3 Algoritmos para o Problema de Roteirização Estática de Veículos com Janelas de Tempo

Este capítulo apresenta diversos algoritmos propostos para resolver o Problema de Roteirização Estática de Veículos com Janelas de Tempo e um algoritmo híbrido que se baseia no paradigma de Otimização por Colônias de Formigas e no método de Descida em Vizinhança Variável Aleatória.

3.1 Introdução

Otimização por Colônias de Formigas (*Ant Colony Optimization - ACO),* são algoritmos baseados no comportamento de formigas quando imersas em colônias de indivíduos semelhantes. Estes algoritmos tem se mostrado eficientes na resolução de problemas de otimização combinatória de elevada complexidade computacional, difíceis de serem resolvidos por técnicas exatas de otimização.

Esta técnica foi inicialmente pesquisada por Dorigo et al. (1991), que propuseram um algoritmo denominado *Ant System* (AS) para a resolução do TSP. Em 1997, Dorigo e Gambardella (1997) criaram o *Ant Colony System* (ACS) a partir do AS, o que representou um novo avanço no desempenho dos algoritmos. Em 1999, Bullnheimer et al. (1999a e 1999b) aplicaram esta técnica pela primeira ao VRP*,* através de um modelo baseado no AS, que ficou denominado por AS-VRP. Em 1999, Gambardella et al. (1999) propõem uma técnica denominada *Multiple Ant Colony System* aplicado ao VRPTW (MACS-VRPTW).

Outras aplicações mais recentes do ACO a problemas de roteirização podem ser encontradas no trabalhos de Bell e McMullen (2004), Abad e Gajpal (2008), Yu e Yang (2011) e Ding et al (2012).

Para o VRPTW, propõe-se um algoritmo híbrido denominado MACS-RVND, que consiste numa extensão melhorada do MACS-VRPTW. Este algoritmo utiliza como fase de busca local o método de Descida em Vizinhança Variável Aleatória (*Random Variable Neighborhood Descent - RVND*) e que foi

proposto por Subramanian (2012). Por sua vez, o RVND é uma variante do método de Descida em Vizinhança Variável (*Variable Neighborhood Descent - VND*) proposto por Mladenović e Hansen (1997).

3.2 Sistema de Múltiplas Colônias de Formigas para o VRPTW

Para explicar o funcionamento do algoritmo MACS-VRPTW serão detalhados os algoritmos que foram utilizados como base para seu desenvolvimento.

3.2.1 Ant System para o TSP

O AS foi o primeiro algoritmo baseado no comportamento de formigas, divulgado por Dorigo et al. (1991), e serviu de base para o desenvolvimento de muitos outros modelos, como o ACS e o AS-VRP. Sua aplicação original foi criada para resolver o Problema do Caixeiro Viajante (TSP). Este algoritmo, assim como os outros algoritmos baseados em formigas, simula ciclos de vida de cada formiga atuante no problema.

Neste algoritmo são realizadas simulações do ciclo de vida de cada formiga, o qual consiste na realização de um caminho a partir do seu ninho até uma fonte de alimento. Cada passo de uma formiga é definido por uma regra probabilística de decisão, pois deve ser escolhido um arco entre todos os outros existentes a partir da sua posição atual. Esta regra baseia-se nos valores da quantidade de feromônios acumulada pela passagem das formigas precedentes e do custo (distância) existente em cada arco. Para cada um destes valores, pode-se dar um peso diferente através dos parâmetros *alfa* e *beta*, respectivamente. O cálculo da atratividade de cada arco pode ser descrito matematicamente por:

$$a_{ij}(t) = \frac{\left[\tau_{ij}(t)\right]^{\alpha}\left[\eta_{ij}\right]^{\beta}}{\sum_{l \in N_i}\left[\tau_{il}(t)\right]^{\alpha}\left[\eta_{il}\right]^{\beta}} \tag{16}$$

A partir da expressão (16) é obtida a atratividade $a_{ij}(t)$ para uma formiga que se encontra em um nó i qualquer, e para todo nó j pertencente a N_i, onde N_i é o conjunto dos nós adjacentes ao nó i. A variável τ_{ij} representa a quantidade de

feromônio acumulada no arco *(i,j)*, e a variável η_{ij} é igual ao inverso do comprimento do arco *(i,j)*. O parâmetro *t* indica que os valores estão variando a cada iteração *t*. Uma vez calculados os valores de *a* de todos os arcos incidentes a *i*, define-se a probabilidade de uma formiga *k* escolher percorrer o arco *ij* como:

$$p_{ij}^{k}(t) = \frac{a_{ij}(t)}{\sum_{l \in N_i^k} a_{il}(t)} \tag{17}$$

Na expressão (17), N_i^k é o subconjunto de N_i que contém somente os nós que ainda não foram visitados pela formiga *k*, ou seja, os nós que não constam na memória da formiga. A Figura 2 ilustra o procedimento:

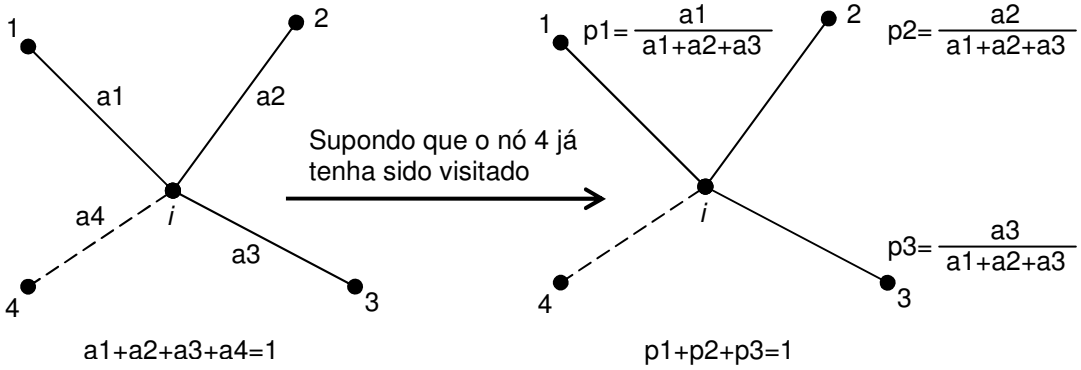

Figura 2 - Cálculo das probabilidades de escolha de um arco, para o *Ant System*

Na Figura 2, o conjunto N_i consiste dos nós 1, 2, 3 e 4, enquanto que N_i^k possui apenas os elementos 1, 2 e 3. Cada um destes três nós tem, então, suas probabilidades p1, p2 e p3 determinadas e, com base nestas probabilidades, o nó seguinte será escolhido. Calcula-se então a probabilidade acumulada a partir das probabilidades de cada arco p_{ij} e sorteia-se um número aleatório entre 0 e 1. O arco escolhido será aquele em que o número sorteado possuir valor maior ou igual à $p_{ij\text{-}1(acumulado)}$ e menor que $p_{ij(acumulado)}$.

Após todas as *k* formigas completarem seus caminhos, a quantidade de feromônios de todos os arcos é atualizada. Esta atualização consiste na evaporação dos feromônios e na deposição de novos feromônios de forma global, a qual é representada pela seguinte expressão:

$$\tau_{ij}(t) \leftarrow (1-\rho)\tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}(t) \tag{18}$$

Na expressão (18), o parâmetro ρ representa a intensidade de evaporação dos feromônios. O somatório representa todas as contribuições individuais de cada formiga *k* que tenha passado pelo arco *(i,j)* na iteração em questão; $\Delta\tau_{ij}^{k}$ é a quantidade de feromônios que a formiga *k* deposita no arco *(i,j),* e é definida como o inverso do comprimento total do percurso L_k que a formiga *k* percorreu. O número total de formigas é definido pelo parâmetro *m*. Quanto mais formigas passarem por um arco, mais feromônios serão depositados no arco, e o tamanho destas deposições é proporcional à qualidade das soluções encontradas. Matematicamente, tem-se:

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} \dfrac{1}{L_k(t)}, & se (i,j) \in \text{percurso feito pela formiga } k \text{ na iteração } t \\ 0, & \text{caso contrário} \end{cases} \tag{19}$$

Esta é a ideia básica do AS, aplicado ao problema do caixeiro viajante. A seguir, apresenta-se outra meta-heurística que foi desenvolvida tendo como base o AS, conhecida como *Ant Colony System*.

3.2.2 Ant Colony System para o TSP

O *Ant Colony System* consiste num melhoramento do *Ant System* proposto por Dorigo e Gambardella (1997). Uma importante diferença entre AS e o ACS é a atualização dos feromônios. No caso do AS, ela é feita de forma global e unificada, após o término de cada iteração. No ACS a atualização dos feromônios se dá tanto de forma local após a ação de cada formiga, quanto de forma global após o fim de cada iteração, reforçando apenas o melhor caminho encontrado pelas formigas.

Entretanto, a principal diferença entre os algoritmos consiste na regra de decisão das formigas. No ACS, esta regra de decisão trabalha simultaneamente com a regra probabilística (desbravadora) do AS e com uma regra determinística (exploratória). Tem-se um novo parâmetro chamado *q0* que varia de 0 a 1, e que define o poder de exploração de novos caminhos. Sorteia-se então um número

aleatório *q* e se este número for maior que *q0*, a regra de decisão da formiga é probabilística, usando as expressões (16) e (17), mas com o expoente α da variável τ_{ij} igual a 1.

Por outro lado, se o número aleatório sorteado *q* for menor que *q0*, a formiga tomará sua decisão baseada no conhecimento disponível sob a forma de depósitos de feromônios e distâncias. Em outras palavras, a regra de decisão da formiga será, para todo *j* pertencente a N_i^k:

$$p_{ij}^k(t) = \begin{cases} 1, \quad se \quad j = \arg\max a_{ij}, \quad onde\, a_{ij} = \dfrac{[\tau_{ij}(t)][\eta_{ij}]^\beta}{\sum_{l \in N_i} [\tau_{il}(t)][\eta_{il}]^\beta} \\ 0, \quad caso \quad contrário \end{cases} \tag{20}$$

Ou seja, a formiga escolherá o nó *j* que maximize o valor de $a_{ij.}$ Essa é uma regra determinística, que define que a formiga irá explorar o conhecimento já adquirido ao invés de procurar por novos caminhos melhores. De forma mais concisa, pode-se resumir a regra de decisão do modelo ACS como:

$$j = \begin{cases} \arg\max_{j \in N_i^k} \{[\tau_{ij}(t)][\eta_{ij}]^\beta\}, \quad se \quad q \le q0 \\ S, \quad caso \quad contrário \end{cases} \tag{21}$$

Na expressão (21), *S* é uma variável aleatória discreta com a probabilidade calculada com a fórmula (17), com o a_{ij} calculado pela fórmula (16), com o parâmetro α igual a 1.

A outra grande diferença para o modelo AS, como dito anteriormente, é a atualização dos feromônios. Primeiramente, o modelo ACS faz uma atualização local em cada arco, ou seja, sempre que uma formiga percorre um arco, a quantidade de feromônios em tal arco é modificada de acordo com a regra abaixo.

$$\tau_{ij}(t) \leftarrow (1-\rho)\tau_{ij}(t) + \rho\tau_0 \tag{22}$$

Na expressão (22), τ_0 é a quantidade inicial de feromônios em cada arco. Esta quantidade inicial de feromônios é definida como o inverso do comprimento da solução encontrada através do método do *Nearest Neighbor* ($1/L_{nn}$), o qual é detalhado na seção 3.2.3. Adicionalmente, pode-se multiplicar este comprimento L_{nn} pelo número de nós do problema, diminuindo ainda mais esta concentração

inicial de feromônios ($1/n.L_{nn}$). A expressão (22) faz com que a concentração de feromônios de um arco sofra uma redução cada vez que tal arco é percorrido por uma formiga. Isso garante que as informações mais antigas vão gradativamente assumindo um peso cada vez menor na decisão atual de cada formiga, e que novas e melhores soluções possam surgir.

No AS, toda formiga era responsável por depósitos de feromônios, que seriam tão maiores quanto melhores fossem os caminhos encontrados por cada formiga. No caso do ACS, esta regra é diferente. O reforço de feromônios neste caso não é feito por cada formiga, mas sim de forma unificada e centralizada. Ao final de cada iteração, o melhor caminho encontrado é o único que tem sua densidade de feromônios incrementada, representando uma diferença importante com relação ao modelo AS. Agora, todos os arcos que compõem o caminho mais curto encontrado têm sua densidade de feromônios atualizada segundo a expressão (13), onde, agora, $\Delta\tau_{ij}(t)$ é igual ao inverso do comprimento do caminho mais curto. Ou seja, quanto melhor for a qualidade da solução, ou quanto menor for o custo desta solução, maior é a contribuição de feromônios que será adicionada aos arcos desta melhor solução.

O ACS foi a base para o desenvolvimento do MACS-VRPTW, o qual será visto na seção seguinte e que é aplicado ao problema de roteirização de veículos com janelas de tempo.

3.2.3 Algoritmo do Vizinho Mais Próximo – NN

O algoritmo do Vizinho Mais Próximo (*Nearest Neighbor)* foi abordado no trabalho de Solomon (1987). Neste algoritmo, parte-se de um nó de origem e os nós são um a um gradativamente incorporados na rota. Sua ideia principal consiste em ordenar os nós adjacentes ao nó incorporado mais recentemente na rota em ordem decrescente de custos e incluir na rota o nó com menor custo, respeitando as restrições de capacidade dos veículos e de janela de tempo dos clientes. Sendo assim, para cada nó visitado, uma nova lista dos nós mais próximos é construída até que todos os nós sejam visitados. No caso do VRP estes custos indicam a distância entre o nó atual e o nó adjacente, mas no VRPTW estes custos são obtidos a partir da expressão (23).

$$d_{ij} = \delta_1.t_{ij} + \delta_2.espera_{\ j} + \delta_3.urgência_{\ j} \quad (23)$$

Onde, t_{ij} é o tempo de viagem, $espera_j=max(a_j-(começo_i+s_i+t_{ij}),0)$, $urgência_i=b_j-(começo_j+s_i+t_{ij})$, $começo_j=max(a_j,começo_i+s_i+t_{ij})$, se j é viável, δ_1, δ_2 e δ_3 são parâmetros que ponderam cada parcela do custo.

3.2.4 Algoritmo de Inserção – PFIH

Solomon (1987) descreve e avalia três heurísticas de inserção para o VRPTW, as quais foram denominadas I1, I2 e I3. O autor conclui que a heurística I1 apresentou os melhores resultados, mas também indica que sua combinação com a heurística *Sweep* (Gillet e Miller, 1974) obtém excelentes soluções com tempo computacional razoável. Esta heurística híbrida foi denominada *Push Forward Insertion Heuristic* (PFIH), a qual é descrita por Thangiah et al. (1994). Nesta heurística, inicia-se com uma rota vazia iniciando e retornando ao depósito e os nós não roteirizados são inseridos sequencialmente, seguindo uma ordem de prioridade e respeitando as restrições de capacidade dos veículos e de janela de tempo. Caso não seja possível adicionar o nó atual nas rotas já criadas, então se cria uma nova rota fechada com o nó atual. A prioridade se inserção é calculada a partir da expressão (24).

$$p_i = -\omega_1.t_{oi} + \omega_2.b_j + \omega_3\left(\left(\left|p_i - p_j\right|/360\right).t_{oi}\right) \quad (24)$$

Onde: ω_1=peso para o tempo de viagem do cliente i ao depósito, ω_2=peso para o instante final da janela de tempo do cliente j, ω_3=peso da coordenada polar normalizada pelo tempo de viagem, p_i=coordenada polar cliente i.

3.2.5 Busca Local – *Cross*

Conforme a proposta original do MACS-VRPTW, o procedimento de busca local consiste na aplicação da vizinhança *Cross* (Taillard et al., 1997). O algoritmo desta vizinhança é detalhado na seção 3.3.3.6.

3.2.6 MACS-VRPTW

Proposto por Gambardella et al. (1999), o MACS-VRPTW é uma extensão do ACS para o problema de roteirização de veículos com janelas de tempo. Seu diferencial consiste no tratamento de múltiplos objetivos, os quais são minimizar a quantidade de veículos e o tempo total das rotas. Neste caso, o primeiro objetivo tem prioridade sobre o segundo. Este algoritmo utiliza-se de duas colônias de formigas, onde cada uma visa a otimização de um dos objetivos do problema.

A proposta inicial do algoritmo deveria usar computação paralela, onde ambas as colônias estariam ativas e procurando por melhores soluções ao mesmo tempo. Visando a simplificação deste processo, apresenta-se a seguir o algoritmo da rotina MACS-VRPTW trabalhando com ciclos (Algoritmo 1).

Algoritmo 1: MACS-VRPTW

```
1. Procedimento MACS-VRPTW()
2. #quantidade_veiculo(s*) retorna o número de veículos da solução s*
3. s* ← Encontre uma solução inicial pela heurística do Vizinho Mais Próximo
4. Enquanto não se cumpre a condição de parada faça
5.       v ← #quantidade_veiculos(s*)
6.       s* ← Execute ACS-VEI(v - 1)
7.       s* ← Execute ACS-TIME(v)
8. Fim enquanto
```

Inicialmente, obtém-se uma solução viável para o VRPTW através da heurística do Vizinho Mais Próximo, a qual é utilizada como referência inicial para redução do tempo de busca pelo MACS-VRPTW. Então a solução é melhorada pelas duas colônias, ACS-VEI e ACS-TIME. Ambas as colônias utilizam depósitos de feromônios independentes, mas elas se comunicam através do compartilhamento da melhor solução encontrada até o momento, operação esta que é gerenciada pelo MACS-VRPTW. A condição de parada pode ser definida como um tempo limite de execução ou um número máximo simulações do ciclo de vida da formiga sem que haja melhoria na solução.

3.2.6.1 Algoritmo ACS-VEI

Dentro de um processo iterativo, a solução atual é melhorada primeiramente pelo algoritmo ACS-VEI (Algoritmo 2), que tenta encontrar uma solução viável que possua menos veículos que a solução atual. O algoritmo inicia com uma

solução obtida pelo método do vizinho mais próximo utilizando um número máximo de veículos *v*. Para cada formiga *k* são construídos caminhos com o objetivo de visitar o maior número possível de clientes e são atualizadas as variáveis *IN* incrementando cada variável IN_j para os clientes j que não foram visitados na solução s_k. Se a solução s_k obtida possuir um número maior de clientes visitados do que a melhor solução do ACS-VEI e caso esta solução seja factível (visita todos os clientes), então foi encontrada uma solução com um veículo a menos do que a melhor solução do MACS-VRPTW *s**. A execução do ACS-VEI é interrompida e a solução é retornada ao MACS-VRPTW. Caso contrário, ao término de cada iteração, realiza-se a atualização global de feromônios que é aplicada duplamente, aplicando a expressão (18) com a melhor solução atual *s** e a melhor solução do ACS-VEI $s^{ACS\text{-}VEI}$.

Algoritmo 2: ACS-VEI

1. **Procedimento** ACS-VEI(*v*)
2. // retorna o número de clientes que foram visitados na solução *s*
3. #clientes_visitados(s)
4. **Inicializa** matriz de feromônios e estruturas de dados utilizando *v*
5. $s^{ACS\text{-}VEI} \leftarrow$ Encontre uma solução inicial com *q* veículos aplicando o Vizinho Mais Próximo
6. **Repita**
7. **Para cada** formiga *k* **faça**
8. // Constrói uma solução *s* para a formiga *k*
9. $s_k \leftarrow$ simulaFormiga(k, buscaLocal=FALSE, IN)
10. // Atualiza os valores de IN para os nós não visitados
11. **Para cada** cliente $j \notin s_k$ **faça**
12. $IN_j \leftarrow IN_j + 1$
13. **Fim para cada**
14. // Atualiza a melhor solução caso tenha sido melhorada
15. **Se** #clientes_visitados(s_k) > #clientes_visitados($s^{ACS\text{-}VEI}$) **então**
16. $s^{ACS\text{-}VEI} \leftarrow s_k$
17. **Para cada** j **faça**
18. $IN_j \leftarrow 0$
19. **Se** $s^{ACS\text{-}VEI}$ é factível **então**
20. **Retorne** $s^{ACS\text{-}VEI}$ **para** MACS-VRPTW
21. **Fim se**
22. **Fim para cada**
23. // Atualização global de feromônios de utilizando ambas as soluções $s^{ACS\text{-}VEI}$ e *s**
24. $\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \rho / Ls^* \qquad \forall (i,j) \in s^*$
25. $\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \rho / Ls^{\text{ACS-VEI}} \qquad \forall (i,j) \in s^{ACS-VEI}$
26. **Até que** um número de iterações *nIter* seja atingido

3.2.6.2 Algoritmo ACS-TIME

Semelhante ao ACS-VEI, no ACS-TIME (Algoritmo 3) são construídos caminhos para as formigas, no entanto, com o objetivo é encontrar alguma solução viável que minimize o tempo total gasto com o número de veículos

encontrado pela primeira colônia. A atualização global de feromônios é realizada apenas utilizando a melhor solução s^*. Terminada a execução deste segundo algoritmo, o ciclo é reiniciado e continua até que algum critério de término seja atendido. Por exemplo, este critério pode ser o número de iterações.

Algoritmo 3: ACS-TIME

1. **Procedimento** ACS-TIME(v)
2. **Inicializa** matriz de feromônios e estruturas de dados utilizando v
3. **Repita**
4. **Para cada** formiga k **faça**
5. // Constrói uma solução s para a formiga k
6. s_k = simulaFormiga(k, buscaLocal=TRUE, 0)
7. **Fim para cada**
8. **Se** s_k é factível **E** $Ls_k < Ls^*$ **então**
9. **Retorne** s_k para MACS-VRPTW
10. // Atualização global da matriz de feromônios utilizando a solução s^*
11. $\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \rho / Ls^* \qquad \forall (i,j) \in s^*$
12. **Até que** um número de iterações *nIter* seja atingido

3.2.6.3 Algoritmo SimulaFormiga

Os algoritmos ACS-VEI e ACS-TIME utilizam o mesmo procedimento construtivo denominado SimulaFormiga (Algoritmo 4). Este procedimento possui funcionamento semelhante ao ACS para o TSP. Entretanto, como o VRPTW envolve múltiplas rotas, e não mais um único caminho passando por todos os nós, é preciso que seja realizada uma adaptação para este problema. Isto é feito através da inclusão de um depósito virtual para cada veículo utilizado no problema, os quais são cópias idênticas do depósito real. Esta adaptação é equivalente a inclusão de m-1 depósitos artificiais, assim como sugerido por Christofides e Eilon (1969). Assim, o funcionamento do ACS no VRPTW pode ser equivalente ao TSP, resultando em um único caminho que visite o depósito várias vezes.

No algoritmo SimulaFormiga, cada formiga inicia seu caminho (ciclo de vida) a partir de um dos depósitos (original ou artificial) escolhido aleatoriamente. A cada passo, a formiga localizada na posição i deve escolher probabilisticamente seu próximo destino j a ser visitado. O conjunto de nós disponíveis para serem visitados será composto pelos nós ainda não visitados e que não violem as restrições de capacidade e janelas de tempo. A atratividade η_{ij} em cada arco é calculada em função da urgência do cliente ser atendido, da distância do arco ij e do número de vezes que o nó j não foi inserido em uma solução, representado pela variável IN_j. As variáveis IN são utilizadas apenas pelo algoritmo ACS-VEI.

Cada vez que uma formiga move de um nó para outro é realizada uma atualização local de feromônios. Caso a formiga não consiga visitar todos os nós, seu ciclo de vida é finalizado, pois não criou uma solução viável. No entanto, seu caminho ainda poderá ser aproveitado através da tentativa de inserção dos nós não visitados. O procedimento de inserção utilizado por Gambardella et al. (1999) é semelhante ao procedimento de Solomon (1987). No entanto, a solução é iniciada com o caminho da formiga e os nós não visitados são ordenados em ordem decrescente da demanda. Finalmente, se a solução é viável para o ACS-TIME, um procedimento de busca local (item 3.2.5) é aplicado para melhorar a solução.

Algoritmo 4: SimulaFormiga

1. **Procedimento** SimulaFormiga(k, buscaLocal, IN ou 0)
2. **Inicializa** o caminho da formiga k em um dos depósitos duplicados i
3. $s_k \leftarrow \{i\}$
4. $começo_k \leftarrow 0$
5. $carga_k \leftarrow 0$
6. // Inicia a construção do caminho s da formiga k
7. **Repita**
8. // Calcular a atratividade dos arcos adjacentes à posição atual da formiga
9. **Para cada** $j \in i$ **faça**
10. $começo_j \leftarrow max(começo_k + t_{ij}, b_j)$
11. $tempo\_delta_{ij} \leftarrow começo_j - começo_k$
12. $distância_{ij} \leftarrow tempo\_delta_{ij} * (a_j - começo_k)$
13. $distância_{ij} \leftarrow max(1.0, (distância_{ij} - IN_j))$
14. $\eta_{ij} \leftarrow 1.0 / distância_{ij}$
15. // Escolher probabilisticamente o próximo nó j utilizando η_{ij} na expressão 13
16. $s_k \leftarrow s_k + \{j\}$
17. $começo_k \leftarrow começo_j$
18. $carga_k \leftarrow carga_k + q_j$
19. **Se** j é um depósito **então**
20. $tempo\_atual_k \leftarrow 0$
21. $carga\_atual_k \leftarrow 0$
22. **Fim se**
23. // Atualização local de feromônios
24. $\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \rho\tau_0$
25. $i \leftarrow j$ // Novo nó para a formiga k
26. **Fim para**
27. **Até que** não existam mais nós factíveis
28. // Se existir nós não visitados, tenta inseri-los na solução s_k
29. $s_k \leftarrow insertion\_procedure(s_k)$
30. // Melhora a solução s_k através do algoritmo de busca local
31. **Se** *buscaLocal*=TRUE e s_k é factível **então**
32. $s_k \leftarrow procedimento\_busca\_local(s_k)$
33. **Fim se**
34. **Retorne** s_k

3.3 Método de Descida em Vizinhança Variável Aleatória

A busca local é um procedimento que consiste na alteração da solução atual através da troca sucessiva de nós ou arcos com objetivo de melhorar a solução. Para cada tipo de movimento é dado o nome de estrutura de vizinhança ou apenas vizinhança. Para problemas de roteirização de veículos existem as vizinhanças do tipo inter-rota, que realizam movimentos entre diferentes rotas e intra-rota, que realizam movimentos na mesma rota.

Utilizando-se das vizinhanças do tipo inter-rota e intra-rota, Mladenović e Hansen (1997) propuseram um algoritmo denominado Descida em Vizinhança Variável *(Variable Neighborhood Descent - VND)*, que consiste na aplicação sequencial de vizinhanças. Subramanian (2012) propôs uma versão do VND denominada Descida em Vizinhança Variável Aleatória *(Random Variable Neighborhood Descent - RVND).* Esta versão utiliza um número aleatório para determinar a ordem em que as vizinhanças serão executadas. Caso a vizinhança sorteada seja bem sucedida, buscas intra-rota são realizadas nas rotas modificadas, caso contrário, a vizinhança é removida da lista da busca entre rotas.

3.3.1 O algoritmo RVND

Seja $N = \{N^{(1)},N^{(2)},N^{(3)},...,N^{(z)}\}$ o conjunto de vizinhanças inter-rota. Sempre que uma vizinhança selecionada falha ao melhorar a solução atual, o algoritmo RVND escolhe aleatoriamente outra vizinhança do mesmo conjunto e continua a procura através do espaço de soluções.

No algoritmo RVND (Algoritmo 5), uma lista de vizinhanças (NL) contendo um número *z* de movimentos inter-rota é inicializada. No laço principal, uma vizinhança $N^{(n)} \in$ NL é escolhida por um número aleatório e o melhor movimento admissível é determinado. Caso haja alguma melhoria na solução, uma busca local intra-rota é realizada e uma NL é povoada com todas as vizinhanças. Caso contrário, $N^{(n)}$ é removida de NL. Um conjunto de Estruturas de Dados Auxiliares (ver item 3.3.2) é atualizado no início do processo e toda vez que uma busca em vizinhança for realizada.

Algoritmo 5: RVND

1. **Procedimento** RVND(*s*)
2. Atualiza *EDA*
3. Inicializa a Lista de Vizinhanças Inter-Rota *(NL)*
4. **Enquanto** *NL* não está vazia **faça**
5. Escolha aleatoriamente uma vizinhança $N^{(n)} \in NL$
6. Busque a melhor solução vizinha *s'* de $s \in N^{(n)}$
7. **Se** $f(s') < f(s)$ **então**
8. $s \leftarrow s'$
9. $s \leftarrow BuscaIntraRota(s)$
10. // Insere em NL todas as estruturas de vizinhança inter-rota
11. Atualiza *NL*
12. **Senão**
13. Remove $N^{(n)}$ de *NL*
14. **Fim se**
15. Atualiza *EDA*
16. **Fim enquanto**
17. **Retorne** *s*

O Algoritmo 6 descreve o procedimento da busca intra-rota. Seja N' um conjunto composto por *z'* estruturas de vizinhança intra-rota. Inicialmente, uma lista NL' é inicializada com todas as estruturas de vizinhança intra-rota. A seguir, enquanto NL' não estiver vazia, uma vizinhança $N'^{(n)}$ é selecionada aleatoriamente e a busca local é realizada exaustivamente até que não sejam encontradas melhorias.

Algoritmo 6: BuscaIntraRota

1. **Procedimento** BuscaIntraRota(*s*)
2. Inicializa a Lista de Vizinhanças Intra-Rota *(NL')*
3. **Enquanto** NL' não está vazia **faça**
4. Escolha aleatoriamente uma vizinhança $N'^{(n)} \in NL'$
5. Busque a melhor solução vizinha *s'* de $s \in N'^{(n)}$
6. **Se** $f(s') < f(s)$ **então**
7. $s \leftarrow s'$
8. **Senão**
9. Remove $N'^{(n)}$ de *NL'*
10. **Fim se**
11. **Fim enquanto**
12. **Retorne** *s*

3.3.2 Estruturas de Dados Auxiliares

A fim de reduzir o esforço computacional do processo de busca com uso das vizinhanças, algumas das estruturas de dados auxiliares (EDA) propostas por Subramanian (2012) foram utilizadas. O autor não abordou problemas de roteirização com janelas de tempo. Para cobrir esta folga, propôs-se a estrutura *TempoEntrega*. As estruturas de dados utilizadas são descritas a seguir.

- *SomaEntregas*[*r*] - Soma das demandas entregues para cada rota *r*. Por exemplo, se *SomaEntregas*[2] = 120, isto significa que a soma das demandas entregues na rota 2 é igual à 120.
- *EntregaMinima*[*r*] - demanda entregue mínima para cada rota *r*. Por exemplo, se *EntregaMinima*[2] = 10, isto significa que 10 é a menor demanda entregue ao longo de todos os clientes da rota 2.
- *EntregaMaxima*[*r*] - demanda máxima entregue na rota *r*.
- *EntregaAcumulada*[*i*][*r*] - entrega acumulada em cada cliente *i* da rota *r*. Por exemplo, se *EntregaAcumulada*[3][5] = 80, isto significa que a soma das demandas nos primeiros cinco clientes da rota 3 corresponde à 80.
- *TempoEntrega*[*v*][*i*] - horário de chegada do veículo *v* no cliente *i*. Por exemplo, se *TempoEntrega*[2][4] = 70, significa que o tempo de chegada no cliente 4 pelo veículo atribuído à rota 2 corresponde ao momento 70.
- *EstadoVizinhança*[*z*][*r*] - indica se a rota *r* foi modificada após a vizinhança *z* falhar na busca por movimentos de melhoria envolvendo a mesma rota. Por exemplo, se *EstadoVizinhança*[2][4] = 1, isto significa que a última vez que a vizinhança $N^{(2)}$ foi aplicada, nenhum movimento de melhoria envolvendo a rota 4 foi encontrado, mas esta rota foi modificada por outra estrutura de vizinhança. Se *EstadoVizinhança*[2][4] = 0, isto significa que a rota 4 não sofreu nenhuma alteração após a última vez que $N^{(2)}$ não obteve sucesso ao aplicar um movimento de melhoria usando outra vizinhança com esta rota.

3.3.3 Estruturas de Vizinhança Inter-rota

Foram empregadas sete estruturas de vizinhança envolvendo movimentos do tipo inter-rota. Cinco delas são baseadas no esquema de λ-intercâmbios *(λ-interchanges)* proposto por Osman (1993) que consiste na troca de λ clientes entre duas rotas, uma é baseada no operador *Cross-exchange* (Taillard et al., 1997) e uma é baseada no movimento *K-Shift* (Subramanian, 2012). Para limitar o número de possibilidades, considerou-se λ=2. Conforme descrito por Cordeau e Laporte (2005) e Subramanian (2012), estas trocas podem ser mais bem descritas utilizando pares ordenados (λ_1, λ_2), com $\lambda_1 \leq \lambda$ *e* $\lambda_2 \leq \lambda$, que representa uma

operação onde λ_1 clientes são transferidos da rota 1 para a rota 2 e λ_2 cliente são transferidos da rota 2 para a rota 1. As possíveis combinações para 2-intercâmbios são os movimentos: (1,0), (1,1), (2,0), (2,1) e (2,2). Seguindo a nomenclatura utilizada por Subramanian (2012), algoritmos dos movimentos (1,1), (2,1) e (2,2) foram chamados de *swap* (troca) e os movimentos (1,0) e (2,0) foram chamados de *shift* (realocação). A seguir são descritos os algoritmos e são ilustrados (Figura 3) cada um dos movimentos do tipo inter-rota.

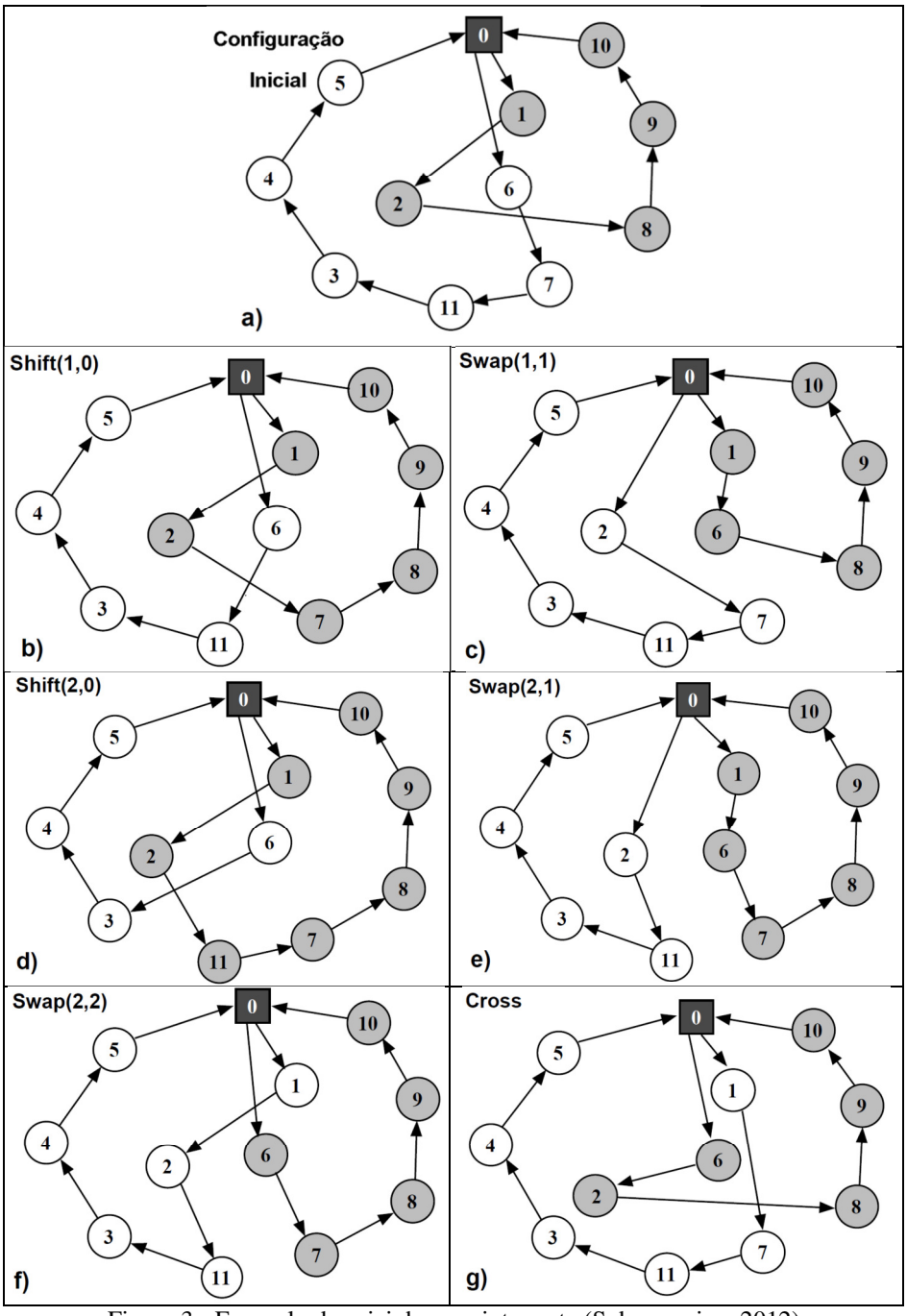

Figura 3 - Exemplo das vizinhanças inter-rota (Subramanian, 2012)

3.3.3.1 *Shift*(1,0)

A vizinhança *Shift*(1,0) consiste em remover um cliente de uma rota e inserir em outra. Na Figura 3.b pode-se observar que o cliente 7 foi inserido na outra rota após os clientes 2 e 8. A aplicação desta vizinhança é detalhada no Algoritmo 7.

Algoritmo 7: Shift(1,0)

```
1:  Procedimento Shift_1_0(s)
2:  Para r1 = 1...v faça
3:      Para r2 = 1...v faça
4:          Se r1 ≠ r2 e (EstadoVizinhança[1][r1] = 1 ou EstadoVizinhança[1][r2] = 1) e
              EntregaMinima[r1] + SomaEntregas[r2] ≤ K então
5:            Para cada cliente k ∈ r1 faça
6:              Se d_k + SomaEntregas[r2] ≤ K então
7:                Para z=1 até o tamanho de r2 faça
8:                  // Avaliar o custo L de uma solução s' vizinha de s, após transferir k∈ r1
                     para a posição z em r2
9:                  Se Ls' < Ls* E s' é factível então
10:                   s* ← s'
11: Se Ls* < Ls então  s ← s*
12: Senão
13:     Para r = 1...v faça
14:         EstadoVizinhança[1][r] = 0
15: Retorne s
```

3.3.3.2 *Shitf*(2,0)

A vizinhança *Shift*(2,0) consiste em remover dois clientes de uma rota e inserir em outra. Na Figura 3.d, os clientes 7 e 11 foram inseridos na outra rota entre os clientes 2 e 8. A aplicação desta vizinhança é detalhada no Algoritmo 8.

Algoritmo 8: Shift(2,0)

```
1:  Procedimento Shift_2_0(s)
2:  Para r1 = 1...v faça
3:      Para r2 = 1...v faça
4:          Se r1 ≠ r2 e (EstadoVizinhança[2][r1] = 1 ou EstadoVizinhança[2][r2] = 1) e
              EntregaMinima[r1]*2 + SomaEntregas[r2] ≤ K então
5:            Para cada par de clientes adjacentes k e l ∈ r1 faça
6:              Se d_k + d_l + SomaEntregas[r2] ≤ K então
7:                Para z=1 até o tamanho de r2 faça
8:                  // Avaliar o custo L de uma solução s' vizinha de s', após transferir
                     k e l ∈ r1 para a posição z em r2
9:                  Se Ls' < Ls* E s' é factível então
10:                   s* ← s'
11: Se Ls* < Ls então s ← s*
12: Senão
13:     Para r = 1...v faça
14:         EstadoVizinhança[2][r] = 0
15: Retorne s
```

3.3.3.3 *Swap*(1,1)

A vizinhança *Swap*(1,1) consiste em trocar um cliente de uma rota com um cliente de outra. Na Figura 3.c, o cliente 2 troca de lugar com o cliente 6. A aplicação desta vizinhança é detalhada no Algoritmo 9.

Algoritmo 9: Swap(1,1)

```
1:  Procedimento Swap_1_1(s)
2:  Para r1 = 1...v faça
3:      Para r2 = r1+ 1...v faça
4:          Se (EstadoVizinhança[3][r1] = 1 ou EstadoVizinhança[3][r2] = 1) e
              EntregaMinima[r1] – EntregaMaxima[r2] + SomaEntregas[r2] ≤ K então
5:              Para cada cliente k ∈ r1 faça
6:                  Se d_k + SomaEntregas[r2] + EntregaMaxima[r2] ≤ K então
7:                      Para cada cliente l ∈ r2 faça
8:                          // Avaliar o custo L de uma solução s' vizinha de s, após trocar k ∈ r1
                            com l ∈ r2
9:                          Se Ls' < Ls* E s' é factível então
10:                             s* ← s'
11: Se Ls* < Ls então
12:     s ← s*
13: Senão
14:     Para r = 1...v faça
15:         EstadoVizinhança[3][r] = 0
16: Retorne s
```

3.3.3.4 *Swap*(2,1)

A vizinhança *Swap*(2,1) consiste em trocar dois clientes adjacentes de uma rota com um cliente de outra. Na Figura 3.e os clientes 6 e 7 são trocados pelo cliente 2. A aplicação desta vizinhança é detalhada no Algoritmo 10.

Algoritmo 10: Swap(2,1)

```
1:  Procedimento Swap_2_1(s)
2:  Para r1 = 1...v faça
3:      Para r2 = 1...v faça
4:          Se r1 ≠ r2 e (EstadoVizinhança[4][r1] = 1 ou EstadoVizinhança[4][r2] = 1) e
              EntregaMinima[r1]*2 – EntregaMaxima[r2] + SomaEntregas[r2] ≤ K então
5:              Para cada par de clientes adjacentes k e l ∈ r1 faça
6:                  Se d_k + d_l + SomaEntregas[r2] – EntregaMaxima[r2] ≤ K então
7:                      Para cada cliente k' ∈ r2 faça
8:                          // Avaliar o custo L de uma solução s' vizinha de s, após trocar os
                            clientes adjacentes k e l ∈ r1 com k' ∈ r2
9:                          Se Ls' < Ls* E s' é factível então
10:                             s* ← s'
11: Se Ls* < Ls então
12:     s ← s*
13: Senão
14:     Para r = 1...v faça
15:         EstadoVizinhança[4][r] = 0
16: Retorne s
```

3.3.3.5 *Swap*(2,2)

A vizinhança *Swap*(2,2) consiste em trocar dois clientes adjacentes de uma rota com dois clientes adjacentes de outra. Na Figura 3.f os clientes 1 e 2 são trocados pelos clientes 6 e 7. Esta vizinhança é detalhada no Algoritmo 11.

Algoritmo 11: Swap(2,2)

```
1:  Procedimento Swap_2_2(s)
2:  Para r1 = 1...v faça
3:      Para r2 = r1 + 1...v faça
4:          Se r1 ≠ r2 e (EstadoVizinhança[5][r1] = 1 ou EstadoVizinhança[5][r2] = 1) e
              EntregaMinima[r1]*2 – EntregaMaxima[r2]*2 + SomaEntregas[r2] ≤ K então
5:            Para cada par de clientes adjacentes k e l ∈ r1 faça
6:              Se d_k + d_l + SomaEntregas[r2] – EntregaMaxima[r2]*2 ≤ K então
7:                Para cada par de clientes adjacentes k' e l' ∈ r2 faça
8:                  // Avaliar o custo L de uma solução s' vizinha de s, após trocar os
                     clientes adjacentes k e l ∈ r1 com k' e l' ∈ r2
9:                  Se Ls' < Ls* E s' é factível então
10:                   s* ← s'
11: Se Ls* < Ls então s ← s*
12: Senão
13:     Para r = 1...v faça  EstadoVizinhança[5][r] = 0
14: Retorne s
```

3.3.3.6 *Cross*

A vizinhança *Cross* consiste em remover o arco entre os clientes adjacentes c1 e c2, pertencentes a uma rota r1, e o arco entre clientes c3 e c4 de uma rota r2. Em seguida, um arco é inserido entre c1 e c4 e outro entre c3 e c2. Na Figura 3.g os arcos 1-2 e 6-7 são removidos e os arcos 6-2 e 1-7 são inseridos. A aplicação desta vizinhança é detalhada no Algoritmo 12.

Algoritmo 12: Cross

```
1:  Procedimento Cross(s)
2:  Para r1 = 1...v faça
3:      Para r2 = 1...v faça
4:          Se r1 ≠ r2 e (EstadoVizinhança[6][r1] = 1 ou EstadoVizinhança[6][r2] = 1) então
5:              Para i=1 até o tamanho de r1 faça
6:                  f ← último cliente da rota r2
7:                  Se DemandaAcumulada[r1][i] + d_f ≤ K então
8:                      Para j=1 até o tamanho de r2 faça
9:                          // Avaliar o custo L de uma solução s' vizinha de s, após remover o
                             arco (k,l) ∈ r1 (posições i e i+1) e o arco (k',l') ∈ r2, (posições j-1 e j)
                             e inserir os arcos (k,l') e (k',l)
10:                         Se Ls' < Ls* E s' é factível então
11:                             s* ← s'
12: Se Ls* < Ls então s ← s*
13: Senão
14:     Para r = 1...v faça  EstadoVizinhança[6][r] = 0
15: Retorne s
```

3.3.3.7 *K-shift*

A vizinhança *K-Shift* consiste em transferir um subconjunto consecutivo de clientes da rota r1 para a rota r2. Na Figura 4, os clientes consecutivos 1, 2 e 7 são transferidos da primeira rota para a segunda rota. A aplicação desta vizinhança é detalhada no Algoritmo 13.

Algoritmo 13: K-Shift

```
1:  Procedimento K-Shift(s)
2:  Para r1 = 1...v faça
3:      Para r2 = 1...v faça
4:          Se r1 ≠ r2 e (EstadoVizinhança[7][r1] = 1 ou EstadoVizinhança[7][r2] = 1) então
5:              Para todo cliente k ∈ r1 faça
6:                  Se d_k + SomaEntregas[r2] ≤ K então
7:                      SomaEntregasTemp ← d_k + SomaEntregas[r2]
8:                      z ←posição de k
9:                      l ← k
10:                     Enquanto SomaEntregasTemp ≤ K e z é uma posição válida de r1 faça
11:                         // Avaliar o custo L de uma solução s' vizinha de s, após transferir os
                               clientes a partir de k até l ∈ r1 para o fim da rota r2
12:                         Se Ls' < Ls* E s' é factível então
13:                             s* ← s'
14:                         Fim se
15:                         z ← z + 1
16:                         l ← cliente associado com a posição z ∈ r1
17:                         SomaEntregasTemp ← SomaEntregasTemp + d_l
18:                     Fim enquanto
19:                 Fim se
20: Se Ls* < Ls então
21:     s ← s*
22: Senão
23:     Para r = 1...v faça
24:         EstadoVizinhança[7][r] = 0
25: Retorne s
```

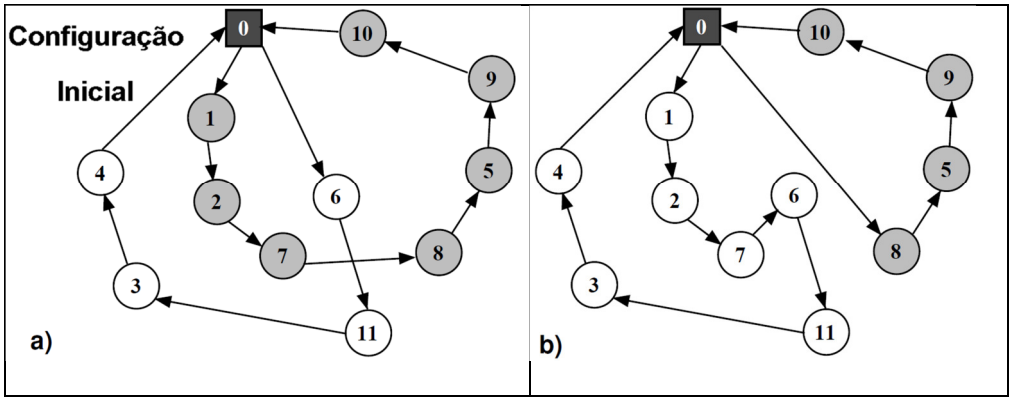

Figura 4 - Exemplo de vizinhança inter-rota K-shift (Subramanian, 2012)

3.3.4 Estruturas de Vizinhança Intra-Rota

Foram empregadas cinco estruturas de vizinhança envolvendo movimentos do tipo intra-rota. Três são baseadas no esquema de Or-Opt proposto por Or (1976), que consiste no reposicionamento de δ clientes dentro de uma mesma rota. Foram aplicados os movimentos com δ=1, 2 e 3. Uma é baseada no movimento do tipo *k*-opt, com k=2, denominado 2-opt (Croes, 1958 e Lin, 1965). Uma aplicação mais sofisticada do movimento do tipo *k*-opt pode ser encontrada em Helsgaun (2009), na qual define a cada iteração o valor de *k* que resultará na melhor melhoria da solução. Outra é baseada no movimento do tipo *Exchange*, proposto por Nagy e Salhi (2005). Estas vizinhanças intra-rotas são aplicadas nas rotas que sofreram modificações pelas vizinhanças inter-rotas. A seguir são descritos os algoritmos e são ilustrados (Figura 5) cada um dos movimentos do tipo intra-rota.

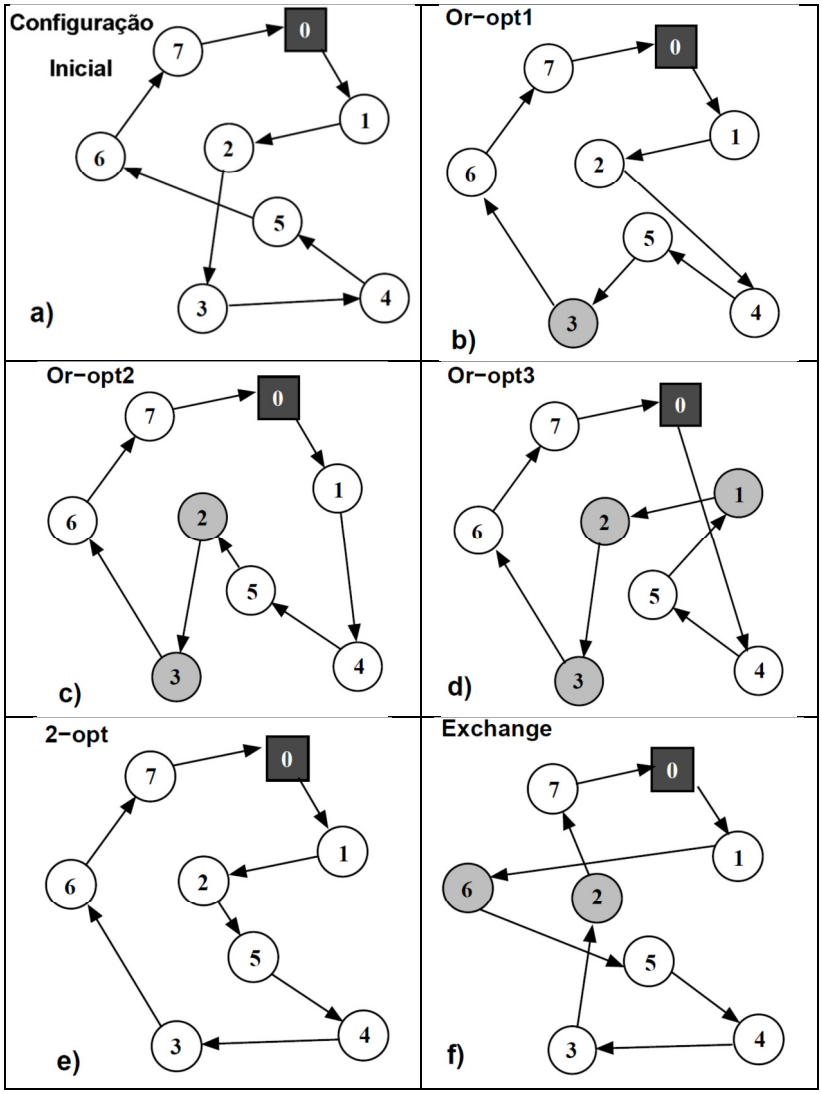

Figura 5 - Exemplo das vizinhanças intra-rota (Subramanian, 2012)

3.3.4.1 Or-Opt1

A vizinhança Or-Opt1 consiste em remover um cliente e posteriormente inseri-lo em outra posição, conforme ilustra a Figura 5.b. A aplicação desta vizinhança é detalhada no Algoritmo 14.

Algoritmo 14: Or-Opt1

```
1: Procedimento Or-Opt1(s)
2: Para r1 = 1...v faça
3:     Para i=1 até o tamanho de r1 faça
4:         Para j=i+1 até o tamanho de r1 faça
5:             // Avaliar o custo L de uma solução s' vizinha de s, após remover o nó da posição i
               e inseri-lo na posição j
6:             Se Ls' < Ls* E s' é factível então
7:                 s* ← s'
8: Retorne s
```

3.3.4.2 Or-Opt2

A vizinhança Or-Opt2 consiste em remover dois clientes e posteriormente inseri-los em outra posição, conforme ilustra a Figura 5.c. A aplicação desta vizinhança é detalhada no Algoritmo 15.

Algoritmo 15: Or-Opt2

```
1: Procedimento Or-Opt2(s)
2: Para r1 = 1...v faça
3:     Para i=1 até o tamanho de r1 faça
4:         Para j=i+2 até o tamanho de r1 faça
5:             // Avaliar o custo L de uma solução s' vizinha de s, após remover os nós das
               posições i e i+1 e inseri-los nas posições j e j+1
6:             Se Ls' < Ls* E s' é factível então
7:                 s* ← s'
8: Retorne s
```

3.3.4.3 Or-Opt3

A vizinhança Or-Opt3 consiste em remover três clientes e posteriormente inseri-los em outra posição, conforme ilustra a Figura 5.c. A aplicação desta vizinhança é detalhada no Algoritmo 16.

Algoritmo 16: Or-Opt3

```
1: Procedimento Or-Opt3(s)
2: Para r1 = 1...v faça
3:     Para i=1 até o tamanho de r1 faça
4:         Para j=i+3 até o tamanho de r1 faça
5:             // Avaliar o custo L de uma solução s' vizinha s, após remover os nós das posições
               i, i+1 e i+2 e inseri-los nas posições j, j+1 e j+2
6:             Se Ls' < Ls* E s' é factível então
7:                 s* ← s'
8: Retorne s
```

3.3.4.4 2-opt

A vizinhança 2-opt consiste em remover um par de arcos e posteriormente inserir outro par de arcos. Na Figura 5.e, são removidos os arcos 2-3 e 5-6 e inseridos os arcos 2-5 e 3-6. A aplicação desta vizinhança é detalhada no Algoritmo 17.

Algoritmo 17: 2-opt

```
1: Procedimento 2-opt(s)
2: Para r1 = 1...v faça
3:     Para i=1 até o tamanho de r1 faça
4:         Para j=tamanho de r1 até 1 faça
5:             // Avaliar o custo L de uma solução s' vizinha de s, após remover os arcos (i,i+1) e
               (j,j-1) e inserir os arcos (i,j-1) e (i+1,j)
6:             Se Ls' < Ls* E s' é factível então
7:                 s* ← s'
8: Retorne s
```

3.3.4.5 Exchange

A vizinhança *Exchange* consiste na permutação entre dois clientes. Na Figura 5.f são trocados de posição os clientes 2 e 6. A aplicação desta vizinhança é detalhada no Algoritmo 18.

Algoritmo 18: Exchange

```
1: Procedimento Exchange(s)
2: Para r1 = 1...v faça
3:     Para i=1 até o tamanho de r1 faça
4:         Para j=i+2 até o tamanho de r1 faça
5:             // Avaliar o custo L de uma solução s' vizinha de s, após inverter uma
               sequencia de clientes que inicia na posição i e finaliza na posição j
6:             Se Ls' < Ls* E s' é factível então
7:                 s* ← s'
8: Retorne s
```

3.3.5 Limite inferior para o número de veículos

Em algoritmo de otimização frequentemente são empregados limites inferior e superior (*lower e upper bounds*) para os problemas estudados. Em problemas que possuem uma função objetivo de minimização, utiliza-se o limite inferior para se conhecer a proximidade da solução atual com a solução ótima. Já para uma função objetivo de maximização é utilizado o limite superior.

No caso do VRPTW, que possui uma função objetivo de minimização, podem ser obtidos limites inferiores para a distância total e para o número de veículos utilizados. O cálculo destes limites é realizado com base nas restrições do

problema. O limite inferior mais simples de ser calculado e muito útil para o VRPTW é o limite inferior para o número de veículos ($lb_{veículos}$). Ele é calculado através da seguinte expressão.

$$lb_{veiculos} = arredondarParaCima\left(\sum_{i=1}^{N} d_i / K\right) \quad (25)$$

Para algoritmos que buscam minimizar o número de veículos, assim como o ACS-VEI (Algoritmo 2), o $lb_{veículos}$ pode ser adicionado como critério de parada, para reduzir o tempo de execução. Quando o número de veículos da solução atual for igual à $lb_{veículos}$, então o algoritmo não precisa mais reduzir o número de veículos, pois já atingiu o mínimo possível para o problema.

3.4 O algoritmo MACS-RVND

Baseando-se nos algoritmos MACS-VRPTW e RVND, propôs-se uma versão melhorada do MACS-VRPTW foi denominada de MACS-RVND. As modificações propostas são detalhadas abaixo.

- A solução inicial utilizada nos algoritmos MACS-VRPTW (Algoritmo 1) e ACS-VEI (Algoritmo 2) passa a ser a melhor solução entre as soluções obtidas através dos algoritmos do Vizinho Mais Próximo (item 3.2.3) e de Inserção PFIH+VND (3.2.4).
- Uso do algoritmo de inserção PFIH+VND (item 3.2.4) no algoritmo SimulaFormiga (Algoritmo 4).
- Aplicação do algoritmo RVND (Algoritmo 5) com diversas vizinhanças na fase busca local do algoritmo SimulaFormiga (Algoritmo 4).
- Criação da estrutura de dados auxiliar *TempoEntrega*, que auxilia na redução do tempo de execução ao verificar as restrições de janelas de tempo.
- Uso do limite inferior de veículos (item 3.3.5) no algoritmo ACS-VEI (Algoritmo 2).

3.5 Resultados Computacionais

O objetivo desta seção é mostrar o desempenho do algoritmo proposto para o VRPTW quanto à qualidade da solução. Um conjunto de problemas foi usado para testar o modelo. O algoritmo foi implementado utilizando a linguagem de programação Java dentro do paradigma de orientação a objetos e foi criado um *framework* para solução de problemas de roteirização, com aproximadamente 13.000 linhas de código. O algoritmo foi executado num servidor PowerEdge T110 II com processador Intel Xeon Quad-Core de 3,4 GHz e 8GB de memória RAM, sob plataforma Linux, com a distribuição do CentOS 6.3 64 bits.

Os problemas adotados para a validação dos algoritmos são as 56 instâncias apresentadas por Solomon (1987). Essas instâncias se dividem em três grupos: um grupo C, onde os clientes se encontram clusterizados, ou seja, estão distribuídos geograficamente em grupos de clientes próximos uns dos outros; um grupo R, onde os clientes são distribuídos aleatoriamente sem formar grupos e distantes uns dos outros e um grupo RC, onde se tem uma mistura dos dois grupos anteriores. Cada grupo possui dois subgrupos. No subgrupo 1, o horizonte de planejamento possui curta duração e no subgrupo 2 possui longa duração. Todas as instâncias possuem 100 clientes. Os valores das melhores soluções foram obtidos no site www.sintef.no (2012).

Os parâmetros utilizados em cada um dos algoritmos que compõem o algoritmo MACS-RVND foram calibrados através da execução do algoritmo com diversos valores utilizando parte ou todas as instâncias de Solomon (1987).

Para os algoritmos do Vizinho Mais Próximo e de PFIH+VND, a cada um dos três parâmetros foi atribuído valores de 0 a 1 com incremento de 0,001, onde a soma deles é igual a um. Para cada uma das 56 instâncias foi obtida a combinação destes parâmetros que resultou na melhor solução.

Para os algoritmos ACS-TIME, ACS-VEI e SimulaFormiga foram analisados os parâmetros ρ (fator de evaporação), *beta* (peso atribuído ao inverso do comprimento do arco), *q0* (fator de exploração), *m* (número de formigas por iteração) e *nIter* (número de iterações), *maxIter* (número máximo de iterações sem que ocorram melhorias sem melhorias na solução atual). Para estes algoritmos, realizou-se a fixação de um parâmetro e a variação com incremento de 0,1 para no

parâmetro avaliado. Utilizou-se uma instância de cada grupo e subgrupo, totalizando 6 instâncias. Os valores utilizados para execução do MACS-RVND, e desvios padrões são apresentados na Tabela 2.

Tabela 2 – Calibração dos parâmetros utilizados pelos algoritmos

Algoritmo		Valor Utilizado
Vizinho Mais Próximo	δ_1 – Peso distância (tempo)	0,761
	δ_2 – Peso espera	0,179
	δ_3 – Peso urgência	0,059
PFIH+VND	ω_1 – Peso distância (tempo)	0,512
	ω_2 – Peso urgência (fim janela)	0,284
	ω_3 – Peso coordenada polar	0,204
MACS-RVND	ρ – fator de evaporação	0,1
	beta – fator de exploração	1
	q0 – fator de exploração	0,9
	m – número de formigas	10
	nIter – número de iterações	10
	maxIter – máximo de iterações	20

A Tabela 3 apresenta as melhores soluções reportadas na literatura e as melhores soluções e médias obtidas pelo algoritmo MACS-RVND. A primeira coluna é o nome da instância. As colunas NV representam o número total de veículos necessário. As colunas DT representam a distância total das rotas. A coluna CPU indica a soma do tempo total de cinco execuções do algoritmo para obtenção da melhor solução para cada instância. A última coluna indica o autor que reportou a melhor solução conhecida na atualidade.

Como podem ser observadas na Tabela 3, as melhores soluções obtidas pelo MACS-RVND possuem valores bem próximos dos melhores resultados da literatura e alguns casos é a melhor solução conhecida. Das 56 instâncias, para 29 delas foram encontradas as melhores soluções conhecidas. Para 23 instâncias foram encontradas soluções com o menor número de veículos conhecido e com GAP de distância total de no máximo 0,62%. Para 4 instâncias foram encontradas soluções com 1 veículo a mais, porém com distância total menor que nas melhores soluções conhecidas, com uma diferença média de -5,21%.

Tendo em vista que as melhores soluções da literatura foram obtidas por diferentes algoritmos, cada qual explorando as especificidades de uma dada instância ou conjunto de instâncias, o algoritmo desenvolvido mostrou-se capaz de produzir soluções finais bastante satisfatórias.

Tabela 3 - Melhores resultados conhecidos e resultados obtidos pelo MACS-VRPTW

	Melhor conhecido		Melhor MACS-RVND		Média MACS-RVND		CPU (s)	Autores
Nome	NV	DT	NV	DT	NV	DT		
C101	10	828,94	10	828,94	10	828,94	23	Rochat e Taillard, 1995
C102	10	828,94	10	828,94	10	828,94	33	Rochat e Taillard, 1995
C103	10	828,06	10	828,06	10	828,06	68	Rochat e Taillard, 1995
C104	10	824,78	10	824,78	10	825,48	83	Rochat e Taillard, 1995
C105	10	828,94	10	828,94	10	828,94	24	Rochat e Taillard, 1995
C106	10	828,94	10	828,94	10	828,94	26	Rochat e Taillard, 1995
C107	10	828,94	10	828,94	10	828,94	24	Rochat e Taillard, 1995
C108	10	828,94	10	828,94	10	828,94	28	Rochat e Taillard, 1995
C109	10	828,94	10	828,94	10	828,94	49	Rochat e Taillard, 1995
C201	3	591,56	3	591,56	3	591,56	57	Rochat e Taillard, 1995
C202	3	591,56	3	591,56	3	591,56	80	Rochat e Taillard, 1995
C203	3	591,17	3	591,17	3	600,21	76	Rochat e Taillard, 1995
C204	3	590,60	3	590,60	3	590,60	126	Rochat e Taillard, 1995
C205	3	588,88	3	588,88	3	588,88	66	Rochat e Taillard, 1995
C206	3	588,49	3	588,49	3	588,49	60	Rochat e Taillard, 1995
C207	3	588,29	3	588,29	3	588,29	66	Rochat e Taillard, 1995
C208	3	588,32	3	588,32	3	588,32	76	Rochat e Taillard, 1995
R101	19	1.645,79	19	1.650,80	19	1.650,80	102	Homberger, 2000
R102	17	1.486,12	17	1.486,12	17	1.486,35	443	Rochat e Taillard, 1995
R103	13	1.292,68	13	1.292,68	13,8	1.232,93	126	Li et al., 2003
R104	9	1.007,24	10	981,23	10	1.014,86	29	Mester et al., 2005
R105	14	1.377,11	14	1.377,11	14	1.379,15	215	Rochat e Taillard, 1995
R106	12	1.251,98	12	1.252,03	12	1.259,33	402	Mester et al., 2005
R107	10	1.104,66	10	1.114,29	10,4	1.105,47	354	Shaw, 1997
R108	9	960,88	9	965,18	9,8	991,53	35	Berger et al., 2001
R109	11	1.194,73	11	1.197,42	11,8	1.172,10	63	Gehring e Homberger, 1999
R110	10	1.118,59	10	1.172,58	10,8	1.099,03	395	Mester et al., 2005
R111	10	1.096,72	10	1.105,46	10,2	1.113,05	485	Rousseau et al., 2002
R112	9	982,14	10	953,63	10	977,20	51	Gambardella et al., 1999
R201	4	1.252,37	4	1.252,37	4	1.252,37	23	Gehring e Homberger, 1999
R202	3	1.191,70	3	1.191,80	3,6	1.129,96	289	Rousseau et al., 2002
R203	3	939,50	3	945,13	3	970,51	50	Woch e Lebkowski, 2009
R204	2	825,52	2	830,00	2,6	803,73	104	Bent e Van Hentenryck, 2001
R205	3	994,42	3	994,43	3	1.033,61	126	Rousseau et al., 2002
R206	3	906,14	3	919,14	3	921,67	896	Schrimpf et al., 2000
R207	2	890,61	2	898,35	2,6	877,16	136	Pisinger e Ropke, 2007
R208	2	726,75	2	726,82	2	727,52	393	Mester et al., 2005
R209	3	909,16	3	909,33	3	912,59	424	Homberger, 2000
R210	3	939,34	3	948,52	3	954,11	777	Mester et al., 2005
R211	2	885,71	3	779,81	3	809,80	127	Woch e Lebkowski, 2009
RC101	14	1.696,94	14	1.696,95	14,8	1.644,73	375	Taillard et al., 1997
RC102	12	1.554,75	12	1.558,07	12,8	1.495,74	246	Taillard et al., 1997
RC103	11	1.261,67	11	1.262,02	11	1.307,65	64	Shaw, 1998
RC104	10	1.135,48	10	1.135,83	10	1.187,45	31	Cordeau et al., 2000
RC105	13	1.629,44	13	1.649,12	13,6	1.599,73	243	Berger et al., 2001
RC106	11	1.424,73	12	1.376,26	12,2	1.398,72	64	Berger et al., 2001
RC107	11	1.230,48	11	1.230,54	11	1.247,44	55	Shaw, 1997
RC108	10	1.139,82	10	1.139,82	10,6	1.149,93	36	Taillard et al., 1997
RC201	4	1.406,91	4	1.406,94	4	1.414,41	143	Mester et al., 2005
RC202	3	1.365,65	3	1.365,65	3,8	1.202,90	350	Gambardella et al., 1999
RC203	3	1.049,62	3	1.052,35	3	1.060,37	198	Czech e Czarnas, 2002
RC204	3	798,41	3	798,56	3	801,79	521	Mester et al., 2005
RC205	4	1.297,19	4	1.297,65	4	1.302,46	385	Mester et al., 2005
RC206	3	1.146,32	3	1.146,32	3	1.211,21	203	Homberger, 2000
RC207	3	1.061,14	3	1.061,14	3	1.113,03	163	Bent e Van Hentenryck, 2001
RC208	3	828,14	3	832,62	3	834,35	541	Ibaraki et al., 2005
Total	405	57.180,84	409	57.128,33	418,4	57.030,7	1.062	

NV = número de veículos utilizados, DT = distância total das rotas

Notas: Configuração do computador utilizado; número de execuções; tempo total de execução em minutos: (1) Pentium 200 MHz, 1 execução, 198 min.; (2) Pentium III 1 GHz, 1 execução, 559 min.; (3) 4×Pentium 400 MHz, 5 execuções, 1.458 min.; (4) AMD 700 MHz, 3 execuções, 106 min.; (5) Pentium 200 MHz, 10 execuções, 312 min.; (6) 5 P500 MHz, 1 execução, 12 min; (7) Pentium 400 MHz,5 execuções, 473 min.; (8) **Intel Xeon Quad-Core de 3,4 GHz, 5 execuções, 177 min.**; (9) Sun Ultra 10, 5 execuções, 1.095 min.; (10) Pentium III 545 MHz, 3 execuções, 594 min.; (11) Pentium 400 MHz, 1 execução, 162 min.; (12) Sun Ultra 10, 10 execuções, 13.381 min.; (13) HP 9000/720, 3 execuções, 102 min.; (14) 4×Pentium 200 MHz, 1 execução, 48 min.; (15) Sun Sparc 10, 1 execução, 248 min.; (16) Sun Ultrasparc 1, 1 execução, 210 min.; (17) Intel Pentium 4 CPU, 2.79 GHz, 30 execuções, 1.477 min.; (18) DEC Alpha, 3 execuções, 362 min.; (19) Motorola PowerPC 604, 5 execuções, 270 min.; (20) Pentium 200, 4 execuções, 373 min.; (21) Silicon Graphics 100 MHz, 1 execução, 138 min.; (22) Sun Sparc 10, 5 execuções, 6 min.; (23) Sun Sparc 10, 1 execução, 9,8 min.

Na Tabela 4, são apresentados os valores médios por classe de instância e a soma total de veículos e tempo total das rotas de todas as instâncias, que foram reportados por diversos autores. Assim como em Bräysy e Gendreau (2005), foram utilizados os resultados obtidos com esforço computacional limitado, não ultrapassando um dia de execução, que foi o máximo encontrado entre os trabalhos nesta literatura. A primeira coluna mostra a posição do algoritmo, seguindo uma ordenação de acordo com uma função objetivo hierárquica. O objetivo primário é minimizar o número total de veículos e objetivo secundário é minimizar a o tempo total de viagem. A segunda coluna mostra as referências dos trabalhos que reportaram estes resultados. As colunas R1, R2, C1, C2, RC1 e RC2 mostram o número médio de veículos e o tempo total médio de viagem para os respectivos seus grupos de instâncias. A coluna SNV/SDT indica o somatório de número de veículos (SNV) e o somatório do tempo de viagem (SDT) de todas as 56 instâncias.

Conforme os resultados apresentados na Tabela 4, observa-se que o método proposto é competitivo com os métodos propostos por diversos autores, incluindo vários autores que encontraram algumas das melhores soluções conhecidas para o VRPTW. Considerando os resultados de 22 trabalhos apresentados, o algoritmo proposto ficou em oitavo lugar. Nós concluímos que o uso do método RVND combinado com o MACS-VRPTW pode melhorar consideravelmente o método original proposto por Gambardella et al. (1999), que está na décima sexta posição.

Tabela 4 – Comparação com resultados de diversos autores com esforço computacional limitado

#	Autores	R1	R2	C1	C2	RC1	RC2	SNV SDT
0	Melhor conhecido	11,92	2,73	10,00	3,00	11,50	3,25	405
	(diversos autores)	1.209,89	951,02	828,38	589,86	1.384,16	1.119,17	57.181
1	Bräysy, 2003	11,92	2,73	10,00	3,00	11,50	3,25	405
		1.222,12	975,12	828,38	589,86	1.389,58	1.128,38	57.710
2	Ibaraki et al., 2002	11,92	2,73	10,00	3,00	11,63	3,25	406
		1.220,02	961,64	828,38	589,86	1.378,72	1.132,17	57.480
3	Gehring e Homberger,	12,00	2,73	10,00	3,00	11,50	3,25	406
	2001	1.217,57	961,29	828,63	590,33	1.395,13	1.139,37	57.641
4	Bräysy et al., 2004a	12,00	2,73	10,00	3,00	11,50	3,25	406
		1.220,20	970,38	828,38	589,86	1.398,76	1.139,37	57.796
5	Homberger e Gehring,	11,92	2,73	10,00	3,00	11,63	3,25	406
	1999	1.228,06	969,95	828,38	589,86	1.392,57	1.144,43	57.876
6	Le Bouthillier e Crainic,	12,08	2,73	10,00	3,00	11,50	3,25	407
	2005	1.209,19	963,62	828,38	589,86	1.389,22	1.143,70	57.412
7	Homberger e Gehring,	12,08	2,82	10,00	3,00	11,50	3,25	408
	2005	1.211,67	950,72	828,45	589,96	1.395,93	1.135,09	57.422
8	**MACS-RVND**	**12,08**	**2,82**	**10,00**	**3,00**	**11,63**	**3,25**	**409**
	proposto	**1.212,38**	**945,06**	**828,38**	**589,86**	**1.381,08**	**1.120,15**	**57.128**
9	Bent e Van Hentenryck,	12,17	2,73	10,00	3,00	11,63	3,25	409
	2004	1.203,84	980,31	828,38	589,86	1.379,03	1.158,91	57.707
10	Li et al., 2003	12,08	2,91	10,00	3,00	11,75	3,25	411
		1.215,14	953,43	828,38	589,86	1.385,47	1.142,48	57.467
11	Berger et al., 2003	12,17	2,73	10,00	3,00	11,88	3,25	411
		1.251,40	1.056,59	828,50	590,06	1.414,86	1.258,15	60.200
12	Rousseau et al., 2002	12,08	3,00	10,00	3,00	11,63	3,38	412
		1.210,21	941,08	828,38	589,86	1.382,78	1.105,22	56.953
13	Liu e Shen, 1999	12,17	2,82	10,00	3,00	11,88	3,25	412
		1.249,57	1.016,58	830,06	591,03	1.412,87	1.204,87	59.318
14	Gehring e Homberger,	12,42	2,82	10,00	3,00	11,88	3,25	415
	1999	1.198,00	947,00	829,00	590,00	1.356,00	1.144,00	56.946
15	Taillard et al., 1997	12,33	3,00	10,00	3,00	11,90	3,38	417
		1.220,35	1.013,35	828,45	590,91	1.381,31	1.198,63	58.614
16	Gambardella et al., 1999	12,38	3,00	10,00	3,00	11,92	3,33	418
		1.210,83	960,31	828,38	591,85	1.388,13	1.149,28	57.583
17	Gong et al., 2012	12,58	3,00	10,00	3,00	12,13	3,38	422
		1.232,28	1.016,66	835,91	593,41	1.385,44	1.169,07	58.677
18	Kilby et al., 1999	12,67	3,00	10,00	3,00	12,13	3,38	423
		1.200,33	966,56	830,75	592,24	1.388,15	1.133,42	57.423
19	Schulze e Fahle, 1999	12,50	3,09	10,00	3,00	12,25	3,38	423
		1.268,42	1.055,90	828,94	589,93	1.396,07	1.308,31	60.651
20	Brandão, 1999	12,58	3,18	10,00	3,00	12,13	3,50	425
		1.205,00	995,00	829,00	591,00	1.371,00	1.250,00	58.562
21	Rochat e Taillard, 1995	12,58	3,09	10,00	3,00	12,38	3,62	427
		1.197,42	954,36	828,45	590,32	1.369,48	1.139,79	57.120
22	Kontoravdis e Bard, 1995	12,58	3,09	10,00	3,00	12,63	3,50	427
		1.325,44	1.164,27	827,30	589,65	1.500,94	1.414,21	64.196
23	Potvin e S., 1996	12,58	3,09	10,00	3,00	12,63	3,38	427
		1.294,70	1.185,90	861,00	602,50	1.465,00	1.476,10	64.679

3.6 Considerações Finais

Neste capítulo foi apresentado uma heurística híbrida baseada na meta-heurística Otimização por Colônia de Formigas, mais especificamente o Sistema de Múltiplas Colônias de Formigas (MACS) com o método de Descida em Vizinhança Variável Aleatória (RVND) para resolver o Problema de Roteirização de Veículos (VRPTW) estático. O algoritmo foi apresentado em detalhes para permitir o entendimento profundo do funcionamento do algoritmo. Através da aplicação do algoritmo a 56 instâncias usadas como *benchmarking* na literatura é possível concluir que o modelo foi capaz de encontrar boas ou mesmo as melhores soluções conhecidas com o número mínimo de veículos. Mostrou-se que a utilização do algoritmo RVND melhorou consideravelmente o MACS-VRPTW.