CARP

YAO ZHAO

Steps

- Preparation
- Construction
- Improvement

Preparation

- In preparation, we need to read in the file and find **the minimum distance** between any two nodes using **Dijkstra** or **Floyd** algorithm.

Dijkstra review

For each unexplored node, explicitly maintain $\pi(v) = \min_{e=(u,v)\,:\,u \in S} d(u) + \ell_e$

- Next node to explore = node with minimum $\pi(v)$.
- When exploring v, for each incident edge e = (v, w), update

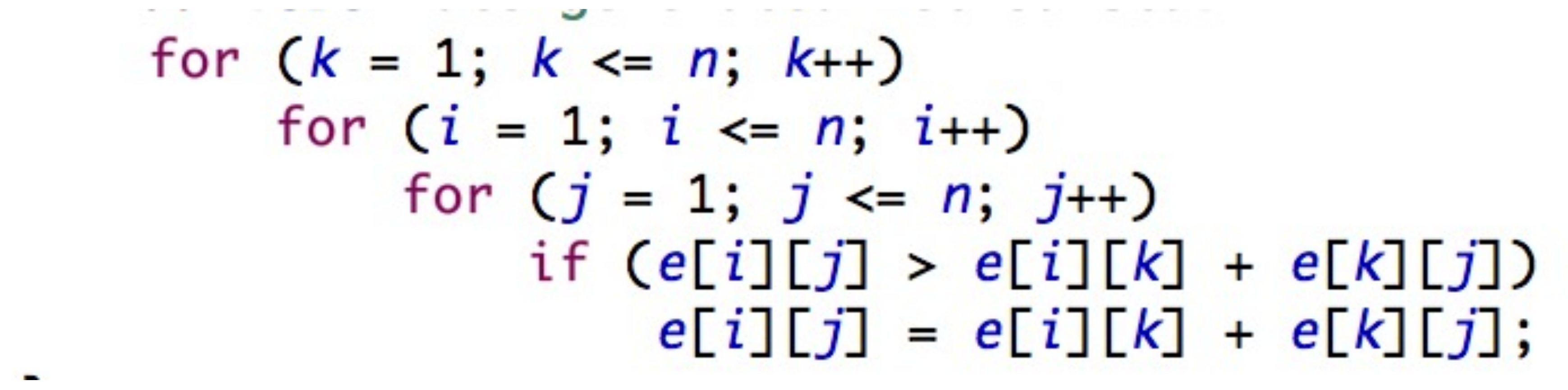
$$\pi(w) = \min \{ \; \pi(w), \; \pi(v) + \ell_e \}.$$

Efficient implementation. Maintain a priority queue of unexplored nodes, prioritized by $\pi(v)$.

Dijkstra review

```
add start node into heap
while (heap is not empty){
    curnode = heap.poll()
    for each child node i of curnode{
        if(node i haven’t been visited){
            the distance of node i = curnode.distance+ weight of edge(curnode, i)
            set node i is visited
            add node i into heap
        }else{
            if(the distance of node i > curnode.distance+ weight of edge(curnode, i){
                update distance of node i
                decrease key of node i in heap
            }//end if
        }//end if
    }//end for
}//end while
```

Floyd Review



```
for (k = 1; k <= n; k++)
    for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
            if (e[i][j] > e[i][k] + e[k][j])
                e[i][j] = e[i][k] + e[k][j];
```

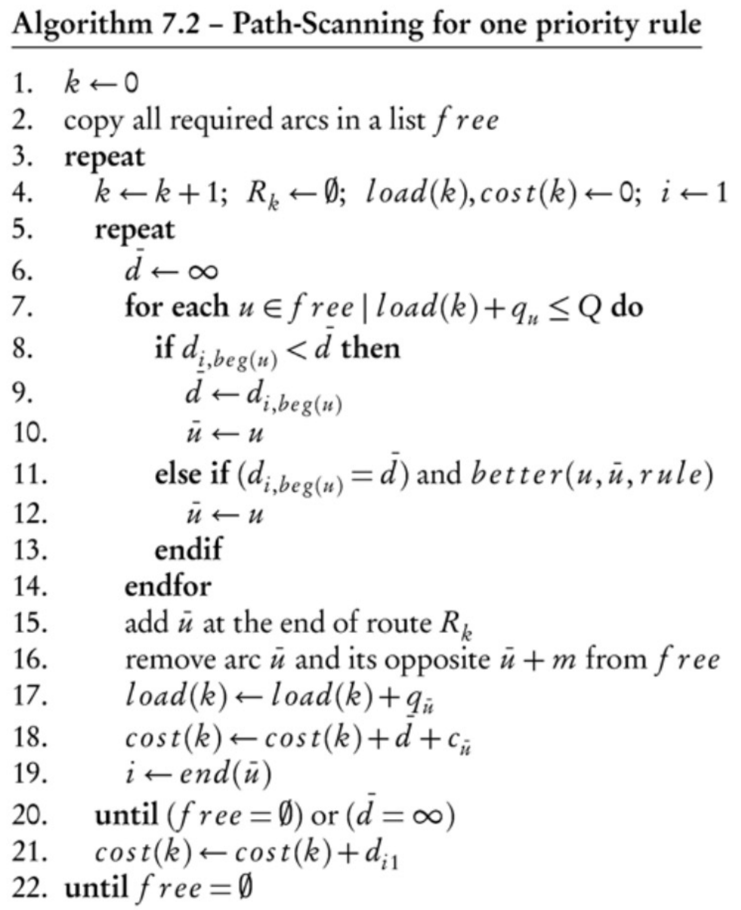
Construction

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

More detail: Arc Routing, [Ángel Corberán and Gilbert Laporte]
P144~P149

Path-Scanning

- Copy all required arcs in a list **free**
- Repeat the following steps to generate paths one by one :
  - Start at the depot
  - Repeat to add the path which is the closest to the end of current path, not yet serviced and compatible with vehicle capacity . If multiple tasks are the closest to the end of current path, **five rules(next page will introduce)** or **Random selection** are used to determine the next task.
  - No task can join the path and go back to the depot



Algorithm 7.2 – Path-Scanning for one priority rule

```
1.  $k \leftarrow 0$
2.  copy all required arcs in a list $free$
3.  repeat
4.     $k \leftarrow k+1;\ R_k \leftarrow \emptyset;\ load(k), cost(k) \leftarrow 0;\ i \leftarrow 1$
5.     repeat
6.        $\bar{d} \leftarrow \infty$
7.        for each $u \in free \mid load(k) + q_u \leq Q$ do
8.           if $d_{i,beg(u)} < \bar{d}$ then
9.              $\bar{d} \leftarrow d_{i,beg(u)}$
10.             $\bar{u} \leftarrow u$
11.          else if $(d_{i,beg(u)} = \bar{d})$ and $better(u, \bar{u}, rule)$
12.             $\bar{u} \leftarrow u$
13.          endif
14.       endfor
15.       add $\bar{u}$ at the end of route $R_k$
16.       remove arc $\bar{u}$ and its opposite $\bar{u}+m$ from $free$
17.       $load(k) \leftarrow load(k) + q_{\bar{u}}$
18.       $cost(k) \leftarrow cost(k) + \bar{d} + c_{\bar{u}}$
19.       $i \leftarrow end(\bar{u})$
20.    until $(free = \emptyset)$ or $(\bar{d} = \infty)$
21.    $cost(k) \leftarrow cost(k) + d_{i1}$
22. until $free = \emptyset$
```

Path-Scanning:Five rules

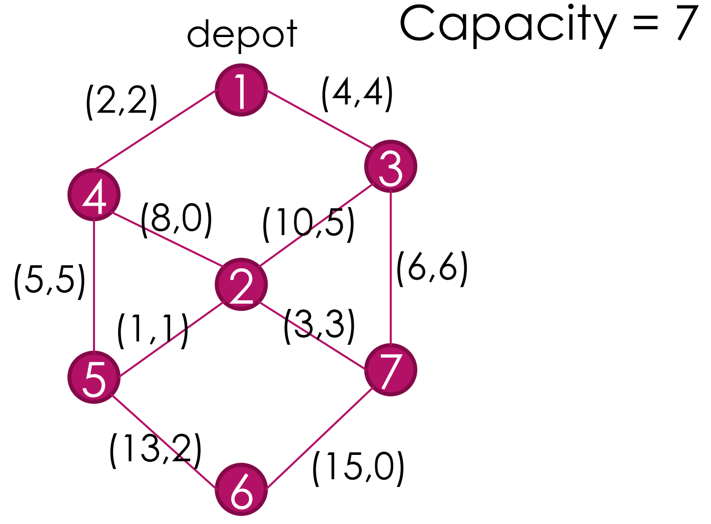
- 1) maximize the distance from the task to the depot;
- 2) minimize the distance from the task to the depot;
- 3) maximize the term $dem(t)/sc(t)$, where $dem(t)$ and $sc(t)$ are demand and serving cost of task t, respectively;
- 4) minimize the term $dem(t)/sc(t)$;
- 5) use rule 1) if the vehicle is less than half- full, otherwise use rule 2)

PS: Any rule can be used to further select candidate tasks, and you need construct multiple initial solutions, the first solution can use rule1, the second solution can use rule2, and so on.

Example: Preparation

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



Read in the file and find the minimum distance between any two points using **Dijkstra** or **Floyd** algorithm.
These distance can be stored in a two-dimensional array.

Example: Path-Scanning(initial)

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)

Route1： ∅

Load of route 1 = 0
Cost of route 1 = 0
The end of current path = 1

Path-Scanning(Route1， Iteration 1)

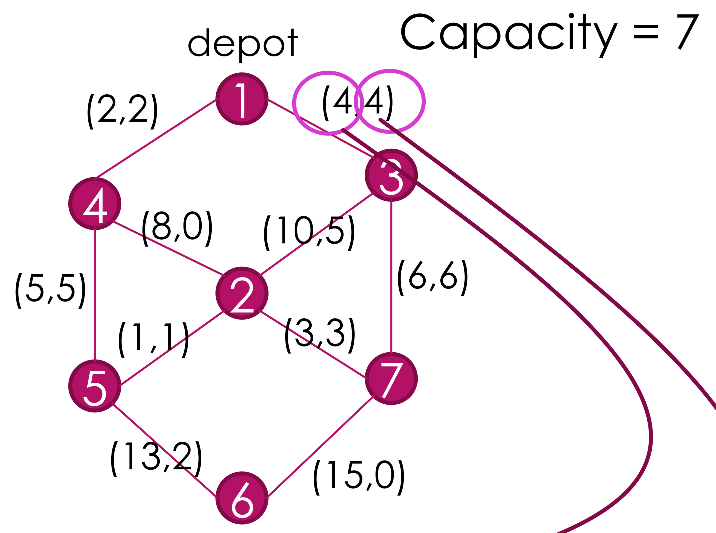
(1,4)and (1,3) are the closest tasks to the end of current path(Node 1)

Using rule5, choose task(1,3) which has maximum distance from the task to the depot

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)

Route1: (1,3)

Cost of route 1 = 4
Load of route 1 = 4
The end of current path = 3



Path-Scanning(Route1， Iteration 2)

residual capacity : 7-4 = 3

Tasks compatible with vehicle residual capacity:
(1,4)(4,1)(2,5)(5,2)(2,7)(7,2)(5,6)(6,5)

(1,4) is the closest tasks to the end of current path(Node 3)

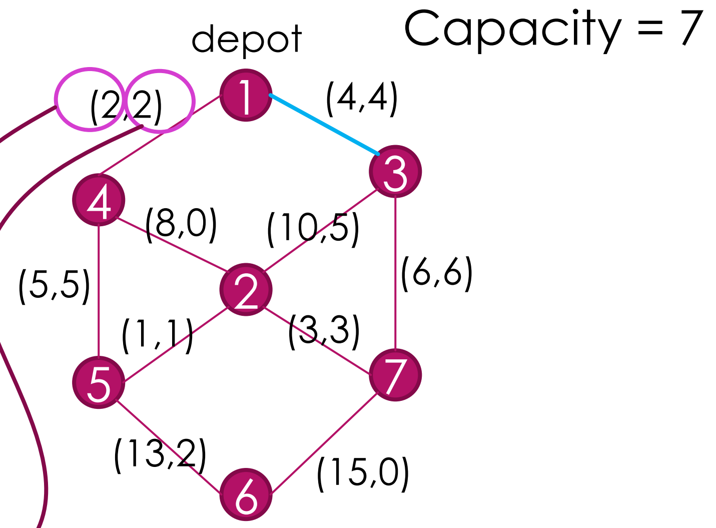
free：

~~(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)

Route1：

(1,3)	(1,4)

Cost of route 1 = 4 + c[3][1] + 2 = 10
Load of route 1 = 4 + 2 = 6
The end of current path = 4



Path-Scanning(Route1， Iteration 3)

residual capacity : 3-2 = 1

Tasks compatible with vehicle residual capacity:
(2,5)(5,2)

(5,2) is the closest tasks to the end of current path(Node 4)

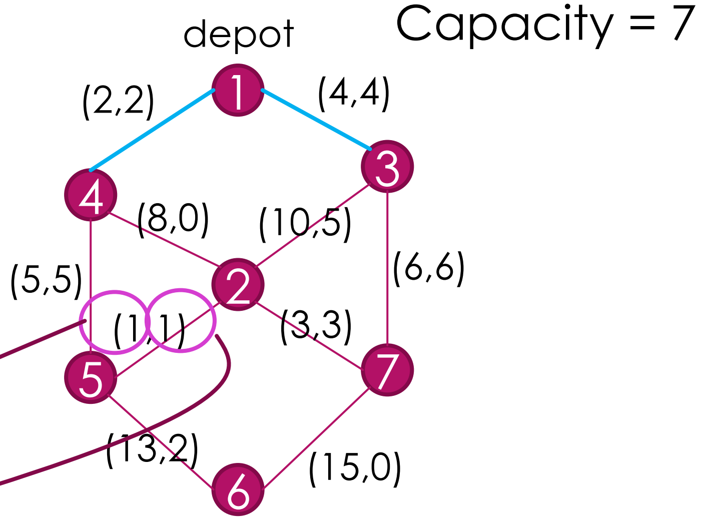
free：

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

Route1：

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

Cost of route 1 = 10 + c[4][5] + 1 = 16
Load of route 1 = 6 + 1 = 7
The end of current path = 2



Path-Scanning(Route1, stop)

residual capacity : 1 - 1 = 0

Stop

free:

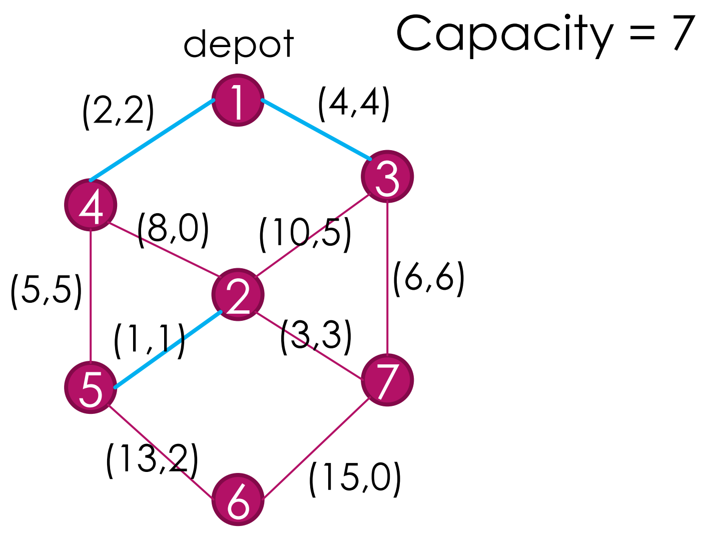
(4,5)	(5,6)	(2,3)	(2,7)	(3,7)
(5,4)	(6,5)	(3,2)	(7,2)	(7,3)

Route1:

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

Cost of route 1 = 16 + c[2][1] = 24
Load of route 1 = 7

When a route is finished, don't remember add the distance from the current end of the route to the depot



Path-Scanning(Route2,　Iteration 1)

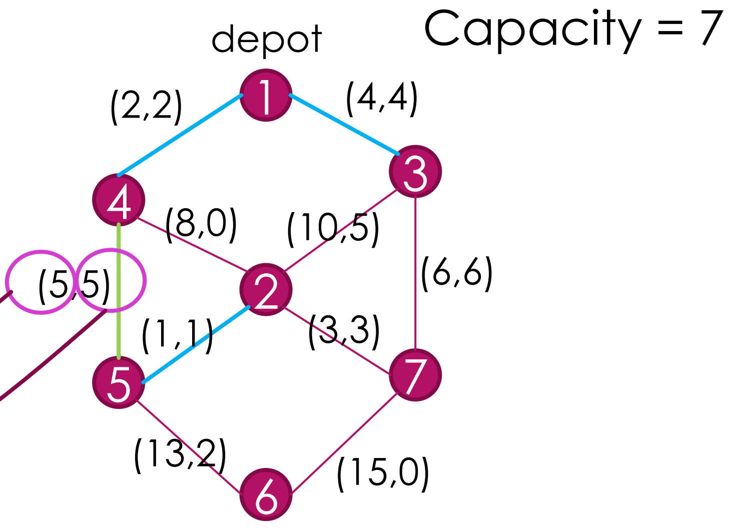
(4,5) is the closest tasks to the end of current path(Node 1)

free:

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

Route2: (4,5)

Cost of route 2 = c[1][4] + 5 = 7
Load of route 2 = 5
The end of current path = 5



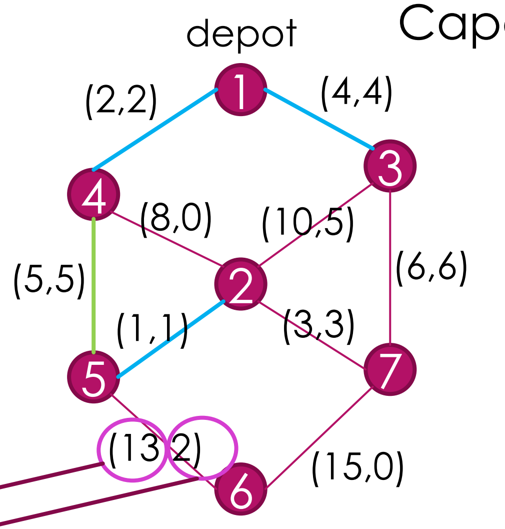
Path-Scanning(Route2, Iteration 2)

residual capacity : 7-5 = 2

Tasks compatible with vehicle residual capacity:
(5,6)(6,5)

(5,6) is the closest tasks to the end of current route(Node 5)

Capacity = 7



free:

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

Route2:

(4,5)	(5,6)

Cost of route 2 = 7 + c[5][5] + 13 = 20
Load of route 2 = 5 + 2 = 7
The end of current path = 6

Path-Scanning(Route2, stop)

residual capacity : 2 - 2 = 0

Stop

free:

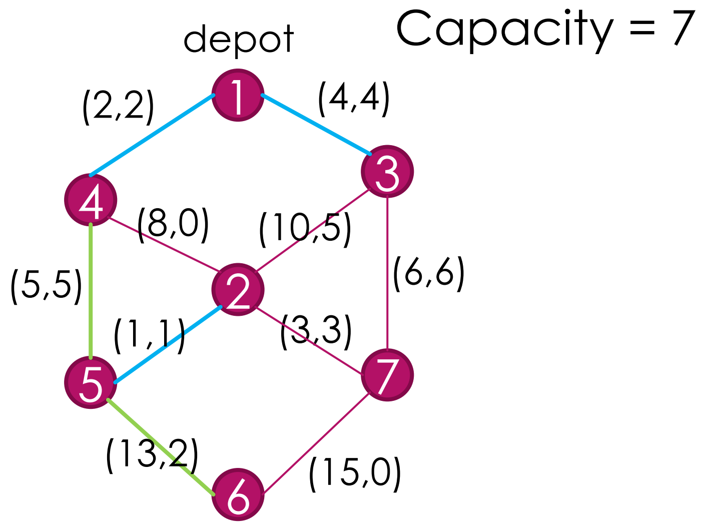
(2,3)	(2,7)	(3,7)
(3,2)	(7,2)	(7,3)

Route2:

(4,5)	(5,6)

Cost of route 2 = 20 + c[6][1] = 40
Load of route 2 = 7

When a route is finished, don't remember add the distance from the current end of the route to the depot

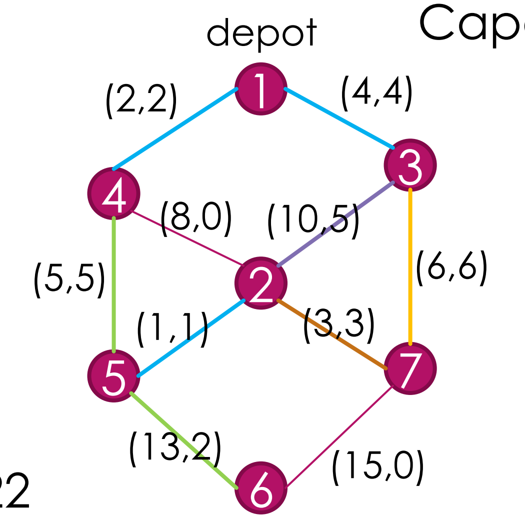


Path-Scanning(End)

free：∅

—	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

- A legal solution can be obtained by applying Path-scanning

(Congratulations, you can get 80+ for program score)

Improvement

- The so-called improvement refers to the reduction of total cost and the specific process is as follows:

s -> move operator -> s’

Move operator is an operator that transforms one solution to another, and then if s’ is better than s, we get an improvement.

Common Move Operators

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

Flip

Try to flip through all tasks to see if there are any improvements

$\nearrow$ (3,1)

R1: $\boxed{(1,3)}$ (1,4) (5,2)

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

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

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

R3: (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

c[][]:

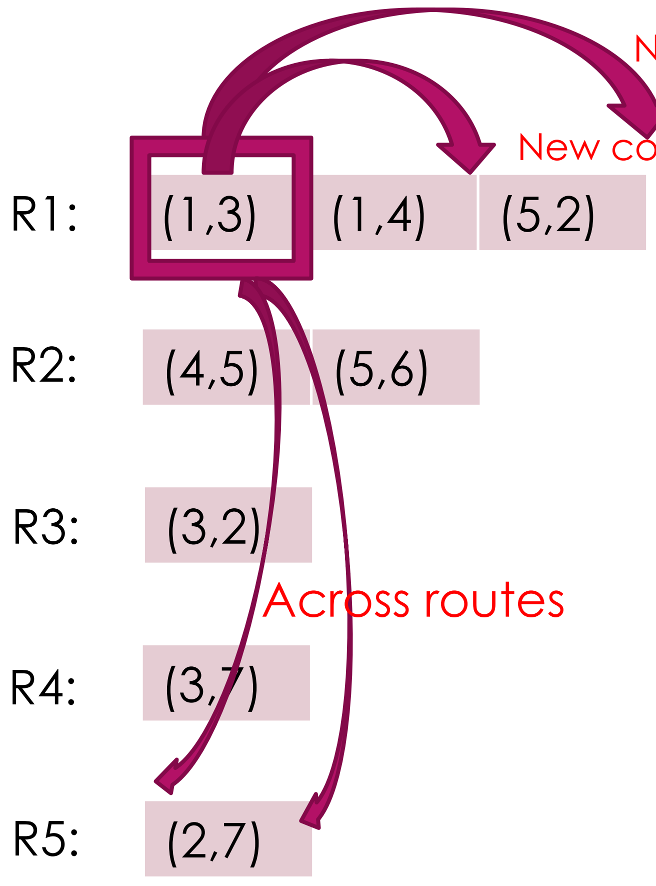
	1	2	3	4	5	6	7
1	0	8	4	2	7	20	10
2	8	0	9	6	1	14	3
3	4	9	0	6	10	21	6
4	2	6	6	0	5	18	9
5	7	1	10	5	0	13	4
6	20	14	21	18	13	0	15
7	10	3	6	9	4	15	0

Single insertion

Try to insert each individual task after other tasks or depot in the same route or another route.

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

New cost(1)= 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



load(1) = 7
cost(1) = 24

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

load(3) = 5
cost(3) =22

load(4) = 6
cost(4) = 20

load(5) = 3
cost(5) = 21

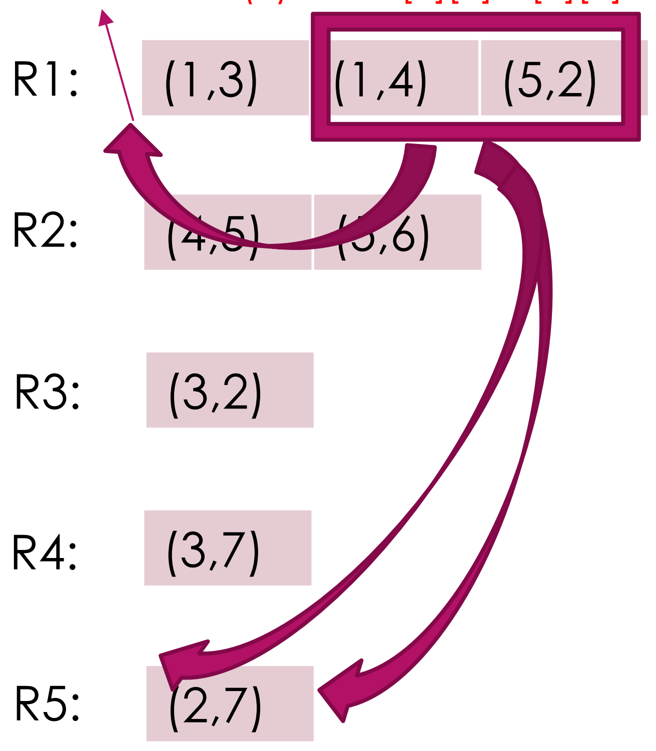
c[][]:

	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

Double insertion

Similar to the Single Insection, two consecutive tasks try to be inserted after other tasks or depot in the same route or another route.

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



load(1) = 7
cost(1) = 24

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

load(3) = 5
cost(3) = 22

load(4) = 6
cost(3) =22

load(5) = 3
cost(5) = 21

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

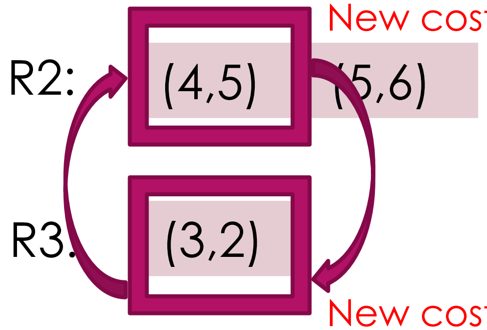
Swap

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

load(1) = 7
cost(1) = 24

New load(2) = 7

New cost(2)= 40-c[1][4]-c[5][5]-e[4][5]+c[1][3]+c[2][5]+e[3][2]=40-2-0-5+4+1+10=48



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

load(3) = 5
cost(3) =22

New load(3) = 5

New cost(3)= 22-c[1][3]-c[2][1]-e[3][2]+c[1][4]+c[5][1]+e[4][5]=22-4-8-10+2+7+5=14

R4: (3,7)

load(4) = 6
cost(4) = 20

R5: (2,7)

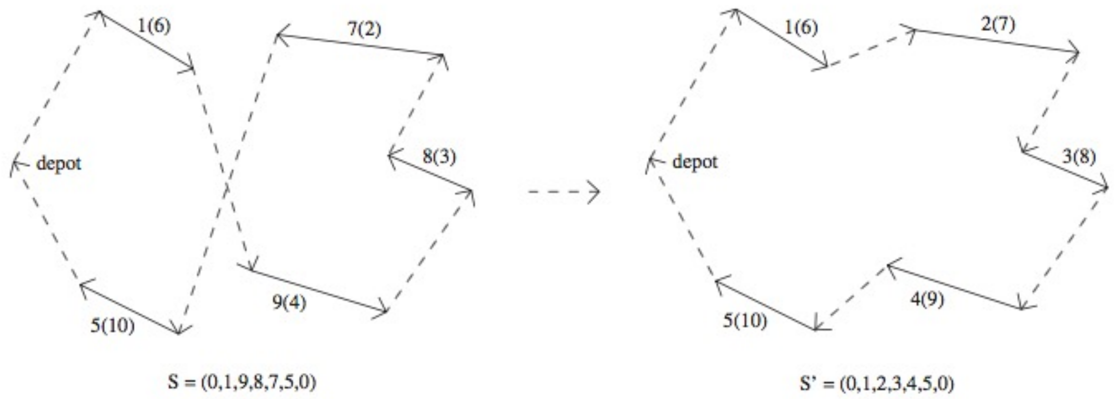
load(5) = 3
cost(5) = 21

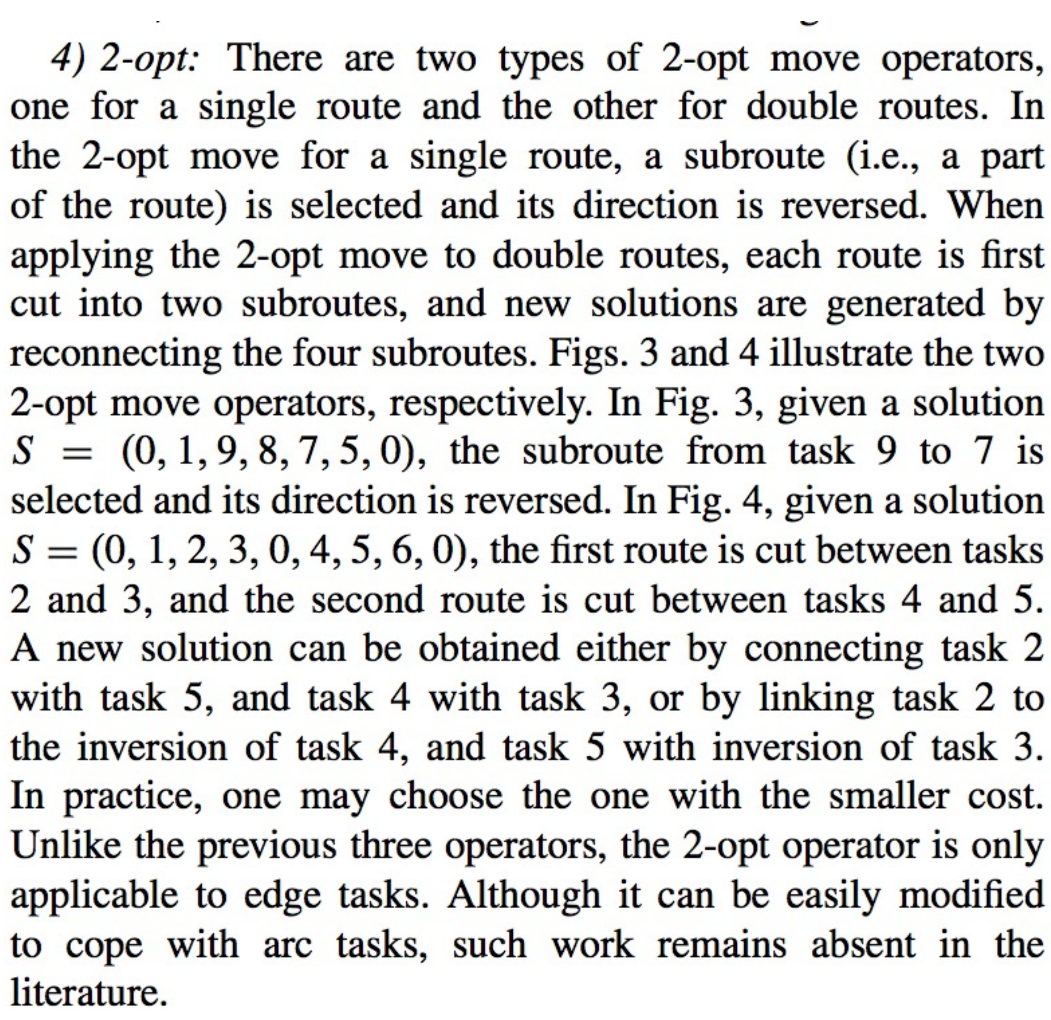
c[][]:

	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

2-opt

- optimal for single route

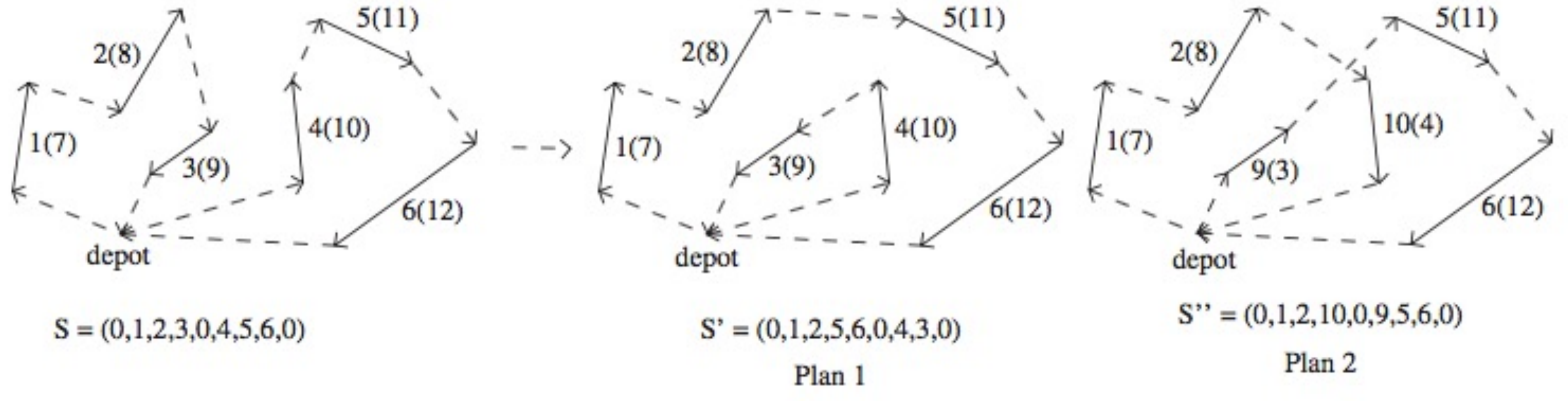




4) 2-opt: There are two types of 2-opt move operators, one for a single route and the other for double routes. In the 2-opt move for a single route, a subroute (i.e., a part of the route) is selected and its direction is reversed. When applying the 2-opt move to double routes, each route is first cut 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》

How to apply operators efficiently?

- Small step operators tend to fall into local optimality
- Big step operators may miss the global optimal solution when approaching the global optimal solution.
- No operator can perform well in all common scenarios.