6
O *Multiple Ant Colony System* (MACS)

O MACS-VRPTW é a extensão do ACS para o problema de roteirização de veículos com janelas de tempo. Como vimos na seção 3.2, na descrição do problema, o objetivo do problema consiste em minimizar tanto a quantidade de veículos utilizada quanto a distância total percorrida por eles (custo total). Estes objetivos possuem uma hierarquia, sendo o primeiro mais importante que o segundo. Esta hierarquia é comumente empregada na literatura, e será aqui também utilizada.

A idéia central do modelo proposto reside na adoção de duas colônias de formigas, cada uma visando a otimização de um dos objetivos do problema. As melhores soluções são compartilhadas entre as duas colônias, mas as informações de ferormônios são independentes. Dessa forma, ambas as colônias trabalham de forma distinta, porém simultânea, na procura por soluções mais interessantes, tendo os resultados de uma colônia grande influência nos caminhos tomados pela outra. Esta é a essência do MACS-VRPTW.

Trazendo estas palavras para uma realidade computacional, o algoritmo deveria idealmente ser capaz de trabalhar com computação paralela, ou seja, ambas as colônias estariam ativas e procurando por melhores soluções ao mesmo tempo. Esta é a idéia proposta por Gambardella *et al* em [4]. Na prática, porém, visando facilitar a implementação do algoritmo, pode-se construir um algoritmo que trabalhe com ciclos, nos quais as colônias estão ativas alternadamente, e a solução encontrada por uma é utilizada em seguida pela outra, e vice-versa. Veremos mais adiante que esta será a forma de implementação utilizada nas simulações.

Cada uma das duas colônias será então simulada separadamente. Basicamente, cada colônia será simulada por uma função, que deverá construir caminhos e analisar a qualidade destes caminhos. A diferença nos objetivos das duas colônias se refletirá principalmente na forma com que esta análise dos caminhos será feita. Entretanto, a construção dos caminhos em ambas as colônias será feito de forma muito parecida, pois a regra de decisão de cada formiga é sempre a mesma, independentemente de a qual colônia ela pertence. Sendo assim,

ambas as colônias usarão uma mesma função auxiliar para a construção dos caminhos percorridos. Este processamento dos caminhos percorridos por cada formiga por esta função auxiliar será feito de forma equivalente ao funcionamento do ACS para o TSP, segundo o qual uma matriz de distâncias/custos e uma de ferormônios são analisadas na procura pelo arco mais atrativo. Entretanto, como o VRP envolve rotas múltiplas, e não mais um único caminho passando por todos os nós, é preciso que o processamento das matrizes seja adaptado. Isto é feito através da inclusão de depósitos virtuais nestas matrizes. Estes depósitos são verdadeiras cópias do depósito real, nas mesmas coordenadas cartesianas, e as matrizes deverão possuir tantos depósitos quantos veículos forem utilizados. Por exemplo, um problema de 10 nós, que será resolvido com 3 veículos, precisará de matrizes não mais de 10x10, mas de 12x12, onde os nós 11 e 12 serão cópias do depósito (nó 1). Dessa forma, o funcionamento do ACS no VRP pode ser equivalente àquele no TSP, resultando em um caminho que incluirá, em sua seqüência proposta de nós a serem percorridos, cópias virtuais do depósito, que representarão retornos ao depósito. Quando um veículo retorna ao depósito, a capacidade e o tempo decorrido são zerados, possibilitando que outra rota seja iniciada.

O modelo precisa ser inicializado com alguma solução inicial. Isto normalmente é feito com um método simples, como o *Nearest Neighbor*. Conforme vimos anteriormente, este método possui algumas peculiaridades para que possa ser aplicado ao VRPTW, mas mantém sua simplicidade de entendimento. Tendo em mãos esta solução inicial, o MACS-VRPTW entra na sua fase cíclica, na qual as duas colônias de formigas entrarão em funcionamento. A solução inicial, primeiramente, passa pelo chamado ACS_VEI, que é um algoritmo que visa encontrar alguma solução factível que utilize um veículo a menos que na solução inicial. Este algoritmo usa uma primeira colônia de formigas para tanto. Feito isso, a solução encontrada passa para a segunda colônia de formigas, materializada sob a forma do algoritmo ACS_TIME, que tem como objetivo minimizar o tempo total gasto pelo dado número de veículos encontrado pela primeira colônia. Cabe lembrar aqui que minimizar o tempo total equivale a minimizar a distância total percorrida, sendo a diferença de utilização uma mera formalidade para manter a coerência das unidades do problema. Terminada a execução deste segundo algoritmo, o ciclo é reiniciado, com a primeira colônia

procurando uma nova solução com um veículo a menos que a melhor solução encontrada pela segunda colônia. Esse ciclo continua até que algum critério de término seja atendido. Por exemplo, este critério pode ser o número de iterações, ou o tempo de processamento. No nosso algoritmo, como veremos adiante, será usado um critério variável, no qual o algoritmo só se dá por terminado após um certo número de ciclos sem que haja nenhuma melhora na solução.

A idéia básica do MACS-VRPTW é a descrita acima. Sua implementação, porém, possui muitas peculiaridades que precisarão ser vistas com mais detalhes. Sendo assim, passamos à próxima seção, na qual estudaremos a fundo uma forma proposta de implementação do algoritmo.

6.1
Um Algoritmo para o MACS-VRPTW

Veremos agora o funcionamento do algoritmo aqui proposto para o problema de roteirização de veículos com janelas de tempo. A formalização do problema foi feita na seção 3.2. Todas as simulações foram elaboradas com o auxílio do software *MatLab 6.0 v12*. Primeiramente, é preciso definir de que forma serão os dados de entrada do algoritmo MACS-VRPTW, e qual será a forma de sua saída.

Como dados de entrada, precisamos ter as características físicas do problema, bem como os dados das janelas de tempo e capacidade de carga dos veículos. Sendo assim, usaremos como dados de entrada: uma matriz N x N contendo todos os custos C_{ij} entre todos os pares de nós do problema (matriz *dist)*; dois vetores 1 x N contendo os valores iniciais b_i e finais e_i das janelas de tempo de cada um dos N nós (vetores *h_inicial* e *h_final*); um vetor 1 x N com as demandas d_i de cada nó, lembrando que a demanda do nó 1, que convencionamos ser o depósito, deve ser igual a zero (vetor *demanda*); uma variável com o valor da capacidade máxima de carga dos veículos, considerados todos iguais (*cap*); finalmente, um vetor 1 x N contendo os tempos de serviço s_i de cada nó, incluindo do depósito (vetor *tempo_serv*). Estes dados são suficientes para identificar completamente o problema. A saída do algoritmo, por sua vez, será composta de três partes: o vetor *caminho*, contendo a seqüência de nós do melhor caminho encontrado; a variável *comp*, com o custo total de tal caminho (comprimento total

do caminho, que conforme mencionado acima, é equivalente ao tempo total que um veículo precisará rodar para percorrer o caminho encontrado) e a variável *carros*, com o número de veículos utilizados na solução encontrada.

Cabe ressaltar aqui que a variável *comp* representa o tempo total em trânsito, que é diretamente proporcional à distância total percorrida pelos veículos; se dobrarmos o valor de *comp*, ou seja, se dobrarmos o tempo de trânsito, também seria dobrada a distância total percorrida, dado que nossos veículos são considerados como tendo uma velocidade média constante. O valor de *comp* poderia indicar o custo com combustível, por exemplo, ou a quilometragem total dos carros. O problema também pode ser visto de outra forma: se somarmos ao valor de *comp* os tempos de serviço de todos os nós, teremos o tempo total necessário para que as rotas sejam percorridas. Isso pode indicar, por exemplo, o número de horas trabalhadas, que precisarão ser pagas aos motoristas e ajudantes. Dessa forma, a saída do algoritmo nos possibilita compreender o problema de diversas formas. Entretanto, o que se deseja minimizar juntamente com o número de veículos utilizados é o comprimento total das rotas, que corresponde ao somatório dos custos C_{ij} de cada aresta percorrida, desconsiderando os tempos de serviço de cada nó. A isso corresponde a variável *comp*.

O algoritmo criado para a rotina MACS-VRPTW segue abaixo. Primeiramente, com base nos seis dados físicos do problema, é chamada a função auxiliar *nearest_neighbor*, que determina um caminho factível através da heurística homônima, aplicada ao VRP (como descrita na seção 3.2.1). Sendo assim, as variáveis de saída *caminho, comp* e *carros* são inicializadas com os valores referentes ao caminho encontrado pelo *Nearest Neighbor*.

```
[caminho,comp,carros] = nearest_neighbor (dist,h_inicial,h_final,demanda,cap,tempo_serv);

flag = 1; %flag se inicia em 1; se houver melhora pelo acs_vei, flag e´ subtraido de 1; acs_time sempre
          %faz flag = flag+1, ou seja, o algoritmo continua ate que flag seja igual a 5, ou apos 4 passagens
          %pelo acs_vei sem melhorias

comp_nn = comp; %comp_nn possui o comprimento encontrado pelo nearest neighbor
while flag~=5
   [caminho,comp,carros,flag] = acs_vei (caminho,comp,carros,flag,comp_nn,dist,h_inicial,h_final,demanda,cap,
tempo_serv);
   [caminho,comp,carros,flag] =acs_time (caminho,comp,carros,flag,comp_nn,dist,h_inicial,h_final,demanda,cap,
tempo_serv);
end
```

Quadro 2 – Rotina principal do MACS-VRPTW

O uso da variável *flag* visa fornecer um critério de término para a rotina. Caso o ciclo do *acs_vei* e do *acs_time* seja realizado por um dado número de vezes sem que seja encontrada uma solução factível com um número de veículos inferior ao utilizado atualmente, a rotina é terminada, e o melhor caminho até o momento é fornecido como resposta, bem como seu comprimento e o número de carros utilizados.

O ciclo principal do algoritmo vem em seguida. Este ciclo, segundo Gambardella *et al* [4], deveria ser realizado através de computação paralela, na qual cada uma das duas funções principais (*acs_time* e *acs_vei*) teria seu processamento realizado de forma independente, e simultâneo ao processamento da outra função. Isso certamente reduzirá o tempo total do MACS-VRPTW, e representará um ganho de eficiência; porém, haverá um aumento de complexidade.

6.1.1
Implementação do *acs-vei*

Abaixo segue o algoritmo criado para a função *acs_vei*. Os dados de entrada são as mesmas informações físicas do problema requisitadas pela função *nearest_neighbor*, além do melhor caminho até o momento (vetor *caminho),* seu comprimento total (variável *comp*), o número de veículos que este caminho utiliza (variável *carros*), o *flag* de finalização de execução, e o comprimento do caminho encontrado pelo *Nearest Neighbor* (variável *comp_nn*, que será usada nas atualizações de ferormônios). Na primeira vez que o ciclo principal do MACS-VRPTW é executado, estes dados referentes ao melhor caminho encontrado até o momento dizem respeito ao caminho encontrado pelo *Nearest Neighbor*, fazendo com que o valor da variável *comp_nn* seja igual ao valor da variável *comp*. Conforme o ciclo é executado, entretanto, estas informações vão sendo atualizadas com as soluções encontradas.

```
function [caminho,comp,carros,flag] = acs_vei(caminho,comp,carros,flag,comp_nn,dist,h_inicial,h_final,
demanda,cap,tempo_serv)
%funçao que procura um caminho que utilize (carros-1) carros, e sinaliza se encontrou tal caminho atraves de um flag

%INICIALIZAÇAO
cont = 0;
j = 0;
while cont~=carros
   j = j+1;
```

```
   if caminho(j)==1
      cont = cont + 1;
   end
end
caminho_atual = caminho(1:j); %soluçao inicial com um veiculo a menos que a melhor soluçao encontrada ate o
momento
comp_atual = comp; %sera subtraido de comp as distancias referentes aos clientes nao visitados, para se ter
comp_atual
for z=j:(length(caminho)-1)
   comp_atual = comp_atual - dist(caminho(j),caminho(j+1));
end

rho = 0.1;
ferormonios = ones(length(demanda)+carros-2); %adiciona `a matriz (carros-2) depositos virtuais
dist_acs = dist; %dist_acs sera a matriz de distancias extendida, com os depositos virtuais
maximo = max(dist_acs(1,:));

for i=(length(demanda)+1):(length(demanda)+carros-2)
   dist_acs(i,:)=dist_acs(1,:);
   dist_acs(:,i)=dist_acs(:,1);
   dist_acs(i,1)=100*maximo;
   dist_acs(1,i)=100*maximo;
end

ferormonios = ferormonios*(1/comp_nn); %inicializa os valores de ferormonio nos arcos
in = zeros(1,length(demanda)); %in registra o numero de vezes que cada no deixou de ser visitado

%CICLO
cont = 0; %contador de iteraçoes, cada uma composta de k=10 formigas
while cont<20
   for k=1:10
      [candidato,comp_candidato] = simula_formiga(rho,comp_nn,in,dist_acs,ferormonios,h_inicial,h_final,
demanda,cap,tempo_serv);
      in = in+1; %soma 1 a todos os elementos de in
      for i=1:length(candidato)
         in(candidato(i))=in(candidato(i))-1; %subtrai 1 dos nos visitados em in
      end
      in(1)=0; %o deposito sempre possui seu in igual a 0
      clientes = length(candidato)-qtd_veiculos(candidato)-1;
      if clientes>(length(caminho_atual)-qtd_veiculos(caminho_atual)-1)
         caminho_atual = candidato;
         comp_atual = comp_candidato; %se a soluçao nao for factivel, comp_candidato podera ate estar errado, mas
nao ha problema
         in = in-in; %faz-se in=0 para todos os nos
         if (length(caminho_atual)-qtd_veiculos(caminho_atual)-1)==(length(demanda)-1) %soluçao factivel
            caminho = caminho_atual;
            comp = comp_atual;
            carros = qtd_veiculos(caminho);
            flag = flag - 1;
            return %a funçao acs_vei e´ terminada, pois uma soluçao foi encontrada
         end
      end
   end
   for i=1:(length(caminho_atual)-1) %global update #1
      ferormonios(caminho_atual(i),caminho_atual(i+1))=(1-rho)*ferormonios(caminho_atual(i), caminho_atual(i+1))
+ rho*(1/comp_atual);
      ferormonios(caminho_atual(i+1),caminho_atual(i))=(1-rho)*ferormonios(caminho_atual(i+1), caminho_atual(i))
+ rho*(1/comp_atual);
   end
   for i=1:(length(caminho)-1) %global update #2
      ferormonios(caminho(i),caminho(i+1))=(1-rho)*ferormonios(caminho(i),caminho(i+1)) + rho*(1/comp);
      ferormonios(caminho(i+1),caminho(i))=(1-rho)*ferormonios(caminho(i+1),caminho(i)) + rho*(1/comp);
   end
   cont = cont+1;
end
```

Quadro 3 – Algoritmo da função *acs-vei*

Na inicialização da função, deseja-se definir um caminho, mesmo que infactível, composto por (*carros* – 1) veículos, para posteriormente avaliar-se se é possível chegar a uma solução factível a partir deste caminho, com este número reduzido de veículos. Para esse fim, o primeiro *while* é introduzido. Se existem *M* veículos na solução inicial *caminho*, significa dizer que o nó correspondente ao depósito (o nó 1 por convenção) aparece M + 1 vezes na seqüência de nós correspondente ao vetor *caminho*, pois é preciso que todas as rotas se iniciem e terminem no depósito. Sendo assim, este primeiro *while* descobre qual a posição *j* do vetor *caminho* que abriga o M-ésimo nó 1, descartando os nós seguintes, que correspondem à última rota e ao último retorno ao depósito. Por exemplo, se possuímos uma solução *caminho* = [1 3 4 1 2 6 7 1 5 8 1], temos que *carros = 3* (pois existem 3 rotas, sendo elas 1-3-4-1, 1-2-6-7-1 e 1-5-8-1); pode-se ver que o depósito aparece 4 vezes no caminho, pois existem 3 veículos em uso. A execução do primeiro *while* para este exemplo dirá então que *j = 8*, pois a M-ésima aparição do depósito no caminho se dá na oitava posição do vetor. Descarta-se então a rota 1-5-8-1, fazendo com que o vetor *caminho_atual* fique sendo o caminho 1-3-4-1-2-6-7-1. Define-se ainda uma variável *comp_atual*, que possui o valor do comprimento do *caminho_atual*, que nada mais é que o valor da variável *comp* inicial subtraída do comprimento dos arcos que foram expurgados da solução (o que é implementado no primeiro *for* da função *acs-vei*).

Ainda na inicialização, é definido o parâmetro *rho*, que representa a importância do caminho mais recente frente ao conhecimento acumulado referente aos caminhos passados. Esta variável é o ρ que será usado na atualização dos ferormônios. O valor escolhido foi de 0,1, ou seja, a cada evaporação de ferormônios, a quantidade de ferormônios que um dado arco possui além da quantidade inicial de ferormônios cai 10%; no caso da deposição de ferormônios, dá-se um peso de 90% ao valor de ferormônios atual e 10% ao valor relacionado ao melhor caminho encontrado na iteração em questão. Quanto maior o valor de *rho*, mais rápida acontece a evaporação de ferormônios, e menos peso se dá para as informações passadas.

É hora de se criar a matriz *ferormônios*, que possuirá as informações da concentração de ferormônios de cada arco. No TSP, esta matriz seria de dimensão N x N, onde *N* é o número de nós do problema. Porém, no VRP, é preciso atentar para os depósitos virtuais, conforme descrito anteriormente. Esta matriz terá então

uma dimensão N+V-1 por N+V-1, onde *V* é o número de veículos com o qual se tentará chegar a uma solução factível. Como o número de veículos da melhor solução até o momento possui um número *carros* de veículos, queremos procurar por uma solução um veículo a menos: com V = *carros* -1. Sendo assim, deve-se adicionar (*carros-2)* cópias do nó 1 à matriz N x N original. Quando todos os N+V-1 nós tiverem sido visitados na construção de um caminho, adiciona-se o retorno final ao depósito na solução encontrada. Como exemplo: se a melhor solução até o momento possui 8 veículos, e o problema consiste em 25 nós, sendo o nó 1 o depósito, deve-se criar uma matriz de ferormônios de dimensão 31x31, pois se deve adicionar 8-2=6 cópias do depósito à matriz, que somadas ao depósito original (o primeiro dos 25 nós), totalizam 7 cópias do depósito, ou 7 veículos. Estes 7 veículos descreverão, então, 7 rotas, todas se iniciando no depósito ou em uma de suas cópias virtuais. Tendo sido criada a matriz *ferormônios*, define-se que todo arco terá uma concentração inicial igual ao inverso do comprimento do caminho encontrado pelo *Nearest Neighbor*.

Alem disso, a matriz *dist* de distâncias/custos também precisa ser expandida para as dimensões N+V-1 x N+V-1, para passar a considerar os depósitos virtuais. Isso também é feito na inicialização da função. É importante perceber que não se deseja que um caminho possua em seqüência dois ou mais depósitos (por exemplo, um caminho que sai do nó 1 e vai diretamente para um nó correspondente a um depósito virtual). Ou seja, não se deseja que um veículo siga de um depósito diretamente para uma cópia sua, pois isso não teria nenhum significado físico, já que os depósitos virtuais são meras cópias do depósito original. A função dos depósitos virtuais é fazer com que o depósito original possa ser visitado mais de uma vez na etapa de construção de caminhos, sendo uma opção de caminho a ser percorrido por um veículo que se encontre em meio a uma rota qualquer. Sendo assim, é importante que não exista um arco entre dois depósitos. Isto é conseguido fazendo com que a distância entre os nós que representam depósitos seja infinita, para que estes arcos nunca sejam escolhidos pelos veículos. No algoritmo proposto, isso é implementado através da definição da distância entre estes nós como um valor muito grande, de 100 vezes o comprimento do maior arco do problema. Isto na prática inviabiliza a escolha destes arcos.

Para finalizar a inicialização do *acs_vei*, define-se ainda o vetor *IN*, de dimensão 1 x N, que terá uma função de indicar quantas vezes seguidas cada nó deixou de ser visitado por uma formiga em seu percurso. Em outras palavras, após a simulação de cada formiga, os nós que não tiverem sido visitados terão suas posições no vetor *IN* incrementadas de uma unidade. Isto será usado posteriormente para motivar as formigas a visitarem os nós que tenham sido pouco visitados pelas formigas, na tentativa de se encontrar uma solução factível. Este artifício só será usado pela função *acs-vei*, que deve procurar por soluções factíveis. No caso da função *acs_time*, como veremos adiante, o vetor *IN* será deixado de lado. Este será a principal diferença entre os métodos de construção de caminhos das duas colônias. Entretanto, isto não impossibilita que uma mesma função seja utilizada para construir caminhos para ambas as colônias, como veremos adiante.

No ciclo principal da função, define-se o número máximo de iterações que serão executadas. Escolheu-se o valor *cont = 20* para tal. Cada uma destas iterações será composta da simulação de *k = 10* formigas. Basicamente, para cada formiga simulada, uma função auxiliar *simula_formiga* é chamada. Esta função recebe como argumentos as variáveis *rho*, *comp_nn*, o vetor *IN*, as matrizes *dist_acs* e *ferormônios*, os vetores *h_inicial, h_final*, *demanda* e *tempo-serv*, e a variável *cap*. Ou seja, a função recebe as informações mais recentes sobre as concentrações de ferormônios e os dados físicos do problema. Esta função calcula então, através das regras de decisão das formigas, o caminho que ela percorreria pelo problema, e fornece como resposta um caminho chamado de *candidato*, bem como seu comprimento. Esta função também será utilizada pela outra colônia de formigas, pois a regra de decisão das formigas, e conseqüentemente a forma com que os caminhos são construídos, não muda de uma colônia para outra. A diferença entre as colônias está na forma com que estas soluções fornecidas pelo *simula_formiga* são analisadas e avaliadas, como foi explicado anteriormente.

Após a simulação da formiga, atualiza-se o vetor *IN*, adicionando uma unidade nas posições do vetor correspondente a nós que não foram visitados no *candidato* (caminho encontrado pela formiga em questão).

Define-se então a variável *clientes*, que terá o número de clientes visitados em *candidato*, excluindo-se de *candidato* os depósitos e suas cópias. Isto é feito com o auxílio da função auxiliar *qtd_veiculos*, que simplesmente calcula a

quantidade de veículos utilizada por um caminho. Isso é feito somando-se o número de vezes que o nó 1 aparece em um caminho, e subtraindo-se em seguida o retorno final ao depósito. A função pode ser implementada através de um algoritmo muito simples, que chamamos de *qtd_veiculos,* e se encontra abaixo:

```
function qtd = qtd_veiculos(caminho)
%funçao que recebe o vetor caminho e retorna o numero de veiculos utilizados

qtd = 0;
for j = 1:length(caminho)
   if caminho(j)==1
      qtd = qtd + 1;
   end
end
qtd = qtd-1;
```

Quadro 4 – Algoritmo da função *qtd_veículos*

Se o número de clientes for maior no *candidato* que no *caminho_atual*, sabe-se que o caminho representado por *candidato* está mais próximo de ser uma solução factível que o caminho proposto em *caminho_atual*. Sendo assim, faz-se *caminho_atual = candidato*, e *comp_atual = comp_candidato*. Reinicia-se então o vetor *IN*.

Em seguida, um teste é feito: se este caminho proposto em *candidato* visitar a totalidade dos clientes (número este que é conseguido pela dimensão do vetor *demanda*, por exemplo), então este caminho é uma solução factível. Ou seja, foi possível encontrar uma solução factível com um número de veículos inferior ao da melhor solução encontrada até o momento. Então, define-se *caminho = caminho_atual*, *comp = comp_atual* e *veículos = qtd_veiculos(caminho)*. Além disso, subtrai-se uma unidade do *flag*, conforme explicado anteriormente, para indicar que uma solução factível foi encontrada. Estas quatro variáveis e vetores são as saídas da função *acs_vei*. A execução da função está, então, terminada, e pode-se passar para a execução do *acs-time*.

Caso, entretanto, as $k = 10$ formigas da primeira iteração forem simuladas, e não seja encontrada uma solução factível, deve-se fazer o *global update*, ou atualização global, dos ferormônios. Isso é feito no final da iteração, através da deposição de ferormônios nos arcos que compõem o melhor caminho encontrado até o momento pela função (*caminho_atual),* que pode não ser factível, bem como nos arcos que fazem parte da melhor solução factível que tenha sido encontrada até o momento (*caminho)*. Esta dupla deposição de ferormônios é sugerida por [4]

como uma forma de otimizar a busca por melhores soluções. A quantidade de ferormônios depositada nestas atualizações é igual ao inverso do comprimento de *caminho_atual* e de *caminho*. Feito isso, passa-se para a próxima iteração do algoritmo, na qual o processo de simulação das *k* formigas é reiniciado, porém com as informações referentes à iteração anterior guardadas na matriz de ferormônios.

6.1.2
Implementação do *acs-time*

O algoritmo criado para a execução da função *acs_time* possui muitas semelhanças com aquele criado para a função *acs_vei*, e se encontra detalhado abaixo:

```
function [caminho,comp,carros,flag] = acs_time(caminho,comp,carros,flag,comp_nn,dist,h_inicial,h_final,demanda,
cap,tempo_serv)
%funçao que, dado um certo numero de carros, procura por um caminho melhor

%INICIALIZAÇAO
flag = flag + 1;

caminho_atual = caminho;
comp_atual = comp;
rho = 0.1;
ferormonios = ones(length(demanda)+carros-1); %adiciona `a matriz (carros-1) depositos virtuais
dist_acs = dist; %dist_acs sera a matriz de distancias extendida, com os depositos virtuais
maximo = max(dist_acs(1,:));

for i=(length(demanda)+1):(length(demanda)+carros-1)
   dist_acs(i,:)=dist_acs(1,:);
   dist_acs(:,i)=dist_acs(:,1);
   dist_acs(i,1)=100*maximo;
   dist_acs(1,i)=100*maximo;
end

ferormonios = ferormonios*(1/comp_nn);
in = zeros(1,length(demanda)); %in sera sempre zero para o acs_time

%CICLO
cont = 0; %contador de iteraçoes, cada uma composta de k=10 formigas
while cont<20
   for k=1:10
      [candidato,comp_candidato] = simula_formiga(rho,comp_nn,in,dist_acs,ferormonios,h_inicial,h_final,demanda,
cap,tempo_serv);
      clientes = length(candidato)-qtd_veiculos(candidato)-1;
      if clientes==(length(demanda)-1) %se a soluçao for factivel
         if comp_candidato<comp_atual
            caminho_atual = candidato;
            comp_atual = comp_candidato;
         end
      end
   end
   for i=1:(length(caminho_atual)-1) %global update
      ferormonios(caminho_atual(i),caminho_atual(i+1))=(1-rho)*ferormonios(caminho_atual(i),caminho_atual(i+1))
+ rho*(1/comp_atual);
      ferormonios(caminho_atual(i+1),caminho_atual(i))=(1-rho)*ferormonios(caminho_atual(i+1),caminho_atual(i))
+ rho*(1/comp_atual);
   end
```

```
    cont = cont+1;
end
caminho = caminho_atual;
comp = comp_atual;
carros = qtd_veiculos(caminho);
```

Quadro 5 – Algoritmo da função *acs_time*

Esta função recebe os mesmos argumentos que a função *acs_vei,* que neste ponto deverão ser referentes ao melhor caminho encontrado por aquela função. Além disso, o *acs_time* terá uma saída equivalente ao *acs_vei*: o melhor caminho encontrado, seu comprimento, o número de veículos utilizado neste caminho, e o *flag* indicativo de melhoria.

Na inicialização, o *flag* é incrementado de uma unidade. Desta forma, fecha-se a lógica de seu funcionamento. Se a função *acs_vei* obtiver melhorias, *flag* é subtraído de uma unidade; em seguida, a função *acs_time* incrementa-o de uma unidade. Enquanto houver melhorias no *acs-vei*, este ciclo se perpetua. No momento que o *acs_vei* não for mais capaz de obter melhorias, a variável *flag* será incrementada seguidamente pelo *acs_time*, até que alcance um valor limítrofe, o que fará com que o MACS-VRPTW seja terminado.

A etapa de inicialização da função é equivalente àquela do *acs_vei*. Entretanto, não é preciso expurgar uma rota do melhor caminho encontrado até o momento, como havia sido feito para a outra função. No caso do *acs_time*, faz-se *caminho_atual* = *caminho* e *comp_atual* = *comp*. Ou seja, define-se o caminho atual como sendo o próprio caminho recebido como argumento da função, sem passar por nenhuma manipulação. Feito isso, define-se *rho*, *dist_acs*, *ferormônios* e *IN* da mesma maneira que no *acs_vei*, com a diferença que deve-se adicionar um número de depósitos virtuais às matrizes *dist_acs* e *ferormônios* igual a (*carros* – 1), e não mais (*carros* – 2), pois deseja-se encontrar um caminho melhor que o atual com um mesmo número de veículos, e não mais com um a menos. Além disso, o vetor *IN* é definido como um vetor 1 x N de zeros, e assim permanecerá durante toda a execução da função. Dessa forma, ele não influenciará em nada na execução do algoritmo, como veremos adiante. Entretanto, o uso deste artifício possibilita que uma única função *simula_formiga* seja criada, atendendo tanto ao *acs_time* quanto ao *acs_vei.*

Terminada a inicialização, passa-se ao ciclo principal do algoritmo. Mais uma vez se define o número máximo de iterações (*cont = 20),* e de formigas com as quais deseja-se trabalhar em cada iteração (*k = 10).* Para cada formiga a função *simula_formiga* é chamada, visando obter um caminho simulado. Este caminho recebe o nome de *candidato*. Se o número de clientes deste *candidato* for igual ao número total de clientes, a solução encontrada é factível. Deseja-se avaliar então se o comprimento deste *candidato* é inferior ao *comp_atual*, que representa o comprimento da melhor solução encontrada até o momento. Se o for, faz-se *caminho_atual* = *candidato* e *comp_atual* = *comp_candidato*. Isto é repetido para todas as *k* formigas de cada iteração.

Ao final de cada iteração, é feita a atualização global de ferormônios. A deposição de ferormônios, neste caso, só é realizada nos arcos pertencentes ao melhor caminho encontrado, que é necessariamente factível (no caso do *acs_vei*, uma dupla atualização global era realizada porque o melhor caminho até o momento não necessariamente era factível). A quantidade de ferormônios depositadas nestes arcos, mais uma vez, é o inverso do comprimento total deste melhor caminho encontrado. Feito isso, incrementa-se o contador de iterações e o ciclo se reinicia.

Pode-se perceber que a execução do *acs-time* é de certa forma mais simples que aquela do *acs-vei,* principalmente devido a uma inicialização mais direta. Porém, o funcionamento dos algoritmos é muito parecido. A grande diferença entre estas funções está no critério utilizado para se definir que uma solução encontrada é melhor que a atual. Para o *acs_vei*, não importa o comprimento da solução, e sim quantos clientes ela abrange. Uma solução com um número maior de clientes no caminho é preferível a uma com um número menor, pois deseja-se chegar a uma solução factível qualquer, que prove que é possível visitar todos os clientes com um número reduzido de veículos. Para o *acs_time,* entretanto, o critério usado no modelo ACS clássico é seguido: uma solução encontrada é preferível à atual se possuir um comprimento total inferior.

Estudaremos com mais detalhes na seção seguinte as funções auxiliares *simula_formiga* e *insertion_procedure* – esta última sendo uma função auxiliar da *simula_formiga*.

6.1.3
Funções auxiliares

Primeiramente, vamos descrever a função *simula_formiga*. Esta função é chamada tanto pela função *acs_vei* quanto pela função *acs_time*. Entretanto, nesta última função o vetor *IN* é um vetor de zeros, fazendo com que ele não tenha influências na execução da *simula_formiga*. Isto ficará mais claro a seguir.

A função *simula_formiga* tem por objetivo definir o caminho que uma formiga percorrerá, com base nas informações atuais de ferormônios – que são parte dos argumentos da função – e nas regras de decisão pseudo-aleatórias apresentadas. O algoritmo se encontra na sua totalidade descrito abaixo:

```
function [candidato,comp_candidato] = simula_formiga(rho,comp_nn,in,dist_acs,ferormonios,h_inicial,h_final,
demanda,cap,tempo_serv)
%funçao que calcula uma rota a partir das matrizes de distancia e ferormonio estendidas (incluindo depositos virtuais).
%a soluçao nao precisa ser factivel, pois o numero de rotas (depositos) ja vem definido pelas dimensoes das matrizes.

h_atual = 0;
cap_atual = 0;
rotas = length(ferormonios(1,:))-length(demanda) + 1; %menor numero de rotas obtido ate o momento
aux = fix(rand(1)*rotas)+1; %aux sera um valor aleatorio entre 1 e o numero de rotas
if aux==1
   no_atual = 1;
else
   no_atual = length(demanda)-1+aux;
end

for i=(length(demanda)+1):length(ferormonios(1,:)) %os parametros sao estendidos, considerando tambem os
depositos virtuais
   in(i)=in(1);
   h_inicial(i)=h_inicial(1);
   h_final(i)=h_final(1);
   demanda(i)=demanda(1);
   tempo_serv(i)=tempo_serv(1);
end

candidato_atual = [no_atual];
comp_candidato_atual = 0;

q0 = 0.9;
beta = 1;
const = 0; %const indica se o insertion procedure foi executado

while (length(candidato_atual)<length(ferormonios))&(const==0)
   for i=1:length(ferormonios)
      urgencia = h_final(i)-h_atual-tempo_serv(no_atual)-dist_acs(no_atual,i);
      carga = demanda(i) + cap_atual;
      if (i~=no_atual)&(~ismember(i,candidato_atual))&(carga<=cap)&(urgencia>=0)
         delivery_time = max(h_atual + tempo_serv(no_atual) + dist_acs(no_atual,i),h_inicial(i));
         delta_time = delivery_time - h_atual;
         distancia = delta_time.*((h_final(i))-h_atual);
         distancia = max(1,(distancia-5*in(i)));
         atratividade(i) = 1/distancia;
      else
         atratividade(i) = 0;
      end
   end
   if max(atratividade)==0 %caso nao haja mais nos factiveis
      [candidato_atual,comp_candidato_atual] = insertion_procedure(candidato_atual,comp_candidato_atual,rotas,
dist_acs,h_inicial,h_final,demanda,cap,tempo_serv);
```

```
    if length(candidato_atual)<length(ferormonios) %caso o insertion procedure nao tenha encontrado soluçao
factivel
        const = 1;
    end
  else
    q = rand(1);
    prob = ferormonios(no_atual,:).*(atratividade.^beta);
    if q<=q0
      [valor,indice] = max(prob);
      candidato_atual = [candidato_atual indice];
    else
      r = rand(1);
      a = 0;
      denominador = 0;
      for i=1:length(prob)
        if ~ismember(i,candidato_atual)
          denominador = denominador + prob(i);
        end
      end
      i=1;
      while r>a
        if ~ismember(i,candidato_atual)
          a = a + ((prob(i))/(denominador));
        end
        i=i+1;
      end
      i=i-1;
      candidato_atual = [candidato_atual i]; %i e´ o proximo no a ser visitado
    end
    proximo = candidato_atual(length(candidato_atual)); %proximo sera o proximo no a ser visitado
    comp_candidato_atual = comp_candidato_atual + dist_acs(no_atual,proximo);
    if (proximo==1)|(proximo>length(ferormonios)-rotas+1) %se o proximo no for um deposito
      h_atual = 0;
      cap_atual = 0;
    else
      h_atual = h_atual + max(dist_acs(no_atual,proximo)+tempo_serv(no_atual),h_inicial(proximo)-h_atual);
      cap_atual = cap_atual + demanda(proximo);
    end
    ferormonios(no_atual,proximo) = (1-rho)*ferormonios(no_atual,proximo) + rho*(1/comp_nn); %local update
    ferormonios(proximo,no_atual) = (1-rho)*ferormonios(proximo,no_atual) + rho*(1/comp_nn); %local update
    no_atual = proximo;
  end
end
candidato_atual = [candidato_atual 1]; %adiciona-se o retorno ao deposito
comp_candidato_atual = comp_candidato_atual + dist_acs(no_atual,1);
for i=1:length(candidato_atual)
  if candidato_atual(i)>length(ferormonios)-rotas+1
    candidato_atual(i)=1;
  end
end
candidato = candidato_atual;
comp_candidato = comp_candidato_atual;
```

Quadro 6 – Algoritmo da função auxiliar *simula_formiga*

Esta função já recebe como argumentos as matrizes de distâncias (*dist_acs)* e de ferormônios (*ferormônios*) com as dimensões expandidas de acordo com o número de depósitos virtuais que serão usados. Estas matrizes tiveram suas dimensões adaptadas no decorrer das funções *acs_vei* e *acs_time*, como vimos anteriormente. Dessa forma, a função *simula_formiga* deve simplesmente percorrer os nós, incluindo os depósitos virtuais, segundo a regra de decisão das

formigas, de forma a compor um caminho. O caminho encontrado pode ser ou não factível, ou seja, pode não passar por todos os nós. Entretanto, as conseqüências da não-factibilidade do caminho encontrado são analisadas pelas funções que chamam o *simula_formiga*, especificamente pela *acs_vei* e pela *acs_time*. No caso da *acs_vei*, por exemplo, a *simula_formiga* é chamada para tentar encontrar um caminho com um veículo a menos que o atual. Caso a resposta da *simula_formiga* seja um caminho não factível, mas visitando mais clientes que o caminho atual, a solução é mantida pelo *acs_vei*. No caso do *acs_time*, entretanto, só interessam as soluções factíveis. Dessa forma, se a resposta da *simula_formiga* for um caminho não-factível, esta será descartada.

O funcionamento da função se inicia pela definição de dois contadores: o *h_atual*, que será uma espécie de relógio registrando o tempo decorrido desde o início de cada rota, e o *cap_atual*, que registrará a capacidade total utilizada pelo veículo desde o início da rota.

Feito isso, deve-se selecionar aleatoriamente um nó de início do algoritmo. Na prática, o início será sempre no depósito, mas como este depósito possui algumas cópias virtuais, deve-se selecionar uma dentre estas possibilidades para se iniciar o caminho. Supondo um problema com 10 nós, sendo o nó 1 o depósito, e 4 rotas, por exemplo, deve-se selecionar um dentre os nós 1, 11, 12 e 13 – estes três últimos sendo as cópias virtuais do depósito. Isso é feito definindo-se inicialmente o número de rotas com o qual se irá trabalhar, o que se faz analisando as dimensões da matriz de ferormônios e do vetor *demanda*, por exemplo. A diferença é o número de depósitos virtuais que foram acrescentados à matriz. Feito isso, gera-se um número aleatório para auxiliar na decisão de qual depósito escolher como ponto de partida.

A próxima etapa da inicialização é a adaptação dos vetores *h_inicial, h_final, IN, demanda* e *tempo_serv* para considerarem também os depósitos virtuais. Isto é feito copiando-se as posições 1 destes vetores no final deles próprios, tantas vezes quantos depósitos virtuais forem utilizados. O primeiro *for* do algoritmo realiza estas operações.

Define-se então o vetor *candidato_atual*, que possui o candidato à solução, sendo inicialmente um caminho de comprimento zero composto somente do nó inicial. Outros nós serão gradativamente adicionados a este caminho, aumentando seu comprimento. Este comprimento será registrado na variável

comp_candidato_atual. Os parâmetros *q0*, *beta* e *const* são ainda definidos. O primeiro é o parâmetro que define a probabilidade de cada formiga ser exploradora, bem como a probabilidade da formiga se utilizar do conhecimento passado para tomar suas decisões. Isto está de acordo com o descrito na seção 5, quando o ACS foi detalhado. Quanto maior for o valor de *q0*, menos desbravadora será a formiga. Suas funções ficarão mais claras no desenvolvimento do algoritmo. O parâmetro *beta*, por sua vez, define a importância relativa do comprimento ponderado dos arcos frente às concentrações de ferormônios. A variável *const* é somente um indicativo de que já se chegou ao fim da construção do caminho, que será fundamental para a terminação da execução da função.

O comando *while* define que o algoritmo deverá ser executado até que o comprimento do *candidato_atual* seja igual às dimensões da matriz *ferormônios*, o que só acontecerá quando todos os nós tiverem sido visitados, incluindo os depósitos virtuais. Entretanto, é possível que, para um dado número de veículos, um caminho não seja capaz de passar por todos os nós do problema. Isso deixaria o problema em um *loop* infinito, pois nunca se chegaria a uma solução factível (que visita todos os nós). Para isso que existe a variável *const*. Quando se chega a esta situação de infactibilidade, uma função chamada *insertion_procedure* é chamada para tentar inserir os nós remanescentes no caminho. Independentemente do fato dela ter tido sucesso ou não nessa tarefa, a variável *const*, que funciona como um *flag*, assume o valor 1. Este valor indica que a execução da *simula_formiga* chegou ao fim. Se este valor for 0, então a rotina é executada novamente.

Deseja-se inicialmente definir a atratividade de cada arco, do ponto de vista de uma formiga que se situa no nó inicial escolhido. A atratividade é uma espécie de comprimento ponderado dos arcos. Ao invés de olharmos simplesmente para os comprimentos, deseja-se levar em consideração também o possível tempo de espera que poderá incorrer devido ao início das janelas de tempo, bem como a urgência devido à proximidade do fim das janelas de tempo dos clientes. Procedimento equivalente foi adotado na adaptação do *Nearest Neighbor* para o VRPTW, como foi descrito na seção 3.2.1. No caso atual, deseja-se definir a atratividade somente para aqueles nós que ainda não constam no caminho percorrido, e que além disso não violam as restrições de capacidade e de janela de tempo. Para esta última restrição, define-se a urgência de cada nó, que nada mais é

que a diferença entre o final da janela de tempo de cada nó e a soma do tempo atual e os tempos de serviço e de trânsito até o nó. Ou seja, se a urgência de um nó, que pode ser entendida como a "folga" de tempo, for maior que zero, este nó pode ser alcançado pelo veículo atual antes que sua janela de tempo se feche. Caso estes critérios de urgência, de capacidade e de presença de um nó no caminho atual forem atendidos, a atratividade do nó é calculada. Caso contrário, ela é definida como zero.

Para o cálculo da atratividade, calcula-se primeiramente o *delivery_time*, que é o momento no qual o veículo poderá ser atendido por um nó. Ele é a hora atual (*h_atual)* somada ao tempo de serviço do nó atual e à distância até o nó em questão, ou então é simplesmente a hora inicial da janela de tempo do nó, o que for maior. Subtraindo-se disto a hora atual, tem-se o valor de *delta_time*, que indica o intervalo de tempo que se passará do momento atual (início do serviço no nó atual) até o início do serviço no nó seguinte em questão. Quanto menor este intervalo, maior será a atratividade. Este intervalo engloba a espera que poderá incorrer ao se visitar o nó seguinte em questão.

Para levarmos em consideração a urgência de cada nó, multiplicamos este *delta_time* pelo valor de $h\_final_i - h\_atual$, onde i é o possível nó de destino, que está tendo sua atratividade calculada. Temos então um número maior que zero, pois estes cálculos só são realizados para os nós com $urgência > 0$. Este número, chamado de *distância*, já reúne informações de comprimento dos arcos e do início e fim das janelas de tempo. Entretanto, um outro fator é levado em consideração: o vetor *IN*, que possui o número de iterações que cada nó passou sem ser visitado por um caminho. Faz-se $distância_i = max(1, distância_i - IN_i)$, ou seja, o valor da *distância* de um nó i é reduzido de uma quantidade proporcional ao número de iterações que ele não foi visitado, e se este resultado for inferior a 1, a *distância* deste nó é definida como 1. Isso estimula a visitação de nós pouco comuns nos caminhos, favorecendo a criação de novas soluções. Como a dimensão de *distância* antes desta operação de subtração de *IN* é da ordem de dezenas ou centenas, ou até mesmo poucos milhares (pois ela é resultado da multiplicação do *delta_time*, da ordem de dezenas, pela *urgência*, também da ordem de dezenas, podendo ambas chegar a pouco mais que uma centena), não haveria muito sentido em subtrair um valor de *IN* da ordem de unidades. A conta ficaria desequilibrada, e a influência de *IN* seria praticamente nula. Então, pode-se adicionar à equação

um multiplicador de *IN*, para darmos mais peso a este vetor. Na implementação do algoritmo, usamos 5**IN*, que mostra ser mais coerente com as dimensões em questão. Finalmente, tendo sido calculada a $distancia_i$ de cada nó, define-se a *atratividade* de cada nó como o inverso da *distância*. Ou seja, a *atratividade* será sempre um valor entre 0 e 1.

O algoritmo testa, então, se o valor máximo de *atratividade* dos nós do problema é zero. Se o for, significa dizer que nenhum nó pode ser visitado, e o caminho não pode mais ser estendido. Isto pode significar tanto que o caminho atual já é factível e todos os nós já foram visitados, quanto que o caminho atual não é factível, e alguns nós ficaram de fora e não poderão ser visitados em seguida. A função auxiliar *insertion_procedure* é então chamada para tentar inserir estes nós faltantes em algum ponto do caminho atual, em uma tentativa de deixar a solução factível. Se mesmo após esta medida a solução continuar infactível, a variável *const* assume o valor 1, como mencionado anteriormente. Se a solução se tornar factível, esta medida não é necessária, pois a própria função *simula_formiga*, no seu primeiro *while*, identificará que uma solução factível já foi encontrada.

Se houver valores de atratividade diferentes de zero, entretanto, a função entra em sua parte principal, que é a construção de caminhos propriamente dita. Conforme definido pela regra de decisão das formigas, multiplica-se o valor da concentração de ferormônios de cada arco pela sua atratividade elevada ao parâmetro *beta*. O resultado é proporcional à probabilidade de escolha de cada arco, resultado esse que será chamado de *prob*, para todos os nós do problema. Escolhe-se um número aleatório de 0 a 1, que será chamado de *q*. Se *q* for menor ou igual ao parâmetro *q0*, o próximo nó a ser visitado pela formiga será simplesmente aquele que possuir o maior valor de *prob*. Caso contrário, se $q > q0$, o próximo nó selecionado será definido probabilisticamente. Neste caso, é preciso primeiramente definir o *denominador*, que será o somatório dos valores de *prob* dos nós que ainda não pertencem ao caminho. Feito isso, sabe-se que a probabilidade de um nó ser escolhido será a razão entre seu valor de *prob* e este denominador. Inicializa-se então um contador *a* e um número aleatório *r* entre 0 e 1. O contador é inicialmente igual a zero. Define-se ainda um contador *i*, inicialmente igual a 1, que deverá percorrer o vetor *prob*. Verifica-se se o valor de *r* é menor que o valor de *a*. Caso ele não seja, soma-se a *a* o valor de

prob(i)/denominador, onde *i* é o primeiro nó que ainda não pertence ao caminho. Caso *r* ainda seja maior que *a*, soma-se a *a* o valor de *prob(i)/denominador* sendo *i* o segundo nó não pertencente ao caminho. Quando $r < a$, define-se o nó escolhido como o nó $(i - 1)$. O nó escolhido pode então ser adicionado ao caminho. Dessa forma, uma escolha aleatória é realizada.

A este nó selecionado dá-se o nome de *próximo*. Acrescenta-se então o comprimento do arco que liga o nó atual até este nó ao *comp_candidato_atual*. Passa-se em seguida para a etapa de atualização dos contadores de tempo e de carga *h_atual* e *cap_atual*. Estes devem ser zerados caso *próximo* seja o depósito ou uma de suas cópias. Caso contrário, deve-se somar a estes contadores os valores devidos, considerando possíveis esperas, tempos de trânsito, tempos de serviço e demandas dos nós.

Para finalizar a iteração, deve-se fazer a atualização local de ferormônios, através da qual a concentração de ferormônios no arco percorrido é decrescida de acordo com o parâmetro *rho*. Define-se então *próximo* como *no_atual* e reinicia-se o *while* da função.

A função precisa ainda adicionar o retorno ao nó 1 ao caminho, para que o caminho seja concluído. Além disso, os nós correspondentes aos depósitos virtuais são todos substituídos pelo nó 1, para indicar que na realidade se referem ao depósito inicial. Está terminada então e execução da função *simula_formiga*, que certamente é o coração operacional de toda a simulação do MACS-VRPTW.

Abordaremos agora o funcionamento da função *insertion_procedure*, que foi chamada pela função *simula_formiga* na tentativa de chegar a uma condição de factibilidade de uma solução. Seu algoritmo se encontra abaixo:

```
function [candidato_atual,comp_candidato_atual] = insertion_procedure(candidato_atual,comp_candidato_atual,rotas,
dist_acs,h_inicial,h_final,demanda,cap,tempo_serv);
%funçao que recebe um caminho incompleto e adiciona da melhor forma possivel os nos ainda nao visitados

qtd = zeros(1,length(demanda)); %indicara as demandas dos nos que ainda nao foram visitados
for i=1:length(demanda)
   if ~ismember(i,candidato_atual)
      qtd(i)=demanda(i);
   end
end

novo_candidato = candidato_atual;

while max(qtd)~=0
   [valor,indice] = max(qtd);
   novo_comp = 5*comp_candidato_atual;
   novo_candidato_temp = novo_candidato;
   for i=1:length(novo_candidato)-1 %o no sera inserido entre i e i+1
```

```
      temp = [novo_candidato(1:i) indice novo_candidato(i+1:length(novo_candidato))]; %acrescenta o no ao vetor
temp
      comp_temp = 0; %tera o comprimento do novo caminho
      flag = 0; %flag sinalizara a infactibilidade da soluçao
      h = 0; %hora atual
      c = 0; %carga atual
      for j=1:length(temp)-1
         if ((h_final(temp(j+1)) - h - tempo_serv(temp(j)) - dist_acs(temp(j),temp(j+1)))>=0)&((c + demanda(temp(j
+1)))<=cap)
            if (temp(j+1)==1)|(temp(j+1)>length(demanda)-rotas+1) %se o no seguinte for um deposito
               h = 0;
               c = 0;
            else
               h = h + max(tempo_serv(temp(j)) + dist_acs(temp(j),temp(j+1)), h_inicial(temp(j+1)) - h);
               c = c + demanda(temp(j+1));
            end
         else
            flag = 1; %indica que a soluçao nao e´ factivel
         end
         comp_temp = comp_temp + dist_acs(temp(j),temp(j+1));
      end
      if (flag==0)&(comp_temp<novo_comp)
         novo_candidato_temp = temp;
         novo_comp = comp_temp;
      end
   end
   novo_candidato = novo_candidato_temp;
   qtd(indice) = 0;
end
candidato_atual = novo_candidato;
comp_candidato_atual = novo_comp;
```

Quadro 7 – Algoritmo da função auxiliar *insertion_procedure*

Inicialmente, cria-se o vetor *qtd* de demandas, que será igual ao vetor *demanda*, com a diferença de que as demandas que já foram atendidas pelo *caminho_atual* estão zeradas. Dessa forma, o vetor indica quais nós ainda precisam ser visitados. Começa-se então a tentativa de inserção pelo nó que possuir a demanda mais elevada.

Selecionado o nó a ser trabalhado, o algoritmo cria um vetor *temp*, que será o caminho atual adicionado do referido nó. Inicialmente, este nó é inserido na posição 2 do vetor, entre o primeiro e o segundo nós do caminho atual. Em seguida, a factibilidade desta configuração é testada, com o auxílio dos contadores *h* e *c*, respectivamente de tempo decorrido e de capacidade utilizada de carga. Se a solução for infactível, um *flag* é acionado com o valor 1, para indicar tal fato. Se for factível, avalia-se o novo comprimento do caminho. Se ele for menor que o melhor caminho encontrado até o momento (caminho este que já inclui o nó em questão inserido em alguma outra posição do caminho), este novo caminho é definido como o *novo_candidato_temp*. Reinicia-se então o ciclo de inserção do nó, tentando-se inseri-lo em outra posição do vetor caminho. Ao final deste ciclo,

o *novo_candidato_temp* deverá conter o caminho original acrescido deste nó, na configuração que apresentou o menor comprimento. Faz-se então *novo_candidato = novo_candidato_temp*, e a demanda do nó em questão é zerada no vetor *qtd*, para indicar que ele já foi visitado. Reinicia-se então o ciclo maior do *while*, na tentativa de inserir mais um nó no caminho. Cabe observar aqui que se um nó não conseguir ser inserido em posição alguma no caminho, mesmo assim sua demanda será zerada, da mesma forma que o seria se este nó tivesse sido inserido no caminho. Isto faz com que mesmo após a execução do *insertion_procedure* a solução possa continuar infactível. Porém, é de interesse que seja desta forma, pois para o *acs_vei*, mesmo uma solução infactível pode ser de interesse.

Uma outra função auxiliar que é utilizada é a *nearest_neighbor*, que já foi explicada em detalhes na seção 3.2.1.

Estas são todas as funções necessárias à implementação do MACS-VRPTW. Na seção seguinte, apresentaremos alguns resultados obtidos na simulação de problemas com o auxílio destas rotinas desenvolvidas como forma de atestar sua eficiência.