Modelos e algoritmos

Problemas de roteamento

Ricardo Camargo (rcamargo@dep.ufmg.br)

2020

Construtivas

- Do vizinho mais próximo
- De inserção
- De economia

De melhoria

- Inserção de nós
- Inserção de arcos
- Troca 3-opt

Lista duplamente encadeada:

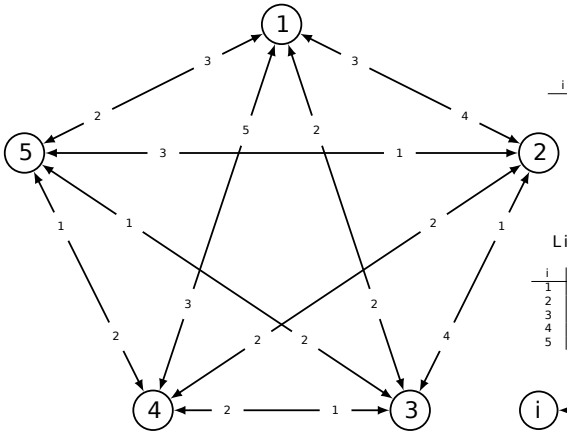
- Solução: $1 \to 3 \to 4 \to 2 \to 5 \to 1$
- nxt $=[\frac{3}{1}, \frac{5}{2}, \frac{4}{3}, \frac{2}{4}, \frac{1}{5}]$
- prd $=[\frac{5}{1}, \frac{4}{2}, \frac{1}{3}, \frac{3}{4}, \frac{2}{5}]$

PercorreRota:

1: $u \leftarrow 1$
2: $v \leftarrow nxt(u)$
3: Imprime u
4: **while** $u \neq v$ **do**
5: Imprime v
6: $v \leftarrow nxt(v)$
7: **end while**
8: Imprime v

Ideia:

Começando a partir de um nó inicial, seleciona o nó ainda não visitado mais próximo das extremidades da rota.



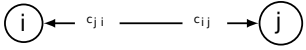
Matriz de custos

i → j	1	2	3	4	5
1	-	4	2	3	2
2	3	-	4	2	3
3	2	1	-	2	1
4	5	2	1	-	1
5	3	1	2	2	-

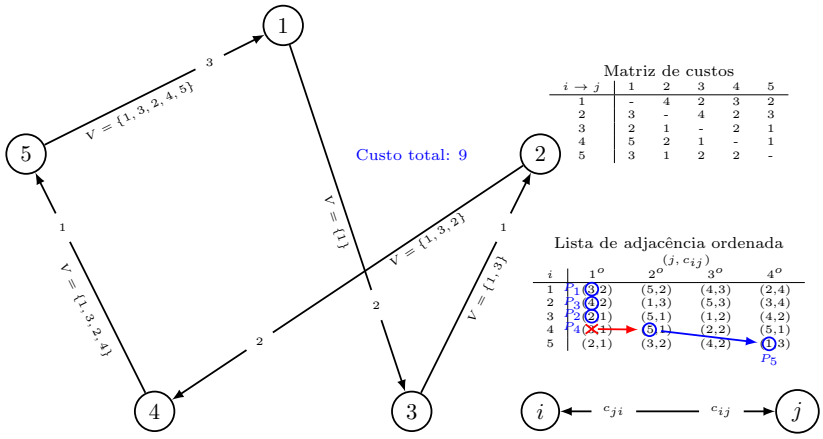
Lista de adjacência ordenada

(j, c_{ij})

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



Passo a Passo:



1: $V \leftarrow \{1, , n\}$ // conjunto ordenado
2: $u \leftarrow \text{first}(V)$
3: $u^0 \leftarrow u$
4: $V \leftarrow V \setminus \{u\}$
5: $\pi \leftarrow 0$ // comprimento da rota
6: **while** $|V| > 0$ **do**
7: $\quad v \leftarrow \underset{j \in V}{\text{argmin}}\{c_{uj}\}$
8: $\quad nxt(u) \leftarrow v$
9: $\quad prd(v) \leftarrow u$
10: $\quad \pi \leftarrow \pi + c_{uv}$
11: $\quad V \leftarrow V \setminus \{v\}$
12: $\quad u \leftarrow v$
13: **end while**
14: $nxt(v) \leftarrow u^0$
15: $prd(u^0) \leftarrow v$
16: $\pi \leftarrow \pi + c_{vu^0}$

Pode-se considerar as duas pontas da rota, no lugar de apenas uma, como no algoritmo acima.

```
17  def nearest_neighbor(n,N,c,sc):

33      nxt = [-1 for i in N];prd = [-1 for i in N];visited = [-1 for i in N]
34
35      (u,ac) = sc[0][0]
36      nxt[0] = u; prd[u] = 0; rtc = ac
37      visited[0] = 1; visited[u] = 1
38      while (True):
39          is_visited = False
40          for (v,ac) in sc[u]:
41              if visited[v] == -1:
42                  visited[v] = 1; is_visited = True
43                  nxt[u] = v; prd[v] = u; rtc += ac
44                  u = v
45                  break
46          if is_visited == False:
47              break
48      nxt[u] = 0; prd[0] = u; rtc += c[u][0]
49      return (rtc,nxt,prd)
```

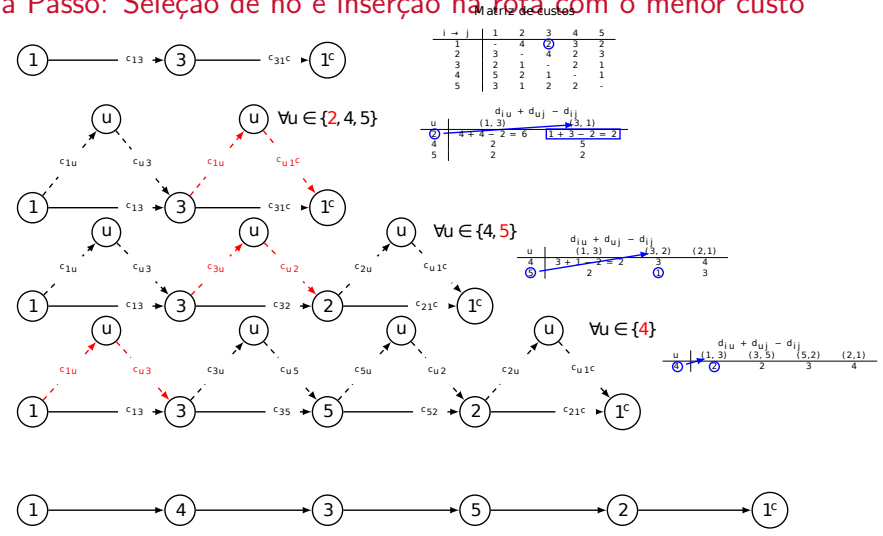
Ideia:

Começando a partir de um nó ou rota inicial, seleciona-se um nó ainda não visitado segundo algum critério pré-estabelecido (mais longe, mais perto, maior menor distância, menor maior distância) e o insere na rota na melhor posição possível.

Ideia

- Inserir no menor aumento de custo possível
- Dado o nó u a ser inserido, $\underset{(i,j)\in\pi}{\operatorname{argmin}}\{c_{iu} + c_{uj} - c_{ij}\}$, onde $\pi = \{i_1, i_2, \ldots, i_p\}$ é a sequência de visitação dos nós até o momento

Passo a Passo: Seleção de nó e inserção na rota com o menor custo



1: R : rota inicial com ao menos dois nós
2: $V \leftarrow N \setminus R$: nós ainda não visitados
3: **while** $|V| > 0$ **do**
4: $\quad u \leftarrow Select(V)$ // Seleciona u segundo algum critério
5: $\quad Insert(u, R)$ // Insere na rota na posição mais barata
6: $\quad V \leftarrow V \setminus \{u\}$
7: **end while**

Critérios de seleção

- Mais perto: seleciona-se o nó com a menor distância relativa a rota;
- Mais longe: seleciona-se o nó cuja menor distância relativa a rota é máxima;
- Mais longe (2): seleciona-se o nó com a maior distância relativa a rota;
- Maior soma: seleciona-se o nó com a maior soma de distâncias relativa a rota;
- Menor soma: igual ao critério maior soma, mas usando a menor soma de distâncias relativa a rota.

Python: insertion.py

```
17  def insertion(n,N,c,sc):

33      nxt = [-1 for i in N]
34      prd = [-1 for i in N]
35      visited = [-1 for i in N]
36      # initial route
37      rtc = first_subroute(visited,nxt,prd,sc,c)
38      while (True):
39         (u,v,j,chg) = select_node(N,visited,
40                         c,nxt,prd,
41                         criterion='nearest')
42         if j == -1: break
43         rtc += insert(u,v,j,
44                       chg,nxt,prd,visited)
45      return (rtc,nxt,prd)
46  def insert(u,v,j,chg,nxt,prd,visited):
47      nxt[u] = j; nxt[j] = v
48      prd[v] = j; prd[j] = u
49      visited[j] = 1
50      return chg
51  def first_subroute(visited,nxt,prd,sc,c):
52      u = 0; (v,ac) = sc[u][0]
53      visited[u] = 1; visited[v] = 1
54      nxt[u] = v; prd[u] = v
55      prd[v] = u; nxt[v] = u
56      return ac + c[v][u]
```

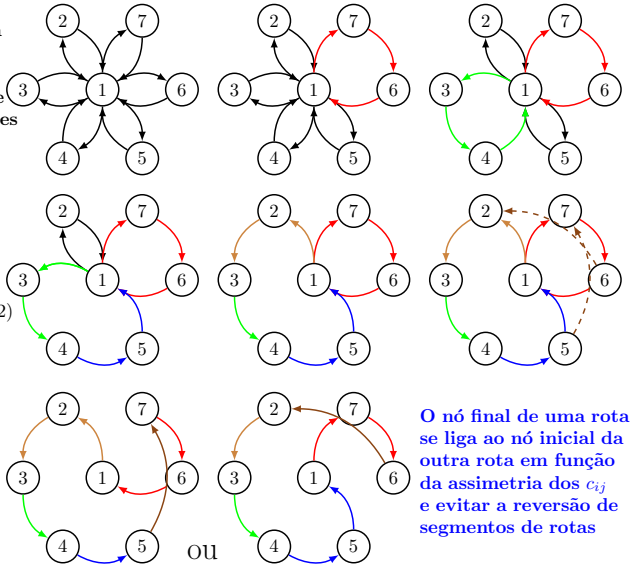
```
57  def select_node(N,visited,c,nxt,prd,
58                  criterion='nearest'):
59      if criterion == 'nearest':
60         best = (-1,-1,-1,float('inf'))
61         for i in N:
62           if visited[i] == -1:
63             best = best_insertion(best,i,nxt,c)
64      return best

87  def best_insertion(e,i,nxt,c):
88      u = 0; v = nxt[u]
89      while (True):
90         chg = c[u][i] + c[i][v] - c[u][v]
91         if (e[3] > chg):
92           e = (u,v,i,chg)
93         u = v
94         v = nxt[v]
95         if (u == 0):
96           break
97      return e
```

Ideia

- Montam-se rotas pendulares (rotas que saem da origem vão até um dado nó e retornam ao nó de origem) inicialmente.
- Depois, elabora-se uma lista de economias obtidas ao se fundir pares dessas rotas.
- Após a elaboração dessa lista, ordena-a da maior economia para a menor.
- Seguindo essa lista, fundem-se as rotas que possuem nós em rotas diferentes e são nós base, isto é, nós que começam ou terminam as rotas. Note que em função dos custos poderem ser assimétricos, a fusão só ocorre se não houver reversão de arcos nas rotas. Para tal, ligam-se o nó final com o nó inicial das rotas.

Lista ordenada das economias obtidas ao se fundir pares de rotas pendulares

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



O nó final de uma rota se liga ao nó inicial da outra rota em função da assimetria dos c_{ij} e evitar a reversão de segmentos de rotas

1: C^T : custo total
2: $S = Ordenado\{(i, j, s_{ij}) : i, j \in N \setminus \{1\}, s_{ij} = d_{1i} + d_{1j} - d_{ij}\}$
3: **for** $i \in N \setminus \{1\}$ **do**
4: $(\pi_i, b_i, e_i) \leftarrow (i, i, i)$ // liga nó a rota, e qual nó começa e termina a rota i
5: $(n_1, n_i, p_1, p_i) \leftarrow (i, 1, i, 1)$ // próximo e predecessor do nó
6: **end for**
7: **for** $(i, j, s) \in S$ **do**
8: **if** $(\pi_i \neq \pi_j) \wedge (e_{\pi_i} = i \wedge b_{\pi_j} = j)$ **then**
9: $(r_1, r_2) \leftarrow$ **if** $(\pi_i < \pi_j)$ **then** (π_i, π_j) **else** (π_j, π_i)
10: $C^T \leftarrow C^T - s$
11: $(b_{r_1}, e_{r_1}, n_i, p_j) \leftarrow (b_{\pi_i}, e_{\pi_j}, j, i)$
12: $(\pi_{r_2}, b_{r_2}, e_{r_2}) \leftarrow (-1, -1, -1)$
13: Reatribui nós de $b_{\pi_{r_1}}$ até $e_{\pi_{r_1}}$ a rota r_1
14: **end if**
15: **end for**

Python: savings.py

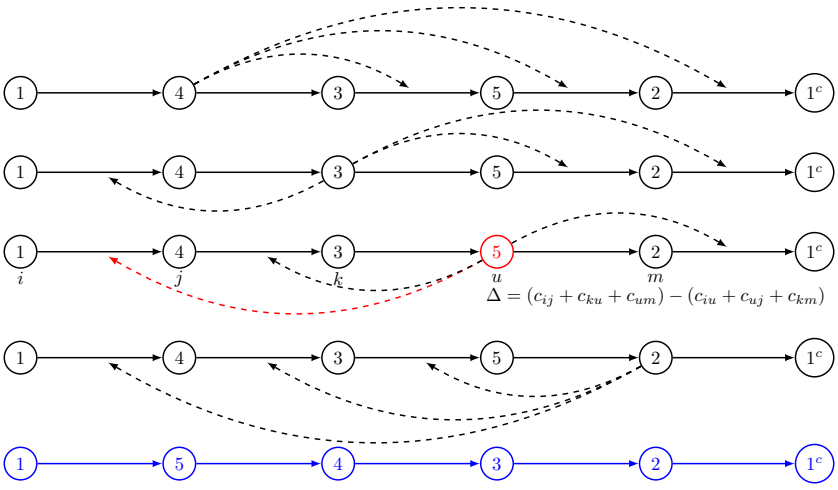
```
17  def savings(n,N,c):

32      nxt = [0 for i in N];prd = [0 for i in N]
33      b = [i for i in N];e = [i for i in N]
34      ra = [i for i in N]
35      nr = n - 1
36
37      rtc = sum([c[0][i] + c[i][0] for i in N if i > 0])
38      S = [(i,j,c[i][0]+c[j][0]-c[i][j]) for i in N for j in N if i != j and i != 0 and j != 0]
39      S.sort(key=lambda x : x[2],reverse=True)
40      for (i,j,s) in S:
41          if ra[i] != ra[j] and e[ra[i]] == i and b[ra[j]] == j:
42             rtc -= s
43             (r1,r2) = (ra[i],ra[j]) if (ra[i] < ra[j]) else (ra[j],ra[i])
44             b[r1],e[r1],nxt[i],prd[j],b[r2],e[r2] = b[ra[i]],e[ra[j]],j,i,-1,-1
45             re_assign(b[r1],e[r1],r1,nxt,ra)
46      nxt[0] = b[1]
47      return    (rtc,nxt,prd)
48  def re_assign(bn,en,r,nxt,ra):
49      v = bn
50      while (v != en):
51         ra[v] = r
52         v = nxt[v]
53      ra[v] = r
```

Métodos:

- Inserção de nós
- Inserção de arcos
- Troca 3-opt

Ideia:

Dado uma rota viável, seleciona-se um nó e o insere em uma posição melhor. Repete-se o processo até não ser possível mais melhorar o custo da rota.


1
4
3
5
2
1^c
1
4
3
5
2
1^c
1
4
3
5
2
1^c
i
j
k
u
m
Δ = (c_ij + c_ku + c_um) − (c_iu + c_uj + c_km)
1
4
3
5
2
1^c
1
5
4
3
2
1^c

1: c^T : custo total
2: $\pi = \{o_1, o_2, \ldots, o_n\}$ // Sequência de visitação na rota π
3: $stop \leftarrow False$
4: **while** $stop = False$ **do**
5: $\quad e^* \leftarrow (-1, -1, -1, -1, c^T)$
6: $\quad$ **for** $u \in \pi$ **do**
7: $\quad\quad e(i, j, k, m, \tilde{c}^T) \leftarrow SearchBestSelection(u)$
8: $\quad\quad$ **if** $e_c^* > e_c$ **then**
9: $\quad\quad\quad e^* \leftarrow e$
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: $\quad$ **if** $e_c^* < C^T$ **then**
13: $\quad\quad Update(\pi, c^T)$
14: $\quad$ **else**
15: $\quad\quad stop \leftarrow True$
16: $\quad$ **end if**
17: **end while**

Python: nodeinsertion.py

```
21  def node_insertion(c,rtc,nxt,prd):

35      outer_loop_stop = False
36      while (outer_loop_stop == False):
37          inner_loop_stop = False
38          best = (rtc,-1,-1,-1,-1,-1)
39          while (inner_loop_stop == False):
40              u = 0
41              v = nxt[u]
42              inner_loop_stop = True
43              while(v != 0):
44                  e = search(v,c,rtc,nxt,prd)
45                  if best[0] > e[0]:
46                      best = e
47                      inner_loop_stop = False
48                  v = nxt[v]
49          if (best[1] != -1):
50              rtc = update(best,nxt,prd)
51          else:
52              outer_loop_stop = True
53      return (rtc,nxt,prd)
```

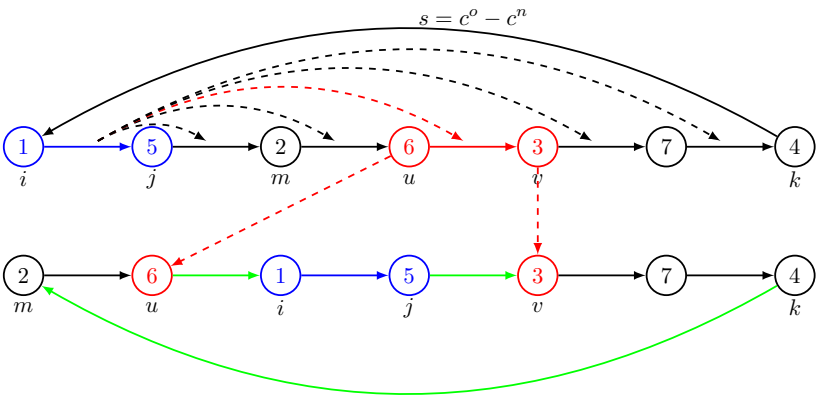
```
54  def update(best,nxt,prd):
55      (rtc,v,k,k,nv,pk,nk) = best
56      nxt[v] = k;prd[k] = v
57      nxt[k] = nv;prd[nv] = k
58      nxt[pk] = nk;prd[nk] = pk
59      return rtc
60  def search(k,c,rtc,nxt,prd):
61     pk = prd[k]; nk = nxt[k];
62     ck = c[pk][k] + c[k][nk]
63     s = (rtc,-1,-1,-1,-1,-1)
64     v = nxt[k]
65     while (pk != v):
66        nv = nxt[v]
67        oc = c[v][nv] + ck
68        nc = c[v][k] + c[k][nv] + c[pk][nk]
69        saved = oc - nc
70        if (s[0] > rtc - saved):
71            s = (rtc - saved,v,k,k,nv,pk,nk)
72        v = nxt[v]
73     return s
```

Ideia:

Ideia semelhante a heurística de inserção de nó, porém usando agora arcos.
Seleciona-se um arco e o insere entre outro arco na melhor posição possível.

$$c^o = c_{ki} + c_{ij} + c_{jm} + c_{uv}$$

$$c^n = \min\{c_{km} + c_{ui} + c_{ij} + c_{jv}, c_{km} + c_{uj} + c_{ji} + c_{iv}\}$$



1: R : rota inicial
2: $A \leftarrow \pi(R)$: sequência de arcos da rota
3: **while** $|A| > 0$ **do**
4: $\quad (i,j) \leftarrow Select(A)$ // Seleciona (i,j) segundo algum critério
5: $\quad Insert((i,j), R)$ // Insere na rota na posição mais barata
6: $\quad A \leftarrow A \setminus \{(i,j)\}$
7: **end while**

Python: arcinsertion.py

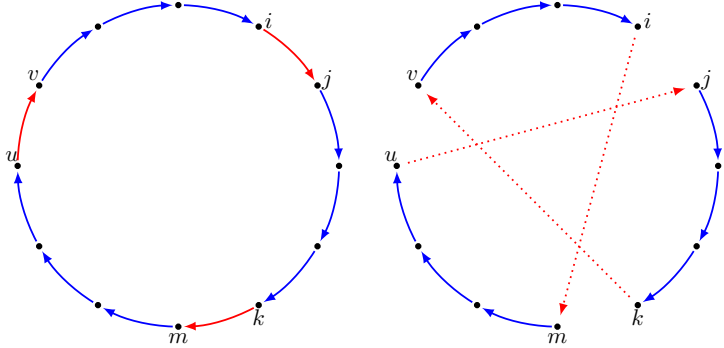
```
25  def arc_insertion(c,rtc,nxt,prd):
40    stop = False
41    while (stop == False):
42      i = 0; j = nxt[i]
43      stop = True
44      best = (rtc,-1,-1,-1,-1,-1,-1)
45      while(j != 0):
46        cs = search(i,j,c,rtc,nxt,prd)
47        if best[0] > cs[0]:
48           best = cs; stop = False
49        i = j;j = nxt[i]
50      if (best[1] != -1):
51        rtc = update(best,nxt,prd)
52    return (rtc,nxt,prd)
53  def update(best,nxt,prd):
54    (rtc,u,v,i,j,k,m,r,s) = best
55    nxt[u] = v;prd[v] = u
56    nxt[i] = j;prd[j] = i
57    nxt[k] = m;prd[m] = k
58    nxt[r] = s;prd[s] = r
59    return rtc
```

```
60  def search(i,j,c,rtc,nxt,prd):
61    u = j;v = nxt[u]
62    k = prd[i];m = nxt[j]
63    cs = (rtc,-1,-1,-1,-1,-1,-1,-1,-1)
64    while (i != v):
65      if (u == j):
66        oc = c[k][i] + c[i][j] + c[u][v]
67        nc = c[k][j] + c[j][i] + c[i][v]
68        saved = oc - nc
69        if (cs[0] > rtc - saved):
70          cs = (rtc - saved,k,j,j,i,i,v,i,v)
71      else:
72        oc = c[k][i] + c[i][j] + c[j][m] + c[u][v]
73        ncij = c[k][m] + c[u][i] + c[i][j] + c[j][v]
74        ncji = c[k][m] + c[u][j] + c[j][i] + c[i][v]
75        (saved,ii,jj) = (oc - ncij,i,j) if ncij < ncji else (oc -
           ↪    ncji,j,i)
76        if (cs[0] > rtc - saved):
77          cs = (rtc - saved,k,m,u,ii,ii,jj,jj,v)
78      u = v;v = nxt[u]
79    return cs
```

Ideia:

Variante do 2-opt, consiste em separar, da melhor forma possível, a rota em 3 segmentos diferentes para depois reconectá-los de forma a obter uma rota mais interessante ou econômica. Aqui, como estamos trabalhando com os custos assimétricos, evitamos movimentos que possam exigir a inversão de segmentos das rotas. Razão pelo qual o 2-opt e combinações de ligação que requerem essa inversão são evitados.

$$\Delta = (c_{ij} + c_{km} + c_{uv}) - (c_{im} + c_{uj} + c_{kv})$$



1: c^T : custo total
2: $\pi = \{o_1, o_2, \ldots, o_n\}$ // Sequência de visitação na rota π
3: *stop* $\leftarrow$ *False*
4: **while** *stop* $=$ *False* **do**
5: $e((i,j),(u,v),(k,m)) \leftarrow SelectArcs(Criteria)$
6: $\Delta \leftarrow (c_{ij} + c_{km} + c_{uv}) - (c_{im} + c_{uj} + c_{kv})$
7: **if** $\Delta > 0$ **then**
8: $Update(\pi, e)$
9: **else**
10: *stop* $\leftarrow$ *True*
11: **end if**
12: **end while**

3-opt

```
32  def three_opt(n,N,c,rtc,nxt,prd):
47      stop = False
48      while stop == False:
49          stop = True
50          best = (rtc,-1,-1,-1,-1,-1,-1)
51          u = 0; v = nxt[u]; stopu = prd[prd[0]]
52          while (u != stopu):
53              i = v; j = nxt[i]; stopi = prd[0]
54              while (i != stopi):
55                  k = j; m = nxt[k]
56                  while (k != 0):
57                      oc = c[u][v]+c[i][j]+c[k][m]
58                      nc = c[u][j]+c[k][v]+c[i][m]
59                      saved = oc - nc
60                      if (best[0] > rtc - saved):
61                          stop = False
62                          best = (rtc - saved,u,j,k,v,i,m)
63                      k = m; m = nxt[k]
64                  i = j; j = nxt[i]
65              u = v; v = nxt[u]
66          if best[1] != -1:
67              rtc = improve_solution(best,nxt,prd)
68      return rtc,nxt,prd
69  def improve_solution(best,nxt,prd):
70      (rtc,u,j,k,v,i,m) = best
71      nxt[u] = j;prd[j] = u
72      nxt[k] = v;prd[v] = k
73      nxt[i] = m;prd[m] = i
74      return rtc
```

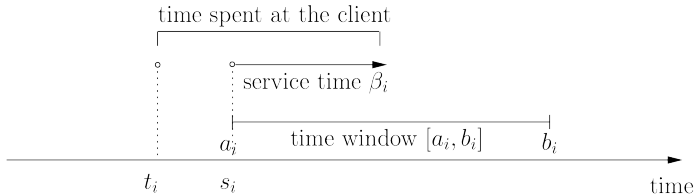
O problema do caixeiro viajante com janelas de tempo e custos assimétricos

O problema do caixeiro viajante com janelas de tempo e custos assimétricos consiste em achar uma rota de custo mínimo que começa e termina no depósito, e atenda a todos os clientes respeintando todas as janelas de tempo dos clientes.

Cada cliente $i \in V$ tem:

- Um tempo de serviço β_i necessário para ser atendido;
- Um tempo de início de disponibilidade para ser atendido a_i: não se pode servir a um cliente antes deste tempo;
- Um tempo limite para ser atendido b_i: o cliente deve ser atendido antes deste tempo.

- Cada cliente só pode ser visitado uma única vez.
- Os custos nos arcos representam também os tempos de viagem ou translado.
- O custo da rota é a soma dos custos dos arcos que a formam.
- Os tempos de disponibilidade e limite definem a janela de atendimento de cada cliente $[a_i, b_i]$ dentro do qual ele deve ser servido.
- Pode-se chegar antes do início de disponibilidade e esperar até o tempo de início para atender ao cliente, porém a data limite tem de ser respeitada.
- Uma rota que desrespeita algum cliente é considerada inviável.



- t_i tempo de chegada no cliente;
- s_i tempo que o atendimento no cliente começou.

Dificuldade:

A dificuldade do problema não depende apenas do número de nós, mas também da "qualidade" das janelas de tempo. Há poucos trabalhos na literatura que resolvem esse problema de forma eficiente e exata.

O que deve ser feito:

Adaptar uma heurística construtiva e uma de melhoria para o problema do caixeiro viajante com custos assimétricos e com janelas de tempo.

Gerando instâncias:

Gere n pontos aleatórios (lat, lon) num plano de 30×30. Depois monte uma rota aleatória. Calcule seu comprimento. Gere valores aleatórios para os tempos de atendimento e janela de tempo de forma que a solução da rota aleatória seja viável.

- R. Ferreira da Silva and S. Urrutia. A General VNS heuristic for the traveling salesman problem with time windows, Discrete Optimization, V.7, Issue 4, pp. 203-211, 2010.
- M.W.P. Savelsbergh. Local search in routing problems with time windows, Annals of Operations Research 4, 285-305, 1985.
- M.M. SOLOMON and J.DESROSIERS, Time Window Constrained Routing and Scheduling Problems Transportation Science Vol. 22, No. 1 (February 1988), pp. 1-13