases} (\mathrm{i}, \ \mathrm{j} = 0,1,..., \ \mathrm{n}; k = 1,2,..., k)$$

模型中，配送中心的编号为 0，客户编号为 1，2，...，n，车辆编号为 1，2，...，k。

目标函数(5—1)表示总费用最低：式(5—2)表示车辆载重能力约束，也就是每条线路的总送货量不允许超过车辆的最大装载量；式(5—3)表示每个客户都要有车辆提供服务；式(5—4)表示车辆的出发点和终点都要是配送中心，也就是闭环运输；式(5—5)和式(5—6)表示如果客户点 i，j，在车辆 k 的行驶路线上，则客户点 i，j，需求量将由车辆 k 进行配送。

5.2 带时间窗的需求可差分的车辆路径问题概述

带时间窗的需求可拆分的车辆路径问题(Split Delivery Vehicle Routing Problem with Time Windows，简称 SDVRPTW)是车辆路径问题的一种，是由带时间窗的车辆路径问题和需求可拆分的车辆路径问题的混合。

5.2.1 带时间窗的车辆路径问题

带时间窗的车辆路径优化模型(Vehicle Routing Problem with Time Windows，简称 VRPTW）是指客户要求在一定的时问范围内被访问，即客户有接收货物的最后时间节点和最早开始时间节点。

VRPTW 问题可以描述为：一个出发点，n 个需要服务的客户，K 辆一定承载能力的汽车。已知：每个客户的位置、客户的需求量、出发点地点、每辆车的载重限制以及客户能够接受服务的时间范围，要求是合理制定车辆行驶方案，使服务总费用尽可能小，并且要求满足下面的条件限制：①每个客户只能被访问一次；②每条路线要从出发点出发，最后回到出发点；③每条线的总配送量不能超过车辆最大承载能力；④必须在客户规定的时间窗内完成配送。

若车辆不能在要求的时间窗内从供应商处取货，也就是违背了时间窗限制，则必然会带来一定损失，这些损失会产生时间效应成本，因此，在追求物流总成本最优的情况下，必须将这部分成本也考虑进去。产生时间效应成本的原因包括一下两个方面。

1.供应商为了降低库存而要求准时配送，若车辆无法在要求的时间窗内取货，则需要赔偿一定的损失。

2.如果车辆提前到达则需要等待，这段时间车辆和人员均处于闲置状态，就会导致闲置成本和机会成本的产生，也可能会产生交通问题，耽误下一个供应点的任务。

我们假定当运输车在规定的时间窗之外到达，则需要按照违反时间的长度接受一定的惩罚，惩罚函数大致如下图 6-2 所示：

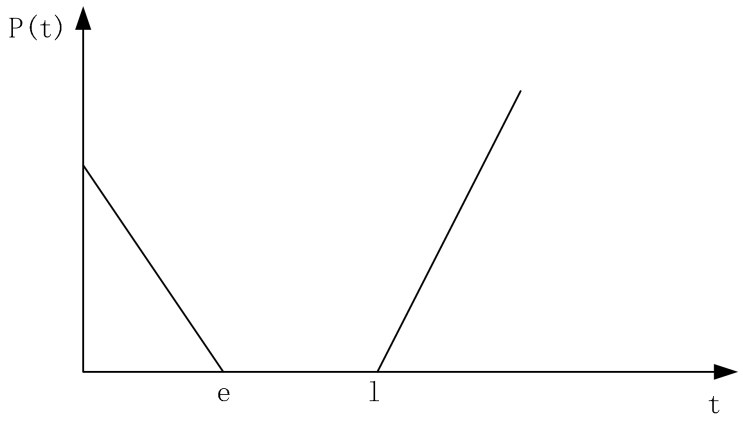


图 5-2 惩罚函数图

对违反时间窗的行为进行惩罚，可是装配厂不至于为了只关注于如何节省路线成本而忽略了服务水平。惩罚成本的权重是企业根据成本和供应商满意度的综合考量定的，整体的趋势是车辆配送时间偏离约定时间窗的范围越大，其惩罚成本越高。定义惩罚函数表达如下：

$$\begin{cases} \mathrm{p}(e_i - s_i), & s_i < e_i \\ 0, & e_i \le s_i \le l_i \\ q, & s_i > l_i \end{cases}$$

上式可统一表示为：

$$\mathrm{p}_i(s_i) = p\max(e_i - s_i, 0) + q\max(s_i - l_i, 0)$$

式中参数的定义为：

si——表示车辆到达客户 *i* 的时间；

ei——客户 *i* 接收服务的最早时间；

li——客户 *i* 接受服务的最晚时间；

p——车辆早于时间窗起始点到达而发生的等待单位时间机会成本；

q——车辆晚于时间窗起始点到达而发生的等待单位时间惩罚值；

如果车辆 *k* 在 *ei* 之前到达客户 *i* 处，则需要等待，从而发生的机会成本为 p(*ei*-s*i*)；在 *li* 之后到达，则产生延误成本 q(s*i*-*li*)；在[*ei*，*li*]内到达，则遵守了时间窗约束，不产生额外成本。其中 p 和 q 的取值大小依实际情况而定，如果客户对时间窗要求比较严苛，则 p 和 q 可以取比较大的值。

5.2.2 需求可拆分的车辆路径问题

在传统 VRP 问题中，有一个隐含的约束条件：每个客户点只能由一辆车提供服务。也就是说不允许将客户的同一批次的需求量拆分成几次配送。但是在实际的配送中可能存在这样两种情况：第一，n 个客户的需求总量刚好等于 K 辆车的装载总量，但是因为该约束条件的存在，必须要 K 辆以上的车辆；但若允许多辆车为同一供应商提供服务，则恰好需要 K 辆车就可以完成任务。第二，当存在某些客户的需求量超过了一辆车的最大载重（装载体积）时，就要对这些客户的需求进行拆分配送，允许多辆车共同完成任务。所以，在现实的配送任务中，对需求进行拆分配送的研究有很好的实际意义。方案详情如下图 5-3 所示：

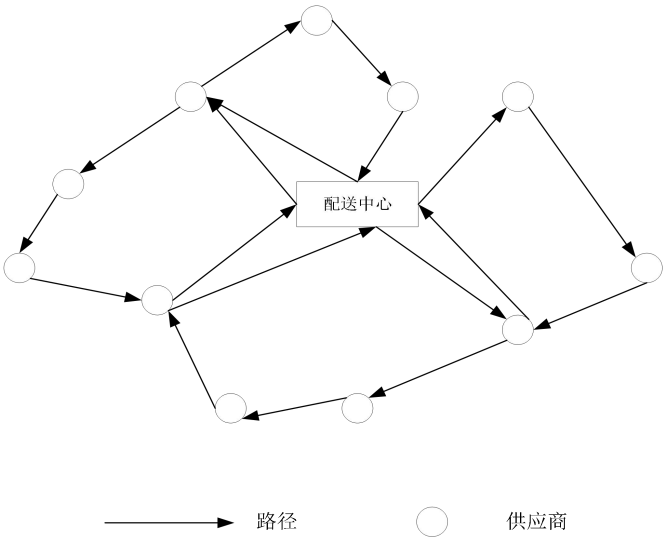


图 5-3 需求可拆分的车辆路径问题模型

5.3 基于 Milk-Run 的 SDVRPTW 问题建模

我们拟将对带时间窗的车辆路径问题（VRPTW）以及需求可拆分的车辆路径问题（SDVRP）结合起来进行路径优化，使其更加符合实际情况。

5.3.1 问题描述

该问题可以描述为：由 *s* 公司的一个总装厂和几个客户点组成待研究的配送网络，已知客户点与配送中心的地理位置。由车 *k* 从总装厂出发，前往客户点 *i*

完成装载任务，然后回到总装厂。车辆 *k* 的行驶轨迹就是一条 Milk Run 循环送货路线，所以 *k* 辆车就对应 *k* 条循环路线。允许对客户的需求进行拆分，由多辆车共同完成配送任务。但在同一条路径中，每一客户节点只能被服务一次。另外，车辆 *k* 要尽可能的在客户 *f* 约定的时间窗[ei，li]内到达，否则将接受一定惩罚。该 SDVRPTW 问题的目标是合理安排车辆行驶路线，使得包括运输成本、库存成本和惩罚成本的总成本最优。

5.3.2 定义变量

N：总装厂对接的所有客户点的集合，N={1,2,...,n}；

N_0：总装厂和客户点的集合，N_0={0,1,2,...,n}；

i,j：单个客户或总装厂，i,j ∈ {0,1,2,...,n}；

K：车辆编号，V={1,2,...,k}，k ∈ V；

C_{ij}：从客户 i 到客户 j 的运输成本（与距离相关），且 i≠j；

α：车辆单位运输距离成本；

Q：车辆 k 的最大载重（或容积）；

h_{ik}：客户 i 的单位库存成本；

Y_{ik}：车辆 k 从客户 i 处提取的货物总量（或总体积）；

d_i：需要从客户 i 处提取的货物总量（或总体积）；

ei，li：客户 i 允许的最早或最晚服务时间；

si：车辆 k 到达客户 i 处的时刻，s_0=0;

t_{ij}：车辆 k 从客户 i 行驶到客户 j 的时间，且 i≠j；

f_i：车辆 k 在客户 i 处停留时间；

w_i：车辆提前到达客户 i 处的等待时间，w_0=0;

T_0^k：车辆 k 总装厂出发的时间；

T_R^k：车辆 k 要求返回总装厂的时间；

$$x_{ijk}=\begin{cases}1，\text{车辆k从点i行驶到}j\text{完成}\\0，\text{否则}\end{cases}（\mathrm{i}，j=0,1,...,n；\ k=1,2,...,k）$$

5.3.3 模型的假设

1）本问题只有一个总装厂，且位置已知。有足够的车辆；

2）车辆规格相同，从总装厂出发最终要回到原点，形成闭环运输；

3）所有供应商位置和其供应零件包装体积和供应数量已知；

4）假定任意两点之间的距离是对称的。

5.3.4 模型的构建

构建模型如下：

$$\min f = \partial \sum_{i \in N_0} \sum_{j \in N_0} \sum_{k \in V} C_{ij} x_{ijk} + \sum_{i \in N} \sum_{k \in V} h_{ik} y_{ik} + \sum_{k \in V} \sum_{i \in N} p_i(s_i) \tag{5-9}$$

s.t.

$$\sum_{k \in V} \sum_{j \in N} x_{ijk} \le k, \quad i = 0 \tag{5-10}$$

$$\sum_{j \in N} x_{ijk} = \sum_{j \in N} x_{jik} \le 1, \quad i = 0 \tag{5-11}$$

$$\sum_{k \in V} \sum_{j \in N_0} x_{ijk} \ge 1, \quad i \in \{1,2,...,n\} \tag{5-12}$$

$$\sum_{k \in V} y_{ik} = d_i, \quad i \in \{1,2,...,n\} \tag{5-13}$$

$$y_{ik} \le d_i \tag{5-14}$$

$$R_l \cap R_m \le 1, \quad l \ne m \tag{5-15}$$

$$\sum_{i \in N_0} \sum_{j \in N_0} d_i x_{ijk} \le Q, \quad k \in \{1,2,...,k\} \tag{5-16}$$

$$\sum_{i \in N_0} x_{ip} - \sum_{j \in N_0} x_{pj} = 0, \quad p \in \{1,2,...,k\}, \quad k \in \{1,2,...,k\}$$

$$\mathrm{T}_0^{\,k}+\sum_{i\in N_0}\sum_{j\in N_0}x_{ijk}(t_{ij}+f_i+w_i)\le T_p^{\,k},\ \ k\in\{1,2,...,n\} \tag{5-18}$$

$$\sum_{\mathrm{k}\in V}\sum_{i\in}x_{ijk}(s_i+f_i+w_i+t_{ij})=s_j,\ \ j\in\{1,2,...,n\} \tag{5-19}$$

$$\mathrm{x}_{ijk}\in\{0,1\},\ \ i,j\in\{1,2,...,n\},\ \ k\in\{1,2,...,k\} \tag{5-20}$$

$$\mathrm{p}_i(s_i)=p\max(ei-si,0)+\mathrm{q}\max(si-li,0) \tag{5-21}$$

对上述模型做一下说明：目标函数(5-9)表示总成本最小，主要由行驶成本，库存成本和惩罚成本构成；约束(5-10)表示所启用的车辆总数不超过 K，并且 K 辆车的总装载重可以满足客户所有需求；约束(5-11)表示车辆的始点和终点都是配送中心；约束(5-12)表示每个客户点至少被一辆车服务，即允许需求被拆分；约束(5-13)表示每一客户点的需求得到满足；约束(5-14)表示需求只能被拆分到访问该点的车辆上，且拆分的量不能超过该点的总需求量；约束(5-15)表示两条循环送货路线中最多只能有一个交点；约束(5-16)表示单位车辆运输总和不超过车辆载重；约束(5-17)表示流守恒条件，也就是车辆到达某客户点后，必须离开该点；约束(5-18)表示车辆出发和返回时间都要在配送中心规定的时间范围内；约束(5-19)表示车辆从客户 i 行驶到客户 j 的时间约束：约束(5-20)表示整数化约束，限制 x_{ijk} 只能取 0 或 1；约束(5-21)是相应的惩罚函数表达式，惩罚力度也可以根据零件属性及需求设为无穷大。

6.算法实现与评价

6.1 位置信息获取

我们首先获得 193 个供应商位置的经纬度信息，图 6-1 为这些供应商在地图上的相应位置：

图 6-2 供应商位置图

做成的散点图大致如下图 6-2 所示（图中橘色标识的是总装厂的位置。）：

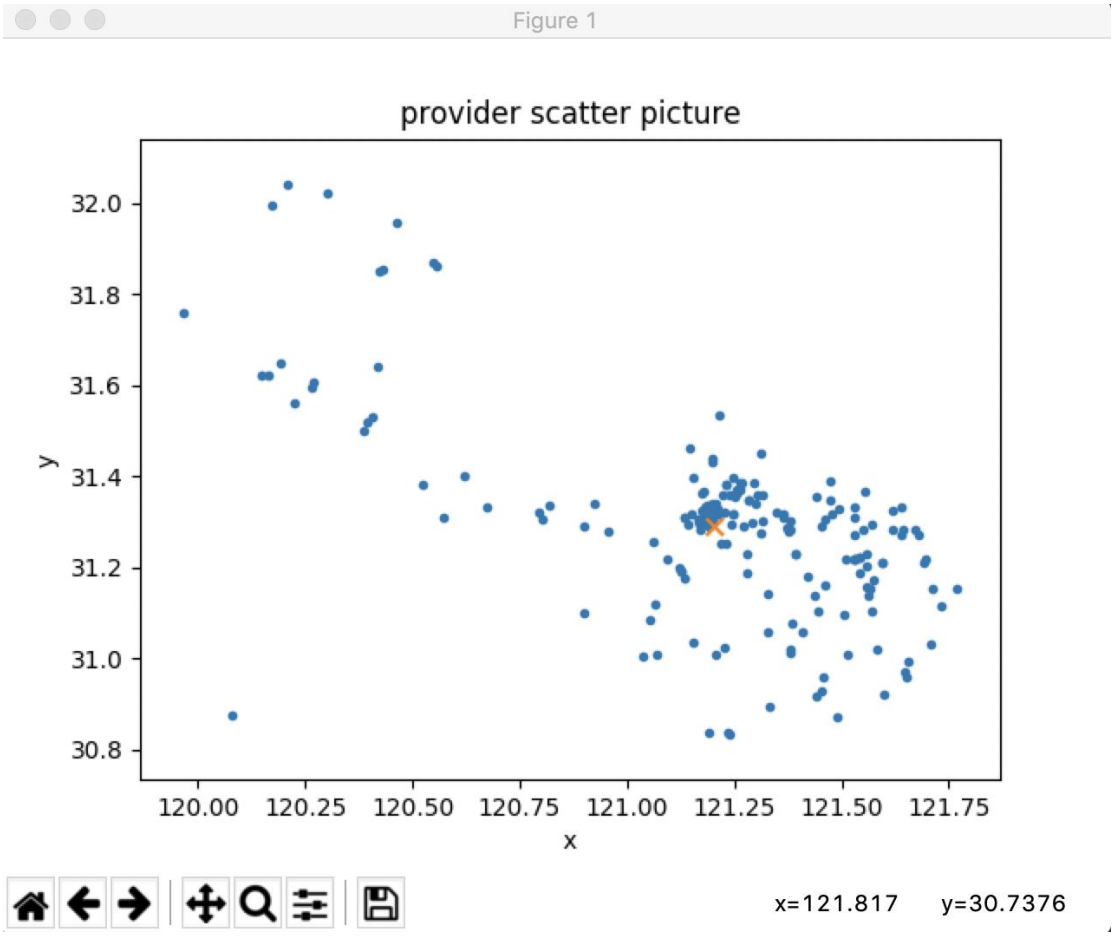


图 6-2 供应商位置散点图

同时，我们按照不同供应商供应零部件种类和数量的不同，按照‘RdYlBu’方案做了相应的区分，改过后的散点图如下图 6-3 所示：

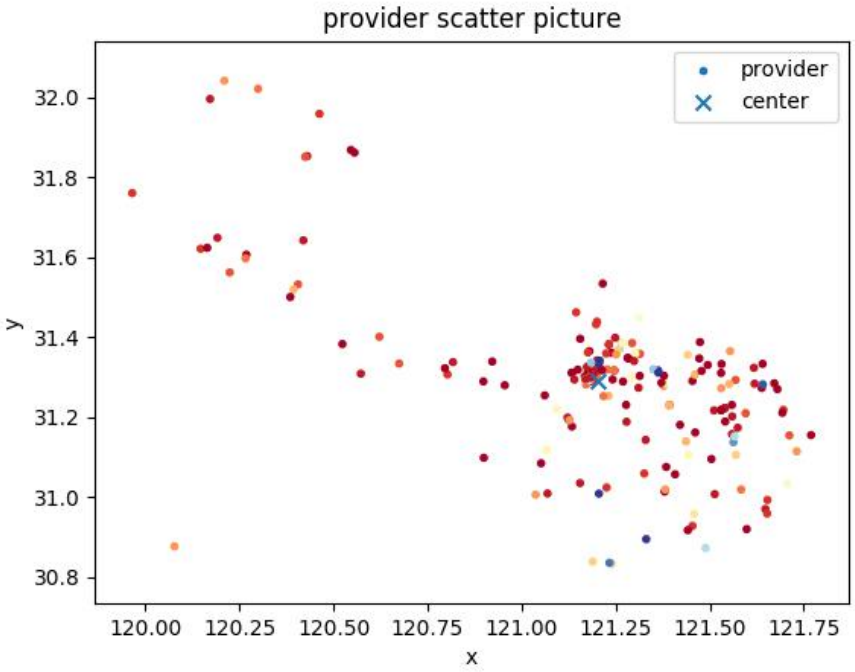


图 6-3 改过后的供应商分布散点图

然后按照经纬度计算曼哈顿距离，形成距离矩阵。

6.2 算法的主要实现过程

所涉及到的算法大概的流程图如下图 6-4 所示：

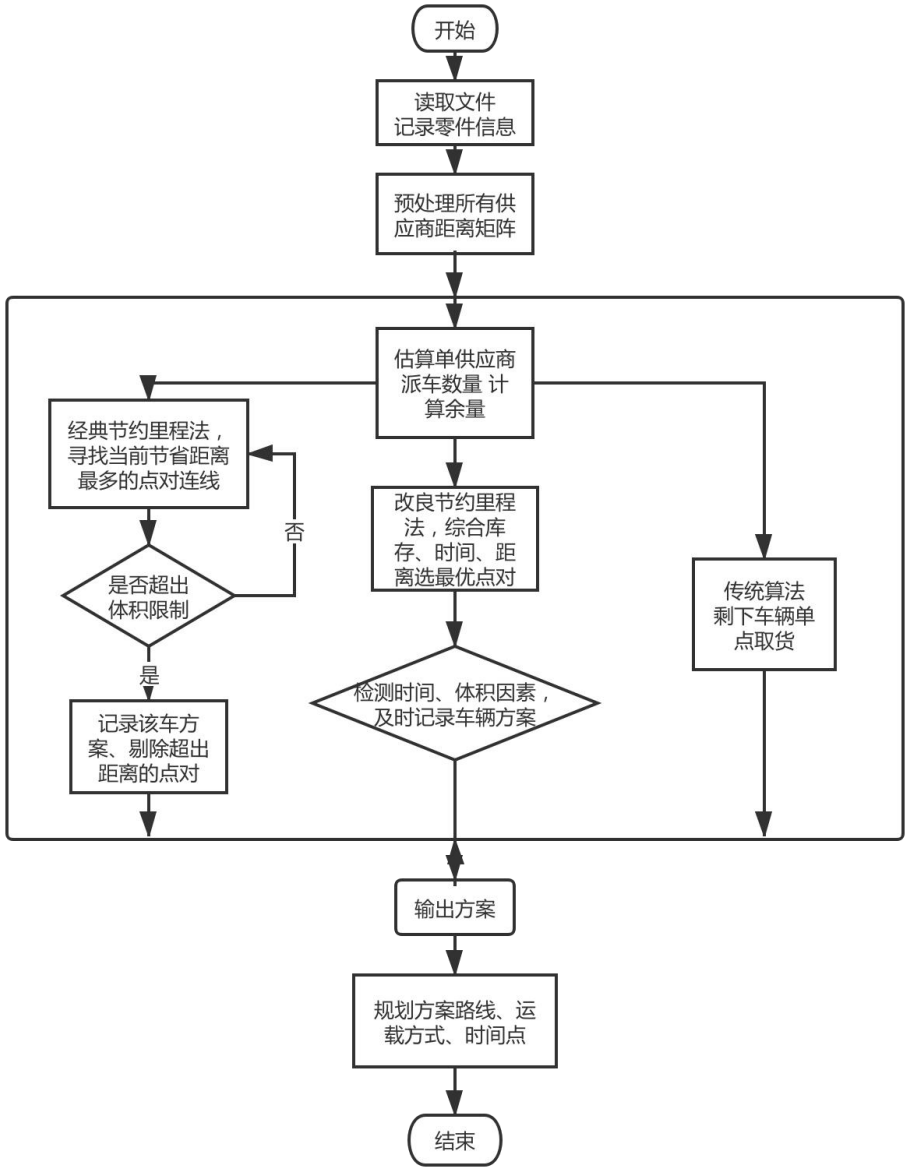


图 6-4 算法实现流程图

具体代码部分如下：

（1）从 excel 读入数据并将计算结果写到 excel 中

```
import pandas as pd
import math


class Data:
```

```
"""
    values 0:零件号
           1:零件名
           2:供应商号
           5:提货点地址
           10:包装长
           11:包装宽
           12:包装高
           13:装箱数/包装
           14:每车用量
           16:频数


    pos 0:供应商编号
        2:经度
        3:维度
"""

# 从 excel 文件读取并保存在一个二维数组
def __init__(self):

    # 获取案例 12 原始数据

    input_file1 = '12.xls'
    df = pd.read_excel(input_file1, sheet_name=0)
    self.values = df.values

    # 获取所有供应商地址数据
```

```
            input_file2 = 'position.xlsx'
            df = pd.read_excel(input_file2, sheet_name=0)
            self.pos = df.values

            id_to_address = []
            for i in range(len(self.pos)):
                for j in range(len(self.values)):
                    if self.pos[i, 0] == self.values[j, 2]:
                        id_to_address.append([self.values[j,       4],
self.values[j, 5]])
                        break
            self.id_to_address = id_to_address

        # 按照经纬度计算实际曼哈顿距离
        def get_dis(self, x1, y1, x2, y2):

            # 地球半径
            r = 6378.137
            # 圆周率
            # pi = 3.1415926
            pi = math.pi

            # 计算经度距离
            d1 = abs(y1 - y2) / 180 * pi * r

            # 计算纬度距离
            r_new = r * math.cos((y1 + y2) / 2 / 180 * pi)
            d2 = abs(x1 - x2) / 180 * pi * r_new
```

```
        return d1 + d2

    # 返回经纬度, 距离矩阵和产品信息
    def get_info(self):

        # x 经度 y 纬度
        x = []
        y = []
        info = []
        for i in range(len(self.pos)):
            x.append(self.pos[i, 2])
            y.append(self.pos[i, 3])
            tmp = []
            for j in range(len(self.values)):
                if self.pos[i, 0] == self.values[j, 2]:
                    tmp2 = list()
                    tmp2.append(self.values[j, 0])
                    tmp2.append(int(self.values[j, 10]))
                    tmp2.append(int(self.values[j, 11]))
                    tmp2.append(int(self.values[j, 12]))
                    tmp2.append(int(self.values[j, 13]))
                    tmp2.append(int(self.values[j, 14]))
                    tmp2.append(int(self.values[j, 16]))
                    tmp.append(tmp2)
            info.append(tmp)

        # info 第一维供应商 第二维对应零件集合
        #       第三维零件信息 0 零件号 1 长 2 宽 3 高 4 装箱数 5
每车用量 6 频数
```

```
        # dis[i, j]代表 i 到 j 的曼哈顿距离，与 info 中的一一对应
        dis = []
        for i in range(len(self.pos)):
            tmp = []
            for j in range(len(self.pos)):
                tmp.append(self.get_dis(x[i], y[i], x[j], y[j]))
            dis.append(tmp)
        return x, y, dis, info


# test code

# my_data = Data()
# print(my_data.values)
# a = []
# for i in range(len(my_data.values)):
#     a1 = int(my_data.values[i, 10])
#     a2 = int(my_data.values[i, 11])
#     a3 = int(my_data.values[i, 12])
#     a.append(a1 * a2 * a3 / 1000)
#
# a = set(a)
# print(a)
# print(len(a))
# 620 种体积的箱子
# print(f)
# print(len(set(f)))
# print(my_data.pos)
```

```
    # x, y, dis, info = my_data.get_info()
    # print(x)
    # print(y)
    # print(info)
    # print(dis)
    # print(len(x), len(y), len(dis), len(info))
    # my_data = Data()
    # print(len(my_data.id_to_address))
```

（2）问题解决器

```
import data
import result
import math
import numpy as np


# 问题解决框架
class Solver:

    def __init__(self, car_x=12, car_y=2.35, car_z=2.5, car_coe=0.9,
machine_x=None, machine_y=None, car_num=600,
                 pro_num=192):
        """
        :param car_x: 单个小卡车长
        :param car_y: 单个小卡车宽
        :param car_z: 单个小卡车高 选取了一种常用尺寸
        :param car_coe: 容量安全系数
        :param machine_x: 主机厂经度
        :param machine_y: 主机厂纬度 如没有按照平均计算
        :param car_num: 平均车的日产量 年产量除以三百天计算
```

```
:param pro_num: 供应商数量

"""

# 提取 excel 信息
my_data = data.Data()
x, y, dis, info = my_data.get_info()
self.my_data = my_data
self.x = x
self.y = y
self.dis = dis
self.info = info

self.car_x = car_x
self.car_y = car_y
self.car_z = car_z
self.car_coe = car_coe
self.pro_num = pro_num
self.car_num = car_num

if machine_x is None:
    tmp = 0
    for i in range(pro_num):
        tmp += x[i]
    self.machine_x = tmp / pro_num
else:
    self.machine_x = machine_x

if machine_y is None:
```

```
            tmp = 0
            for i in range(pro_num):
                tmp += y[i]
            self.machine_y = tmp / pro_num
        else:
            self.machine_y = machine_x

    # 传统运输方式 简单体积 以里程为代价 不管频次/时间窗
    def get_trad_simpleV_mileageCost_noFre(self):

        total_dis_tmp = 0
        total_car_fre = 0
        for i in range(self.pro_num):
            dis = self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[i], y2=self.y[i])
            now_set = self.info[i]
            total_v = 0
            for part in now_set:
                total_v += part[1]/1000 * part[2]/1000 * part[3]/1000 *
self.car_num * part[5] / part[4]
            total_car_fre += math.ceil(total_v / (self.car_x *
self.car_y * self.car_z * self.car_coe))
            total_dis_tmp += dis * 2 * math.ceil(total_v / (self.car_x
* self.car_y * self.car_z * self.car_coe))
            # print('test', i, total_v)
            # if i == 191:
            #     print(now_set)

        res = result.Result()
```

```
        res.total_dis = total_dis_tmp
        res.total_car_fre = total_car_fre
        return res

    # 节约里程法 简单体积 以里程为代价 不管频次/时间窗
    def get_saveDis_simpleV_mileageCost_noFre(self):

        total_dis_tmp = 0
        total_car_fre = 0
        per_car = self.car_x * self.car_y * self.car_z * self.car_coe

        left = []
        for i in range(self.pro_num):
            dis = self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[i], y2=self.y[i])
            now_set = self.info[i]
            total_v = 0
            for part in now_set:
                total_v += part[1] / 1000 * part[2] / 1000 * part[3] /
1000 * self.car_num * part[5] / part[4]
            total_car_fre += math.floor(total_v / per_car)
            left.append(total_v - math.floor(total_v / per_car) *
per_car)
            total_dis_tmp += dis * 2 * math.floor(total_v / per_car)
        mid_dis = total_dis_tmp

        tag = []
        for i in range(self.pro_num):
            tag.append(0)
```

```
        for i in range(self.pro_num):
            max_v = 0
            now = -1
            tmp_v = per_car
            for j in range(self.pro_num):
                if tag[j] == 0 and left[j] > max_v:
                    now = j
                    max_v = left[j]
            if now == -1:
                break

            tmp_v -= max_v
            left[now] = 0
            tag[now] = 1
            tmp_dis     =     self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[now], y2=self.y[now])
            # pri = []
            # pri.append(tmp_dis)
            while tmp_v > 0:
                min_d = 100000000000
                now_tmp = -1
                for j in range(self.pro_num):
                    if tag[j] == 0 and self.dis[now][j] < min_d and
tmp_v >= left[j]:
                        now_tmp = j
                        min_d = self.dis[now][j]
                if now_tmp == -1:
                    break
                tmp_dis += min_d
```

```
                # pri.append(min_d)
                if left[now_tmp] > tmp_v:
                    tmp_v = 0
                    left[now_tmp] -= tmp_v
                else:
                    tmp_v -= left[now_tmp]
                    left[now_tmp] = 0
                    tag[now_tmp] = 1
                now = now_tmp
            tmp_dis     +=     self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[now], y2=self.y[now])
            #       pri.append(self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[now], y2=self.y[now]))
            # print(pri)
            total_dis_tmp += tmp_dis
            total_car_fre += 1

        res = result.Result()
        res.total_dis = total_dis_tmp
        res.total_car_fre = total_car_fre
        res.mid_dis = mid_dis
        return res

    # 节约里程法 简单体积 以里程为代价 管频次/时间窗
    def get_saveDis_simpleV_mileageCost_Fre(self, storagePrice_coe =
0.042, disPrice_coe = 5.2):

        # 设定货车出发时间 0 点 工作时间点（8-17） 8 12 10 14 11 9
        # 设定平均装货时间 12 分钟/供应商 货车平均时速 60km/h
```

```
        avg_v = 60
        avg_loading_time = 0.333333
        duetime_point = [8, 9, 10, 11, 12, 14]
        duetime_map = [[0], [0, 4], [0, 2, 4], [0, 2, 4, 5], [0, 2, 3,
4, 5], [0, 1, 2, 3, 4, 5]]
        start_point = 0
        car_plan = []
        plan_timezone = []
        plan_timepoint = []
        total_dis_tmp = 0
        total_car_fre = 0
        storage_timePlusV = 0

        # 按简单体积分割每个供应商的产品和时间
        left = []
        for i in range(self.pro_num):
            tmp_per_left = [0, 0, 0, 0, 0, 0]
            now_set = self.info[i]
            for part in now_set:
                for j in range(part[6]):
                    tmp_per_left[duetime_map[part[6]-1][j]]        +=
self.car_num * part[5] * part[1] / 1000 * part[2] / 1000\
                                        * part[3] / 1000 / part[4] /
part[6]
            for j in range(6):
                storage_timePlusV += tmp_per_left[j] * (duetime_point[j]
- start_point)
            left.append(tmp_per_left)
        # print(left)
```

```
        # 一部分直送 直接剔除
        per_car = self.car_x * self.car_y * self.car_z * self.car_coe
        for i in range(self.pro_num):
            sum_v = np.sum(left[i])
            per_car_number = math.floor(sum_v / per_car)
            car_route = [i]
            for j in range(per_car_number):
                car_plan.append(car_route)
                time_tmp = self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[i],
                                                y2=self.y[i]) / avg_v
                plan_timezone.append(2 * (time_tmp + avg_loading_time))
                storage_timePlusV -= 2 * (time_tmp + avg_loading_time)
* per_car
                plan_timepoint.append([time_tmp, 2*time_tmp +
avg_loading_time])
            total_car_fre += per_car_number
            total_dis_tmp += 2 * per_car_number *
self.my_data.get_dis(x1=self.machine_x, y1=self.machine_y,

x2=self.x[i], y2=self.y[i])
            tmp_v = per_car_number * per_car
            for j in range(6):
                if tmp_v >= left[i][j]:
                    tmp_v -= left[i][j]
                    left[i][j] = 0
                else:
                    left[i][j] -= tmp_v
```

```
                tmp_v = 0
        mid_dis = total_dis_tmp
        # print(mid_dis)

        # print(left)

        # 可以进行精细装填 剔除高体积部分

        # 可以进行聚类

        # 按照符合时间的节约里程法进行贪心
        per_car = self.car_x * self.car_y * self.car_z * self.car_coe
        # t 数组表示该边是否可行 0 可行 1 不可行
        t = []
        for i in range(self.pro_num):
            tmp_t = []
            for j in range(self.pro_num):
                d1      =      self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[i], y2=self.y[i])
                d2      =      self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[j], y2=self.y[j])
                if d1 + d2 > self.dis[i][j]:
                    tmp_t.append(0)
                else:
                    tmp_t.append(1)
            t.append(tmp_t)

        tag = []
        for i in range(self.pro_num):
```

```
            tag.append([0, 1, 1, 1, 1, 1])
            left[i][0] = np.sum(left[i])

        # print(tag)
        for i in range(self.pro_num):
            # 初始化
            tmp_v = per_car
            tmp_time = avg_loading_time
            tmp_dis = 0
            plan = []
            tmp_plan_timepoint = []
            # 选最大的
            max_v = 0
            min_dis_point_j = -1
            min_dis_point_k = -1
            for j in range(self.pro_num):
                for k in range(6):
                    if tag[j][k] == 0 and left[j][k] > max_v:
                        max_v = left[j][k]
                        min_dis_point_j = j
                        min_dis_point_k = k
            if min_dis_point_j == -1:
                break
            tmp_time    +=    self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[min_dis_point_j],

y2=self.y[min_dis_point_j]) / avg_v + avg_loading_time
            tmp_plan_timepoint.append(tmp_time - 2 * avg_loading_time)
            tmp_v -= left[min_dis_point_j][min_dis_point_k]
```

```
            tag[min_dis_point_j][min_dis_point_k] = 1
            left[min_dis_point_j][min_dis_point_k] = 0
            plan.append(min_dis_point_j)
            tmp_dis    +=    self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[min_dis_point_j],

y2=self.y[min_dis_point_j])

            # 不停的选最优的
            now_point = min_dis_point_j
            due_time = duetime_point[min_dis_point_k]
            while True:
                min_dis = 10000
                min_dis_point_j = -1
                min_dis_point_k = -1
                for j in range(self.pro_num):
                    for k in range(6):
                        if tag[j][k] == 0 and self.dis[now_point][j] <
min_dis and t[now_point][j] > -1 and tmp_v > left[j][k]\
                                and due_time - start_point > tmp_time +
avg_loading_time + self.dis[now_point][j] / avg_v\
                                    +
self.my_data.get_dis(x1=self.machine_x,            y1=self.machine_y,
x2=self.x[j], y2=self.y[j]) / avg_v:
                                    min_dis = self.dis[now_point][j]
                                    min_dis_point_j = j
                                    min_dis_point_k = k
                if min_dis_point_j == -1:
                    break
```

```
                tmp_time += min_dis / avg_v + avg_loading_time
                tmp_plan_timepoint.append(tmp_time - 2 * avg_loading_time)
                now_point = min_dis_point_j
                if duetime_point[min_dis_point_k] < due_time:
                    due_time = duetime_point[min_dis_point_k]
                tmp_v -= left[min_dis_point_j][min_dis_point_k]
                tag[min_dis_point_j][min_dis_point_k] = 1
                left[min_dis_point_j][min_dis_point_k] = 0
                plan.append(min_dis_point_j)
                tmp_dis += min_dis
            tmp_time += self.my_data.get_dis(x1=self.machine_x, y1=self.machine_y, x2=self.x[now_point],
                                             y2=self.y[now_point]) / avg_v
            tmp_plan_timepoint.append(tmp_time - avg_loading_time)
            tmp_dis += self.my_data.get_dis(x1=self.machine_x, y1=self.machine_y, x2=self.x[now_point],
                                            y2=self.y[now_point])
            # print(tmp_dis)
            total_dis_tmp += tmp_dis
            total_car_fre += 1
            car_plan.append(plan)
            plan_timezone.append(tmp_time)
            plan_timepoint.append(tmp_plan_timepoint)
            storage_timePlusV -= tmp_time * (per_car - tmp_v)

        # print(tag)
        # print(car_plan)
```

```
        # 返回运行结果
        res = result.Result()
        res.total_dis = total_dis_tmp
        res.total_car_fre = total_car_fre
        res.mid_dis = mid_dis
        res.car_plan = car_plan
        res.plan_timezone = plan_timezone
        res.plan_timepoint = plan_timepoint
        # print(total_dis_tmp * disPrice_coe)
        res.total_price = storage_timePlusV * storagePrice_coe +
total_dis_tmp * disPrice_coe
        return res

    # 节约里程库存法 简单体积 以里程和库存为代价(不允许缺货) 管频次/
时间窗
    def             get_saveDisSto_simpleV_mileageStockCost_Fre(self,
storagePrice_coe = 0.042, disPrice_coe = 5.2):

        # 设定货车出发时间 0 点 工作时间点（8-17） 8 12 10 14 11 9
        # 设定平均装货时间 12 分钟/供应商 货车平均时速 60km/h
        # 里程价格系数 5.2 元/km 库存价格系数 0.042 元/(平方米*小时)
        storage_timePlusV = 0
        avg_v = 60
        avg_loading_time = 0.333333
        duetime_point = [8, 9, 10, 11, 12, 14]
        duetime_map = [[0], [0, 4], [0, 2, 4], [0, 2, 4, 5], [0, 2, 3,
4, 5], [0, 1, 2, 3, 4, 5]]
        start_point = 0
        car_plan = []
```

```
        plan_timezone = []
        plan_timepoint = []
        total_dis_tmp = 0
        total_car_fre = 0

        # 按简单体积分割每个供应商的产品和时间
        left = []
        for i in range(self.pro_num):
            tmp_per_left = [0, 0, 0, 0, 0, 0]
            now_set = self.info[i]
            for part in now_set:
                for j in range(part[6]):
                    tmp_per_left[duetime_map[part[6] - 1][j]] +=
self.car_num * part[5] * part[1] / 1000 * part[2] / \
                                                                1000
* part[3] / 1000 / part[4] / part[6]
            for j in range(6):
                storage_timePlusV += tmp_per_left[j] * (duetime_point[j]
- start_point)
            left.append(tmp_per_left)
        # print(left)

        # 一部分直送 直接剔除
        per_car = self.car_x * self.car_y * self.car_z * self.car_coe
        for i in range(self.pro_num):
            sum_v = np.sum(left[i])
            per_car_number = math.floor(sum_v / per_car)
            car_route = [i]
            for j in range(per_car_number):
```

```
                car_plan.append(car_route)
                time_tmp = self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[i],
                                                y2=self.y[i]) / avg_v
                plan_timezone.append(2 * (time_tmp + avg_loading_time))
                plan_timepoint.append([time_tmp, 2 * time_tmp +
avg_loading_time])
                storage_timePlusV -= 2 * (time_tmp + avg_loading_time)
* per_car
            total_car_fre += per_car_number
            total_dis_tmp += 2 * per_car_number *
self.my_data.get_dis(x1=self.machine_x, y1=self.machine_y,

x2=self.x[i], y2=self.y[i])
            tmp_v = per_car_number * per_car
            for j in range(6):
                if tmp_v >= left[i][j]:
                    tmp_v -= left[i][j]
                    left[i][j] = 0
                else:
                    left[i][j] -= tmp_v
                    tmp_v = 0
        mid_dis = total_dis_tmp
        # print(mid_dis)

        # print(left)

        # 可以进行精细装填 剔除高体积部分
```

```
        # 可以进行聚类

        # 按照符合时间的节约里程法进行贪心
        per_car = self.car_x * self.car_y * self.car_z * self.car_coe
        # t 数组表示该边是否可行 0 可行 1 不可行
        t = []
        for i in range(self.pro_num):
            tmp_t = []
            for j in range(self.pro_num):
                d1      =      self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[i], y2=self.y[i])
                d2      =      self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[j], y2=self.y[j])
                if d1 + d2 > self.dis[i][j]:
                    tmp_t.append(0)
                else:
                    tmp_t.append(1)
            t.append(tmp_t)

        tag = []
        for i in range(self.pro_num):
            tag.append([0, 0, 0, 0, 0, 0])
            # left[i][0] = np.sum(left[i])

        # print(tag)
        for i in range(self.pro_num):
            # 初始化
            tmp_v = per_car
            tmp_time = avg_loading_time
```

```
            tmp_dis = 0
            plan = []
            tmp_plan_timepoint = []
            # 选最大的
            max_v = 0
            min_dis_point_j = -1
            min_dis_point_k = -1
            for j in range(self.pro_num):
                for k in range(6):
                    if tag[j][k] == 0 and left[j][k] > max_v:
                        max_v = left[j][k]
                        min_dis_point_j = j
                        min_dis_point_k = k
            if min_dis_point_j == -1:
                break
            tmp_time    +=    self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[min_dis_point_j],

y2=self.y[min_dis_point_j]) / avg_v + avg_loading_time
            tmp_plan_timepoint.append(tmp_time - 2 * avg_loading_time)
            tmp_v -= left[min_dis_point_j][min_dis_point_k]
            tag[min_dis_point_j][min_dis_point_k] = 1
            left[min_dis_point_j][min_dis_point_k] = 0
            plan.append(min_dis_point_j)
            tmp_dis    +=    self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[min_dis_point_j],

y2=self.y[min_dis_point_j])
```

```
            # 不停的选最优的
            now_point = min_dis_point_j
            due_time = duetime_point[min_dis_point_k]
            while True:
                min_dis = 10000
                min_dis_point_j = -1
                min_dis_point_k = -1
                for j in range(self.pro_num):
                    for k in range(6):
                        if tag[j][k] == 0 and self.dis[now_point][j] <
min_dis and t[now_point][j] > -1 and tmp_v > \
                                left[j][k] \
                                and due_time - start_point > tmp_time +
avg_loading_time + self.dis[now_point][
                            j] / avg_v \
                                +
self.my_data.get_dis(x1=self.machine_x,            y1=self.machine_y,
x2=self.x[j],
                                                       y2=self.y[j])
/ avg_v:
                            min_dis = self.dis[now_point][j]
                            min_dis_point_j = j
                            min_dis_point_k = k
                if min_dis_point_j == -1:
                    break
                tmp_time += min_dis / avg_v + avg_loading_time
                tmp_plan_timepoint.append(tmp_time     -     2     *
avg_loading_time)
                now_point = min_dis_point_j
```

```
                if duetime_point[min_dis_point_k] < due_time:
                    due_time = duetime_point[min_dis_point_k]
                tmp_v -= left[min_dis_point_j][min_dis_point_k]
                tag[min_dis_point_j][min_dis_point_k] = 1
                left[min_dis_point_j][min_dis_point_k] = 0
                plan.append(min_dis_point_j)
                tmp_dis += min_dis
            tmp_time    +=    self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[now_point],
                                                   y2=self.y[now_point])  /
avg_v
            tmp_plan_timepoint.append(tmp_time - avg_loading_time)
            tmp_dis    +=    self.my_data.get_dis(x1=self.machine_x,
y1=self.machine_y, x2=self.x[now_point],
                                                  y2=self.y[now_point])
            # print(tmp_dis)
            total_dis_tmp += tmp_dis
            total_car_fre += 1
            car_plan.append(plan)
            plan_timezone.append(tmp_time)
            storage_timePlusV -= tmp_time * (per_car - tmp_v)
            plan_timepoint.append(tmp_plan_timepoint)

        # print(tag)
        # print(car_plan)
        # 返回运行结果
        res = result.Result()
        res.total_dis = total_dis_tmp
        res.total_car_fre = total_car_fre
```

```
        res.mid_dis = mid_dis
        res.car_plan = car_plan
        res.plan_timezone = plan_timezone
        res.plan_timepoint = plan_timepoint
        # print(total_dis_tmp * disPrice_coe)
        res.total_price = storage_timePlusV * storagePrice_coe / 2 +
total_dis_tmp * disPrice_coe
        return res


# test code
if __name__ == "__main__":
    #       test       for       get_trad_simpleV_mileageCost_noFre
get_saveDis_simpleV_mileageCost_noFre
    sol = Solver()
    res1 = sol.get_trad_simpleV_mileageCost_noFre()
    print('传统算法', str(res1.total_dis)+'km')
    # print(res.total_car_fre)
    res2 = sol.get_saveDis_simpleV_mileageCost_noFre()
    print('节约里程法', str(res2.total_dis)+'km')
    #       print('      节      约      百      分      数      ',
(res1.total_dis-res2.total_dis)/res1.total_dis)
    #     print('    实    际    节    约    百    分    数    ',
(res1.total_dis-res2.total_dis)/(res1.total_dis - res2.mid_dis))
    # print(res.total_car_fre)
    #
    # test for get_saveDis_simpleV_mileageCost_Fre
    # sol = Solver()
    res3 = sol.get_saveDis_simpleV_mileageCost_Fre()
```

```
    print('节约里程法(考虑时间)', str(res3.total_dis)+'km')
    print('节约里程法(考虑时间)', str(res3.total_price)+'元')
    #       print('       节       约       百       分       数       ',
(res1.total_dis-res3.total_dis)/res1.total_dis)
    #   print('  较  传  统  实  际  节  约  百  分  数  ',
(res1.total_dis-res3.total_dis)/(res1.total_dis - res3.mid_dis))
    #  print(' 较 节 约 里 程 法 实 际 节 约 百 分 数 ',
(res2.total_dis-res3.total_dis)/(res2.total_dis - res3.mid_dis))
    # print(res3.total_car_fre)
    res4 = sol.get_saveDisSto_simpleV_mileageStockCost_Fre()
    print('节约库存里程法(考虑时间)', str(res4.total_dis)+'km')
    print('节约库存里程法(考虑时间)', str(res4.total_price)+'元')
```

（3）再次从 excel 中读入数据

```
import pandas as pd
import math


class Data:

    """
        values 0:零件号
               1:零件名
               2:供应商号
               5:提货点地址
               10:包装长
               11:包装宽
               12:包装高
               13:装箱数/包装
               14:每车用量
```

```
                16:频数


        pos 0:供应商编号
            2:经度
            3:维度
    """

    # 从 excel 文件读取并保存在一个二维数组
    def __init__(self):

        # 获取案例 12 原始数据

        input_file1 = '12.xls'
        df = pd.read_excel(input_file1, sheet_name=0)
        self.values = df.values

        # 获取所有供应商地址数据

        input_file2 = 'position.xlsx'
        df = pd.read_excel(input_file2, sheet_name=0)
        self.pos = df.values

        id_to_address = []
        for i in range(len(self.pos)):
            for j in range(len(self.values)):
                if self.pos[i, 0] == self.values[j, 2]:
                    id_to_address.append([self.values[j,       4],
self.values[j, 5]])
```

```
                break
        self.id_to_address = id_to_address

    # 按照经纬度计算实际曼哈顿距离
    def get_dis(self, x1, y1, x2, y2):

        # 地球半径
        r = 6378.137
        # 圆周率
        # pi = 3.1415926
        pi = math.pi

        # 计算经度距离
        d1 = abs(y1 - y2) / 180 * pi * r

        # 计算纬度距离
        r_new = r * math.cos((y1 + y2) / 2 / 180 * pi)
        d2 = abs(x1 - x2) / 180 * pi * r_new

        return d1 + d2

    # 返回经纬度，距离矩阵和产品信息
    def get_info(self):

        # x 经度 y 纬度
        x = []
        y = []
        info = []
        for i in range(len(self.pos)):
```

```
            x.append(self.pos[i, 2])
            y.append(self.pos[i, 3])
            tmp = []
            for j in range(len(self.values)):
                if self.pos[i, 0] == self.values[j, 2]:
                    tmp2 = list()
                    tmp2.append(self.values[j, 0])
                    tmp2.append(int(self.values[j, 10]))
                    tmp2.append(int(self.values[j, 11]))
                    tmp2.append(int(self.values[j, 12]))
                    tmp2.append(int(self.values[j, 13]))
                    tmp2.append(int(self.values[j, 14]))
                    tmp2.append(int(self.values[j, 16]))
                    tmp.append(tmp2)
            info.append(tmp)

        # info 第一维供应商 第二维对应零件集合
        #       第三维零件信息 0 零件号 1 长 2 宽 3 高 4 装箱数 5
每车用量 6 频数

        # dis[i, j]代表 i 到 j 的曼哈顿距离，与 info 中的一一对应
        dis = []
        for i in range(len(self.pos)):
            tmp = []
            for j in range(len(self.pos)):
                tmp.append(self.get_dis(x[i], y[i], x[j], y[j]))
            dis.append(tmp)
        return x, y, dis, info
```

```
    # test code


    # my_data = Data()
    # print(my_data.values)
    # a = []
    # for i in range(len(my_data.values)):
    #     a1 = int(my_data.values[i, 10])
    #     a2 = int(my_data.values[i, 11])
    #     a3 = int(my_data.values[i, 12])
    #     a.append(a1 * a2 * a3 / 1000)
    #
    # a = set(a)
    # print(a)
    # print(len(a))
    # 620 种体积的箱子
    # print(f)
    # print(len(set(f)))
    # print(my_data.pos)
    # x, y, dis, info = my_data.get_info()
    # print(x)
    # print(y)
    # print(info)
    # print(dis)
    # print(len(x), len(y), len(dis), len(info))
    # my_data = Data()
    # print(len(my_data.id_to_address))
```

（4）调整求解结果，向 excel 输出方案，数据可视化

```
import math
```

```
import data
import result
import solver


class Solution:

    def __init__(self, res, dat):
        """
        :param res:必须是 result.Result 类 用来存放求解结果
        :param dat:必须是 data.Data 类 用来存放原始数据

        """
        self.result = res
        self.data = dat
        self.car_plan = None

    def get_time(self, number):
        s = str(math.floor(number))
        left = number - math.floor(number)
        left = left * 60
        left = int(left)
        if left < 10:
            s = s + ':'+'0' + str(left)
        else:
            s = s + ':' + str(left)
        return s

    def get_name(self, rank):
```

```
    id_to_address = self.data.id_to_address
    if rank == -1:
        return '总厂'
    else:
        return id_to_address[rank][0].strip()

def get_address(self, rank):
    id_to_address = self.data.id_to_address
    if rank == -1:
        return '安吉物流零部件'
    else:
        return id_to_address[rank][1].strip()

# 根据求解方案 生成合适的派车方案
def sol_balance_mileageCost(self):
    car_plan = self.result.car_plan
    for plan in car_plan:
        plan.append(-1)
    plan_timezone = self.result.plan_timezone
    plan_timepoint = self.result.plan_timepoint
    seq = []
    num = len(plan_timepoint)
    for i in range(num):
        seq.append(i)
    for i in range(num):
        min_time = 100
        min_pos = -1
        for j in range(i+1, num, 1):
            if plan_timezone[seq[j]] < min_time:
```

```
                    min_time = plan_timezone[seq[j]]
                    min_pos = j
            tmp = seq[i]
            seq[i] = seq[min_pos]
            seq[min_pos] = tmp

        mid_car_plan = []
        timezone = 8
        first = 0
        last = num - 1
        while first <= last:
            tmp_car_plan = []
            tmp_timezone = timezone
            tmp_car_plan.append(seq[last])
            tmp_timezone -= plan_timezone[seq[last]]
            last -= 1
            while first <= last and tmp_timezone > plan_timezone[seq[first]]:
                tmp_car_plan.append(seq[first])
                tmp_timezone -= plan_timezone[seq[first]]
                first += 1
            mid_car_plan.append(tmp_car_plan)

        final_car_plan = []
        num = len(mid_car_plan)
        for i in range(num):
            tmp_car_plan = []
            tmp_timezone = 0
            for j in mid_car_plan[i]:
```

```
                for k in range(len(plan_timepoint[j])):

tmp_car_plan.append([self.get_time(plan_timepoint[j][k]              +
tmp_timezone),

self.get_name(car_plan[j][k]),

self.get_address(car_plan[j][k])])
                tmp_timezone += plan_timezone[j]
            final_car_plan.append(tmp_car_plan)
        self.car_plan = final_car_plan

    # 向 txt 输出列车时刻表
    def print_to_txt(self, file_name):
        f = open(file_name, "w")
        num = len(self.car_plan)
        for i in range(num):
            for point in self.car_plan[i]:
                print(point[0], point[1], point[2], file=f)
            print('', file=f)
            print('----------------------------car ' + str(i+1) +
'-----------------------------', file=f)
            print('', file=f)



# test code
# test_solver = solver.Solver()
# test_data = data.Data()
# test_result = test_solver.get_saveDis_simpleV_mileageCost_Fre()
```

```
# test_solution = Solution(test_result, test_data)
# # print(test_solution.result.plan_timezone)
# test_solution.sol_balance_mileageCost()
# test_solution.print_to_txt('car_solution.txt')
```

（5）求解结果记录

```
    class Result:

        def __init__(self):

            # 总里程
            self.total_dis = None

            # 总车次
            self.total_car_fre = None

            # 中段总里程
            self.mid_dis = None

            # 运输路线方案
            self.car_plan = None

            # 运输方案时长 时间点
            self.plan_timezone = None
            self.plan_timepoint = None

            # 运输方案截止时间
            self.plan_duetime = None

            # 运输方案预计库存里程成本
```

```
self.total_price = None
```

7.灵敏度分析

7.1 库存及成本分析

汽车企业的库存是指存储状态下的零件，采用有效的库存策略，在合理占用场地面积和库存资金的前提下保持充足的库存从而安排连续性生产，对于汽车企业的发展起着至关重要的作用。

传统模式下，供应商凭经验送货，没有明确的管理制度，一般在这种情况下，为了提高货车的装载率以及租用面积的最大化利用，库存量最多时能够达到一个星期的量左右，导致整车厂的仓库占用率过高，且库存水平波动较大，物料流动性低，不便于整车厂仓储面积及相关成本的控制和管理。

而实行 Milk-Run 模式则能够实现取货货车定时定次去供应商处取货，并及时送回整车厂直接上线或拆包上线，整车厂物流相关人员与集货供应商提前进行集货信息的保证了生产信息与供应信息的即时更新。因此，零件库存量只需满足取货周期所需要的货物周转即可，一般将其控制在两天左右，即以 2 天作为安全库存。既降低了整车厂的库存压力，缓解仓储面积紧张的现象，又能够满足连续性生产线的需求，整合了物流资源，在路径优化的基础上合理安排物流车辆，有效实施了仓储和物流相关成本的节约与管控。下图 7-1 为库存量变化对比图：

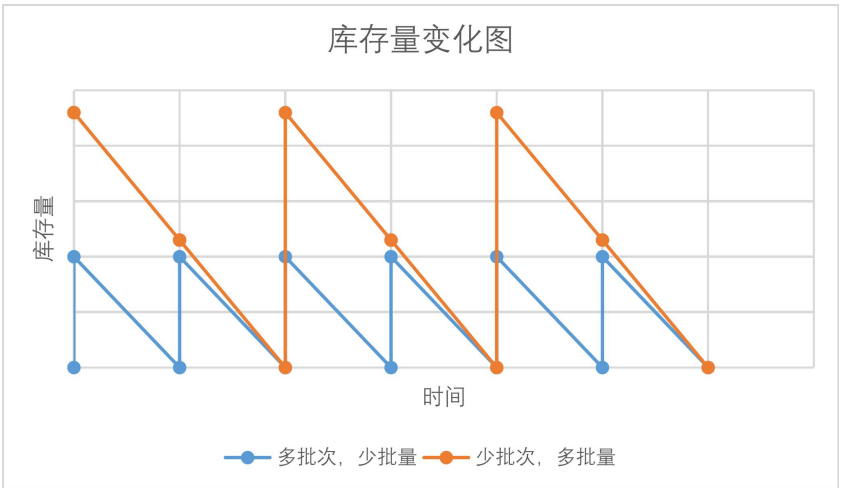


图 7-1 库存量变化对比图

如图 7-2 为具体的库存策略图：

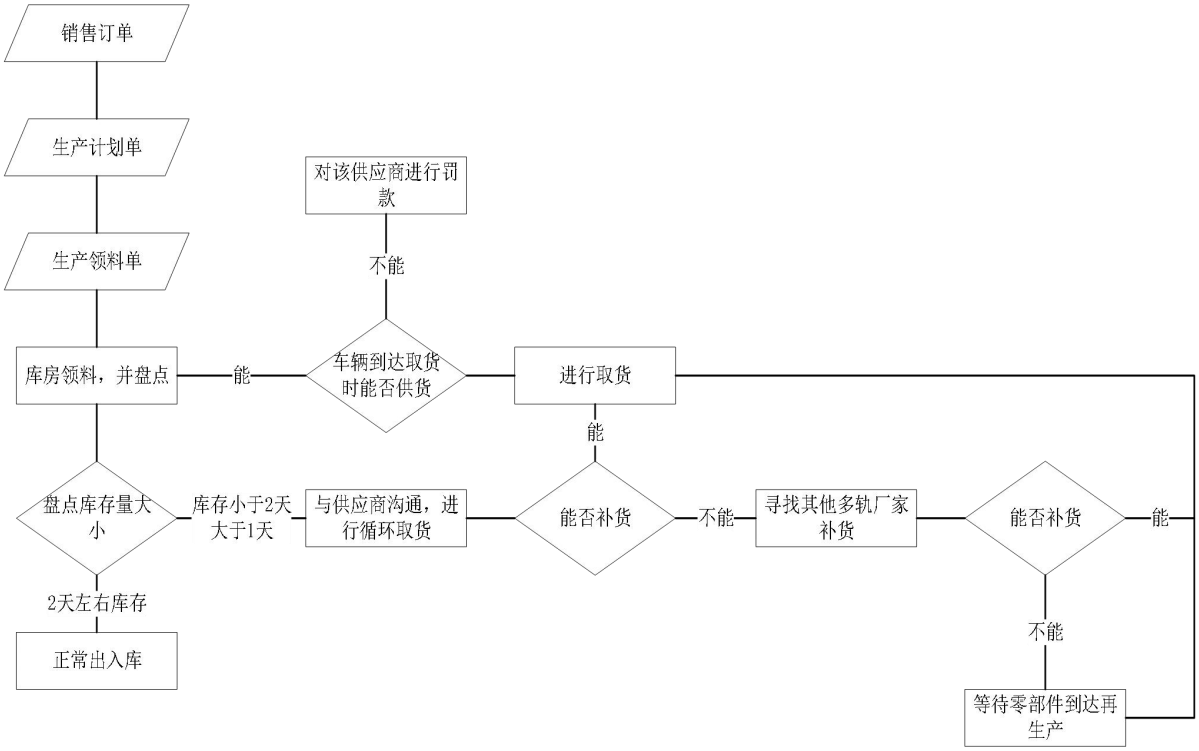


图 7-2 库存策略图

7.2 成本分析

资金是企业发展壮大的重要环节，成本控制是资金链中企业关注的重点，企业的发展扩大和业务周转必然需要资金的支持。

传统的汽车零件入场模式下，整车厂直接跟供应商进行物流费用的清算，包括零件费用和仓库存储管理费用，物流费用直接包含在零件费用之中，并没有进行分离，只有供应商清楚实际花费，忽视了物流成本的管理与控制，从而产生不必要的成本支出。

而在 Milk-Run 模式下，物流由整车厂统一管理配置，实现了物流成本的剥离，根据生产计划的需要和各方面协调，进行物流资源的分配和管理，整车厂掌握了物流管理的主动权。图 7-3 为整个过程的成本费用图：

图 7-3 成本费用图

从上图可知，整车厂采购零件的过程中主要考虑对库存管理费用和物流运输费用的控制，一般物流运输采用第三方物流的形式，由此可见，整车厂在成本的管理和控制上占据主导地位。

通过对成本结构的分析，对 Milkrun 模式下的具体成本费用细化分析，设置相关参数如下：

a_i——每辆车的里程和（i=1,2,3…n_1，n_1表示 Milkrun 模式下节约里程法使用的车辆总数）

b_{ij}——每批零件的体积（i=1,2,3…n_2，n_2表示零件的数量；j=1,2,3…p，p 表示零件运输批次）

t_{ij}——每批零件到的时刻（i=1,2,3…n_2，n_2表示零件的数量；j=1,2,3…p，p 表示零件运输批次）

C——Milkrun 模式下的总成本

$k_{里程}$——运输费用计算采用的里程系数

$k_{库存}$——库存管理费用计算采用的库存系数

在 Milkrun 入场模式下，利用考虑时间窗和库存的节约里程法得到的总成本：

$$C = k_{里程}\sum_{i=1}^{n1} a_i + k_{库存}\sum_{i=1}^{n2}(12 * \sum_{j=1}^{p} b_{ij} - \sum_{j=1}^{p} b_{ij} * t_{ij})$$

1. 不考虑时间窗的节约里程法

通过节约里程法的计算，可得出 Milkrun 的取货路径和距离为 50758.48km（结果保留两位小数），与传统算法的 58459.35km 比较可知，实行循环取货可比自主送货节约 13.2%的运输距离，从而缩短了运输成本。同时提高了运输车辆的利用率，减少尾气排放，改善交通拥堵情况，既符合满足企业管理需要又社会环保理念。

2. 考虑时间窗的节约里程法

在合理的规划下，卡车按照既定的路线和时间依次到不同的供应商处收取货物，既能够缩短运输距离也能够对整车厂和供应商的货物交互起到更好的调控作用，有利于双方的工作安排和人力物力时间等资源的节约。

通过代码计算输出可知，考虑时间窗的 Milkrun 总里程为 50756.95km，总成本为 295470.49 元。

7.3 输出数据分析

1. 改变整车厂的日产量，即整车厂的生产能力，得出如下两图。图 7-4 为传统供货模式、只考虑里程的 Milk-Run 模式及考虑库存和里程的 Milk-Run 模式这三种模式下的产量-总里程图，图 7-5 为考虑里程的 Milk-Run 模式及考虑库存和里程的 Milk-Run 模式下的产量—总成本图。

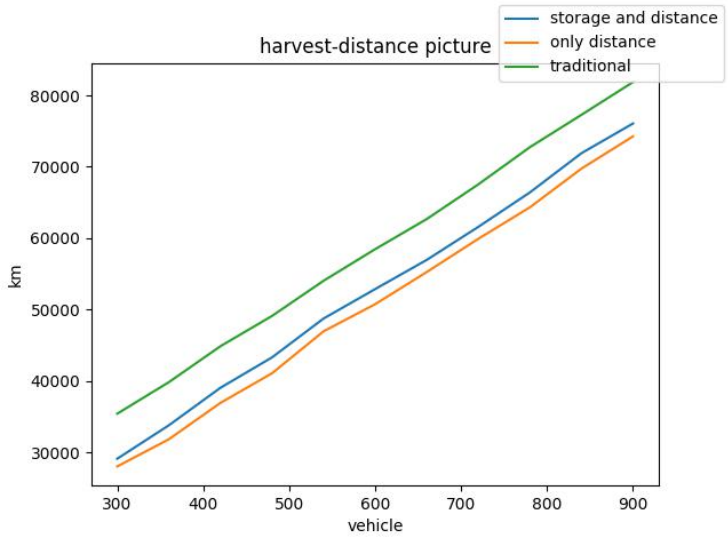


图 7-4 三种模式下的产量-总里程图

（传统供货模式、只考虑里程的 Milk-Run 模式及考虑库存和里程的 Milk-Run 模式）

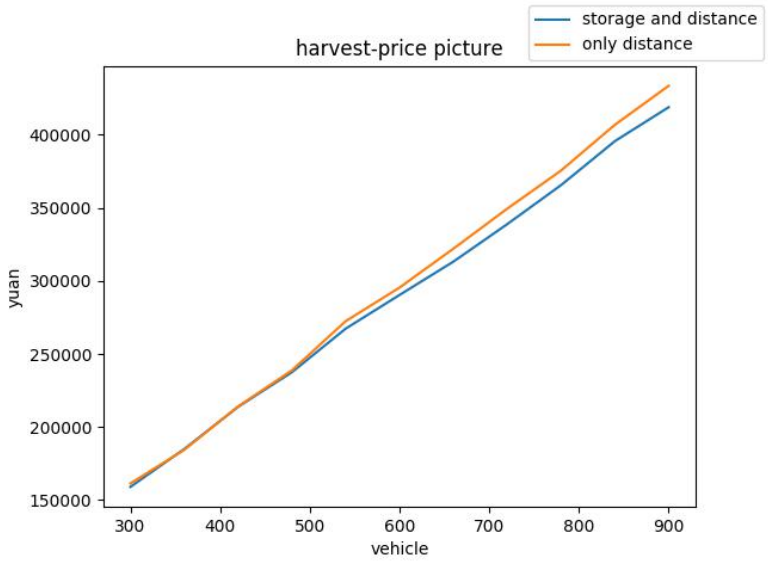


图 7-5 产量—总成本图

（考虑里程的 Milk-Run 模式及考虑库存和里程的 Milk-Run 模式）

由图表可知，随着整车厂的生产能力增强，所需的零件量增加，货车行驶的总里程数基本呈现线性增加，同时库存管理费用和物流运输费用也随之增加。传统模式下的总里程数最多，物流成本花费较高，资金占用严重；只考虑距离时利用节约里程法得出的总里程数最少。但综合两图，可以看出虽然综合考虑库存和距离时利用节约里程法得出的总里程数并非最优，但相对于只考虑距离时总成本降低了，且由输出数据可知，当日产量为 600 辆车时，增加里程 2115.1km，降低成本 5067.44 元。

2. 改变货车集装箱的规格，即物流运输能力，得出如下两图。分别是在其他因素不变的情况下，改变集装箱的规格和体积，取货过程中总里程和总成本的变化。

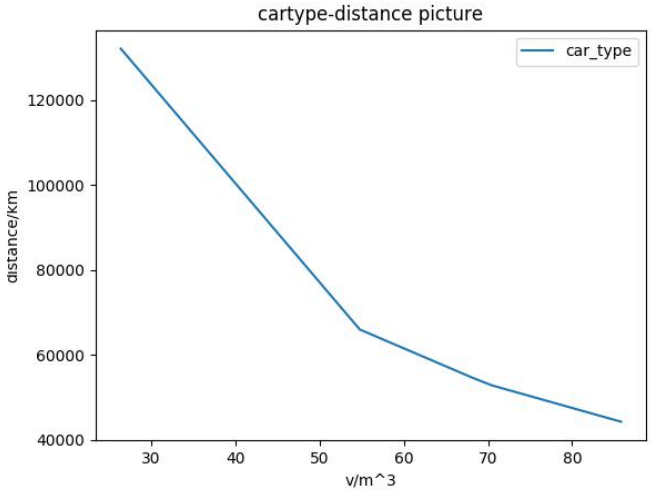


图 7-6 内体积—总里程图

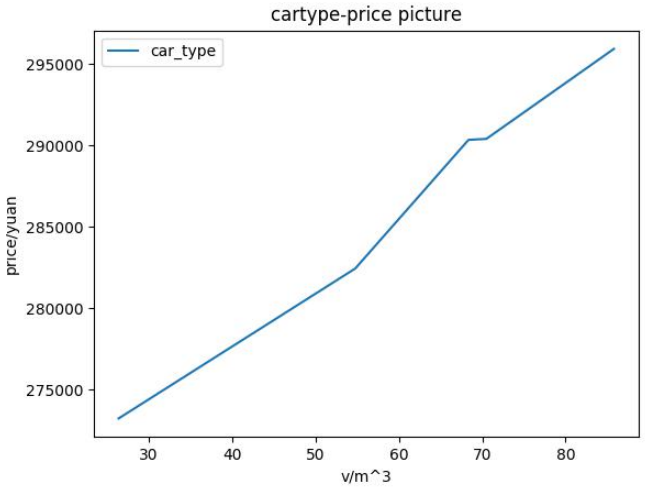


图 7-7 内体积—总成本图

由图表可知，随着集装箱运输体积，即物流运输能力的增大，总里程数会随之减小，总成本却在增加，说明虽然运输总里程减少了，运输成本减少，但由此造成的库存成本增加，导致总成本反而增加了，在实际问题的考虑中需要综合两方面因素选择合适的车型进行物流作业。

3. 改变整车厂的选址，即中心点的位置，得出图 7-8

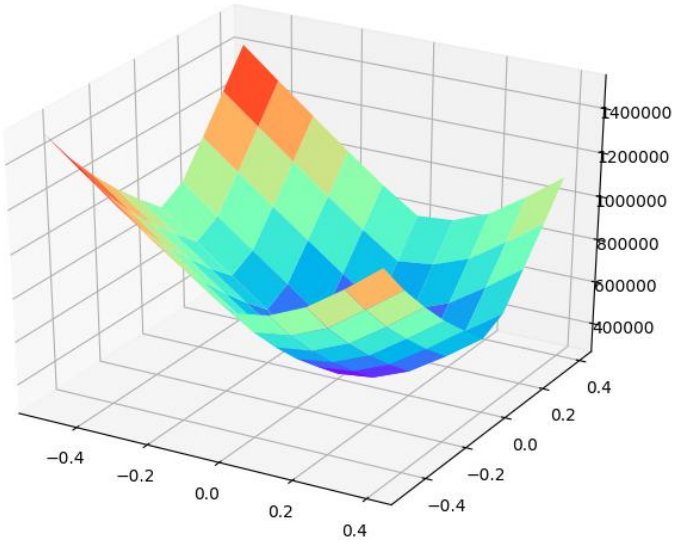


图 7-8 位置—总成本三维图

由三维图表可知，选址应该综合考虑总成本最低以及实际的客观地理因素，从而实现进一步的优化。

8.异常配送过程的应急与优化

8.1 异常取货流程的原因分析

实施循环取货模式下的汽车零部件配送过程中，一些异常运行流程通常会出现。现将造成这些异常的因素分成可控因素和不可控因素两大类。其中，不可控因素是指由于循环取货模式外部因素造成的取货异常，这种原因一般修复成本很高，汽车总装厂一般难以控制，比如突发的自然状况、政府对出入车辆的临时管制等。可控因素是指由于循环取货模式内部参与方造成的取货异常，总装厂可以通过自身的管理与优化、多方协调与控制，实现对这些因素的控制，从而保证生产的顺利进行。

8.1.1 不可控因素

在取货配送过程中，非自身原因而是外部原因造成的异常流程称为不可控因素。主要包括由于运输过程的意外突发情况和其他客观原因导致的取货流程异常。

（1）天气原因。如雾霾、大雨雪、洪涝、冰灾、地震等。天气原因将影响汽车总装厂的取货运输车辆的交通情况，使车辆不能按时到达各个供应商节点取货，延长了循环取货周期时间。

（2）交通管制。如重大活动或这事务发生时，政府对出入车辆的临时管制等。当相关公路限行时，会使取货车辆改走其他非最优线路，影响车辆到达时间。

(3)运输过程风险。运输频次增加，会提高交通拥堵或者事故概率。这时可能就需要专业物流公司，来执行循环取货工作，尽可能降低运输风险。

8.1.2 可控因素

可控因素是指造成流程异常的内部原因，主要包括供应商方面导致的流程异常，可以靠汽车总装厂的管理优化、积极协调与沟通，控制住这些因素。

供应商配合程度低。总装厂生产/销售不规律，需求量波动大，直接影响到供应商的备货时间和装卸货时间。导致时间窗不能按计划执行，根本无法追踪时间窗口的准时到货率。一般的解决方案是设定弹性的时间窗口，并且根据弹性时间窗口备一定库存。

零部件质量和数量问题。循环取货的司机验货时，主要检验质量和数量。零部件的质量，除了外观破损受淋的情况，其他问题是无法检验的。如果零部件件生产商对发动机、变速器等关键零部件质量的控制不到位，这会导致总装厂生产出来的汽车产品质量部不合格。数量方面，司机只负责小包装数，但是具体每个包装有多少零件，司机无法检查。

零部件成本不透明。有些供应商不愿意透明产品成本，避免因产品成本透明所带来对自身利益的损失。循环取货的成本降低体现在零部件单价的下降，而老的供应商在零部件定价过程中，并未剥离运输费用和包装费用。这时总装厂应更新物流供应商协议。

运输意外纠纷顾虑。供应商担心零部件字运输过程中发生意外所带来的责任纠纷和额外损失。针对此问题，在合同中明确取货不同参与方的责任，尽可能使流程细化，总装厂要更好发挥协调者作用。

装载率问题。循环取货模式，必须统一料箱规格，鼓励供应商使用塑料或者金属可重复使用的料箱，为节省空箱配送费用，可设计可折叠的料箱，在波包装

方面，对供应商提出明确的包装规范要求。

物料挪用风险。物料的挪用，零部件供应商要拆东补西，与供应商自己送货还是总装厂上门取货，没有直接关系。这时就需要依靠双方的采购合同协议和履约承诺来约束供应商行为。

8.2 不可控因素异常的应急方案

循环取货最重要的要求就是满足总装厂公司 JIT 中要求在正确的时间送正确的产品到正确的地方。在循环取货过程中设计到多家供应商，任何一家的流程延误就可能造成其他供应商的安排和损失，严重时还可能造成停线风险。

当出现因各种原因造成生产商的紧急加单、减单和并单等超过当天零部件正常需求量的 25%时；或当天气、道路等出现异常情况时；可立即启动应急方案。目前我们采用应急方案的主要方法主要包括：

(1) 调整路线，避开异常情况（大雾时段、造桥、阻塞路段等）；

(2) 调整路线，提前出发，将差异时间计算在内；

(3) 命令就近路线司机进行紧急援救；

(4) 申请供应商自运，或利用外部车辆(如租赁出租车)进行运输。

对于异常运行流程，可由专业人员根据实际情况，向系统手工输入异常的零部件需求信息，以帮助系统正常运行；或直接根据运行指南进行人工出单，以保证生产的顺利进行。

9.总结与展望

9.1 总结

随着中国汽车行业的不断发展和竞争加剧，加上传统的供应商直送入场物流方式因为送货成本高，汽车厂商库存量大，灵活性差，本文在阅读了大量文献和企业调研的基础上，先对本次项目的背景、目标、基本流程进行简要介绍，接着对入厂物流影响因素、库存及成本控制采用具体方法进行分析与总结，然后对 Milk run 模式的特点、相关理论与方法技术以及实施步骤、规划设计做了详尽阐

述，对基于 Milk-Run 的 SDVRPTW 的问题建模路径规划进行重点研究，由于分析数据量极为庞大，借助编程工具，用改进后的 CW 算法及进行求解，得出最优运输路线，通过和供应商直供模式的运输成本对比，验证了循环取货模式的可行性与经济性。本文研究方法和结果如下：

以汽车零部件入场物流成本的居高不下问题为基础，在柔性生产、精益生产、准时制供应的背景下，针对需求为多频次、小批量类型的汽车零部件特点，提出将循环取货模式应用于零部件入厂物流。

全面综合介绍了循环取货项目的具体实施条件和规划方案，以及该方案的核心问题-取货路径规划问题（VRP）以及目前学术界对该问题的求解方法的研究成果，本文选用的是改进后的 CW 节约算法对模型求解。

参考某汽车总装厂所需的零部件供应商的分布网点取货地址，以及具体零件的包装尺寸、送货频次等信息，首先阐述方案实施过程的车型、包装、装箱、责任划分、KPI 等指标多个要素（要改），其次建立循环取货路径路径规划模型(VRP)，并用改进后的 CW 节约法求得最优提货路径。

总结以上研究，通过循环取货与供应商直送模式运输成本对比，证明循环取货模式具有明显的优势和汽车入厂物流行业引入循环取货模式的必要性。

9.2 展望

Milk-Run 是一个复杂的运作系统，在实际运作时涉及到的环节和要考虑的因素很多，要研究和解决的问题较多，设计运输、库存、成本分析、运作管理、实时监控、信息系统等等各方面。限于能力和时间约束，本文仅对其中部分问题做了研究，还有很多方面有待进一步解决，主要包括以下几方面：

在路径优化中，本文仅考虑了一个汽车整装厂，单一的取货运输车型，并且假设当多辆车同时到达整装厂时，不会出现拥堵情况。在后期研究中，可以将更多符合实际情况的因素考虑进去，分析多车型情况，并且考虑车辆排队等待服务的情况。

在构建模型时，为了方便计算，对很多参数进行了简化处理，比如统一了运输车辆、车辆行驶速度、单位行驶成本和操作时间等。因此综合考虑各种复杂因素及进行行建模和算法求解还需要进一步研究，以更加符合现实情况。

在车辆装载方面，仅以零部件体积作为装载限制，没有考虑不同零部件的重量、包装方式、以及对运输的要求等情况，这反而对车辆装载有很大的影响。在后面的研究中，应该综合考虑零件特点，选择合适的装载方式。

10. 参考文献

[1]李若萌.基于 MILKRUN 的 S 汽车企业入厂物流规划[D].河南:郑州大学,2014.

[2]陈丽.汽车零部件入厂物流的循环取货模式应用研究[D].北京:对外经济贸易大学,2018.

[3]王双金.B 汽车制造企业零部件入厂物流循环取货路径优化研究[D].北京:北京交通大学,2016. DOI:10.7666/d.Y3126573.

[4]魏宇.基于改进节约算法的汽车零部件循环取货路线问题研究[J].内燃机与配件,2018,(21):202-205.
DOI:10.3969/j.issn.1674-957X.2018.21.100.

[5]徐敏.供应商主导下的 Milk Run 计划及车辆路径问题研究[D].江苏:南京农业大学,2016.