CARP问题求解

赵耀

基本步骤

- 第一步，准备
- 第二步，构造
- 第三步，改进

迪杰斯特拉算法

- 初始时，S只包含源点，即S＝{v}，v的距离为0。U包含除v外的其他顶点，即:U={其余顶点}，若v与U中顶点u有边，则<u,v>正常有权值，若u不是v的出边邻接点，则<u,v>权值为∞。
- 从U中选取一个距离v最小的顶点k，把k，加入S中（该选定的距离就是v到k的最短路径长度）。
- 以k为新考虑的中间点，修改U中各顶点的距离；若从源点v到顶点u的距离（经过顶点k）比原来距离（不经过顶点k）短，则修改顶点u的距离值，修改后的距离值位源点到顶点k的距离加上k到u边上的权。
- 重复步骤b和c直到所有顶点都包含在S中。

任意两点间的距离

- 可对每个点都执行一遍迪杰斯特拉算法，即可得到任意两点间的最短距离
- 也可应用Floyd算法

通用的Path-Scanning的算法

- 将所有的弧都拷贝到未分配的列表中，假设列表名为free
- 重复如下步骤挨个生成路径：
  - 初始化起点为1（depot的指定位置）
  - 重复加入满足容量限制且距离到上一个任务的终点距离最近的任务，如果存在距离相等的任务，应用其他的优选准则（下页介绍5个）挑选相对更好的任务（或随机选择等距离的任务）
  - 没有任务能在满足约束的条件下加入到路径，回到起点

Algorithm 7.2 – Path-Scanning for one priority rule

$k \leftarrow 0$
copy all required arcs in a list $free$
repeat
    $k \leftarrow k+1;\ R_k \leftarrow \emptyset;\ load(k), cost(k) \leftarrow 0;\ i \leftarrow 1$
    repeat
        $\bar{d} \leftarrow \infty$
        for each $u \in free \mid load(k) + q_u \le Q$ **do**
            if $d_{i,beg(u)} < \bar{d}$ **then**
                $\bar{d} \leftarrow d_{i,beg(u)}$
                $\bar{u} \leftarrow u$
            else if $(d_{i,beg(u)} = \bar{d})$ and $better(u, \bar{u}, rule)$
                $\bar{u} \leftarrow u$
            endif
        endfor
        add $\bar{u}$ at the end of route R_k
        remove arc $\bar{u}$ and its opposite $\bar{u}+m$ from $free$
        $load(k) \leftarrow load(k) + q_{\bar{u}}$
        $cost(k) \leftarrow cost(k) + \bar{d} + c_{\bar{u}}$
        $i \leftarrow end(\bar{u})$
    until $(free = \emptyset)$ or $(\bar{d} = \infty)$
    $cost(k) \leftarrow cost(k) + d_{i1}$
until $free = \emptyset$

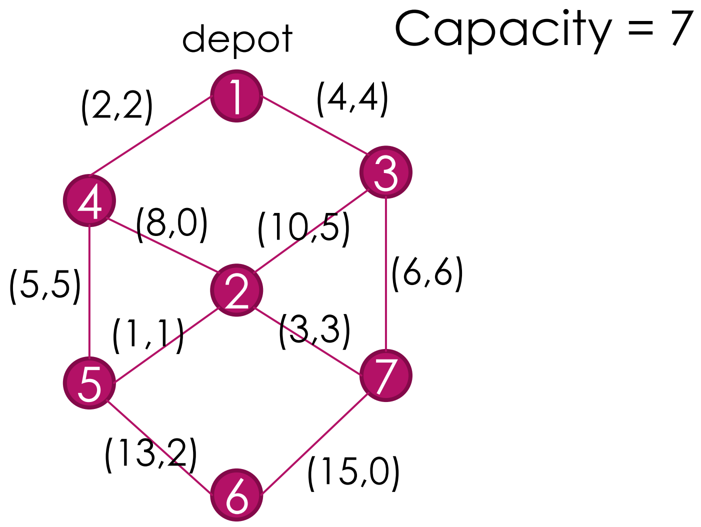
可以应用不同的rule

- 1) maximize the distance from the task to the depot;
- 2) minimize the distance from the task to the depot;
- 3) maximize the term $dem(t)/sc(t)$, where $dem(t)$ and $sc(t)$ are demand and serving cost of task t, respectively;
- 4) minimize the term $dem(t)/sc(t)$;
- 5) use rule 1) if the vehicle is less than half- full, otherwise use rule 2)
- 可以任选一个rule用于进一步挑选候选任务，多个初始解时，可以第1个解应用rule1，第二个解应用rule2等等。

举个例子：

c[][]:

	1	2	3	4	5	6	7
1	0	8	4	2	7	20	10
2	8	0	9	6	1	14	3
3	4	9	0	6	10	21	6
4	2	6	6	0	5	18	9
5	7	1	10	5	0	13	4
6	20	14	21	18	13	0	15
7	10	3	6	9	4	15	0



Path-Scanning(初始化)

free：

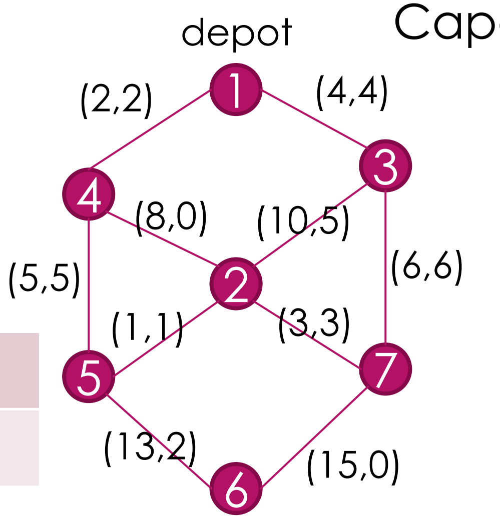
(1,4)	(1,3)	(4,5)	(5,6)	(2,3)	(2,5)	(2,7)	(3,7)
(4,1)	(3,1)	(5,4)	(6,5)	(3,2)	(5,2)	(7,2)	(7,3)

R1： ∅

load(1) = 0
cost(1) = 0
i = 1

Path-Scanning(路径1，迭代1)

(1,4)与(1,3)是到1最近的2个task，假设按照rule5去选任务，
则可以选择任务(1,3)， (1,3)回程距离较远。



free:								
	(1,4)	~~(1,3)~~	(4,5)	(5,6)	(2,3)	(2,5)	(2,7)	(3,7)
	(4,1)	~~(3,1)~~	(5,4)	(6,5)	(3,2)	(5,2)	(7,2)	(7,3)

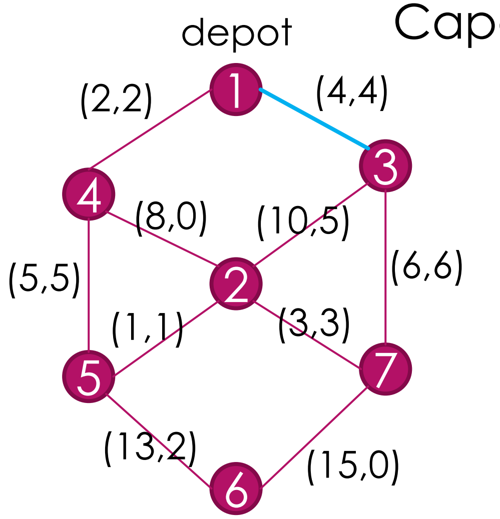
R1 : (1,3)

load(1) = 4
cost(1) = 4
i = 3

Path-Scanning(路径1，迭代2)

剩余容量3，满足容量的task有
(1,4)(4,1)(2,5)(5,2)(2,7)(7,2)(5,6)(6,5)，距离3最近的task是
(1,4)

Capacity = 7



free：

~~(1,4)~~	(4,5)	(5,6)	(2,3)	(2,5)	(2,7)	(3,7)
~~(4,1)~~	(5,4)	(6,5)	(3,2)	(5,2)	(7,2)	(7,3)

R1 ：

(1,3)	(1,4)

load(1) = 6
cost(1) = 4 + c[3][1] + 2 = 10
i = 4

Path-Scanning(路径1，迭代3)

剩余容量1，满足容量的task有 (2,5)(5,2)，距离4最近的task是(5,2)

free：

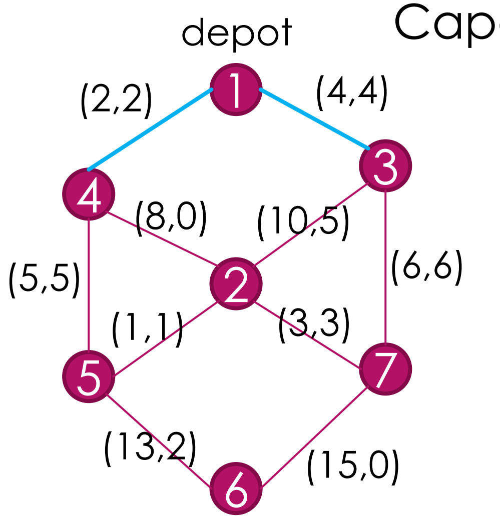
(4,5)	(5,6)	(2,3)	~~(2,5)~~	(2,7)	(3,7)
(5,4)	(6,5)	(3,2)	~~(5,2)~~	(7,2)	(7,3)

R1 ：

(1,3)	(1,4)	(5,2)

load(1) = 7
cost(1) = 10+c[4][5]+1= 16
i = 2

Capacity = 7



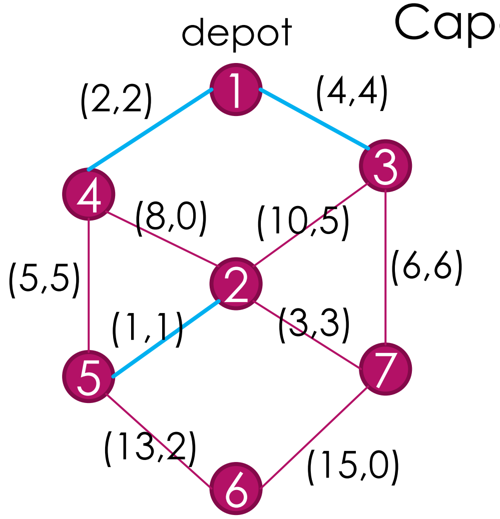
Path-Scanning(路径1,结束)

剩余容量0

free：	(4,5)	(5,6)	(2,3)	(2,7)	(3,7)
	(5,4)	(6,5)	(3,2)	(7,2)	(7,3)
R1 ：	(1,3)	(1,4)	(5,2)		

load(1) = 7
cost(1) = 16 + c[2][1] = 16 + 8 = 24

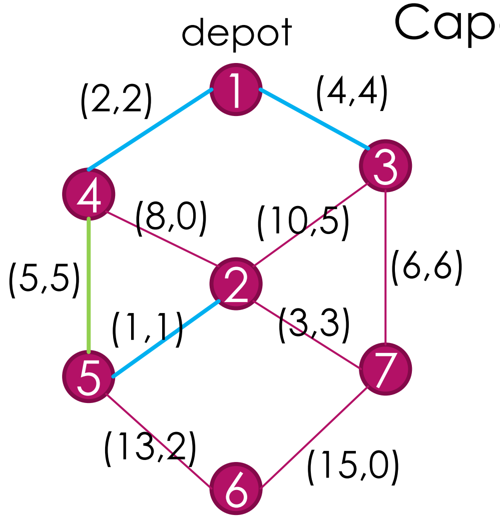
Capacity = 7



Path-Scanning(路径2,迭代1)

(4，5) 到1最近的task，c[1][4] = 2

Capacity = 7



free：

~~(4,5)~~	(5,6)	(2,3)	(2,7)	(3,7)
~~(5,4)~~	(6,5)	(3,2)	(7,2)	(7,3)

R2 ：

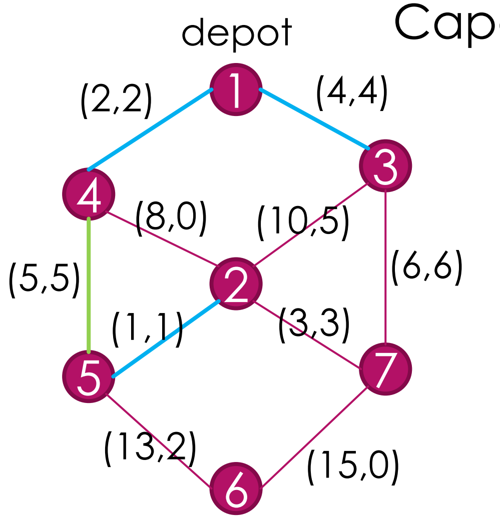
(4,5)

load(2) = 5
cost(2) = c[1][4] + 5 = 7
i = 5

Path-Scanning(路径2,迭代2)

Capacity = 7

剩余容量2，满足容量需求的task为(5,6)(6,5)，(5,6) 到5最近，
c[5][5] = 0



free：

~~(5,6)~~	(2,3)	(2,7)	(3,7)
~~(6,5)~~	(3,2)	(7,2)	(7,3)

R2 ：

(4,5)	(5,6)

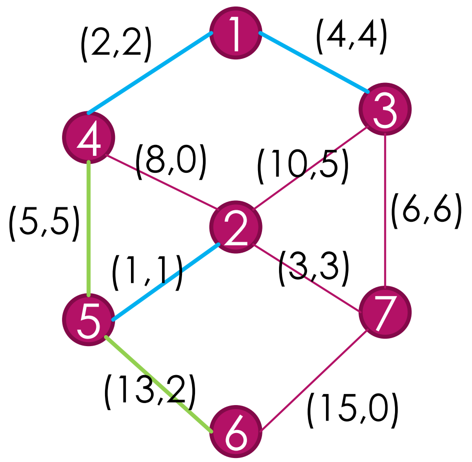
load(2) = 7
cost(2) = 7 + c[5][5] + 13 = 20

Path-Scanning(路径2,结束)

剩余容量0

free：	(2,3)	(2,7)	(3,7)
	(3,2)	(7,2)	(7,3)
R2 ：	(4,5)	(5,6)	

load(2) = 7
cost(2) = 20 + c[6][1] = 40

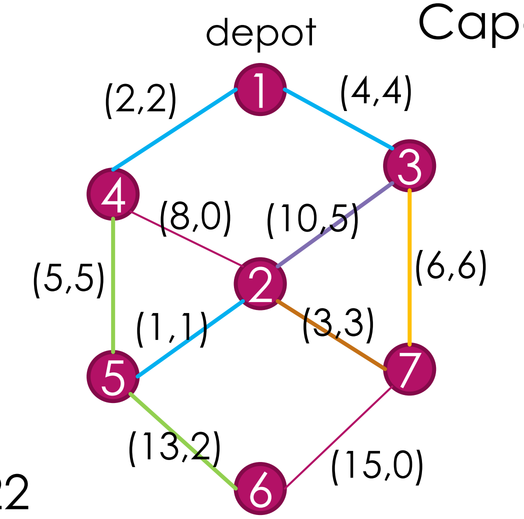


Path-Scanning(算法结束)

free：ø

R1:	(1,3)	(1,4)	(5,2)	load(1) = 7 cost(1) = 24
R2:	(4,5)	(5,6)		load(2) = 7 cost(2) = 40
R3:	(3,2)			load(3) = 5 cost(3) = c[1][3]+10+c[2][1]=22
R4:	(3,7)			load(4) = 6 cost(4) = c[1][3]+6+c[7][1]=20
R5:	(2,7)			load(5) = 3 cost(5) = c[1][2]+3+c[7][1]=21

Capacity = 7



Path-Scanning

- Path-scanning 已经可以保证获得一个可行解（恭喜，完成到这个程度就及格了！）
- 应用贪心算法的基础
- 用于局部搜索算法计算初始种群

其他构造方法

- Augment-Merge
- Ulusoy’s route-first cluster-secound method
- Construct-strike

详见：《Arc Routing》[Ángel Corberán and Gilbert Laporte] P144~P149

常用的算子（Move Operator）

- Flip
- Single insertion
- Double insertion
- Swap
- 2-opt

Flip

(3,1)

R1:	(1,3)	(1,4)	(5,2)
R2:	(4,5)	(5,6)	
R3:	(3,2)		
R4:	(3,7)		
R5:	(2,7)		

load(1) = 7
cost(1) = 24 ->24-c[1][1]-c[3][1]+c[1][3]+c[1][1]=24

load(2) = 7
cost(2) = 40

load(3) = 5
cost(3) = c[1][3]+10+c[2][1]=22

load(4) = 6
cost(4) = c[1][3]+6+c[7][1]=20

load(5) = 3
cost(5) = c[1][2]+3+c[7][1]=21

可以遍历Flip所有的任务，看是否有改进

Single insertion

R1: (1,3) (1,4) (5,2)

R2: (4,5) (5,6)

R3: (3,2)

R4: (3,7)

R5: (2,7)

也可以跨路径

cost(1) = 24 ->24-c[1][1]-c[2][1]-c[3][1]+c[1][1]+c[2][1]+c[3][1]
= 24 (舍弃)

load(1) = 7
cost(1) = 24 ->24-c[1][1]-c[3][1]-c[4][5]+c[1][1]+c[4][1]+c[3][5]
= 24 – 4 – 5 + 2 + 10 =27(舍弃)

load(2) = 7
cost(2) = 40

load(3) = 5
cost(3) = c[1][3]+10+c[2][1]=22

load(4) = 6
cost(4) = c[1][3]+6+c[7][1]=20

load(5) = 3
cost(5) = c[1][2]+3+c[7][1]=21

把每个单个任务尝试插入到本路径的其他任务后面或depot后，或者插入到其他的路径任务后，或depot后

Double insertion

R1: (1,3) (1,4) (5,2)

R2: (4,5) (5,6)

R3: (3,2)

R4: (3,7)

R5: (2,7)

load(2) = 7
cost(2) = 40

load(3) = 5
cost(3) = c[1][3]+10+c[2][1]=22

load(4) = 6
cost(4) = c[1][3]+6+c[7][1]=20

load(5) = 3
cost(5) = c[1][2]+3+c[7][1]=21

跟Single insection类似，把连续2个任务尝试插入到本路径的其他任务后面或depot后，或者插入到其他的路径任务后，或depot后

Swap

R1: (1,3) (1,4) (5,2)
load(1) = 7
cost(1) = 24

R2: $\boxed{(4,5)}$ (5,6)
load(2) = 7
cost(2) = 40

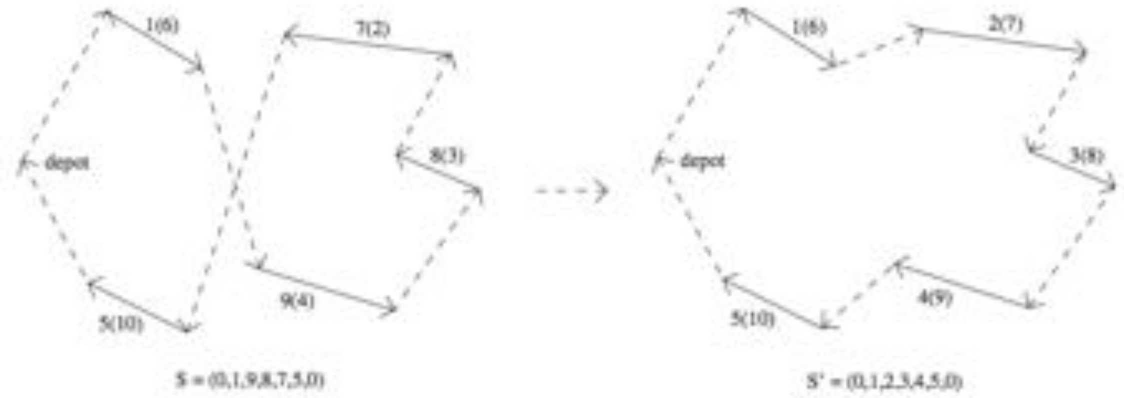
R3: $\boxed{(3,2)}$
load(3) = 5
cost(3) =22

R4: (3,7)
load(4) = 6
cost(4) = 20

R5: (2,7)
load(5) = 3
cost(5) = 21

2-opt

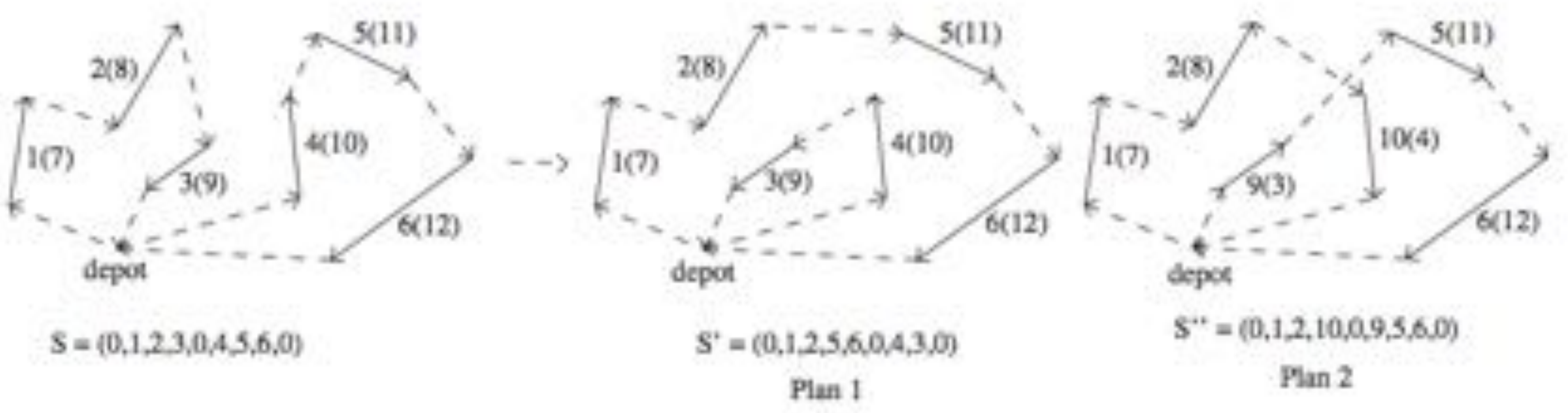
- optimal for single route



4) 2-opt: There are two types of 2-opt move operators, one for a single route and the other for double routes. In the 2-opt move for a single route, a subroute (i.e., a part of the route) is selected and its direction is reversed. When applying the 2-opt move to double routes, each route is first cut into two subroutes, and new solutions are generated by reconnecting the four subroutes. Figs. 3 and 4 illustrate the two 2-opt move operators, respectively. In Fig. 3, given a solution $S = (0, 1, 9, 8, 7, 5, 0)$, the subroute from task 9 to 7 is selected and its direction is reversed. In Fig. 4, given a solution $S = (0, 1, 2, 3, 0, 4, 5, 6, 0)$, the first route is cut between tasks 2 and 3, and the second route is cut between tasks 4 and 5. A new solution can be obtained either by connecting task 2 with task 5, and task 4 with task 3, or by linking task 2 to the inversion of task 4, and task 5 with inversion of task 3. In practice, one may choose the one with the smaller cost. Unlike the previous three operators, the 2-opt operator is only applicable to edge tasks. Although it can be easily modified to cope with arc tasks, such work remains absent in the literature.

2-opt

- optimal for double route



P.23 and P.24 are copy from 《Memetic Algorithm with Extended Neighborhood Search for Capacitated Arc Routing Problems》

- 怎样有效的应用各种算子?
- 小步的算子容易陷入局部最优
- 大步的算子有可能在接近全局最优解时错过全局最优解
- 没有一种算子能够在各种通用的场景都表现得很好
- 通用：先小步找到局部最优解，再大步跳出来，如果跳出成功（什么叫跳出成功：新的解比原有解更优），再小步找到新区域的局部最优解

Thank You!