

使用DP的问题特征：

- ① 无后效性。
- ② 最优子结构。
- ③ 子问题的重叠性。

第14章 动态规划

动态规划 (Dynamic Programming, DP) 是运筹学中的一种重要的最优化数学方法，主要用来求解多阶段决策问题。20世纪50年代初，美国数学家 R. Bellman 等在研究多阶段决策过程的优化问题时，提出了著名的最优化原理，从而创立了动态规划 (Bellman 1952, Bellman 1966)。动态规划算法可以非常高效地求解 SPPRC，从而加快 VRPTW 的求解。同时，也可以用于提高其他子问题为 SPPRC 的相关问题的求解效率。

14.1 动态规划

动态规划分为前向动态规划、后向动态规划和双向动态规划。动态规划的应用非常广泛，包括工程技术、经济、工业生产、军事以及自动化控制等领域，并在最短路问题、Lot-Sizing 问题、背包问题、资金管理问题、资源分配问题和复杂系统可靠性等问题中取得了显著的应用效果。

运用动态规划求解的问题一般具备如下特点：① 无后效性：某个阶段的状态一旦确定后，就不会受该状态之后的决策影响，同时，该阶段之后的决策和状态发展不受该阶段之前各状态的影响。② 最优子结构：如果某个问题的最优解可以由其子问题的最优解推出，那么该问题就具备最优子结构。简言之，一个最优策略的子策略总是最优的，就叫作最优子结构，也叫作最优化原理。③ 子问题的重叠性：动态规划在实现过程中需要存储各种状态，这些状态会占用计算空间，如果要使得动态规划算法能够比较高效，那么各种状态就要有较高的复用性，以便减少计算空间的占用。但该性质不是必需的，只是在一定程度上决定了算法的效率。

动态规划的求解思路如下。

(1) 划分阶段：将问题按照时间或者空间特征分为若干相互联系的阶段，这些阶段需要有序或者可排序，否则不便于动态规划过程的递推。其中描述阶段的变量称为阶段变量，阶段变量一般是离散变量。

(2) 状态表示：状态主要用来描述问题各阶段所处的客观情况或自然条件，这些情况或条件具备无后效性，即当前状态不会受决策者后续决策的影响。

(3) 状态转移：状态表示确定之后，问题从一个阶段向下一个阶段转移时，伴随着状态的跳转，状态的跳转需要状态转移方程（又叫状态递推方程）来表示，状态转移方程一般需要在划分阶段、状态表示之后，根据状态之间的跳转关系进行决策分析确定。

(4) 确定边界：初始状态和最终状态的确定，以限制问题的开始和结束。

14.1.1 动态规划求解最短路问题

给定有向图 $D = (V, A)$ ， V 是图中的点集， A 是弧集，每个弧段都有一个相应的非负距离（权值），要求从一个初始点 s 开始到另一个终止点 t 之间，找到一条使得距离最短（权值最小）的路，就是最短路问题 (Shortest Paths Problem)。下面用一个例子详细解释 DP 的过程。

假设小明要驾车从城市 1 去城市 10，中间有多条路径可以选择，每条路径途经的城市（城市 2~9）和路程长度是不一样的，如图 14.1 所示。其中城市 2、3、4 可以驾车在一天内到达，城市 5、6、7 需要驾车两天才能到达，城市 8、9 需要三天时间到达，到达城市 10 则需要四天时间。要使得总行驶距离最小，小明应该如何规划这四天的行程？

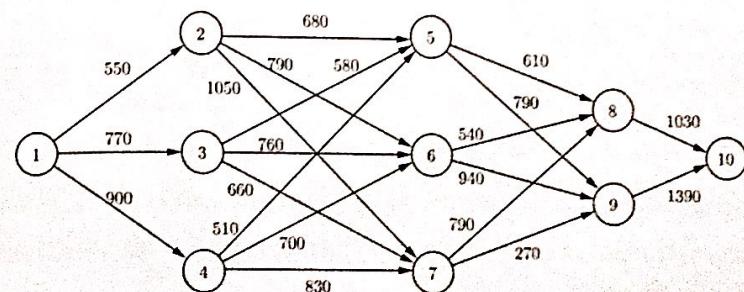


图 14.1 最短路问题：网络结构图

14.1.2 问题建模和求解

接下来我们运用上面理论部分所讲的动态规划的思路来分析这个例题。首先划分阶段，题目中提及各城市到达的时间是不一样的，从一天到四天不等，因此很明显我们可以将时间（天数）作为划分阶段的标准，即将问题划分为四个阶段 ($t = \{1, 2, 3, 4\}$)。然后确定状态，我们将状态设定为某一天开始时小明从某个城市出发到达目的地时所驾车驶过的累积路程，我们定义为 $f_t(i), i \in V$ 。接下来分析状态转移过程，每一天的结束将伴随小明从一个城市去到另一个城市，他所驾车驶过的路程就进行了跳转。我们采用后向的动态规划思路来分析递推公式，假定后一个阶段一定取得最短路，那么当前阶段的状态转移方程可以写为 $f_t(i) = \min\{f_{t+1}(j) + c_{ij}\}$ ，其中， j 是从 t 到 $t+1$ 阶段所有可以从 i 到达的点， c_{ij} 是点 i 与点 j 之间的距离。最后确定开始和结束条件，因为是后向动态规划，所以开始于第四阶段，结束于第一阶段。

下面我们从第 4 阶段开始分别进行分析动态规划的整个决策求解过程。

第 4 阶段：小明将从城市 8 或者城市 9 出发去往城市 10，因此我们需要分别计算 $f_4(8)$ 和 $f_4(9)$ 。

$$f_4(8) = \min\{f_3(10) + c_{8,10}\} = \min\{0 + 1030\} = 1030$$

$$f_4(9) = \min\{f_3(10) + c_{9,10}\} = \min\{0 + 1390\} = 1390$$

其中, $f_5(10)$ 已经到达目的地, 所以其值为 0。

第3阶段: 小明将从城市 5 或者城市 6 或者城市 7, 去往城市 8 或者城市 9, 因此要获得从当前城市到达城市 10 的最短路, 需要对城市 5、6、7 分别计算状态转移方程。

$$f_3(5) = \min \begin{cases} f_4(8) + c_{58} = 1030 + 610 = 1640* \\ f_4(9) + c_{59} = 1390 + 790 = 2180 \end{cases}$$

其中, * 表示 $f_3(5)$ 的最小值, 即从城市 5 到城市 10 的最短路径为 $5 \rightarrow 8 \rightarrow 10$, 将其作为备选路径, 以下 * 表示相同含义。

$$f_3(6) = \min \begin{cases} f_4(8) + c_{68} = 1030 + 540 = 1570* \\ f_4(9) + c_{69} = 1390 + 940 = 2330 \end{cases}$$

$f_3(6)$ 的最小值为 1570, 从城市 6 到城市 10 的最短路径为 $6 \rightarrow 8 \rightarrow 10$, 将其作为途经城市 6 到目的地的备选路径。

$$f_3(7) = \min \begin{cases} f_4(8) + c_{78} = 1030 + 790 = 1820 \\ f_4(9) + c_{79} = 1390 + 270 = 1660* \end{cases}$$

$f_3(7)$ 的最小值为 1660, 从城市 7 到城市 10 的最短路径为 $7 \rightarrow 9 \rightarrow 10$, 将其作为途经城市 7 到目的地的备选路径。

第2阶段: 现在我们已经知道了从第3阶段的城市 5、6、7 分别到达目的地城市 10 的最优路线, 那么继续进行后向动态规划的公式递推, 我们将获得从第2阶段各城市到达目的地的最短路径。第2阶段的城市有 2、3、4, 因此需要我们分别计算 $f_2(2)$ 、 $f_2(3)$ 、 $f_2(4)$ 。

$$f_2(2) = \min \begin{cases} f_3(5) + c_{25} = 1640 + 680 = 2320* \\ f_3(6) + c_{26} = 1570 + 790 = 2360 \\ f_3(7) + c_{27} = 1660 + 1050 = 2710 \end{cases}$$

$f_2(2)$ 的最小值为 2320, 从城市 2 到城市 10 的最短路径可从 $f_3(5)$ 进行追溯, 此路径即为 $2 \rightarrow 5 \rightarrow 8 \rightarrow 10$, 我们将其作为途经城市 2 到目的地的备选路径。

$$f_2(3) = \min \begin{cases} f_3(5) + c_{35} = 1640 + 580 = 2220* \\ f_3(6) + c_{36} = 1570 + 760 = 2330 \\ f_3(7) + c_{37} = 1660 + 660 = 2320 \end{cases}$$

$f_2(3)$ 的最小值为 2220, 从城市 3 到城市 10 的最短路径为 $3 \rightarrow 5 \rightarrow 8 \rightarrow 10$, 将其作为途经城市 3 到目的地的备选路径。

$$f_2(4) = \min \begin{cases} f_3(5) + c_{45} = 1640 + 510 = 2150* \\ f_3(6) + c_{46} = 1570 + 700 = 2270 \\ f_3(7) + c_{47} = 1660 + 830 = 2490 \end{cases}$$

$f_4(4)$ 的最小值为 2150, 从城市 4 到城市 10 的最短路径为 $4 \rightarrow 7 \rightarrow 8 \rightarrow 10$, 将其作为途经城市 4 到目的地的备选路径。

第1阶段: 我们目前已经知道了从城市 2、3、4 到达目的地 10 的最短路径, 接下来只需要继续进行后向动态规划过程求解 $f_1(1)$ 就可以获得从 1 到 10 的最短路径。

$$f_1(1) = \min \begin{cases} f_2(2) + c_{12} = 2320 + 550 = 2870* \\ f_2(3) + c_{13} = 2220 + 770 = 2990 \\ f_2(4) + c_{14} = 2150 + 900 = 3050 \end{cases}$$

至此, 我们已经探明了从城市 1 到城市 10 的最短距离为 2870, 其路径可以通过追溯 $f_2(2)$ 获得, 即为 $1 \rightarrow 2 \rightarrow 5 \rightarrow 8 \rightarrow 10$ 。所得最短路如图 14.2 中粗线条所示路径。

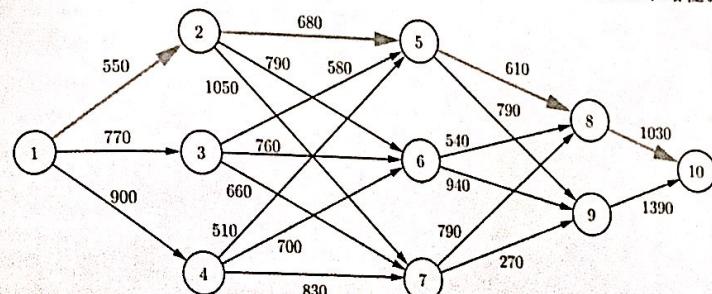


图 14.2 最短路

14.1.3 一个较大规模的例子

上面的例子比较简单, 仅用来展示动态规划解决问题的思路, 但它可能无法有效地展示运用动态规划的优势所在。其实, 对于较大规模的问题, 动态规划也能够比较有效地进行求解, 比如再举一个稍大规模的例子。

如图 14.3 所示, 从点 0 到点 26 的一个分阶段的网络, 各相邻两阶段之间互通, 任意可通达的两点 (i, j) 之间的距离已知 (c_{ij}) , 求从初始点 0 到终止点 26 的最短路径。

一种很容易想到的方法就是枚举其中的所有路径, 然后对比得出最短路径。但我们注意到, 该网络中, 从 0 到 26 有 5^5 条可行路径, 每条路径的计算都要进行 5 条弧的加和, 因此总共需要进行 $5 \times 5^5 = 15625$ 次加和计算, 以及 $5^5 - 1 = 3124$ 次比较运算, 才能获得最短路径。很明显这种方法不太实用, 尤其在实际应用中, 网络规模比这个大得多。而如果我们运用上面分析的动态规划的思路进行求解, 首先将问题划分为 0~6 共 7 个阶段, $f_6(26)$ 已经到达目的地不需要进行决策, 运用后向动态规划从阶段 5 开始, $f_5(i), i \in \{21, 22, 23, 24, 25\}$ 不需要进行计算, 只需要计算 4 次, 阶段 4 至阶段 1 的计算可以用通用递推公式 $f_t(j) = \min\{f_{t+1}(k) + c_{jk}, t \in \{1, 2, 3, 4\}, j \in \{5t-4, 5t-3, 5t-2, 5t-1, 5t\}, k \in \{5t+1, 5t+2, 5t+3, 5t+4, 5t+5\}\}$ 进行计算, 对于每个 t, j, k 进行组合, 总共有 $4 \times 5 \times 5 = 100$ 次求和计算以及 $4 \times 5 \times 4 = 80$ 次比较运算, 最后阶段 0 从点 0 到 1、2、3、4、5 共有

5 次计算即可探索清晰总的路径信息，选取最短路径只需再进行 4 次比较，因此总的计算次数为 $100 + 5 = 105$ 次求和计算以及 $80 + 4 = 84$ 次比较。这些计算仅仅占据枚举法的 $189/18749 \approx 0.01$ ，因此动态规划在较大规模问题上具有显著的优势。

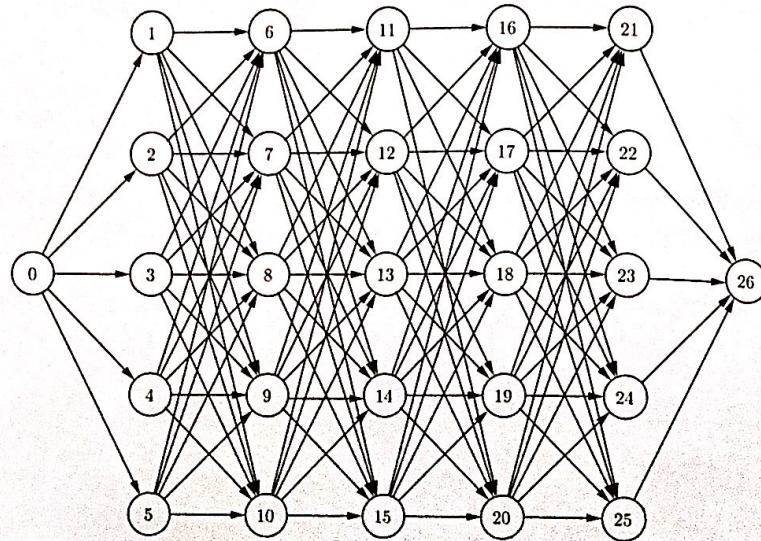


图 14.3 最短路：一个较大规模的网络

14.2 动态规划的实现

14.2.1 伪代码

下面给出用动态规划求解上述例子的伪代码和代码。首先给出伪代码如下。

Algorithm 12 Dynamic Programming 算法

Input: 网络中的弧矩阵

Output: 最短距离和路径信息

- 1: 输入网络数据，初始化 nodeNum , stageNum , nodeNumInOneStage
- 2: 定义数据结构 PathData，其包含 pathInfo , pathDistance 等
- 3: 初始化： $\text{pathList} \leftarrow \emptyset$
- 4: **for** $i = \text{nodeNum} - 1 - \text{nodeNumInOneStage}; i < \text{nodeNum} - 1; i++$ **do**
- 5: $t = \text{stageNum}$
- 6: new PathData pd
- 7: $\text{pd.pathDistance} = f_t(i) = c_{i, \text{nodeNum}-1}$ and $\text{pd.pathInfo.add}(\text{nodeNum}-1)$
- 8: $\text{pathList.add}(\text{pd})$
- 9: **end for**

```

10: for int t = stageNumber-1; t > 0; t -- do
11:   for int i = nodeNumInOneStage*t - (nodeNumberInOneStage - 1);
12:     i < nodeNumInOneStage*t + 1; i ++ do
13:       new PathData pd1
14:       pd1.pathDistance = f_t(i) = min{f_{t+1}(j) + c_{ij}},
15:       ∀j ∈ {nodeNumInOneStage*t + 1, …, nodeNumInOneStage*t + 5}
16:       pd1.pathInfo.add(i)
17:       pathList.add(pd1)
18:     end for
19:   end for
20:   for int i = 1; i < nodeNumInOneStage+1; i ++ do
21:     new PathData pd2
22:     pd2.pathDistance = f_0(0) = min{f_1(i) + c_{0i}} and pd2.pathInfo.add(i)
23:     pathList.add(pd2)
24:   end for
25: return pathList.get(lastElement).pathDistance & pathInfo

```

14.2.2 Java 代码

1. PathData 类

先构建路径拓展所需的数据结构，即 PathData 类，代码如下：

Path Data.java

```

/**
 * @author: Yongsen Zang
 * @School: Tsinghua University
 * @操作说明：动态规划求解最短路问题之定义数据结构
 *
 */
package dynamicProgrammingForSP;
import java.util.ArrayList;

public class PathData {
    int t;
    int nodeID;
    double pathDistance;
    ArrayList<Integer> pathInfo = new ArrayList<>();
}

```

2. GenerateRandomData 类

由于网络图没有给定各弧段的长度，我们为每个弧段随机生成一个长度，即 GenerateRandomData 类，代码如下：

GenerateRandomData.java

```

1 /**
2 * @author: Yongsen Zang
3 * @School: Tsinghua University
4 * @操作说明: 动态规划求解最短路问题之生成随机矩阵
5 *
6 */
7 package dynamicProgrammingForSP;
8 import java.math.BigDecimal;
9
10 public class GenerateRandomData {
11
12     public static double[][] GenerateDistanceMatrix(int nodeNumber, int
13         stageNumber, int nodeNumberInOneStage){
14         double arcDistance[][] = new double[nodeNumber][nodeNumber];
15
16         for(int i=0; i<nodeNumber; i++){
17             for(int j=0; j<nodeNumber; j++){
18                 arcDistance[i][j] = Integer.MAX_VALUE;
19             }
20
21             for(int t=1; t<stageNumber; t++){
22                 for(int i=nodeNumberInOneStage*t-
23                     nodeNumberInOneStage; i<=nodeNumberInOneStage*t; i++){
24                     for(int j=nodeNumberInOneStage*t+1; j<=
25                         nodeNumberInOneStage*t+nodeNumberInOneStage; j++){
26                         arcDistance[i][j] =
27                             GenerateRandomArcDistance();
28                     }
29                 }
30             }
31             for(int i=1; i<nodeNumberInOneStage+1; i++){
32                 arcDistance[0][i] = GenerateRandomArcDistance();
33                 arcDistance[nodeNumberInOneStage*(stageNumber-1)+i][
34                     nodeNumber-1] = GenerateRandomArcDistance();
35             }
36         }
37         return arcDistance;
38     }
39 }

```

```

/*
 * 生成 300 ~ 1500 的随机数
 * @return 随机数
 */
public static double GenerateRandomArcDistance() {
    int max=1500,min=300;
    double ran = (Math.random()*(max-min)+min);
    BigDecimal b = new BigDecimal(ran);
    ran = b.setScale(2, BigDecimal.ROUND_HALF_UP).doubleValue();
    return ran;
}

```

3. DynamicProgrammingForSP 类

动态规划求解的主要过程，即 main 方法，代码如下：

DynamicProgrammingForSP.java

```

/*
 * @author: Yongsen Zang
 * @School: Tsinghua University
 * @操作说明: 动态规划求解最短路问题
 *
 */
package dynamicProgrammingForSP;
import java.util.ArrayList;
public class DynamicProgrammingForSP {
    static int nodeNumber = 27;
    static int stageNumber = 5;
    static int nodeNumberInOneStage = 5;
    static double[][] cij = new double[nodeNumber][nodeNumber];
    public static void main(String[] args){
        // 获取算例数据
        cij = GenerateRandomData.GenerateDistanceMatrix(nodeNumber,
            stageNumber, nodeNumberInOneStage);
        for(int i=0; i<nodeNumber; i++){
            for(int j=0; j<nodeNumber; j++){
                System.out.print(cij[i][j]+", ");
            }
        }
    }
}

```

```

23         System.out.println();
24     }
25
26     // 定义一个 ArrayList 用来存储各个阶段生成的路径信息
27     ArrayList<PathData> pathList = new ArrayList<PathData>();
28
29     // 按阶段进行后向动态规划
30     // 先进行最后一个阶段，即第 5 阶段
31     for(int i=nodeNumber-1-nodeNumberInOneStage; i<nodeNumber-1;
32         i++){
33         PathData pada = new PathData();
34         pada.t = stageNumber;
35         pada.nodeID = i;
36         pada.pathDistance = cij[i][nodeNumber-1];// nodeNumber-1==26 in this example
37         pada.pathInfo.add(nodeNumber-1);
38         pada.pathInfo.add(i);
39         pathList.add(pada);
40     }
41     // 进行第 4 阶段到第 1 阶段的递推计算
42     for(int t=stageNumber-1; t>0; t--){
43         for(int i=nodeNumberInOneStage*t-(nodeNumberInOneStage-1); i<nodeNumberInOneStage*t+1; i++){
44             PathData pd = new PathData();
45             pd = FindShortestSubpath(pathList, i, t);
46             pathList.add(pd);
47         }
48     }
49     // 进行第 0 阶段的计算，即从点 0 到点 1~5 的计算
50     for(int i=1; i<nodeNumberInOneStage+1; i++){
51         PathData pd = FindShortestSubpath(pathList, 0, 0);
52         pathList.add(pd);
53     }
54     // 输出最终结果
55     System.out.println("The shortest path distance is: " +
56     pathList.get(pathList.size()-1).pathDistance);
57     System.out.print("The shortest path is: ");
58     for(int i=pathList.get(pathList.size()-1).pathInfo.size()-1;
      i>=0; i--){
          System.out.print(pathList.get(pathList.size()-1).
pathInfo.get(i) + "->");
      }
  
```

```

}
}

/**
 * 计算每个出发节点到达目的地的最短路径，当前出发节点就是要拓展的节点
 * @param pathList
 * @param i, 当前要扩展的节点
 * @param t, 当前的阶段
 * @return PathData
 */
public static PathData FindShortestSubpath(ArrayList<PathData>
pathList, int i, int t){
    PathData pathdata = new PathData();
    double newLength = 0;
    int nodeID = 0;
    ArrayList<Integer> jpath = new ArrayList<Integer>();
    ArrayList<Integer> extendPath = new ArrayList<Integer>();
    double agency = Integer.MAX_VALUE;
    for(int j=nodeNumberInOneStage*t+1; j<nodeNumberInOneStage*t+
+6; j++){
        double subdis = 0;

        for(PathData pa: pathList){
            if(pa.nodeID == j){
                subdis = pa.pathDistance;
                jpath = pa.pathInfo;
                break;
            }
        }

        newLength = subdis + cij[i][j];
        if(newLength <= agency){
            agency = newLength;
            nodeID = i;
            extendPath = DeepCopy(jpath);
        }
    }
    pathdata.nodeID = nodeID;
    pathdata.pathDistance = agency;
    for(int node: extendPath){
        pathdata.pathInfo.add(node);
    }
}

```

```

99         pathdata.pathInfo.add(nodeID);
100        pathdata.t = t;
101
102        return pathdata;
103    }
104
105    //深拷贝路径信息的方法
106    public static ArrayList<Integer> DeepCopy(ArrayList<Integer> arr){
107        ArrayList<Integer> newArr = new ArrayList<Integer>();
108        for(int a: arr){
109            newArr.add(a);
110        }
111        return newArr;
112    }
113}

```

注意：需要注意的是，在实际应用中，有些时候会遇到不太容易划分阶段的情况，这时也可以不进行阶段的划分，但在确定状态转移方程时会稍微复杂点，同时所需的计算次数也会相应增加，即计算复杂度可能会有一定的增加，如果读者感兴趣，可以自己尝试不划分阶段求解上述最短路问题，这里不做详细阐述。此外，本章只是抛砖引玉地针对最短路问题设计了动态规划算例进行讲解，其应用场景还有很多，比如背包问题、TSP、排程问题等，感兴趣的读者可以自行探索研究。

14.3 动态规划求解 TSP

上文我们介绍了动态规划求解最短路问题的基本原理和代码实现。本节我们以 TSP 问题为例，继续深入探讨动态规划。

14.3.1 一个简单的 TSP 算例

我们以一个非常简单的例子开始。考虑有 4 个点的网络（例子和原理介绍参考自 <https://www.youtube.com/watch?v=XaXsJJh-Q5Y>），如图 14.4 所示。

其对应的距离矩阵为

$$D = \begin{bmatrix} 0 & 8 & 12 & 17 \\ 4 & 0 & 10 & 9 \\ 7 & 14 & 0 & 13 \\ 9 & 7 & 11 & 0 \end{bmatrix}$$

注意：上述距离矩阵不是对称的。

枚举法 \rightarrow 顺推
 \rightarrow 逆推

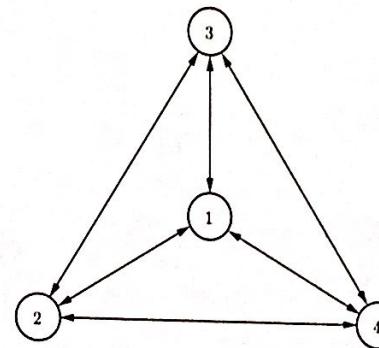


图 14.4 TSP: 一个小例子

如果我们把点 1 作为起始点，枚举所有的路径，就如图 14.5 中展示的那样。如果从上到下枚举，复杂度为 $O(n!)$ ，其中 n 为顾客点的个数。

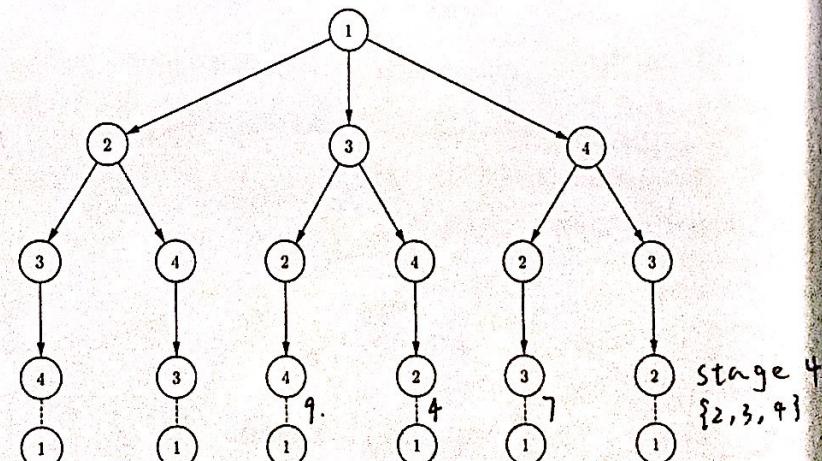


图 14.5 TSP 小例子：枚举所有路径

但是如果我们从下往上逆推，问题的复杂度就会有所不同。容易得出，当顾客点为 4 的时候，任意一个解对应的路径中的节点个数为 5，例如 $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$ 。另外，任意一个解，其最后到达的点一定是起点 1。因此我们可以将整个过程分为 5 个阶段，每个阶段的状态，就是该阶段所在的顾客点。

第 5 阶段，最终到达的一定是点 1。

第 4 阶段，剩余还没有访问的点集就是 {1}，但是在第 4 阶段当前所处的位置却有 3 种不同的情况，即 {2, 3, 4}。如果第 4 阶段在点 4，则从 4 出发，到达终点 1 的距离为 $d_{4,1} = 9$ ；同理，从 2 和 3 出发的距离分别为 $d_{2,1} = 4$ 和 $d_{3,1} = 7$ 。图 14.6 中第 4 层虽然有 6 个节点，但实际上我们只计算 3 次。

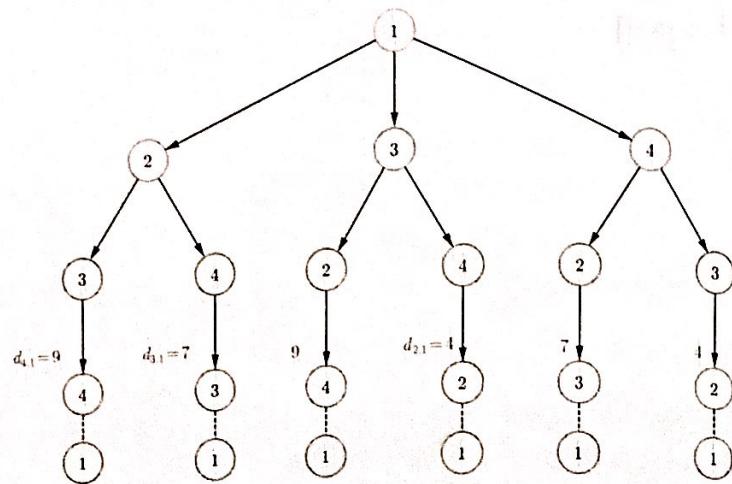


图 14.6 TSP 小例子：第 4 阶段

第 3 阶段，如图 14.7 所示。剩余还没有访问的点集就有所不同。当前所在的点 $v_c \in \{2, 3, 4\}$ 。并且剩余还没访问的点的个数为 2，其中有一个点是 1，因此我们考虑除了 1 以外，还没有被访问的点的集合。如果 $v_c = 3$ ，除了 1 以外，还没有被访问的点的集合大小为 1，集合 S_t 可以为 {2} 或者 {4}。

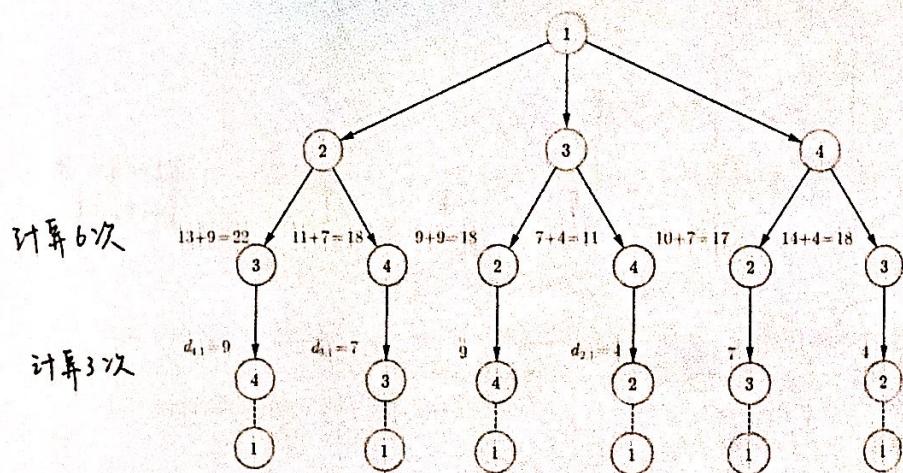


图 14.7 TSP 小例子：第 3 阶段

- (1) 如果 $v_c = 3, S_t = \{4\}$ ，则第 3 个点访问顾客点 3，然后经过顾客点 4，回到顾客点 1 的最小距离为 $d_{3,4} + d_{4,1} = 13 + 9 = 22$ 。
- (2) 如果 $v_c = 4, S_t = \{3\}$ ，则第 3 个点访问顾客点 4，然后经过顾客点 3，回到顾客

点 1 的最小距离为 $d_{4,3} + d_{3,1} = 11 + 7 = 18$ 。

(3) 如果 $v_c = 2, S_t = \{4\}$ ，则第 3 个点访问顾客点 2，然后经过顾客点 4，回到顾客点 1 的最小距离为 $d_{2,4} + d_{4,1} = 9 + 9 = 18$ 。

(4) 如果 $v_c = 4, S_t = \{2\}$ ，则第 3 个点访问顾客点 4，然后经过顾客点 2，回到顾客点 1 的最小距离为 $d_{4,2} + d_{2,1} = 7 + 4 = 11$ 。

(5) 如果 $v_c = 2, S_t = \{3\}$ ，则第 3 个点访问顾客点 2，然后经过顾客点 3，回到顾客点 1 的最小距离为 $d_{2,3} + d_{3,1} = 10 + 7 = 17$ 。

(6) 如果 $v_c = 3, S_t = \{2\}$ ，则第 3 个点访问顾客点 3，然后经过顾客点 2，回到顾客点 1 的最小距离为 $d_{3,2} + d_{2,1} = 14 + 4 = 18$ 。

第 2 阶段，如图 14.8 所示。当前所在的点 $v_c \in \{2, 3, 4\}$ 。并且剩余还没访问的点（除了顾客点 1 以外）的个数为 2。

(1) 如果 $v_c = 2, S_t = \{3, 4\}$ ，即第 2 个点访问顾客点 2，然后以任意顺序访问顾客集合 {3, 4}，回到顾客点 1 的最小距离为 $d_{\min}^2 = \min\{10 + 22, 9 + 18\} = 27$ 。

(2) 如果 $v_c = 3, S_t = \{2, 4\}$ ，即第 2 个点访问顾客点 3，然后以任意顺序访问顾客集合 {2, 4}，回到顾客点 1 的最小距离为 $d_{\min}^3 = \min\{14 + 18, 13 + 11\} = 24$ 。

(3) 如果 $v_c = 4, S_t = \{2, 3\}$ ，即第 2 个点访问顾客点 4，然后以任意顺序访问顾客集合 {2, 3}，回到顾客点 1 的最小距离为 $d_{\min}^4 = \min\{7 + 17, 11 + 18\} = 24$ 。

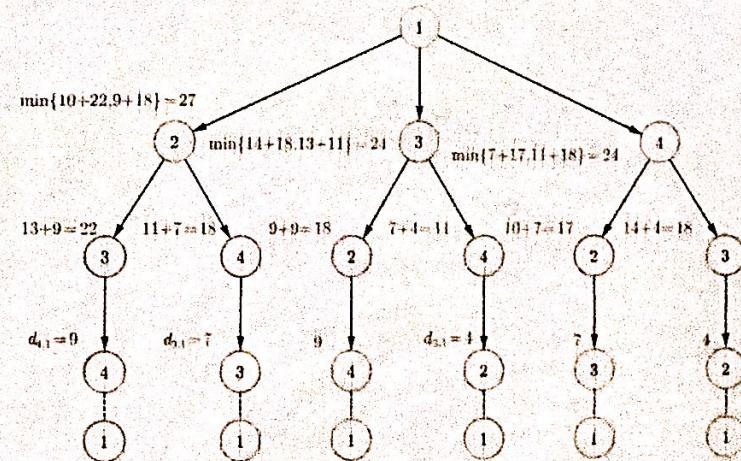


图 14.8 TSP 小例子：第 2 阶段

第 1 阶段，如图 14.9 所示。当前所在的点只能是点 1，因此 $v_c = 1$ 。并且剩余还没访问的点（除了顾客点 1 以外）的个数为 3，集合为 $S_t = \{2, 3, 4\}$ ，因此最小距离为 $d_{\min} = \min\{d_{\min}^2 + d_{1,2}, d_{\min}^3 + d_{1,3}, d_{\min}^4 + d_{1,4}\} = \min\{8 + 27, 12 + 24, 17 + 24\} = 35$ 。

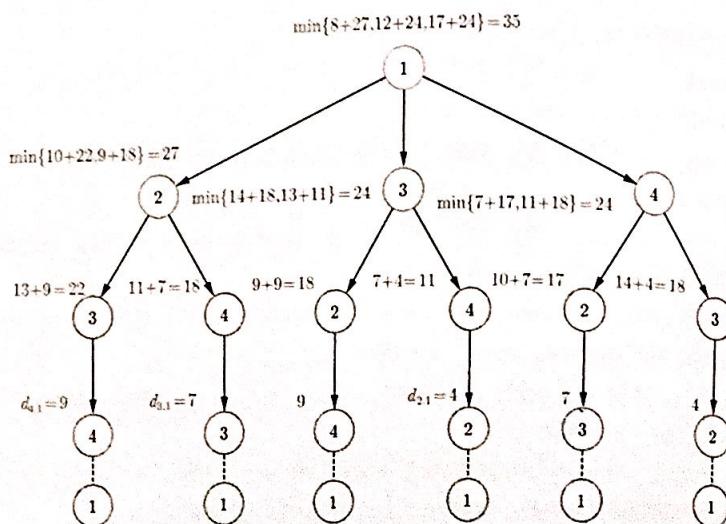


图 14.9 TSP 小例子: 第 1 阶段

接下来我们就可以给出通项公式。我们用 $f(v_c, S_l)$ 表示当前节点为 v_c 、剩余未访问的节点集合为 S_l 时的最小距离，则有

$$f(1, \{2, 3, 4\}) = \min\{d_{1,k} + f(k, \{2, 3, 4\} - \{k\}), k = 2, 3, 4\}$$

更一般地，有

$$f(i, \emptyset) = \min_{k \in S_i} \{d_{i,k} + f(k, S_i - \{k\})\} \quad (14.1)$$

Δ 和剩余未访问的节点集合。

接下来我们用上述递推公式来求解一遍上述例子。我们从第 4 阶段开始， $v_c \in \{2, 3, 4\}$ 。
 $S_l = \emptyset$ ，则计算如下：

- (1) $f(2, \emptyset) = d_{2,1} = 4;$
- (2) $f(3, \emptyset) = d_{3,1} = 7;$
- (3) $f(4, \emptyset) = d_{4,1} = 9.$

第 3 阶段， $v_c \in \{2, 3, 4\}$ ，则分下面的情况讨论。

- (1) $v_c = 2, S_l = \{3\}$ 或 $\{4\}$ ：
 - ① $f(2, \{3\}) = d_{2,3} + \min\{f(3, \emptyset)\} = 10 + 7 = 17;$
 - ② $f(2, \{4\}) = d_{2,4} + \min\{f(4, \emptyset)\} = 9 + 9 = 18.$
- (2) $v_c = 3, S_l = \{2\}$ 或 $\{4\}$ ：
 - ① $f(3, \{2\}) = d_{3,2} + \min\{f(2, \emptyset)\} = 14 + 4 = 18;$
 - ② $f(3, \{4\}) = d_{3,4} + \min\{f(4, \emptyset)\} = 13 + 9 = 22.$
- (3) $v_c = 4, S_l = \{2\}$ 或 $\{3\}$ ：
 - ① $f(4, \{2\}) = d_{4,2} + \min\{f(2, \emptyset)\} = 7 + 4 = 11;$

$$\textcircled{2} \quad f(4, \{3\}) = d_{4,3} + \min\{f(3, \emptyset)\} = 11 + 7 = 18.$$

第 2 阶段， $v_c \in \{2, 3, 4\}$ ，则分下面的情况讨论：

$$(1) v_c = 2, S_l = \{3, 4\}，则 f(2, \{3, 4\}) = \min\{d_{2,3} + f(3, \{4\}), d_{2,4} + f(4, \{3\})\} = \min\{10 + 22, 9 + 18\} = 27;$$

$$(2) v_c = 3, S_l = \{2, 4\}，则 f(3, \{2, 4\}) = \min\{d_{3,2} + f(2, \{4\}), d_{3,4} + f(4, \{2\})\} = \min\{14 + 18, 13 + 11\} = 24;$$

$$(3) v_c = 4, S_l = \{2, 3\}，则 f(4, \{2, 3\}) = \min\{d_{4,2} + f(2, \{3\}), d_{4,3} + f(3, \{2\})\} = \min\{7 + 17, 11 + 18\} = 24.$$

第 1 阶段， $v_c = 1$ ，则计算如下：

$$\begin{aligned} f(1, \{2, 3, 4\}) &= \min\{d_{1,2} + f(2, \{3, 4\}), d_{1,3} + f(3, \{2, 4\}), d_{1,4} + f(4, \{2, 3\})\} \\ &= \min\{8 + 27, 12 + 24, 17 + 24\} = 35. \end{aligned}$$

至于最优路径，我们顺着第 1 阶段向第 5 阶段回溯，可以得到，最优路径为 $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 1$ 。

可以明显看出，在上述过程中，我们在第 4 到第 1 阶段的计算次数共为 $3+6+6+1=16$ 次。而如果用排列组合的方法，计算次数为 $4! = 24$ 次。显然，使用动态规划的方法，显著降低了计算复杂度。经过分析，不难得出，动态规划求解 TSP 的复杂度为 $\mathcal{O}(n^2 \cdot 2^n)$ 。非常可惜，虽然动态规划降低了求解复杂度，但是从本质上来说，动态规划求解 TSP 仍旧是一个指数时间复杂度的算法。

14.3.2 伪代码

前面章节介绍了动态规划求解 TSP 的原理。本节和下一节，我们来探讨如何将其实现。首先，我们根据之前的理论介绍，整理出动态规划求解 TSP 的伪代码。

Algorithm 13 动态规划求解 TSP，算法复杂度 $\mathcal{O}(n^2 \cdot 2^n)$

Input: 图 $G = (V, A)$ ，起始节点 $s = 1$

Output: 最优路径 r^* ，其中每个节点均被访问且只被访问一次

```

1: for  $i = 2, 3, \dots, |V|$  do
2:    $f(i, \emptyset) \leftarrow d_{i,s}$ 
3:   ( $i, \emptyset$ ) 的下一个节点  $\leftarrow s$ 
4: end for
5: for  $k = V, V-1, V-2, \dots, 2$  do
6:   for 当前节点  $i = 2, 3, \dots, |V|$  do
7:     for  $S_l \subset V \setminus \{i\}$  和  $|S_l| = |V| - k$  do
8:        $f(i, S_l) = \min_{v \in S_l} \{d_{i,v} + f(v, S_l - \{v\})\}$ 
9:       ( $i, S_l$ ) 的下一个节点  $\leftarrow \arg \min_v \{d_{i,v} + f(v, S_l - \{v\})\}$ 
10:    end for
  
```

```

11. end for
12. end for
13.  $S_l \leftarrow V - \{s\}$ 
14.  $f(s, S_l) \leftarrow \min_{v \in S_l} \{d_{s,v} + f(v, S_l - \{v\})\}$ 
15.  $(s, S_l)$  的下一个节点  $\leftarrow \arg \min_v \{d_{s,v} + f(v, S_l - \{v\})\}$ 
16. 通过检查当前节点的下一个节点，获得最优路径  $r^*$ 
17. return 最优路径  $r^*$  和最优目标值  $f(s, S_l)$ 

```

14.3.3 Python 实现：示例算例

Python 实现动态规划求解 TSP 的代码如下：

TSP Dynamic Programming.py

```

...
Author      : Liu Xinglu
Institute   : Tsinghua University
Date        : 2020-10-21
...

import copy
import re
import math
import itertools

Nodes = [1, 2, 3, 4]

nodeNum = len(Nodes)

dis_matrix = [[0, 8, 12, 17],
              [4, 0, 10, 9],
              [7, 14, 0, 13],
              [9, 7, 11, 0] ]

def TSP_Dynamic_Programming(Nodes, dis_matrix):
    Label_set = {}
    nodeNum = len(Nodes)
    org = 1
    # cycle stage : V, V-1, ..., 2
    for stage_ID in range(nodeNum, 1, -1): # 逆序遍历 stage_ID
        print('stage :', stage_ID)

```

```

for i in range(2, nodeNum + 1):
    current_node = i
    left_node_list = copy.deepcopy(Nodes)
    left_node_list.remove(i)
    if (org in left_node_list):
        left_node_list.remove(org)
    left_node_set = set(left_node_list)
    # obtain the all the subset of the left node set
    subset_all = list(map(set, itertools.combinations(left_node_set,
                                                       nodeNum - stage_ID)))
    # print('current_node:', current_node, 'it left_node_set :',
    #       left_node_set)
    for subset in subset_all:
        if (len(subset) == 0):
            key = (stage_ID, current_node, 'None')
            next_node = org
            Label_set[key] = [dis_matrix[i - 1][org - 1], next_node]
        else:
            key = (stage_ID, current_node, str(subset))
            min_distance = 1000000
            for temp_next_node in subset:
                subsub_set = copy.deepcopy(subset)
                subsub_set.remove(temp_next_node)
                if (subsub_set == None or len(subsub_set) == 0):
                    subsub_set = 'None'
                sub_key = (stage_ID + 1, temp_next_node, str(subsub_set))
                if (sub_key in Label_set.keys()):
                    if (dis_matrix[current_node - 1][temp_next_node -
                        1] + Label_set[sub_key][0] < min_distance):
                        min_distance = dis_matrix[current_node - 1][
                            temp_next_node - 1] + Label_set[sub_key][0]
                        next_node = temp_next_node
                        Label_set[key] = [min_distance, next_node]
                # print('current_node:', current_node, 'it left_node_set :',
                #       subset)

            # stage 1 :
    stage_ID = 1
    current_node = org
    subset = set(copy.deepcopy(Nodes))

```

```

65     subset.remove(org)
66     final_key = (stage_ID, current_node, str(subset))
67     min_distance = 1000000
68     for temp_next_node in subset:
69         subsub_set = copy.deepcopy(subset)
70         subsub_set.remove(temp_next_node)
71         if (subsub_set == None or len(subsub_set) == 0):
72             subsub_set = 'None'
73             sub_key = (stage_ID + 1, temp_next_node, str(subsub_set))
74             if (sub_key in Label_set.keys()):
75                 if (dis_matrix[current_node - 1][temp_next_node - 1] + Label_set
76 [sub_key][0] < min_distance):
77                     min_distance = dis_matrix[current_node - 1][temp_next_node -
78 1] + Label_set[sub_key][0]
79                     next_node = temp_next_node
80                     Label_set[final_key] = [min_distance, next_node]
81
82     # get the optimal solution
83     opt_route = [org]
84     not_visted_node = set(copy.deepcopy(Nodes))
85     not_visted_node.remove(org)
86     next_node = Label_set[final_key][1]
87     while(True):
88         opt_route.append(next_node)
89         if(len(opt_route) == nodeNum + 1):
90             break
91         current_stage = len(opt_route)
92         not_visted_node.remove(next_node)
93         if (not_visted_node == None or len(not_visted_node) == 0):
94             not_visted_node = 'None'
95         next_key = (current_stage, next_node, str(not_visted_node))
96         next_node = Label_set[next_key][1]
97
98     opt_dis = Label_set[final_key][0]
99
100    print('objective :', Label_set[final_key][0])
101    print('optimal route :', opt_route)
102
103    return opt_dis, opt_route
104
105 # call function to solve TSP

```

```

105 opt_dis, opt_route = TSP_Dynamic_Programming(Nodes, dis_matrix)
106 print('\n\n----- optimal solution -----')
107 print('objective :', opt_dis)
108 print('optimal route :', opt_route)

```

运行结果如下：

```

result
----- optimal solution -----
1: objective : 35
2: optimal route : [1, 2, 4, 3, 1]

```

14.3.4 Python 实现：中大规模算例

我们以 Solomon 的 VRP 标杆算例来测试中大规模的情况。完整代码如下：

```

Dynamic Programming for Solving TSP

# * author : Liu Xinglu
# * Institute: Tsinghua University
# * Date : 2020-9-15
# * E-mail : hsinglul@163.com

# # Read data Function
import pandas as pd
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import copy
import re
import math
import itertools

class Data:
    customerNum = 0;
    nodeNum = 0;
    vehicleNum = 0;
    capacity = 0;
    cor_X = [];
    cor_Y = [];
    demand = [];
    serviceTime = [];


```

```

25     readyTime = [];
26     dueTime = [];
27     disMatrix = [[[]]]; # 读取数据
28
29 # function to read data from .txt files
30 def readData(data, path, customerNum):
31     data.customerNum = customerNum;
32     data.nodeNum = customerNum + 1;
33     # data.nodeNum = customerNum + 2;
34     f = open(path, 'r');
35     lines = f.readlines();
36     count = 0;
37     # read the info
38     for line in lines:
39         count = count + 1;
40         if(count == 5):
41             line = line[:-1].strip();
42             str = re.split(r" +", line);
43             data.vehicleNum = int(str[0]);
44             data.capacity = float(str[1]);
45             elif(count >= 10 and count <= 10 + customerNum):
46                 line = line[:-1];
47                 str = re.split(r" +", line);
48                 data.cor_X.append(float(str[2]));
49                 data.cor_Y.append(float(str[3]));
50                 data.demand.append(float(str[4]));
51                 data.readyTime.append(float(str[5]));
52                 data.dueTime.append(float(str[6]));
53                 data.serviceTime.append(float(str[7]));
54
55     # compute the distance matrix
56     data.disMatrix = [[0] * data.nodeNum for p in range(data.nodeNum)];
57     # 初始化距离矩阵的维度, 防止浅拷贝
58     # data.disMatrix = [[0] * nodeNum] * nodeNum; 这个是浅拷贝, 容易重复
59     for i in range(0, data.nodeNum):
60         for j in range(0, data.nodeNum):
61             temp = (data.cor_X[i] - data.cor_X[j])**2 + (data.cor_Y[i] -
62             data.cor_Y[j])**2;
63             data.disMatrix[i][j] = round(math.sqrt(temp), 1);
64             # if(i == j):
65             # data.disMatrix[i][j] = 0;
66             # print("%6.2f" % (math.sqrt(temp)), end = " ");

```

```

temp = 0;

return data

def printData(data, customerNum):
    print("下面打印数据\n");
    print("vehicle number = %d" % data.vehicleNum);
    print("vehicle capacity = %d" % data.capacity);
    for i in range(len(data.demand)):
        print('{0}\t{1}\t{2}\t{3}'.format(data.demand[i], data.readyTime[i],
            data.dueTime[i], data.serviceTime[i]));

print("-----距离矩阵-----\n");
for i in range(data.nodeNum):
    for j in range(data.nodeNum):
        #print("%d %d" % (i, j));
        print("%6.2f" % (data.disMatrix[i][j]), end = " ");
    print()

# # Read data
data = Data()
path = 'Solomon标准VRP算例/solomon-100/In/r101.txt'
customerNum = 20
readData(data, path, customerNum)
printData(data, customerNum)

# # Build Graph
# 构建有向图对象
Graph = nx.DiGraph()
cnt = 0
pos_location = {}
nodes_col = {}
nodeList = []
for i in range(data.nodeNum):
    X_coor = data.cor_X[i]
    Y_coor = data.cor_Y[i]
    name = str(i)
    nodeList.append(name)
    nodes_col[name] = 'gray'
    node_type = 'customer'
    if(i == 0):
        node_type = 'depot'

    pos_location[i] = (X_coor, Y_coor)
    Graph.add_node(name, pos=(X_coor, Y_coor), type=node_type)
    if(i != 0):
        Graph.add_edge('depot', name)
    else:
        Graph.add_edge(name, 'depot')
    cnt += 1

```

```

106 Graph.add_node(name
107     , ID = i
108     , node_type = node_type
109     , time_window = (data.readyTime[i], data.dueTime[i])
110     , arrive_time = 10000    # 这个是时间标签1
111     , demand = data.demand
112     , serviceTime = data.serviceTime
113     , x_coor = X_coor
114     , y_coor = Y_coor
115     , min_dis = 0          # 这个是距离标签2
116     , previous_node = None # 这个是前序节点标签3
117 )
118
119 pos_location[name] = (X_coor, Y_coor)
120 # add edges into the graph
121 for i in range(data.nodeNum):
122     for j in range(data.nodeNum):
123         if(i != j):
124             Graph.add_edge(str(i), str(j)
125                         , travelTime = data.disMatrix[i][j]
126                         , length = data.disMatrix[i][j]
127 )
128
129 # plt.rcParams['figure.figsize'] = (0.6 * trip_num, trip_num) # 单位是inch
130 nodes_col['0'] = 'red'
131 # nodes_col[str(data.nodeNum-1)] = 'red'
132 plt.rcParams['figure.figsize'] = (10, 10) # 单位是inch
133 nx.draw(Graph
134     , pos=pos_location
135     # , with_labels = True
136     , node_size = 50
137     , node_color = nodes_col.values()    #'y'
138     , font_size = 15
139     , font_family = 'arial'
140     # , edge_color = 'grey'  #'grey' # b, k, m, g,
141     , edgelist = []
142     # , nodelist = nodeList
143 )
144 fig_name = 'network_' + str(customerNum) + '_1000.jpg'
145 plt.savefig(fig_name, dpi=600)
146 plt.show()
147

```

```

188         if (subsub_set == None or len(subsub_set) == 0):
189             subsub_set = 'None'
190             sub_key = (stage_ID + 1, temp_next_node, str(subsub_
set))
191             if (sub_key in Label_set.keys()):
192                 if (dis_matrix[current_node - 1][temp_next_node -
1] + Label_set[sub_key][0] < min_distance):
193                     min_distance = dis_matrix[current_node - 1][
temp_next_node - 1] + Label_set[sub_key][0]
194                     next_node = temp_next_node
195                     Label_set[key] = [min_distance, next_node]
196
197             # print('current_node:', current_node, '\t left_node_set :',
# subset)
198
199             # stage 1 :
200             stage_ID = 1
201             current_node = org
202             subset = set(copy.deepcopy(Nodes))
203             subset.remove(org)
204             final_key = (stage_ID, current_node, str(subset))
205             min_distance = 1000000
206             for temp_next_node in subset:
207                 subsub_set = copy.deepcopy(subset)
208                 subsub_set.remove(temp_next_node)
209                 if (subsub_set == None or len(subsub_set) == 0):
210                     subsub_set = 'None'
211                     sub_key = (stage_ID + 1, temp_next_node, str(subsub_set))
212                     if (sub_key in Label_set.keys()):
213                         if (dis_matrix[current_node - 1][temp_next_node - 1] + Label_set
[sub_key][0] < min_distance):
214                             min_distance = dis_matrix[current_node - 1][temp_next_node -
1] + Label_set[sub_key][0]
215                             next_node = temp_next_node
216                             Label_set[final_key] = [min_distance, next_node]
217
218             # get the optimal solution
219             opt_route = [org]
220             not_visted_node = set(copy.deepcopy(Nodes))
221             not_visted_node.remove(org)
222             next_node = Label_set[final_key][1]
223             while(True):

```

```

opt_route.append(next_node)
if(len(opt_route) == nodeNum + 1):
    break
current_stage = len(opt_route)
not_visted_node.remove(next_node)
if (not_visted_node == None or len(not_visted_node) == 0):
    not_visted_node = 'None'
next_key = (current_stage, next_node, str(not_visted_node))
next_node = Label_set[next_key][1]

opt_dis = Label_set[final_key][0]

print('objective :', Label_set[final_key][0])
print('optimal route :', opt_route)

return opt_dis, opt_route

opt_dis, opt_route = TSP_Dynamic_Programming(Nodes, dis_matrix)
print('\n\n ----- optimal solution ----- \n')
print('objective :', opt_dis)
print('optimal route :', opt_route)

```

上述代码中，我们以 R101 作为测试算例，取前 20 个点构建网络，求解了一个 21 个点的 TSP（加上 depot 是 21 个点）。运行结果如下：

result
1 stage : 21
2 stage : 20
3 stage : 19
4 stage : 18
5 stage : 17
6 stage : 16
7 stage : 15
8 stage : 14
9 stage : 13
10 stage : 12
11 stage : 11
12 stage : 10
13 stage : 9
14 stage : 8
15 stage : 7
16 stage : 6
17 stage : 5

```

18 stage : 4
19 stage : 3
20 stage : 2
21 objective : 262.9
22 optimal route : [1, 14, 7, 6, 18, 17, 15, 16, 3, 5, 13, 4, 10, 21, 2, 11,
12,
23 20, 8, 9, 19, 1]
24 ----- optimal solution -----
25 objective : 262.9
26 optimal route : [1, 14, 7, 6, 18, 17, 15, 16, 3, 5, 13, 4, 10, 21, 2, 11,
12,
27 20, 8, 9, 19, 1]

```

14.4 标签算法求解带资源约束的最短路问题

前面章节介绍了动态规划求解 TSP 的原理，我们讲到，该方法仍旧是一个指数时间复杂度的算法，求解效率并不令人满意。对于 TSP，目前为止也没有能够求解超大规模算例的高效精确算法。但是对于一些其他路径规划相关的问题，却存在伪多项式时间的动态规划算法，或者较为高效的动态规划算法。本节需要讨论的带资源约束的最短路问题（Shortest Path Problem with Resource Constraints, SPPRC）就是一类存在伪多项式时间精确算法的问题。1986 年，Desrochers 博士在其博士论文中第一次提出 SPPRC (Desrochers, 1986)，之后，该问题被广泛地研究和拓展，也衍生出许多变种。特别地，SPPRC 的一个著名的变种，即带资源约束的基本最短路问题 (Elementary Shortest Path Problem with Resource Constraints, ESPPRC)，是 VRPTW 的列生成算法框架的子问题。ESPPRC 比 SPPRC 更难求解，因为 ESPPRC 要求路径中的节点至多只被访问一次，而 SPPRC 中允许同一个节点被访问多次。ESPPRC 已经被认定为强 NP-hard 问题 (Dror, 1994)。求解 SPPRC 和 ESPPRC 的动态规划算法有一个新的名称，即标签算法。在列生成算法的相应章节中，我们讲到，标签算法包括标签校正算法和标签设定算法（例如 Dijkstra）。标签算法本质上也是动态规划算法。本节介绍的标签算法属于标签校正算法。

14.4.1 带资源约束的最短路问题

本节以一个简单案例来介绍 SPPRC。考虑如图 14.10 所示的网络（例子来源于文献 (Desaulniers et al., 2006)）。其中，点 s 表示起点 (source 或 origin)，点 t 表示终点 (sink 或 destination)。节点 i 上方的数组表示该点的最早开始服务时间 a_i 和最晚结束时间 b_i ，即该点的时间窗为 $[a_i, b_i]$ ，例如点 1 的时间窗为 $[6, 14]$ 。图中弧 (i, j) 上的元组 (t_{ij}, c_{ij}) ，分别代表弧 (i, j) 上的行驶时间和成本（如距离等），例如，弧 $(s, 1)$ 的行驶时间和成本构成的元组为 $(8, 3)$ 。SPPRC 的目标就是找到一条从起点 s 出发到达终点 t 的成本最小的路径，并且该路径中途经的所有点都必须在其时间窗内被访问。

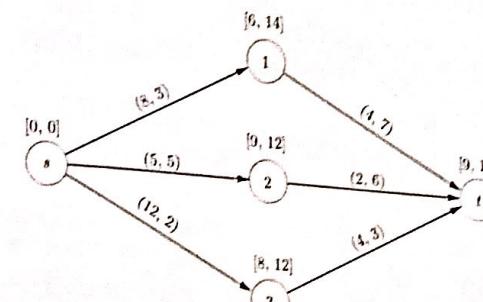


图 14.10 SPPRC 小例子

在上述 SPPRC 中，时间和成本可以看作两个资源，而考虑上述两个资源的 SPPRC，一般又叫作 Shortest Path Problem with Time Windows (SPPTW)，该问题由 Desrosiers 等在 1983 年首次提出 (Desrosiers et al., 1983; Desrosiers et al., 1984)。其中，时间资源是有限的，而成本资源是无限的。Desrosiers 等学者之后又将其进行了拓展，将其一般化为考虑多种资源 (Desrochers, 1986) 的情形，在那之后，关于 SPPRC 的研究日益增多。例如，文献 (Ioachim et al., 1998) 提出了考虑节点的时间依赖成本的 SPPTW；文献 (Dumas et al., 1991) 研究了考虑搭载和配送因素的 SPPTW。

SPPRC 可以描述为：给定一个有向图 $G = (V, A)$ ，其中 $V = \{1, 2, \dots, N\}$ 是图中点的集合， $A = \{(i, j) | \forall i, j \in V, i \neq j\}$ 是图中弧的集合。 $R = \{1, 2, \dots, K\}$ 表示所有资源的集合。 G 中每条弧 (i, j) 都对应一个资源向量 (r_1, r_2, \dots, r_K) ，表示该条弧上的所有资源的消耗量。每个点 i 对应一个资源约束（例如时间窗）。给定起点 s 和终点 t ，SPPRC 的目标就是找到一条从起点 s 出发到达终点 t 的成本最小的路径，并且该路径中途经的所有点都必须满足该点的资源约束。

这里以之前介绍过的 VRPTW 的子问题为例，给出 SPPRC 的整数规划模型。根据列生成相关章节的介绍，考虑时间、行驶成本和载重量 3 种资源的 SPPRC 的整数规划模型可以写成

$$\min \sum_{i \in N} \sum_{j \in N} c_{ij} x_{ij} \quad (14.2)$$

$$\text{s.t. } \sum_{i \in C} d_i \sum_{j \in N} x_{ij} \leq q \quad (14.3)$$

$$\sum_{j \in N} x_{0j} = 1 \quad (14.4)$$

$$\sum_{i \in N} x_{ih} - \sum_{j \in N} x_{hj} = 0, \quad \forall h \in C \quad (14.5)$$

$$\sum_{i \in N} x_{i, n+1} = 1 \quad (14.6)$$

$$s_i + t_{ij} - M(1 - x_{ij}) \leq s_j, \quad \forall i, j \in N \quad (14.7)$$

$$a_i \leq s_i \leq b_i, \quad \forall i \in N \quad (14.8)$$

$$x_{ij} \in \{0, 1\}, \quad \forall i, j \in N \quad (14.9)$$

根据上述模型，可以调用求解器求解 SPPRC。但是本章并不打算用求解器求解，而是用标签法求解。

首先回到图 14.10。SPPRC 实际上就是在所有从起始点 s 出发到终点 t 的所有路径当中，选择一条最短的路径，并且满足一系列的资源约束。上述例子中，唯一的资源约束就是时间约束。在其他问题中，可能会有另外的资源，如载重、换班之间的停顿时间、搭载和配送约束等。这些资源都是随着路径的扩展，按照某种函数关系变化的，这种函数我们称为资源扩展函数（Resource Extension Function, REF）。在例子（图 14.10）中，我们用 T_i 表示在点 i 处时间资源的使用量（即到达点 i 的时间），则弧 (i, j) 的 REF 可以定义为

$$f_{ij}(T_i) = T_i + t_{ij}$$

其中， f_{ij} 就是弧 (i, j) 上的 REF。这个 REF 就是根据弧 (i, j) 中点 i 的访问时间，来更新点 j 的访问时间的函数。这个函数可以计算出在 T_i 时刻从点 i 出发，到达点 j 的最早时间。如果一条路径中，到达点 i 的时间超过了 b_i ，则这个路径就不可行。但是，如果到达时间在 a_i 之前，则允许在点 i 等待一段时间。也就是说，到达点 i 的时间可以小于访问点 i 的时间。

在图 14.10 中，从起点到终点有 3 条路径。第 1 条路径 $P_1 = (s, 1, t)$ 是资源可行的路径。的确，令 $T_s = 0$ （在点 s 处的唯一可行值），我们很容易就可以得到在该条路径下，到达点 1 和 t 的时间 $(T_1 = 8, T_t = 12)$ 。这两个值是根据所有的资源扩展函数（REF）得到的 $(f_{s1}(T_s) \text{ 和 } f_{1t}(T_1))$ 。

第 2 条路径 $P_2 = (s, 2, t)$ 也是满足资源约束的路径。然而，在点 2 会有一些等待时间，到达点 2 的时间可以通过 $f_{s2}(T_s) = 5$ 计算，但是 $f_{s2}(T_s) = 5 < a_2 = 9$ ，在这种情况下，点 2 的访问时间可以设置成 $T_2 = 9$ ，并且在这条路径中，点 t 的访问时间可以设置成 $T_t = T_2 + 2 = 11$ 。

第 3 条路径 $P_2 = (s, 3, t)$ 是资源不可行的，在这条路径上， $T_s = 0, T_3 = 12 \geq f_{s3} = 12$ ，到达点 t 的最早时间为 $f_{3t}(T_3) = f_{3t}(12) = 12 + 4 = 16$ ，由于点 t 的时间窗为 $[9, 15]$ ，因此这条路径是不可行的。

考核 2 条可行路径 P_1, P_2 ， P_1 的成本为 $3 + 7 = 10$ ，比 P_2 的成本 $5 + 6 = 11$ 小，因此只考虑成本的话， P_1 是最优的。由于路径 P_2 有更早的到达时间（到达终点 t ），如果图 14.10 只是一个大网络中的一个子网络，在拓展的时候，路径 P_2 有可能是一个可行路径，但是路径 P_1 有可能不是（由于 P_1 使用了更多的资源，可能导致后续拓展违背资源约束）。

以上描述让我们初步了解了 SPPRC 的难度。SPPRC 很接近于多标准问题。仅考虑单一标准的话，无法判断哪条路径是更好的，也就是很多时候，路径之间是不可比的，这也是 SPPRC 的困难之处。

不同类型的 SPPRC 可以根据下面的特征进行分类：

（1）资源累积的方式（不同的累积方式导致资源可行路径（Resource Feasible Path）定义不同）：

（2）是否存在额外的路径结构约束（Path-structural Constraint），如非基本路径、基本路径等；

（3）目标函数；

（4）基础网络结构。

这里用 $P = (v_0, v_1, \dots, v_p)$ 表示一条路径，其中某个点可能会出现多次。路径 P 的长度为 p 。接下来我们针对上面提出的 4 点来进行阐述。

1. 资源可行路径

在接下来的描述中，我们将区分资源的可行性与路径结构约束的可行性。资源约束可以通过资源消耗和资源间隔（也叫资源上下限、资源窗口等，例如 SPPTW 中的行驶时间 t_{ij} 和时间窗 $[a_i, b_i]$ ）来表示。设资源的数量为 R 。向量 $T = (T^1, T^2, \dots, T^R)^T \in \mathbb{R}^R$ 称为资源向量，对应的资源分配量就是决策变量，称为资源决策变量 x^T 。所有资源的可用量为 $S = (S^1, S^2, \dots, S^R)^T \in \mathbb{R}^R$ 。如果对于每一种资源 i ，都有 $T^i \leq S^i$ ，则我们说 $T \leq S$ 。例如，如果资源窗口为 $[a, b]$ ，则资源向量可以定义为集合 $\{T \in \mathbb{R}^R : a \leq T \leq b\}$ 。

接下来要引入的概念是资源间隔（Resource Interval），也叫资源窗口（Resource Window）。每个节点 $i \in V$ 的资源窗口可以表示为 $[a_i, b_i]$ ，其中 $a_i, b_i \in \mathbb{R}^R, a_i \leq b_i$ 。资源的消耗总是随着弧 (i, j) 变化的，前文中也有所介绍。我们称为资源扩展函数（Resource Extension Function, REF）。弧 (i, j) 的 REF 是一个向量，即 $f_{ij} = (f_{ij}^r)_{r=1}^R$ 。REF $f_{ij} : \mathbb{R}^R \rightarrow \mathbb{R}$ 是依赖于资源向量 $T_i \in \mathbb{R}^R$ 的，也就是说，是取决于从起点 s 到点 i 的路径上的资源的累计消耗量的。因此 $f_{ij}(T_i) \in \mathbb{R}^R$ 可以解释为沿着路径 (s, \dots, i, j) 所累积的资源消耗。这里以之前介绍的 SPPTW 为例，仅考虑时间资源，则其 REF 为

$$f_{ij}(T_i) = T_i^r + t_{ij}^r \quad (14.10)$$

其中， t_{ij}^r 是与弧 (i, j) 相关的常数，在 SPPTW 中，就是弧 (i, j) 上的行驶时间。一般情况下，REF 是资源可分离的，即不同资源之间不存在相互依赖关系。

给定路径 $P = (v_0, v_1, v_2, \dots, v_p)$ ，该路径中包含 $p + 1$ 个不同的位置 i ，即 $i = 0, 1, 2, \dots, p$ 。路径 P 是资源可行（Resource-feasible）的条件是：存在资源向量 $T_i \in [a_{v_i}, b_{v_i}], \forall i = 0, 1, 2, \dots, p$ ，使得 $f_{v_i, v_{i+1}}(T_i) \leq T_{i+1}, \forall i = 0, 1, 2, \dots, p$ 。我们定义 $\mathcal{T}(P)$ 为路径 P 的最后一个节点 v_p 的所有可行资源向量的集合，即

$$\mathcal{T}(P) = \{T_p \in [a_{v_p}, b_{v_p}] : \exists T_i \in [a_{v_i}, b_{v_i}], f_{v_i, v_{i+1}}(T_i) \leq T_{i+1}, \forall i = 0, 1, 2, \dots, p-1\} \quad (14.11)$$

令 $\mathcal{F}(u, v)$ 为从节点 u 到节点 v 的所有资源可行路径的集合。

2. 路径结构约束

路径结构约束 (Path-structural Constraint) 是关于路径可行性的进一步建模需求, 是独立于资源约束的。一般情况下, 路径结构约束不能简单地通过删除网络中的一些弧和点来等价地处理。通常情况下, 我们考虑的路径结构约束都是基本路径 (Elementary Path)。Elementary Path 就是路径中没有环路的路径, 换句话说, 就是所有被访问的点, 至多只被访问了一次。相反地, 一个环 (cycle) 就是一条起点和终点相同的路径, 即路径 $(v_0, v_1, v_2, \dots, v_p)$ 是环, 则 $v_0 = v_p, p > 1$ 。我们将路径长度小于或等于 k 的环称为 k -cycle。

这里用 \mathcal{G} 表示满足路径结构约束的所有可行路径。

对于 Elementary SPPRC (ESPPRC), 则 $\mathcal{G} = \{\text{Elementary Path}\}$ 。在无环图中 (Acyclic Graph), 所有路径都是基本路径, 因此 SPPRC 和 ESPPRC 是相同的。但是对于有环图来讲, ESPPRC 已经被证明是强 NP-hard 问题 (Dror, 1994), 并由 Beasley 和 Christofides 于 1989 年首先研究和解决。在许多 VRP 的应用中, 其定价问题就是 ESPPRC。

对于 SPPRC, $\mathcal{G} = \{\text{All Path}\}$, 即不考虑路径结构约束。许多车辆和人员排班问题的子问题都是 SPPRC (Desrosiers et al., 1984)。

由于 ESPPRC 很难求解 (在某些情况下是非常困难的), 在有环图上建模的 VRP 也常常用求解 SPPRC 的方法代替, 因为 SPPRC 可以用伪多项式时间的精确算法进行求解。用 SPPRC 替代 ESPPRC 实际上是对 VRP 子问题的一种松弛。将 ESPPRC 松弛成 SPPRC 虽然会使得子问题更容易求解, 但是这种松弛不一定总是好的。比如在使用分支定价算法求解 VRP 时, 子问题如果松弛成 SPPRC, 很可能会导致获得的下界比较松, 而且很有可能导致分支树 (Branch and Bound tree) 变得非常大, 反而对求解效率造成不利的影响 (Desaulniers et al., 2006)。

除了 ESPPRC 和 SPPRC, 还有一些路径结构约束, 例如:

- (1) k 环消除的 SPPRC (SPPRC with k -cycle elimination, SPPRC- k -cyc);
- (2) 带禁止路径的 SPPRC (SPPRC with forbidden paths, SPPRCFP);

(3) 带先后顺序和配对约束的 SPPRC (SPPRC with Precedence constraints and pairing constraints), 这一类问题主要针对取货和送货的路径结构约束。

这里我们仅介绍 k 环消除约束的 SPPRC。

对于 k 环消除约束的 SPPRC, 则 $\mathcal{G} = \{k\text{-cycle-free path}\}$, 即没有 k -cycle 的所有路径的集合。这也是解决 ESPPRC 比较难求解的一个折中方案, 即禁止长度较短的环。根据 Solomon VRP 标杆算例的测试结果, 仅取 k 较小的值, 也可以比较显著地提升 Master Problem 的下界。这说明, 除了求解纯 SPPRC 之外, 还需要对 SPPRC 的解进行额外的约束, 也就是删除一些环, 这相对于直接求解 ESPPRC 来讲会容易很多。Houck et al., 1980 首次讨论了 $k = 2$ 的情况, 而这种做法也被之后的研究广泛采用 (Kolen et al., 1987; Desrochers et al., 1992)。

对于目标函数的分类, 这里不做解释, 感兴趣的读者可以阅读文献 (Desaulniers et al., 2006)。

3. 基础网络

SPPRC 还可以根据其基础网络 (Underlying Network) 是无环的还是有环的加以区分。有环的网络意味着在 G 中存在无穷多个不同的路径 (不一定是资源可行路径和满足路径结构约束的路径)。因此, SPPRC 可能是无界的 (如果网络中存在负环, 则 SPPRC 可能是无界的)。这里不考虑无界的情况。

对于一些特殊的情况, 可以通过离散化的手段, 将有环图转化为无环图, 而求解原网络图上的 ESPPRC, 就等价于求解离散化之后的网络图上的 SPPRC。如果存在至少一个非递减资源 r (即 $f_{ij}^r - T_i^r > 0$, 或者 $t_{ij}^r > 0$), 该网络图 $G = (V, A)$ 就有可能被转化成一个无环的时空网络 (Acyclic Time-space Network)。对于每个节点 $v \in V$, 我们将该点对应的资源 r 的资源窗口 (Resource Interval) 离散化成 p 段, 用一些副本 $\text{copy}^1(v), \text{copy}^2(v), \dots, \text{copy}^p(v)$ 来代替 G 中的点 v 。然而, 这种转换只是一种形式手段, 转换完成后, 仍然可以套用文献 (Desaulniers et al., 1998) 中提出的方法进行建模。当然, 按照这种方法转化之后, 求解转化后的无环的时空网络上的 SPPRC, 就等价于求原网络图上的 ESPPRC。

14.4.2 标签算法

求解 SPPRC 的动态规划算法又叫标签算法。该算法从一个初始路径 $P = (s)$ 开始, 一步一步地沿着所有的可行方向拓展初始路径, 从而生成所有的可行路径。该算法的效率取决于识别和舍弃那些对于构建或者生成帕累托最优 (Pareto-optimal) 路径集合无用的路径的能力。识别无用路径是通过优超准则 (Dominance Rule) 实现的, 该准则非常依赖于路径结构约束和 REF 的特性。

为了方便描述, 标签算法中的路径及其资源向量用 label (标签) 进行编码或者表示, 通常将一个部分路径及其资源向量称为一个标签 (label)。

对于给定的路径 $P = (v_0, v_1, v_2, \dots, v_p)$, 我们用 $v(P) = v_p$ 表示路径 P 的最后一个节点。如果路径 P 是路径 $Q = (w_0, w_1, w_2, \dots, w_q)$ 的一个可行拓展, 则有

$$(Q, P) = (w_0, w_1, w_2, \dots, w_q, v_0, v_1, v_2, \dots, v_p) \in \mathcal{F}(w_0, v_p) \cup \mathcal{G}$$

Q 的所有可行拓展的集合表示为 $\mathcal{E}(Q) = \{P : (Q, P) \in \mathcal{F}(w_0, v(P)) \cup \mathcal{G}\}$ 。

标签算法依赖于对两个集合的操作。第一个集合表示为 \mathcal{U} , 是未处理的路径的集合, \mathcal{U} 中的路径都是未被扩展的。第二个集合 \mathcal{P} 是有用路径的集合。有用的路径 $P \in \mathcal{P}$ 是已经被处理过的。它们已经被确定为是帕累托最优路径或可能是帕累托最优路径的前缀 (注意, 帕累托最优路径可能会有非帕累托最优的前缀)。集合 \mathcal{U} 和 \mathcal{P} 在标签算法过程中都是动态变化的。最终 \mathcal{U} 变为空时, 算法结束, 而最优解一定在集合 \mathcal{P} 中, 我们只需要将其筛选出来即可。

标签算法中, 我们首先初始化集合 $\mathcal{U} = P_0, \mathcal{P} = \emptyset$, 其中 $P_0 = (s)$ 是初始路径。每一条路径 $P = (v_0, v_1, v_2, \dots, v_p) \in \mathcal{F}$ 都是从初始路径 P_0 拓展而来的, 即 $(v_1, v_2, \dots, v_p) \in \mathcal{E}(P_0)$ 。

在处理完所有拓展后，我们需要执行最后的筛选，也就是从有用路径集合 \mathcal{P} 中，筛选出帕累托最优解 \mathcal{S} （有可能有多个）。在每一轮的循环中，拓展步骤结束之后，我们可以对 \mathcal{U} 和 \mathcal{P} 中的标签（也就是路径）进行优超（Dominance）。这一步往往是加速算法的关键。

14.4.3 标签算法的伪代码

上一节我们详细介绍了标签算法的原理，本节我们根据之前的介绍，整理出标签算法的伪代码。

Algorithm 14 SPPRC/ESPPRC 的通用动态规划算法（标签算法）

Input: 网络数据（弧集，距离信息）和客户数据（时间窗），起始点 s ，终止点 t
Output: 最短路 optPath

1. 初始化：设置 $\mathcal{U} \leftarrow \{(s)\}$ 和 $\mathcal{P} \leftarrow \emptyset$
2. while \mathcal{U} 非空 do
 3. (**** 拓展路径 ****)
 4. 选择一条部分路径 $Q \in \mathcal{U}$ 并将 Q 从 \mathcal{U} 中删除
 5. for 弧 $(v_Q, w) \in A$ 是从点 v_Q 出去的 do
 6. if (Q, w) 是可行拓展 then
 7. 将 (Q, w) 添加至 \mathcal{U}
 8. end if
 9. end for
 10. $\mathcal{P} \leftarrow \mathcal{P} \cup \{Q\}$
 11. (**** 优超 ****)
 12. if 任意的条件（可自己定义）then
 13. 对 $\mathcal{U} \cup \mathcal{P}$ 中最后一个节点为 v 的部分路径执行优超算法
 14. end if
 15. end while
 16. (**** 筛查 ****)
 17. $\text{optPath} \leftarrow$ 对 \mathcal{P} 进行筛查，识别出最优解
 18. return optPath

如果求解的问题是 ESPPRC，则只需将伪代码中第 6 行改为“如果拓展 (Q, w) 是可行的，且满足基本路径约束”。另外，优超准则部分也需要做相应的修改。

针对以上伪代码，有以下几点需要注意的地方。

(1) 如果只执行路径扩展步骤，而不执行优超步骤，我们将得到所有的可行路径，即 $\mathcal{P} = \mathcal{F}$ 。

(2) 在路径扩展时，可以根据不同的规则选择下一个被拓展的路径，也就是说，在选择 $Q \in \mathcal{U}$ 时，可以有不同的策略。不同的选择策略、基础网络和 REF 将会使得算法变成标签设定算法或者标签校正算法。

(3) 在算法进行过程中，优超算法（Dominance Algorithm）可以随时被调用。为了减少计算量，可以设置优超算法被调用的时机，以便于优超算法能够同时删除多条路径。

(4) 对于筛选步骤，是有一些比较有效的算法的，例如从 \mathcal{P} 中识别出帕累托最优路径（Bentley, 1980; Kung et al., 1975）。

14.4.4 标签设定算法和标签校正算法

上一节我们提到了标签设定（Label Setting）和标签校正（Label Correction）算法。二者都是动态规划算法，但是又有一些区别。

在标签设定算法中，那些选择要扩展的标签（在路径扩展步骤中）一直保留到标记过程结束。在后续的优超算法调用中，它们将不会被识别为可删除或者可丢弃的。Dijkstra 算法就是一种典型的标签设定算法。不能保证这种行为的标签算法称为标签校正算法，也就是说，被扩展的标签在算法调用中可能会被丢弃。

14.4.5 优超准则和优超算法

优超准则（Dominance Rule）和优超算法（Dominance Algorithm）是提升 SPPRC 和 ESPPRC 求解的重要武器。如果一条路径 Q 既不能产生帕累托最优解 $\text{PO}(v(Q))$ ，也不能产生可行扩展 $Q' \in \mathcal{E}(Q)$ ，使得路径 (Q, Q') 可以产生帕累托最优解 $\text{PO}(v(Q'))$ ，则路径 Q 就可以被删除。如果引入的优超准则和优超算法非常有效，则在迭代过程中 \mathcal{U} 和 \mathcal{P} 的集合中的元素的个数会明显减少，从而可以显著地加快求解速度。

一般来讲，优超准则是通过比较两个具有相同尾节点的路径 P 和 $Q(v(P) = v(Q))$ 的资源向量 $T(P)$ 、 $T(Q)$ 和可行拓展集合 $\mathcal{E}(P)$ 、 $\mathcal{E}(Q)$ 来识别无用路径的。本节来探讨 SPPRC、ESPPRC 和 SPPRC-2-cyc 共 3 种情况下的优超准则。还有一种是 SPPRC- k -cyc，本章不做详细阐述，感兴趣的读者可以参考文献（Desaulniers et al., 2006）。

1. SPPRC

给定两个不同的路径 $P, Q \in \mathcal{U} \cup \mathcal{P}$ ， $v(P) = v(Q)$ ，且 $T(P) \leq T(Q)$ ，并且满足下面两种情况，那么我们就可以删除路径 Q 。

(1) $T(P) \leq T(Q)$ 表示 $T(Q)$ 不可能产生更好的帕累托最优解。

(2) 如果通过探索 Q 的所有可行扩展，得出 $\mathcal{E}(Q) \subseteq \mathcal{E}(P)$ ，则可以删除 Q 。特别的，如果没有任何路径结构约束，并且 REF 都是非递减的，则显然有 $\mathcal{E}(Q) \subseteq \mathcal{E}(P)$ 。

例如，在 SPPTW 中， $T(P) \leq T(Q)$ 就表示与 Q 相比， P 所用时间短，花费成本也小。并且由于 SPPTW 中资源全是非递减的，因此就有 $\mathcal{E}(Q) \subseteq \mathcal{E}(P)$ 。根据上文介绍，我们得知， Q 可以被删除。

2. ESPPRC

对于 ESPPRC 来讲， $T(P) \leq T(Q)$ 并不能说明 $\mathcal{E}(Q) \subseteq \mathcal{E}(P)$ ，原因是，在 ESPPRC 中，路径 $P \in \mathcal{G}$ 只能被拓展到那些还没有被访问的点处。我们用 $V(P)$ 表示已经被访问的

点的集合。因此, ESPPRC 中, 删去 Q 的条件为 $T(P) \leq T(Q)$, 且 $V(P) \subseteq V(Q)$ (这两个条件同时满足, 就可以说明 $\mathcal{E}(Q) \subseteq \mathcal{E}(P)$)。其他详细解释见文献 (Desaulniers et al., 2006)。

3. SPPRC-2-cyc

对于 2-cycle Elimination 的情形, 优超准则的一种直观描述如下: 只保留一条帕累托最佳路径 P_1 和一条次优路径 P_2 , 这两条路径是从不同的前序节点 (Predecessor node) 扩展而来的。对于任意路径 $P = (v_0, v_1, v_2, \dots, v_p), p \geq 1$, 节点 v_{p-1} 被称为前序节点, 用 $\text{pred}(P)$ 表示。则任给三条路径 P_1, P_2, Q , 满足 $v(P_1) = v(P_2) = v(Q)$, 且 $T(P_1), T(P_2) \leq T(Q), \text{pred}(P_1) \neq \text{pred}(P_2)$, 则我们可以删去 Q 而保留 P_1, P_2 。原因是 $\text{pred}(P_1) \neq \text{pred}(P_2)$ 表明 $\mathcal{E}(P) \subseteq \mathcal{E}(P_1) \cup \mathcal{E}(P_2)$ 。

14.4.6 Python 实现标签算法求解 SPPRC

介绍完算法理论细节之后, 现在进入实战环节。本节我们给出 Python 实现标签算法求解 SPPRC 的完整代码。我们用 Solomon VRPTW 标杆算例来测试算法。本代码为入门代码, 不保证效率, 读者需要自行改进。

1. 读取算例数据

读取算例数据代码如下。

```
readData
1 class Data:
2     customerNum = 0;
3     nodeNum = 0;
4     vehicleNum = 0;
5     capacity = 0;
6     cor_X = [];
7     cor_Y = [];
8     demand = [];
9     serviceTime = [];
10    readyTime = [];
11    dueTime = [];
12    disMatrix = [[]]; # 读取数据
13
14    # function to read data from .txt files
15    def readData(data, path, customerNum):
16        data.customerNum = customerNum;
17        data.nodeNum = customerNum + 2;
18        f = open(path, 'r');
19        lines = f.readlines();
20        count = 0;
```

```
# read the info
for line in lines:
    count = count + 1;
    if(count == 5):
        line = line[:-1].strip();
        str = re.split(r" +", line);
        data.vehicleNum = int(str[0]);
        data.capacity = float(str[1]);
    elif(count >= 10 and count <= 10 + customerNum):
        line = line[:-1];
        str = re.split(r" +", line);
        data.cor_X.append(float(str[2]));
        data.cor_Y.append(float(str[3]));
        data.demand.append(float(str[4]));
        data.readyTime.append(float(str[5]));
        data.dueTime.append(float(str[6]));
        data.serviceTime.append(float(str[7]));

        data.cor_X.append(data.cor_X[0]);
        data.cor_Y.append(data.cor_Y[0]);
        data.demand.append(data.demand[0]);
        data.readyTime.append(data.readyTime[0]);
        data.dueTime.append(data.dueTime[0]);
        data.serviceTime.append(data.serviceTime[0]);

# compute the distance matrix
data.disMatrix = [[0] * data.nodeNum for p in range(data.nodeNum)];
# 初始化距离矩阵的维度, 防止浅拷贝
# data.disMatrix = [[0] * nodeNum] * nodeNum; 这个是浅拷贝, 容易重复
for i in range(0, data.nodeNum):
    for j in range(0, data.nodeNum):
        temp = (data.cor_X[i] - data.cor_X[j])**2 + (data.cor_Y[i] - data.cor_Y[j])**2;
        data.disMatrix[i][j] = math.sqrt(temp);
        # if(i == j):
        #     data.disMatrix[i][j] = 0;
        # print("%6.2f" % (math.sqrt(temp)), end = " ");
        temp = 0;

return data;
```

```

61 def printData(data, customerNum):
62     print("下面打印数据\n");
63     print("vehicle number = %d" % data.vehicleNum);
64     print("vehicle capacity = %d" % data.capacity);
65     for i in range(len(data.demand)):
66         print('{0}\t{1}\t{2}\t{3}'.format(data.demand[i], data.readyTime[i],
67                                         data.dueTime[i], data.serviceTime[i]));
68
69     print("-----距离矩阵-----\n");
70     for i in range(data.nodeNum):
71         for j in range(data.nodeNum):
72             # print("%d %d" % (i, j));
73             print("%6.2f" % (data.disMatrix[i][j]), end = " ");
74         print();
75
76 # reading data
77 data = Data()
78
79 path = 'Solomon标准VRP算例/solomon-100/In/c101.txt'
80 customerNum = 100
81 readData(data, path, customerNum)
82 printData(data, customerNum)

```

2. 构建网络图

我们用 Python 的 networkx 包来构建网络图，实现数据的可视化。代码如下。

buildGraph

```

1 # 构建有向图对象
2 Graph = nx.DiGraph()
3 cnt = 0
4 pos_location = {}
5 nodes_col = {}
6 for i in range(data.nodeNum):
7     X_coor = data.cor_X[i]
8     Y_coor = data.cor_Y[i]
9     name = str(i)
10    nodes_col[name] = 'gray'
11    node_type = 'customer'
12    if(i == 0):
13        node_type = 'depot'
14    Graph.add_node(name
15                  , ID = i

```

```

, node_type = node_type
, time_window = (data.readyTime[i], data.dueTime[i])
, arrive_time = 10000 # 这个是时间标签1
, demand = data.demand
, serviceTime = data.serviceTime
, x_coor = X_coor
, y_coor = Y_coor
, min_dis = 0
# 这个是距离标签2
, previous_node = None # 这个是前序节点标签3
)

pos_location[name] = (X_coor, Y_coor)
# add edges into the graph
for i in range(data.nodeNum):
    for j in range(data.nodeNum):
        if(i == j or (i == 0 and j == data.nodeNum - 1) or (j == 0 and i == data.nodeNum - 1)):
            pass
        else:
            Graph.add_edge(str(i), str(j)
                           , travelTime = data.disMatrix[i][j]
                           , length = data.disMatrix[i][j]
                           )
# 画出图
# plt.rcParams['figure.figsize'] = (0.6 * trip_num, trip_num) # 单位是inch
nodes_col['0'] = 'red'
nodes_col[str(data.nodeNum-1)] = 'red'
plt.rcParams['figure.figsize'] = (10, 10) # 单位是inch
nx.draw(Graph
         , pos=pos_location
         , with_labels = True
         , node_size = 200
         , node_color = nodes_col.values() # 'y'
         , font_size = 15
         , font_family = 'arial'
         , edge_color = 'grey' #'grey' # b, k, m, g,
         )
fig_name = 'network_' + str(customerNum) + '.jpg'
plt.savefig(fig_name, dpi=600)
plt.show()

```

网络图如图 14.11 所示。

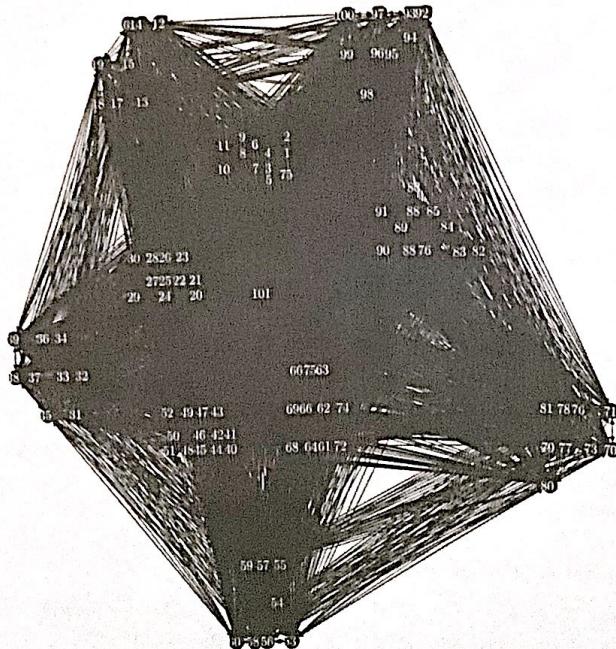


图 14.11 网络图: 100 个顾客

3. 标签算法

标签算法代码如下。

Labelling Algorithm

```

1 class Label:
2     path = []
3     time = 0
4     dis = 0
5
6
7 def labelling_SPPRC(Graph, org, des):
8     # initialize Queue
9     Queue = []
10    # creat initial label
11    label = Label()
12    label.path = [org]
13    label.dis = 0
14    label.time = 0
15
16
17    Queue.append(label)
18    Paths = {}
19    cnt = 0
20
21    while(len(Queue) > 0):
22        cnt += 1
23        current_path = Queue[0]
24        Queue.remove(current_path)
25        # extend the label
26        last_node = current_path.path[-1]
27        for child in Graph.successors(last_node):
28            extended_path = copy.deepcopy(current_path)
29            arc_key = (last_node, child)
30            # justify whether the extension is feasible
31            arrive_time = current_path.time + Graph.edges[arc_key]['travelTime']
32            time_window = Graph.nodes[child]['time_window']
33            if((child not in extended_path.path) and arrive_time >= time_
34                window[0] and arrive_time <= time_window[1]):
35                # the extension is feasible
36                # print('extension is feasible', 'arc : ', arc_key)
37                extended_path.path.append(child)
38                extended_path.dis += Graph.edges[arc_key]['length']
39                extended_path.time += Graph.edges[arc_key]['travelTime']
40                Queue.append(extended_path)
41                print('extended_path : ', extended_path.path)
42            else:
43                pass
44                # print('extension is infeasible', 'arc : ', arc_key)
45        Paths[cnt] = current_path
46
47
48        # dominate step
49        ...
50
51        Add dominate rule function
52        ...
53
54        Queue, Paths = dominate(Queue, Paths)
55
56
57        # filtering Paths, only keep solutions from org to des, delete other
58        # paths
59        PathsCopy = copy.deepcopy(Paths)
60        for key in PathsCopy.keys():
61            if(Paths[key].path[-1] != des):
62                Paths.pop(key)
63

```

```

54     # choose optimal solution
55     opt_path = {}
56     min_distance = 10000000
57     for key in Paths.keys():
58         if(Paths[key].dis < min_distance):
59             min_distance = Paths[key].dis
60             opt_path['1'] = Paths[key]
61
62     return Graph, Queue, Paths, PathsCopy, opt_path
63

```

4. 优超准则

我们加入优超准则 (Dominate rule)。若路径 P, Q 满足

$$t(P) \leq t(Q) \quad (14.12)$$

$$c(P) \leq c(Q) \quad (14.13)$$

$$v(P) = v(Q) \quad (14.14)$$

则 Q 被删除。其中, $t(P)$ 和 $c(P)$ 分别表示路径 P 的时间和成本, $v(P)$ 和 $v(Q)$ 分别表示路径 P 和 Q 访问的最后一个节点。优超准则的具体实现函数如下:

dominate rule

```

1 def dominate(Queue, Paths):
2     QueueCopy = copy.deepcopy(Queue)
3     PathsCopy = copy.deepcopy(Paths)
4
5     # dominate Queue
6     for label in QueueCopy:
7         for another_label in Queue:
8             if(label.path[-1] == another_label.path[-1] and label.time <
another_label.time and label.dis < another_label.dis):
9                 Queue.remove(another_label)
10                print('dominated path (Q) : ', another_label.path)
11
12     # dominate Paths
13     for key_1 in PathsCopy.keys():
14         for key_2 in PathsCopy.keys():
15             if(PathsCopy[key_1].path[-1] == PathsCopy[key_2].path[-1]
and PathsCopy[key_1].time < PathsCopy[key_2].time
and PathsCopy[key_1].dis < PathsCopy[key_2].dis
and (key_2 in Paths.keys())):
16
17
18

```

```

    Paths.pop(key_2)
    print('dominated path (P) : ', PathsCopy[key_1].path)
    return Queue, Paths

```

5. 算例测试

取算例 C101 中的前 10 个点作为测试数据对标签算法进行测试。

Test Labelling Algorithm

```

org = '0'
des = str(data.nodeNum - 1)
Graph, Queue, Paths, PathsCopy, opt_path = labelling_SPPRC(Graph, org, des)

for key in Paths.keys():
    print(Paths[key].path)

print('optimal path : ', opt_path['1'].path )
print('optimal path (distance): ', opt_path['1'].dis)
print('optimal path (time): ', opt_path['1'].time)

```

最终结果如下。

Result of Labelling Algorithm

```

optimal path : ['0', '5', '11']
optimal path (distance): 30.2
optimal path (time): 30.2

```

运行时间为 5 秒。

接着测试 600 个点的大算例。用 c600 为例, 结果如下。

Result of 600 customers instance

```

optimal path : ['0', '59', '13', '173', '90', '86', '490', '391', '572',
322, '267', '492', '551', '599']
optimal path (distance): 512.0103739069432
optimal path (time): 512.0103739069432
running time : 20min 12s

```

接着测试 1000 个点的最大算例。用 c1000 为例, 结果如下。

```

Result of 1000 customers instance

optimal path : ['0', '183', '1001', '997', '836', '638', '155', '87', '69',
    '1', '755', '976', '70', '704', '458', '629', '107', '99', '798', '829',
    '289', '51', '515', '649', '591', '1000']

optimal path (distance): 827.3615049912353
optimal path (time): 827.3615049912353

Running Time: 1h 31min 33s

```

14.5 Python 实现标签算法结合列生成求解 VRPTW

在第13章中介绍了用列生成算法求解VRPTW。我们用列生成算法的思想，将VRPTW分解成为主问题和子问题，其中，子问题是一个ESPPRC。在第13章，我们直接调用求解器对ESPPRC进行了求解，但是由于ESPPRC也是NP-hard问题，所以求解效果并不理想。本节我们尝试用前文介绍过的标签算法来加速子问题的求解。当然我们也可以直接将子问题松弛成SPPRC，也就是去掉每条路径中节点至多只能被访问一次的约束。执行松弛操作的目的是为了加速子问题求解，但是同时可能会导致主问题得到的下界非常差。关于这点，我们不做详细讨论。为了进一步加速整个求解过程，我们采用一种常见的启发式算法——最近邻居（Nearest Neighbor）算法来生成RMP的初始可行列。

14.5.1 初始化 RMP

我们用最近邻居算法生成初始可行列，其伪代码如下。其主要思想就是首先创建一条初始路径 $[0, 0]$ ，然后不断地寻找最近的节点，插入当前的路径中，直到违背载重约束为止，就生成了一辆车的路径。重复该过程，直到所有的客户都被访问。

Algorithm 15 最近邻居算法

Input: 给定：图 $G = (V, E)$, vehicle_num, customer_num, 车场点 O , 车容量 Q

Output: R

```

1: (***** 初始化 *****)
2: 客户集合  $C \leftarrow \{1, 2, \dots, \text{customer\_num}\}$  和  $t_c \leftarrow 0$ 
3: 路径集合  $R \leftarrow \emptyset$ 
4: while  $C$  非空 do
5:   单条路径  $P \leftarrow [0, 0]$ 
6:   (***** 插入步骤 *****)
7:   while  $P.\text{load} < Q$  且  $C$  非空 do
8:     找到最佳插入客户  $v$  和对应的插入位置 pos
9:      $v, pos, \text{arrive\_time} \leftarrow \text{Insertion Position}(G, C, P, t_c)$ 
10:    if  $P.\text{load} + v.\text{demand} > Q$  且 pos 为空 then

```

```

11:      break
12:    else
13:       $P.\text{insert}(v, pos)$ 
14:       $C.\text{remove}(v)$ 
15:      更新  $t_c \leftarrow \text{arrive\_time} + \text{inserted\_customer}.service\_time$ 
16:      更新车辆载重  $P.\text{load} \leftarrow P.\text{load} + v.\text{demand}$ 
17:    end if
18:  end while
19:   $R \leftarrow R \cup P$ 
20: end while
21: return  $R$ 

```

其中， $\text{Insertion Position}(G, C, P)$ 是为了找到最佳的插入位置和最佳的插入顾客。其输入参数为网络图 G 、未被访问的顾客集合 C 和当前路径 P 。其伪代码如下。

Algorithm 16 插入位置（Insertion Position）选择算法

Input: 图 $G = (V, E)$, 候选顾客集合 C , 路径 P （将要执行插入操作）, 当前时间 t_c

Output: inserted_customer , inserted_position , arrive_time

```

1: (***** 初始化 *****)
2:  $\text{inserted\_position} \leftarrow \text{NULL}$ 
3:  $\text{inserted\_customer} \leftarrow \text{NULL}$ 
4: 最近距离  $d_{\min} \leftarrow \infty$ 
5: if  $\text{len}(P) \leqslant 2$  then
6:    $\text{inserted\_customer} \leftarrow$  距离车场最远的客户点（需要满足时间窗约束）
7:    $\text{inserted\_position} \leftarrow 1$ 
8:    $\text{arrive\_time} \leftarrow \max\{t_c + t_0.\text{inserted\_customer}, a_{\text{inserted\_customer}}\}$ 
9:   return  $\text{inserted\_customer}$ ,  $\text{inserted\_position}$ ,  $\text{arrive\_time}$ 
10: end if
11: for 顾客  $p \in C$  do
12:    $v \leftarrow$  路径  $P$  的最后一个客户点
13:   if  $d_{v,p} < d_{\min}$  并且满足时间窗约束 then
14:      $d_{\min} \leftarrow d_{v,p}$ 
15:      $\text{inserted\_customer} \leftarrow p$ 
16:      $\text{inserted\_position} \leftarrow \text{len}(P) - 1$ 
17:      $\text{arrive\_time} \leftarrow \max\{t_c + t_0.\text{inserted\_customer}, a_{\text{inserted\_customer}}\}$ 
18:   end if
19: end for
20: return  $\text{inserted\_customer}$ ,  $\text{inserted\_position}$ ,  $\text{arrive\_time}$ 

```

利用最近邻居算法可以非常快地找到一个较好的可行解，作为初始列加入 RMP 中。

14.5.2 标签算法求解子问题

VRPTW 的子问题是一个 ESPPRC，我们利用前文介绍的标签算法求解该问题。在列生成算法的每一步迭代中，我们首先通过求解 RLMP，得到 RLMP 所有约束的对偶变量 λ_i ，然后利用 λ_i 更新子问题（ESPPRC）的目标函数，即

$$\min \sum_{i \in N} \sum_{j \in N} (c_{ij} - \lambda_i) x_{ij} \quad (14.15)$$

求解子问题得到的解，就构成新列，加入 RLMP 中，重复此过程，直到没有 Reduced Cost 为负的列产生为止。

初始列的生成也可以采用其他启发式算法，例如 Solomon 提出的 I1 和 I2 算法（Solomon and Marius, 1987），Saving Heuristic（节约算法）等（Clarke and Wright, 1964）。

第15章 分支定价算法

之前我们介绍了列生成算法，这是一种强大的求解大规模整数规划的算法。但是，我们也指出了列生成算法的缺陷。就是在生成新列的过程中，暂时将 RMP 松弛成 LP（即 RLMP），但是在最终形式的 RMP 中，我们将所有变量均设置成 0-1 变量，然后进行求解。我们提到，求解整数规划版本的 RMP 得到的整数解，是原问题最优解的一个上界，但并不一定是原问题的最优解。因此仅仅用列生成算法，并不能保证得到原问题的全局最优解。不过在下面这种情况下，是可以确定列生成算法的解同时也是全局最优解的，那就是当我们最终形式 RMP 的整数约束去掉，将其松弛成 RLMP，如果 RLMP 的最优解同时也是整数解，则此时列生成算法得到的解就是全局最优解。

遗憾的是，最终形式的 RMP 的线性松弛 RLMP 不总是存在整数最优解。那么，我们不禁要问，如何能够保证总是得到全局最优解呢？一个比较好的解决方法就是将列生成算法与分支定界算法嵌套在一起使用。当最终 RLMP 的解是小数时，我们对取值为小数的变量进行分支，然后在 BB tree 的每个叶子节点处，在执行了分支操作的基础上，继续执行列生成算法，得到新的最终 RMP，紧接着，再次求解这个新的 RMP 对应的 RLMP。我们不断地分支、更新上界和下界，直到算法结束，最终得到的解一定是原问题的最优解。上面的思想，正是著名的分支定价算法（Branch and Price Algorithm）。本章就来详细介绍该算法。

15.1 分支定价算法基本原理概述

分支定价算法就是将列生成算法和分支定界算法嵌套在一起使用，共同配合，从而得到原问题的最优解的一种强大的算法。因此，分支定价算法可以简单理解为列生成算法 + 分支定界算法。

接下来我们简要介绍分支定价算法的原理，然后以 VRPTW 为例，详细介绍分支定价算法的完整流程。

我们知道，列生成算法的目的是找出那些具有负检验数的列（这些列相当于线性规划的单纯形法迭代过程中可以进基的基变量），并动态地将这些列添加到 RMP 中。可惜的是，该过程并不能保证没有任何漏列的情况，即，完备主问题的最优基中基变量对应的列有一部分没有被加进 RMP。这也是为什么最基本的列生成算法很多时候不能保证最优化，只能得到一个比较接近最优解的解的原因。在列生成算法一章中，我们也给了两种更为理论的解释。

第1种解释。假定所有列的集合为 Ω , RMP中的列的集合为 Ω' ,且主问题(MP)的解必须是整数,则 $MP(\Omega)$ 与 $RMP(\Omega')$ 的最优解相同的条件是: $MP(\Omega)$ 的最优解被包含在 $RMP(\Omega')$ 的所有整数可行解构成的凸包 $Conv(\Omega')$ 中。但是如果上述条件不成立,即 Ω' 对应的列构成的模型的整数可行解构成的凸包没有囊括 $MP(\Omega)$ 的最优解,则 $RMP(\Omega')$ 的解就不是全局最优解。其直接原因就是有一些最优解中被选中的列没有被加进RMP。

第2种解释。全局最优解 z^* 的下界为 $LB = z_{RLMP} + \sum_i \bar{c}_i^*$,其中 \bar{c}_i^* 为第*i*个子问题的目标函数值;上界为整数规划RMP的最优解,即 $UB = z_{RMP}$ 。如果列生成算法结束后,对应的最终的RLMP的最优解不是整数解,则有 $z_{RLMP} < z_{RMP}$,而由于列生成算法已经结束,因此 $\bar{c}_i^* \approx 0$,所以 $LB < UB$,这说明当前目标值 z_{RMP} 不一定是全局最优值。

根据第1种解释,要找到最优解的关键就在于,如何找到那些在列生成过程中的漏网之鱼(即被遗漏的最优基中的基变量),并加进RMP中。我们的做法仍旧是用列生成算法去找,但是基于当前的最终RMP,是无法再找到新的具有负检验数的列的。不过,我们可以将目前的最终RMP中的“捣蛋鬼”识别出来,将其剔除,形成新的最终RMP,然后接着用列生成算法,基于新的最终RMP,再生成若干新列。这些“捣蛋鬼”指的就是当前RMP的线性松弛的最优解中取值为小数的列。这里有一点非常关键,就是这些取值为小数的列就一定不是完备主问题最优基中的基变量吗?答案是不一定。也就是取值为小数的列也有可能会出现在全局最优整数解中,只是因为我们没有完整地将最优基中的基变量生成出来,导致这部分最优基变量与非最优基变量组成的RMP的整数可行解的凸包没有包含 $MP(\Omega)$ 的最优解,所以这部分最优基变量取值为小数。

上述描述中,我们提到,我们可以将“捣蛋鬼”从当前RMP中剔除掉,然后继续用列生成算法生成若干新列。那么我们又要问,如此操作之后,新生成的最终RMP一定能保证得到最优整数解吗?答案依然是不能。原因也在上文中提到了,就是当前RMP中取值为小数的列,同样有可能是 $MP(\Omega)$ 的最优基中的基变量,我们将其删去并且禁止之后的列生成过程中不能再生成该列,一旦禁止的这一列正好是 $MP(\Omega)$ 最优基中的基变量,那么最终得到的RMP的整数可行解的凸包就不能包含 $MP(\Omega)$ 的最优整数解,所以,按照这种做法最终不会得到最优解。因此,为了保证最优性,我们还需要进行另外一种互补情况的讨论,即,在另一种情况下,我们规定取值为小数且被上一种情况禁止的列一定包含在该种情况下的最终RMP中。以上两种情况就包含了所有的可能,也就保证了没有排除任何整数可行解。当然,在上述两种情况的每种情况下,还有可能再次产生小数解,此时我们可以用相同的方法再执行一次。我们循环上述过程,直到遍历完成所有的可能,就可以得到整数最优解。上述过程实际上就是分支定界算法的思想,每次产生小数解之后,我们分两种情况讨论,就是执行分支操作。在每个节点处,我们首先基于父节点的RMP做分支操作,然后接着执行列生成算法的操作,生成一些额外的列。经过不断地更新上界、下界和分支,我们最终可以得到原问题的最优解。这就是分支定价算法的原理。

这里需要特别说明的是,上面的描述中提到的对列进行分支的方法,一般会导致BB

tree极不平衡,降低求解效率,因此实际科研中,我们一般采取更巧妙的办法进行分支。

为了方便介绍,我们还是以VRPTW为例来详细介绍分支定价算法。注意,本章模型忽略了车辆数限制,在实际问题中,读者可将其加入到模型中。

15.2 分支定价算法求解VRPTW

15.2.1 VRPTW的通用列生成建模方法

本节中,我们采取基于路径的建模方式对VRPTW进行建模。模型中的每一列都对应一条可行路径。我们用 $C = \{1, 2, \dots, n\}$ 表示所有顾客的集合。假设所有的车都是同质的,即容量等参数均相同。用 R 表示所有的可行路径的集合,也就是说 R 中的所有路径都是满足时间窗约束和车辆的载重约束的。这里需要特别说明, R 中的元素可能非常多。如果每个节点的时间窗都非常大,那么 R 中的元素个数会随着顾客数量的增大呈现阶乘级别的增大。对于可行路径集合中的任意一条可行路径 $r \in R$,我们引入下面的0-1变量:

$$y_r = \begin{cases} 1, & \text{如果路径 } r \text{ 在最优解中被选中} \\ 0, & \text{其他} \end{cases} \quad (15.1)$$

其中, $r = (0, i_1, i_2, \dots, 0)$ 是一条完整的从depot(点0)出发,最终回到depot的闭合路径。并且,路径 $r \in R$ 的成本(长度)为 c_r 。那么VRPTW的基于路径的模型则可以写成下面的集分割问题(Set Partitioning Problem):

$$\min \sum_{r \in R} c_r y_r \quad (15.2)$$

$$\text{s.t. } \sum_{r \in R} \theta_{ir} y_r = 1, \quad \forall i \in C \quad (15.3)$$

$$y_r \in \{0, 1\}, \quad \forall r \in R \quad (15.4)$$

其中, θ_{ir} 是一个参数,该参数可以根据每一条路径 $r \in R$ 的具体信息获得。即,如果顾客*i*在路径 r 中被访问到了,那么 $\theta_{ir} = 1$,否则 $\theta_{ir} = 0$ 。

上述建模方法非常简洁,但是当所有可行路径集合 R 中的元素数量非常庞大的时候,我们很难直接穷举出完整的 R 。一个可行的做法是,先列出一些初始的可行路径,然后再使用列生成算法的思想,用循环的方式,动态地寻找新的可行路径,并将其添加到RMP和 R 中。

根据上述思路,可以将VRPTW分解成为一个主问题和一个子问题(定价问题)。主问题的模型是基于当前已经列出的可行路径集合 R' 构造出来的,且 $R' \subset R$ 。我们在之前的章节提到过,由于该主问题仅包含了一部分可行路径,因此又称作限制性主问题(Restricted Master Problem, RMP)。RMP的模型可以写成

$$\min \sum_{r \in R'} c_r y_r \quad (15.5)$$

$$\text{s.t. } \sum_{r \in R'} \theta_{ir} y_r = 1, \quad \forall i \in C \quad (15.6)$$

$$y_r \in \{0, 1\}, \quad \forall r \in R' \quad (15.7)$$

为了给子问题提供对偶变量的取值信息，需要将 RMP 松弛成线性规划，也就是将约束 (15.7) 松弛成下面的形式：

$$0 \leq y_r \leq 1, \quad \forall r \in R' \quad (15.8)$$

事实上，该约束还可以进一步松弛为

$$y_r \geq 0, \quad \forall r \in R' \quad (15.9)$$

从 (15.8) 到 (15.9) 的松弛是等价的转化，具体解释见文献 (Feillet, 2010)。为方便读者理解，本文中依然采用 (15.8) 的形式。根据上文的介绍，RMP 的线性松弛可以写成

$$\min \sum_{r \in R'} c_r y_r \quad (15.10)$$

$$\text{s.t. } \sum_{r \in R'} \theta_{ir} y_r = 1, \quad \forall i \in C \quad (15.11)$$

$$0 \leq y_r \leq 1, \quad \forall r \in R' \quad (15.12)$$

求解 (15.10)，我们可以得到约束 (15.11) 对应的 $|C|$ 个对偶变量 $\pi_i, \forall i \in C$ 。

VRPTW 的子问题就是寻找一条检验数为最小且为负的列，即子问题的目标函数为

$$\min_{r \in R} c_r - \sum_{i \in C} \pi_i \theta_{ir} \quad (15.13)$$

注意，这里是在所有可行路径 R 中去寻找检验数最小的路径。

所有可行路径的集合 R 其实是由一系列约束的可行域描述的，更具体地说，就是由子问题的模型描述的。该子问题就是一个带资源约束的基本最短路问题 (Elementary Shortest Path Problem with Resource Constraints, ESPPRC)。根据之前列生成算法章节中的描述，可以将上面的子问题 (ESPPRC) 的模型写成

$$\min \sum_{i \in N} \sum_{j \in N} (c_{ij} - \pi_i) x_{ij} \quad (15.14)$$

$$\text{s.t. } \sum_{i \in C} d_i \sum_{j \in N} x_{ij} \leq q \quad (15.15)$$

$$\sum_{j \in N} x_{0j} = 1 \quad (15.16)$$

$$\sum_{i \in N} x_{ih} - \sum_{j \in N} x_{hj} = 0, \quad \forall h \in C \quad (15.17)$$

$$\sum_{i \in N} x_{i,n+1} = 1 \quad (15.18)$$

$$s_i + t_{ij} - M(1 - x_{ij}) \leq s_j, \quad \forall i, j \in N, i \neq j \quad (15.19)$$

$$a_i \leq s_i \leq b_i, \quad \forall i \in N \quad (15.20)$$

$$x_{ij} \in \{0, 1\}, \quad \forall i, j \in N, i \neq j \quad (15.21)$$

15.2.2 分支定价算法完整流程及伪代码

前面章节简单回顾了 VRPTW 的列生成细节。下面详细介绍分支定价算法的具体流程和相应的伪代码。

显然，当 RLMP 的解中所有 x_r 取值均为整数时，该解才是原问题的可行解。根据前文中的描述，我们用分支定价算法来求解该问题。分支定价算法的主要思路如下：

- (1) 首先为 RMP 生成一些初始的列，并构建初始 RMP(15.5)；
- (2) 求解 RMP 的松弛问题 (15.10)，得到对偶变量；
- (3) 根据对偶变量更新子问题目标函数，求解子问题，得到一些检验数为负的路径，将其加入 RMP 中；
- (4) 循环步骤 (2) 和 (3)，直到没有新的检验数为负的路径生成为止；
- (5) 求解最终的 RMP 的线性松弛 (15.10)，如果得到了整数解，则算法结束，返回最优解；如果解不是整数解，则进行分支，创建子节点以及对应的子节点的模型，更新上界和下界，并在每个节点循环步骤 (2) 和 (3)。直到 BB tree 中的叶子节点集合为空，或者上界和下界相等，算法终止。

注意：在创建分支子节点时，首先需要执行分支约束对应的操作（比如说将分支的列从 RMP 中删除），并在该节点的子问题中加入相应的分支约束。然后，可以保留该子节点的 RMP 中剩余的列，在此基础上，继续执行列生成算法，生成新的列，并加入该子节点的 RMP 中。

图 15.1 是分支定价算法的程序框图。

注意：上面框图中，在分支中做的事情比较多。主要分为下面的几部分。

- (1) 根据分支策略，创建子节点。
- (2) 在每个子节点中，添加对应的分支约束。以分左支为例（分支变量不能再出现）。我们首先需要将分支变量涉及的列从 RMP 中删除，更新 RMP，并将更新后的 RMP 作为该节点的 RMP（有时候，由于问题本身的特性，还需要在 RMP 中添加其他相关的约束）。然后，还需要在子问题中也添加由于分支产生的相关约束，并将更新后的子问题作为该节点的子问题。
- (3) 执行完上述操作后，还需要将加入了分支约束且删除了分支列的 RMP 和加入了分支约束的 Subproblem 的两个新的节点，加入 BB tree 的队列中。

由于假设所有车辆都是同质的，因此所有车辆的子问题是相同的，所以可以认为只有一个子问题。下面给出分支定价算法的伪代码。

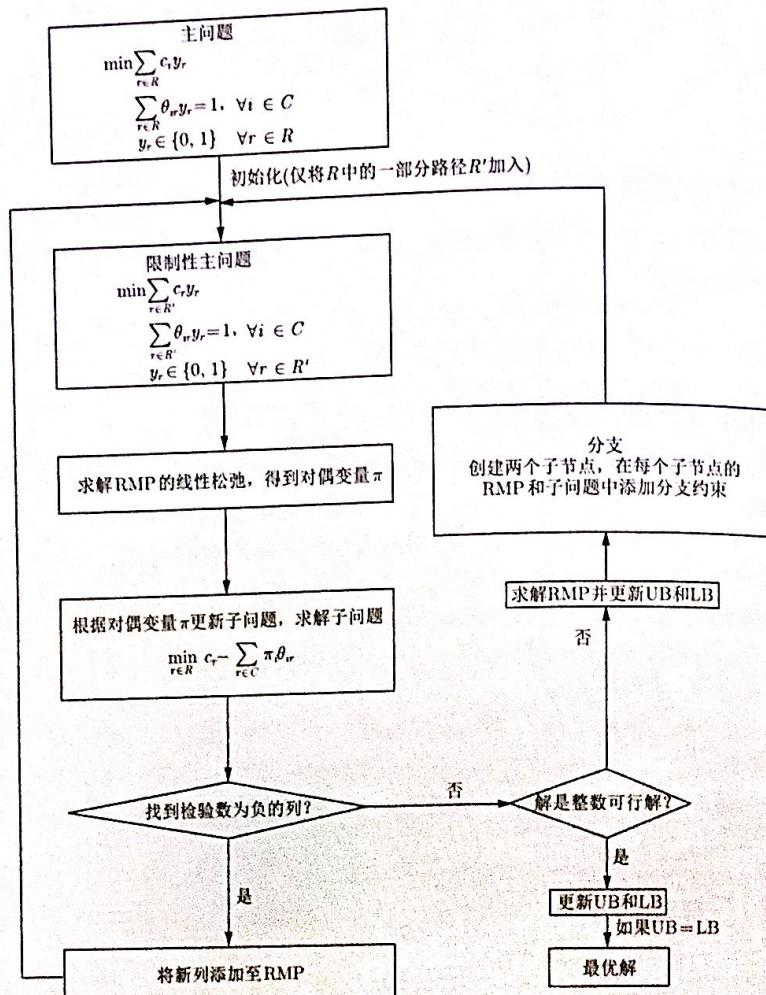


图 15.1 分支定价算法的程序框图

下面，我们给出分支定价算法的伪代码。

Algorithm 17 分支定价算法

- 1: 初始化: 生成初始列 A (例如用启发式), 构建 RMP_0
- 2: 设置 $\epsilon \leftarrow$ 一个很小的负容差 (如, -0.001)
- 3: 设置 BB tree 的节点集合 $Q \leftarrow \emptyset$
- 4: $UB \leftarrow \infty$
- 5: $LB \leftarrow -\infty$ (or 0)

```

6: 当前最优解  $incumbent \leftarrow Null$ 
7: 对偶变量  $\pi \leftarrow$  求解  $RMP_0$  的线性松弛, 即,  $RLMP_0$ 
8: 求解 Subproblem(s)  $SP_0$  (已包含对偶变量  $\pi$ )
9: 创建 BB tree 的根节点  $N_0 \leftarrow \{RMP_0, SP_0\}$ 
10: 将根节点加入 BB tree 的节点集合中  $Q \leftarrow Q \cup \{N_0\}$ 
11: while  $Q$  非空或  $UB-LB > |\epsilon|$  do
12:    $P \leftarrow Q.pop()$  /* 深度优先 */
13:    $RMP \leftarrow P.RMP$ 
14:    $SP \leftarrow P.SP$ 
15:    $\sigma \leftarrow$  求解  $SP$  并得到目标值
16:   while  $\sigma < \epsilon$  do /* 列生成算法开始 */
17:     得到新列 (可多列) 并加入  $RMP$  中
18:     对偶变量  $\pi \leftarrow$  求解  $RMP$ 
19:     用对偶变量  $\pi$  更新  $SP$ 
20:      $\sigma \leftarrow$  求解  $SP(\pi)$  并得到目标值
21:   end while /* 列生成算法结束 */
22:    $y^* \leftarrow$  求解最终  $RMP$  的线性松弛 ( $RLMP$ ) 并得到解
23:   if  $y^*$  是整数可行解 then
24:     if  $UB > y^*$  对应的目标函数 then
25:        $UB \leftarrow y^*$  对应的目标函数
26:        $incumbent \leftarrow y^*$ 
27:     end if
28:     点  $P$  被剪枝 (根据最优性剪枝)
29:   else if  $y^*$  为空 then
30:     点  $P$  被剪枝 (根据非可行性剪枝)
31:   else if  $y^*$  是小数解 then
32:     if  $UB < y^*$  对应的目标函数 then
33:       点  $P$  被剪枝 (根据界限剪枝)
34:     else
35:       ( **** Branch Scheme **** )
36:       根据分支策略创建子节点, 其集合为  $N$ 
37:       for 每个子节点  $i \in N$  do
38:         更新  $RMP_i$  (加入分支约束)
39:         更新  $SP_i$  (加入分支约束导致的相关约束)
40:          $N_i \leftarrow \{RMP_i, SP_i\}$ 
41:       end for
  
```

```

42:   end if
43:    $Q \leftarrow Q \cup N$ 
44:    $LB_{temp} \leftarrow$  所有叶子节点的最小局部下界  $\min_{i \in Q} \left\{ \left( z_{RLMP}^* + \sum_{k=1}^K \tilde{\sigma}_k^* \right)_i \right\}$ 
45:   if  $LB_{temp} > LB$  then
46:     更新  $LB \leftarrow LB_{temp}$ 
47:   end if
48: end if
49: end while
50: return incumbent

```

上述伪代码中，`incumbent`就是当前最优解。我们通过优先队列来存储 BB tree 的节点，并且每次`pop()`都弹出优先队列首元素，并将其从队列中删除。

15.2.3 分支策略

前面的章节中我们详细描述了分支定价算法的具体流程。但是在分支策略的部分并没有进行详细的阐述。本节具体介绍几种典型的分支策略。分支定价算法中的分支操作和一般的分支定界算法中的分支策略还是有一些不同的。在分支定价算法中，主问题的变量和子问题是具有一定关联的。我们求解 RLMP，得到了小数解，如果针对 RLMP 中的变量进行分支，要想在子节点中被禁止的列不再出现，那就需要在子问题中也加入相应的分支约束。因此，分支定价算法中的分支，是牵一发而动全身的。

在 VRPTW 中 BB tree 的一个节点处，用列生成算法生成了所有检验数为负的列，然后求解最终的 RMP 的线性松弛，得到了最优解 y^* 。如果 y^* 是整数，则不用分支；如果 y^* 是小数，我们就需要针对 y^* 进行分支。我们令

$$r_0 = \{i_0 = \text{depot} \rightarrow i_1 \rightarrow i_2 \rightarrow \dots \rightarrow i_{p+1} = \text{depot}\}$$

是当前 RMP 的所有决策变量取值中分数部分最不可行的决策变量对应的路径（也可以是取值最接近 0.5 的路径）。假定 y_{r_0} 的取值为 \bar{y}_{r_0} ，我们就取变量 y_{r_0} 为分支变量。下面我们将来介绍 2 种分支策略。

1. 基于路径的分支

前文中已经介绍了一种很直观的分支方法，即在左分支中禁止路径 r_0 被生成，右分支中一定要生成 r_0 。对于左分支，只需要加入一个约束即可，即

$$\sum_{e \in r_0} x_e \leq |S_{r_0}| - 1$$

其中， $|S_{r_0}|$ 为路径 r_0 中包含的顾客点的个数，并且要将 r_0 对应的列以及 y_{r_0} 从左分支的 RMP 中删去。

对于右分支，为了保证 r_0 一定被生成，我们可以将 r_0 保留在右分支的 RMP 中，并且在右分支的子问题中，删去 S_{r_0} 中的所有顾客点。

在前文提到，这种分支方法非常不好，会导致左右子树非常不平衡。这点不难理解，我们在左分支中仅加入了一条约束，但是在右分支中却删去了若干顾客点，这导致左分支的模型仅仅发生了微小的变化，而右分支的问题规模明显减小，求解难度明显减小，从而导致左右子树极不平衡。正是由于这个原因，基于路径的分支一般不被采用。

2. 基于弧段的分支

基于弧段的分支是实际中常用的一种分支策略，该分支策略下，左右子树相对较为平衡。接下来我们来详细地介绍该分支策略 (Desaulniers et al., 2006)。

由于 $0 < \bar{y}_{r_0} < 1$ ，且 RMP 的约束右端项全部为 1，因此在 RMP 的其他所有列中，一定至少存在一列，该列对应的路径中至少包含了路径 r_0 中一个客户点。即，RMP 中一定有至少一条其他路径至少包含了客户集合 $S(r_0) = \{i_1, i_2, \dots, i_p\}$ 中的一个客户点。

接下来以一个简单的例子来介绍分支定价算法求解 VRPTW 时，是如何分支的。假设我们从 RMP 中抽出 4 列，且这 4 列对应的路径为

$$\begin{aligned} route_1 &= \{0 - 3 - 2 - 5 - 6 - 0\} \\ route_2 &= \{0 - 1 - 2 - 4 - 6 - 0\} \\ route_3 &= \{0 - 1 - 3 - 5 - 4 - 0\} \\ route_4 &= \{0 - 1 - 6 - 5 - 3 - 0\} \end{aligned}$$

它们对应的决策变量 y^* 的取值为

$$(y^*)^T = [0, 0.6, 0.4, 0]$$

依照上面的描述，选择的分支变量即为 $y_2 = 0.6$ 。相应地， y_2 对应的路径为

$$route_2 = \{0 - 1 - 2 - 4 - 6 - 0\}$$

下面写出 RMP 中跟这 4 列相关的部分，即

$$\begin{array}{lllllll} \min & c_1 y_1 & + & c_2 y_2 & + & c_3 y_3 & + & c_4 y_4 & \dots \\ \text{s.t.} & y_2 & + & y_3 & + & y_4 & \dots & = & 1 \\ & y_1 & + & y_2 & + & & & & = 1 \\ & & & & & y_3 & + & y_4 & \dots = 1 \\ & & & & & y_2 & + & y_3 & \dots = 1 \\ & & & & & y_1 & + & y_3 & \dots = 1 \\ & & & & & y_1 & + & y_2 & + & y_4 & \dots = 1 \\ & & & & & 0 \leq & y_1, & y_2, & y_3, & y_4, & \dots \leq 1 \end{array} \quad (15.22)$$

容易观察到, route_2 中访问的所有点的集合为 $\{1, 2, 4, 6\}$, 其中客户点 2 除了在 route_2 中, 在 route_1 中也被访问到了。

由于 $0 < y_2 = 0.6 < 1$, 因此对于除了路径 r_0 以外的路径, 一定存在一条路径 r , 满足 $y_r > 0$, 并且至少与路径 r_0 经过了一个相同的点。因此, 一定存在 $q \in \{1, 2, \dots, p\}$, 满足路径 r 包含客户 i_q 但是不包含弧 (i_q, i_{q+1}) 或者弧 (i_{q-1}, i_q) 。换句话说, 就是路径 r_0 与路径 r 不完全相同。

我们已经选出了分支路径 y_{r_0} , 具体来讲就是 y_2 。接下来穷举 RMP 中所有已经生成的列, 选择第一条满足下面条件的路径 r 作为要生成分支弧段的路径:

- (1) $\bar{y}_r > 0$;
- (2) 与路径 r_0 至少有一个重合的顾客点。

然后选择 (i_q, i_{q+1}) 作为分支弧段, 其中, $q \in \{1, 2, \dots, p\}$ 是满足 $(i_q, i_{q+1}) \notin r$ 的最小的下标 (或者如果有 $(i_q, i_{q+1}) \in r, \forall q \in \{1, 2, \dots, p\}$, 可以选择弧 $(i_0 = \text{depot}, i_1)$)。

我们用上面的例子来解释这个过程。我们选取 $r_0 = \text{route}_2 = \{0 - 1 - 2 - 4 - 6 - 0\}$ 作为分支路径, 另外的 3 条路径中 route_1 也包含 $\{2\}$ 这个客户点。我们发现 route_1 和 route_2 有不重合的点, 正好满足上面的所有条件, 因此选择 route_1 作为 r 。

接下来, 子节点将会按照以下规则产生: 在第 1 个分支中, 弧 (i_q, i_{q+1}) 被禁止访问。在第 2 个分支中, 如果在一个解对应的路径中同时访问了点 i_q 和 i_{q+1} , 则这两个点只有在被弧 (i_q, i_{q+1}) 连接的情况下, 这条路径才能被产生, 否则这条路径不能被产生。换句话说, 在第 2 个分支中, 我们将禁止下列弧被最优解选中:

- (1) (i_q, t) , where $t \neq i_{q+1}$;
- (2) (s, i_{q+1}) , where $s \neq i_q$ 。

或者可以理解为, 在该分支的子问题中, 我们将上述弧直接删除。

继续上面的例子:

$$r_0 = \text{route}_2 = \{0 - 1 - 2 - 4 - 6 - 0\}$$

选择第一个与其至少共享一个节点并且至少有一个点不同的路径, 因此选择

$$r = \text{route}_1 = \{0 - 3 - 2 - 5 - 6 - 0\}$$

路径 r 和路径 r_0 共享客户点 2, 路径 r 包含点 2 但是不包含弧段 $(1, 2)$ 和 $(2, 4)$, 我们可以选择弧段 $(1, 2)$ 当作分支弧段 (当然也可以选择 $(2, 4)$)。

在第 1 个分支中, 弧 $(1, 2)$ 是被禁止的。因此我们需要对 RMP 和子问题都做相应的操作。首先在第 1 个分支的 RMP₁ 中, 将列 y_2 及对应的列删去。并且在对应的子问题 Subproblem₁ 中, 将弧 $(1, 2)$ 禁止, 也就是在 ESPPRC 问题中添加分支约束如下:

$$x_{12} = 0$$

这样操作以后, 在第 1 个分支子节点中, 基于 RMP₁ 调用列生成算法, 生成新的列, 就不会再出现 $\text{route}_2 = \{0 - 1 - 2 - 4 - 6 - 0\}$ 这一列了。

在第 2 个分支中, 必须要保证顾客点 1 和顾客点 2 只有被连续访问 (也就是弧 $(1, 2)$ 被选中时) 时, 顾客点 1 和顾客点 2 才能被同时加入子问题的最优解对应的路径当中, 否则顾客点 1 和顾客点 2 就不能被同时访问到。例如, 如果访问顺序是 $1 \rightarrow 5 \rightarrow 2$, 顾客点 1 和顾客点 2 在同一路经中被访问了, 但是顾客点 1 的后序节点不是顾客点 2, 那么这条路径就不能被 Subproblem₂ 生成, 并被添加到 RMP₂ 中。此外, 我们需要将第 2 个分支的 RMP 中用到弧 $(1, 2)$ 的其他列全部删除。由于上述例子中不存在这样的列, 该步操作可略过。继续后面的分支操作, 在 Subproblem₂ 中, 我们必须禁止满足以下两种情况的弧段被选中:

- (1) 进入顾客点 2, 但是不是从顾客点 1 进入的;
- (2) 从顾客点 1 出去, 但是不是从顾客点 1 到顾客点 2 的。

因此, 在这个分支子节点处的 RMP₂ 中, 我们要保留列 $\text{route}_2 = \{0 - 1 - 2 - 4 - 6 - 0\}$, 也就是保留决策变量 y_2 。并且在分支节点对应的子问题 Subproblem₂ 中, 我们需要添加一系列的分支约束, 即

$$x_{i2} = 0, \quad \forall (i, 2) \in E, i \neq 1$$

$$x_{1j} = 0, \quad \forall (1, j) \in E, j \neq 2$$

这样就能保证在第 2 个分支节点中, 只要顾客点 1 或者 2 被添加到了一个新列中, 那么在这个新列中, 顾客点 1 和 2 一定是被连续访问的, 并且是先访问顾客点 1, 再访问顾客点 2。当然, 上述操作也可以通过直接从底层网络中删除相关弧实现。

这个分支策略是非常实用的, 因为该策略很好地利用了主问题和子问题之间的联系。通过上述介绍可以看到, 在分支定价算法中, 分支的步骤是同时涉及主问题和子问题的, 在分支的时候, 需要根据相应的分支规则, 同时对节点处的 RMP 和子问题都做处理。

上面这种基于弧段的分支略微有些烦琐, 接下来我们介绍另一种更易理解的基于弧段的分支策略 (Feillet, 2010), 与前文介绍的分支策略不同, 我们令左分支中的弧 (i_q, i_{q+1}) 被禁止访问, 右分支中的弧 (i_q, i_{q+1}) 被强制访问。为了方便描述, 我们首先引入下面的参数:

(1) a_{ij}^r : 如果路径 r 中包含弧 (i, j) , 则 $a_{ij}^r = 1$, 否则 $a_{ij}^r = 0$;

(2) $f_{ij} = \sum_{r \in R} a_{ij}^r y_r$, 表示弧 (i, j) 在当前节点的 RMP 线性松弛问题的最优解中被选中的次数, 或者弧 (i, j) 上的流量 (有可能是小数), 显然 $0 \leq f_{ij} \leq 1, \forall (i, j) \in E$ 。

选取分支弧段的方法也略有不同, 这里我们选取 f_{ij} 最接近 0.5 的弧段作为分支弧段并令该弧段为 (i_q, i_{q+1}) 。基于此, 令左子节点中 $f_{i_q, i_{q+1}} = 0$, 即在左子节点的 RMP 中加入分支约束

$$\sum_{r \in R'} a_{i_q, i_{q+1}}^r y_r = 0$$

相应地, 令右子节点中 $f_{i_q, i_{q+1}} = 1$, 即在右子节点的 RMP 中加入分支约束

$$\sum_{r \in R'} a_{i_q, i_{q+1}}^r y_r = 1$$

这种分支约束更易于理解和实现，也更简洁，仅在左右分支节点中各添加了一条约束，并且子问题不用做任何改动。相比第一种基于弧段的分支策略而言，第二种分支策略中左右子树更加平衡。因此，在实际中更建议使用第二种基于弧段的分支策略。

实际上，在第二种基于弧段的分支策略下，子问题也可以做相应的改动，以加快求解。即，在左子节点的子问题中，删去弧 (i_q, i_{q+1}) 。在右子节点的子问题中，删去弧 (i_q, t) , where $t \neq i_{q+1}$ 和 (s, i_{q+1}) , where $s \neq i_q$ 。

15.2.4 界限更新

在分支定界的过程中，采用最基本的界限 (Bounds) 更新，具体来讲，在每步的迭代中，令

$$\text{UB} = z_{\text{RMP}}^* \quad (15.23)$$

$$\text{LB} = z_{\text{RLMP}}^* + \sum_{k=1}^K \tilde{\sigma}_k^* \quad (15.24)$$

其中， z_{RMP}^* 为 RMP 的整数可行解对应的目标值， z_{RLMP}^* 为 RMP 的线性松弛问题 RLMP 的最优解对应的目标值， $\tilde{\sigma}_k^*$ 为子问题 k 的最优目标值，即子问题 k 的检验数。

因为篇幅所限，不在此处给出分支定价算法求解 VRPTW 的完整代码。在附配的资源中提供了相关开源代码的链接，请感兴趣的读者自行查阅。