Inteligência Computacional para Otimização

Marcone Jamilson Freitas Souza

02 de fevereiro de 2024

Marcone Jamilson Freitas Souza, Departamento de Computação, Instituto de Ciências Exatas e Biológicas, Universidade Federal de Ouro Preto, CEP: 35.400-000, Ouro Preto, Minas Gerais. E-mail: marcone@ufop.edu.br, Homepage: http://www.decom.ufop.br/prof/marcone.

Observação: Referencie estas notas de aula como: [1]

[1] Souza, M. J. F. Inteligência Computacional para Otimização: meta-heurísticas. Departamento de Computação, Universidade Federal de Ouro Preto, Ouro Preto, Minas Gerais, 2024. Disponível em http://www.decom.ufop.br/prof/marcone/Disciplinas/InteligenciaComputacional/InteligenciaComputacional.pdf.

Sumário

1 Introdução

Muitos problemas práticos são modelados da seguinte forma: Dado um conjunto S de variáveis discretas s (chamadas soluções) e uma função objetivo $f : S \leftarrow \mathbb{R}$, que associa cada solução $s \in S$ a um valor real $f(s)$, encontre a solução $s^* \in S$, dita ótima, para a qual $f(s)$ tem o valor mais favorável (valor mínimo, no caso de o problema ter como objetivo a minimização de f, ou valor máximo, no caso de o problema ter como objetivo a maximização de f.

Grande parte desses problemas são combinatórios, sendo classificados na literatura como NP-difíceis, e assim, ainda não existem algoritmos que os resolvam em tempo polinomial.

Para citar um exemplo, seja o conhecido Problema do Caixeiro Viajante (PCV). O PCV é descrito por um conjunto de n cidades e uma matriz de distância entre elas, tendo o seguinte objetivo: o caixeiro viajante deve sair de uma cidade, dita cidade origem, visitar cada uma das $n-1$ cidades restantes apenas uma única vez e retornar à cidade origem percorrendo a menor distância possível. Em outras palavras, deve ser encontrada uma rota fechada (ciclo hamiltoniano) de comprimento mínimo que passe exatamente uma única vez por cada cidade.

Para mostrar a dificuldade de solução do PCV, assuma que a distância de uma cidade i à outra j seja simétrica, isto é, que $d_{ij} = d_{ji}$. Assim, o número total de rotas possíveis é $(n-1)!/2$. Para se ter uma idéia da magnitude dos tempos envolvidos na resolução do PCV por enumeração completa de todas as possíveis soluções, para $n = 20$ tem-se 6×10^{16} rotas. Um computador que avalia uma rota em 10^{-8} segundos gastaria cerca de 19 anos para encontrar a melhor rota! Mesmo considerando os rápidos avanços tecnológicos dos computadores, uma enumeração completa de todas essas rotas é inconcebível para valores elevados de n. Nos problemas da classe NP-difícil, não é possível garantir que a rota de custo mínimo seja encontrada em tempo polinomial. Assim, no pior caso, todas as possíveis soluções devem ser analisadas.

É possível dar uma certa "inteligência" a um método de enumeração, utilizando, por exemplo, as técnicas *branch-and-bound* ou *branch-and-cut*, de forma a reduzir o número de soluções a analisar no espaço de soluções. Com isto, pode ser possível resolver problemas de dimensões mais elevadas. Entretanto, dada a natureza combinatória do problema, pode ser que, no pior caso, todas as soluções tenham que ser analisadas. Este fato impede o uso exclusivo desses métodos, dito exatos, dado o tempo proibitivo de se encontrar a solução ótima.

Portanto, em problemas desta natureza, o uso de métodos exatos se torna bastante restrito. Por outro lado, na prática, em geral, é suficiente encontrar uma "boa" solução para o problema, ao invés do ótimo global, o qual, para esta classe de problemas, somente pode ser encontrado após um considerável esforço computacional.

Este é o motivo pelo qual os pesquisadores têm concentrado esforços na utilização de heurísticas para solucionar problemas deste nível de complexidade. Definimos heurística como sendo uma técnica inspirada em processos intuitivos que procura uma boa solução a um custo computacional aceitável, sem, no

entanto, estar capacitada a garantir sua otimalidade, bem como garantir quão próximo ela está da solução ótima.

O desafio é produzir, em tempo reduzido, soluções tão próximas quanto possível da solução ótima. Muitos esforços têm sido feitos nessa direção e heurísticas muito eficientes foram desenvolvidas para diversos problemas. Entretanto, a maioria delas é muito específica para um problema particular, não sendo eficientes (ou mesmo aplicáveis) na resolução de uma classe mais ampla de problemas.

Somente a partir da década de 1980 intensificaram-se os estudos no sentido de se desenvolver procedimentos heurísticos com uma certa estrutura teórica e com caráter mais geral, sem prejudicar a principal característica destes, que é a flexibilidade.

Esta meta tornou-se mais realista a partir da reunião de conceitos das áreas de Otimização e Inteligência Artificial, viabilizando a construção das chamadas melhores estratégias ou dos métodos "inteligentemente flexíveis", comumemente conhecidos como "meta-heurísticas".

Esses métodos, situados em domínios teóricos ainda pouco explorados pela literatura, possuem como característica básica estruturas com uma menor rigidez que as encontradas nos métodos clássicos de otimização sem, contudo, emergir em uma flexibilidade caótica.

Dentre os procedimentos enquadrados como meta-heurísticas que surgiram ao longo das últimas décadas, destacam-se: Algoritmos Genéticos (AGs), *Simulated Annealing* (SA), Busca Tabu (BT, *Tabu Search*), *Greedy Randomized Adaptive Search Procedures* (GRASP), Busca Local Iterada (ILS, *Iterated Local Search*), Busca em Vizinhança Variável (VNS, *Variable Neighborhood Search*), *Late Acceptance Hill-Climbing* (LAHC), Colônia de Formigas (ACO, *Ant Colony Optimization*), Busca Dispersa (SS, *Scatter Search*) e Otimização por Nuvem de Partículas (DPSO, *Discrete Particle Swarm Optimization*).

As duas primeiras meta-heurísticas fundamentam-se em analogias com processos naturais, sendo que os AGs são procedimentos inspirados em princípios da evolução natural. O SA explora uma analogia com a termodinâmica, enquanto a BT faz uso de estruturas de memória para explorar o espaço de soluções. Como será visto mais adiante, cada meta-heurística utiliza uma estratégia diferente para explorar o espaço de soluções de problemas de otimização.

O restante destas notas de aula estão organizadas como segue. Na Seção 2 são apresentadas as heurísticas construtivas, destinadas à geração de uma solução inicial para um problema de otimização. Na Seção 3 são apresentadas as heurísticas clássicas de refinamento, destinadas à melhoria de uma solução. Na Seção 4 são apresentadas as principais meta-heurísticas referenciadas na literatura e na última seção, técnicas de intensificação e diversificação.

2 Heurísticas Construtivas

Uma heurística construtiva tem por objetivo construir uma solução, elemento por elemento. A forma de escolha de cada elemento a ser inserido a cada passo varia de acordo com a função de avaliação adotada, a qual, por sua vez, depende do problema abordado. Nas heurísticas clássicas, os elementos candidatos são geralmente ordenados segundo uma função gulosa, que estima o benefício da inserção de cada elemento, e somente o "melhor" elemento é inserido a cada passo.

A Figura 1 mostra o pseudocódigo para a construção de uma solução inicial para um problema de otimização que utiliza uma função gulosa $g(.)$. Nesta figura, t_{melhor} indica o membro do conjunto de elementos candidatos com o valor mais favorável da função de avaliação g, isto é, aquele que possui o menor valor de g no caso de o problema ser de minimização ou o maior valor de g no caso de o problema ser de maximização.

```
procedimento ConstrucaoGulosa(g(.),s);
1  s ← ∅;
2  Inicialize o conjunto C de elementos candidatos;
3  enquanto (C ≠ ∅) faça
4      g(t_melhor) = melhor{g(t) | t ∈ C};
5      s ← s ∪ {t_melhor};
6      Atualize o conjunto C de elementos candidatos;
7  fim-enquanto;
8  Retorne s;
fim ConstrucaoGulosa;
```

Figura 1: Heurística de construção gulosa de uma solução inicial

A forma de construir uma solução varia conforme o problema abordado. Para ilustrar o funcionamento de uma heurística construtiva utilizaremos o Problema da Mochila como exemplo. Neste problema, há uma mochila de capacidade b e um conjunto de n objetos que podem ser colocados na mochila. A cada objeto j está associado um peso w_j e um valor de retorno (benefício) p_j. Considerando a existência de uma unidade de cada objeto, o objetivo é determinar o conjunto de objetos que devem ser colocados na mochila de forma a maximizar o valor de retorno respeitando a capacidade da mochila.

Seja, então, uma mochila de capacidade $b = 23$ e os 5 objetos da tabela a seguir, com os respectivos pesos e benefícios.

Objeto (j)	1	2	3	4	5
Peso (w_j)	4	5	7	9	6
Benefício (p_j)	2	2	3	4	4

Para construir uma solução, adicionemos à mochila, a cada passo, o objeto mais valioso que não ultrapasse a capacidade da mochila. Em caso de empate,

escolheremos o objeto com menor peso. Reordenando os objetos de acordo com este critério, obtemos:

Objeto (j)	5	4	3	1	2
Peso (w_j)	6	9	7	4	5
Benefício (p_j)	4	4	3	2	2

Representemos uma solução s por um vetor binário de n posições.

Passo 1 : Adicionemos, primeiramente, o objeto 5, que traz o maior benefício p_j, isto é, é o mais valioso dentre todos os objetos
$s = (00001)^t$
$f(s) = 4$
Peso corrente da mochila = 6 < b = 23

Passo 2 : Adicionemos, agora, o objeto 4, que dentre os objetos remanescentes ainda não avaliados tem o maior valor de retorno p_j
$s = (00011)^t$
$f(s) = 8$
Peso corrente da mochila = 15 < b = 23

Passo 3 : Seguindo a estrutura de construção, adicionemos, agora, o objeto 3, para o qual p_3 é o maior dentre os valores de retorno p_3, p_2 e p_1
$s = (00111)^t$
$f(s) = 11$
Peso corrente da mochila = 22 < b = 23

Passo 4 : O objeto a ser alocado agora seria o primeiro. No entanto, esta alocação faria superar a capacidade da mochila. Neste caso, devemos tentar alocar o próximo objeto com o maior valor de p_j ainda não analisado, que é o objeto 2. Como também a alocação deste objeto faria superar a capacidade da mochila e não há mais objetos candidatos, concluímos que a solução anterior é a solução final, isto é: $s^* = (00111)^t$ com $f(s^*) = 11$.

Uma outra forma muito comum de se gerar uma solução inicial é escolher os elementos candidatos aleatoriamente. Isto é, a cada passo, o elemento a ser inserido na solução é aleatoriamente selecionado dentre o conjunto de elementos candidatos ainda não selecionados. A grande vantagem desta metodologia reside na simplicidade de implementação. Segundo testes empíricos, a desvantagem é a baixa qualidade, em média, da solução final produzida, que, em geral, requer um maior esforço computacional na fase de refinamento.

A Figura 2 mostra o pseudocódigo para a construção de uma solução inicial aleatória para um problema de otimização.

Ilustremos, agora, aplicações de heurísticas construtivas para o Problema do Caixeiro Viajante. Serão apresentadas três heurísticas: (a) Heurística do Vizinho Mais Próximo; (b) Heurística de Nemhauser e Bellmore e (c) Heurística da Inserção Mais Barata.

As heurísticas serão ilustradas considerando um exemplo com 6 cidades e as distâncias dadas pela tabela a seguir:

procedimento *ConstrucaoAleatoria*($g(.)$,s);
1 $s \leftarrow \emptyset$;
2 Inicialize o conjunto C de elementos candidatos;
3 enquanto ($C \neq \emptyset$) faça
4 Escolha aleatoriamente $t_{escolhido} \in C$;
5 $s \leftarrow s \cup \{t_{escolhido}\}$;
6 Atualize o conjunto C de elementos candidatos;
7 fim-enquanto;
8 Retorne s;
fim *ConstrucaoAleatoria*;

Figura 2: Heurística de construção aleatória de uma solução inicial

Cidade	1	2	3	4	5	6
1	0	2	1	4	9	1
2	2	0	5	9	7	2
3	1	5	0	3	8	6
4	4	9	3	0	2	5
5	9	7	8	2	0	2
6	1	2	6	5	2	0

(a) **Heurística do Vizinho Mais Próximo**:
Nesta heurística, parte-se da cidade origem e adiciona-se a cada passo a cidade k ainda não visitada cuja distância à última cidade visitada é a menor possível. O procedimento de construção termina quando todas as cidades forem visitadas, situação na qual é feita a ligação entre a última cidade visitada e a cidade origem.

No exemplo considerado, considerando-se a cidade 1 como a cidade origem, tem-se:

i) Passo 1: Adicione a cidade 3 à rota, já que sua distância à cidade 1 é a menor (A cidade 6 dista de mesmo valor, e também poderia ser escolhida).

ii) Passo 2: Adicione a cidade 4 à rota, já que sua distância à cidade 3 é a menor dentre as cidades ainda não visitadas (no caso, as cidades 2, 4, 5 e 6).

iii) Passo 3: Adicione a cidade 5 à rota, já que sua distância à cidade 4 é a menor dentre todas as cidades ainda não visitadas (no caso, as cidades 2, 5 e 6)

iv) Passo 4: Adicione a cidade 6 à rota, já que sua distância à cidade 5 é a menor dentre todas as cidades ainda não visitadas (no caso, as cidades 2 e 6)

v) Passo 5: Adicione a cidade 2 à rota, já que esta é a única cidade ainda não visitada

vi) Passo 6: Faça a ligação da cidade 2 (última cidade visitada) à cidade 1 (cidade origem)

Ao final destes seis passos teremos produzido a solução $s = (1\ \ 3\ \ 4\ \ 5\ \ 6\ \ 2)$. Para esta solução, a distância total percorrida é:

$$dist = d_{13} + d_{34} + d_{45} + d_{56} + d_{62} + d_{21} = 1 + 3 + 2 + 2 + 2 + 2 = 12.$$

Complexidade da Heurística do Vizinho Mais Próximo:

Iteração	# operações	Observações
1	$n-1$	Há $n-1$ ligações para serem analisadas
2	$n-2$	Há $n-2$ ligações para serem analisadas
$\cdots$	$\cdots$	
$n-1$	1	Há apenas uma cidade ainda não visitada
Total	$1 + 2 + \cdots + (n-1) = n(n-1)/2$ operações	

A soma anterior é uma progressão aritmética cujo primeiro elemento a_1 é 1, último elemento a_{nelem} é $n-1$, a razão é igual a 1 e o número de termos $nelem$ é $n-1$. A soma dos termos vale $S = \left(\frac{a_1+a_{nelem}}{2}\right) nelem = \left(\frac{1+(n-1)}{2}\right)(n-1) = n(n-1)/2$. A heurística do vizinho mais próximo tem, portanto, complexidade $O(n^2)$.

(b) **Heurística de Bellmore e Nemhauser**:
Nesta heurística, adicionamos à rota corrente a cidade k ainda não visitada que esteja mais próxima dos extremos da subrota, isto é, a cidade k se liga a uma cidade que esteja em uma extremidade da subrota ou à outra extremidade.

No exemplo considerado, considerando-se a cidade 1 como a cidade origem, tem-se:

i) Passo 1: Adicione a cidade 3 à rota, já que sua distância à cidade 1 é a menor (A cidade 6 dista de mesmo valor, e também poderia ser escolhida em lugar da cidade 3).

ii) Passo 2: Das cidades ainda não visitadas (2, 4, 5 e 6), a cidade 6 é a que menos dista de um extremo da rota (cidade 1) e a cidade 4 é a que menos dista do outro extremo da rota (cidade 3). Como a distância $d_{61} = 1 < d_{34} = 3$, então a cidade 6 é a escolhida e deve ser conectada à cidade 1, isto é, a rota corrente passa a ser: $s = (6 \rightarrow 1 \rightarrow 3)$.

iii) Passo 3: Das cidades ainda não visitadas (2, 4 e 5), a cidade 2 é a que menos dista de um extremo da rota (cidade 6) e a cidade 4 é a que menos dista do outro extremo da rota (cidade 3). Como a distância $d_{26} = 2 < d_{34} = 3$, então a cidade 2 é a escolhida e deve ser conectada à cidade 6, isto é, a rota corrente passa a ser: $s = (2 \rightarrow 6 \rightarrow 1 \rightarrow 3)$. A cidade 5 também poderia ter sido escolhida para se conectar à cidade 6, pois tem a mesma distância da cidade 2 à cidade 6.

iv) Passo 4: Das cidades ainda não visitadas (4 e 5), a cidade 5 é a que menos dista de um extremo da rota (cidade 2) e a cidade 4 é a que menos dista do outro extremo da rota (cidade 3). Como a distância $d_{34} = 3 < d_{52} = 7$, então a cidade 4 é a escolhida e deve ser conectada à cidade 3, isto é, a rota corrente passa a ser: $s = (2 \rightarrow 6 \rightarrow 1 \rightarrow 3 \rightarrow 4)$.

v) Passo 5: A única cidade ainda não visitada é a cidade 5. Ela dista 7 unidades de um extremo da rota (cidade 2) e 2 unidades do outro extremo (cidade 4). Logo, a cidade 5 deve ser conectada à cidade 4, isto é, a rota corrente passa a ser: $s = (2 \rightarrow 6 \rightarrow 1 \rightarrow 3 \rightarrow 4 \rightarrow 5)$.

vi) Passo 6: Como todas as cidades já foram visitadas, resta agora somente conectar as duas extremidades (cidades 5 e 2) para formar um ciclo hamiltoniano.

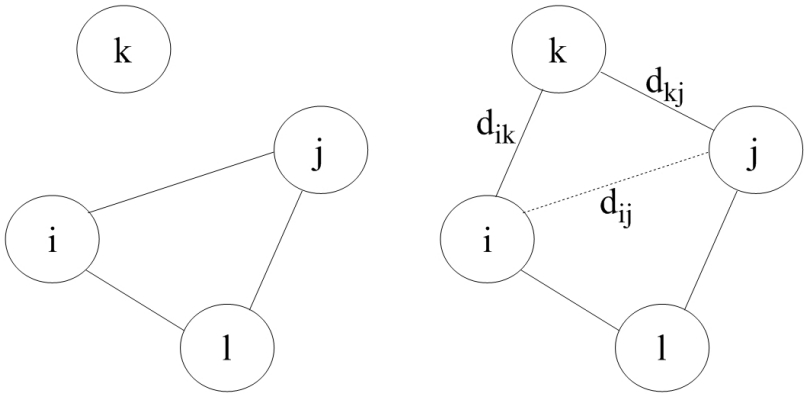
Ao final destes seis passos, teremos produzido a solução $s = (2\ \ 6\ \ 1\ \ 3\ \ 4\ \ 5)$. Para esta solução, a distância total percorrida é:

$$dist = d_{26} + d_{61} + d_{13} + d_{34} + d_{45} + d_{52} = 2 + 1 + 1 + 3 + 2 + 7 = 16.$$

(c) **Heurística da Inserção Mais Barata**:
Nesta heurística, parte-se de uma subrota inicial envolvendo três cidades e, a cada passo, adiciona-se uma cidade k ainda não visitada entre as cidades i e j da subrota cujo custo de inserção s_{ij}^k dado pela Eq. (1) seja o menor possível.

$$s_{ij}^k = d_{ik} + d_{kj} - d_{ij} \tag{1}$$

As figuras a seguir ilustram a inserção da cidade k entre as cidades i e j.



(a) Antes da inserção (b) Depois da inserção

A subrota inicial pode ser formada por um procedimento construtivo qualquer. Por exemplo, parta da cidade origem e adicione à subrota a cidade mais próxima dessa origem. A seguir, considerando as duas extremidades (cidade origem e última cidade inserida), adicione a cidade ainda não visitada cuja soma das distâncias às duas extremidades seja a menor.

No exemplo considerado, considerando-se a cidade 1 como a cidade origem, constrói-se uma solução com os seguintes passos:

i) Passo 1: Adicione a cidade 3 à rota, já que sua distância à cidade 1 é a menor (A cidade 6 tem a mesma distância, e também poderia ser escolhida).

ii) Passo 2: Das cidades ainda não visitadas (2, 4, 5 e 6), a cidade 2 é a aquela cuja distância às cidades extremas 1 e 3 é a menor, no caso, $d_{21} + d_{32} = 2 + 5 = 7$. Então, a cidade 2 é a escolhida e deve ser conectada às cidades 3 e 2, isto é, a subrota corrente é: $s = (1 \rightarrow 3 \rightarrow 2)$, com a cidade 2 ligada à cidade 1. Com os passos 2 e 3 encerra-se a construção de uma subrota inicial envolvendo três cidades. A distância total percorrida nessa solução parcial s é: $d(s) = d_{13} + d_{32} + d_{21} = 1 + 5 + 2 = 8$.

iii) Passo 3: Das cidades ainda não visitadas (4, 5 e 6), calculemos o custo de inserção s_{ij}^k entre todas as cidades i e j da subrota. A tabela a seguir mostra os custos de inserção.

i	k	j	$s_{ij}^k = d_{ik} + d_{kj} - d_{ij}$
1	4	3	$s_{13}^4 = 4 + 3$ - $1 = 6$
1	5	3	$s_{13}^5 = 9 + 8$ - $1 = 16$
1	6	3	$s_{13}^6 = 1 + 6$ - $1 = 6$
3	4	2	$s_{32}^4 = 3 + 9$ - $5 = 7$
3	5	2	$s_{32}^5 = 8 + 7$ - $5 = 7$
3	6	2	$s_{32}^6 = 6 + 2$ - $5 = 3$
2	4	1	$s_{21}^4 = 9 + 4$ - $2 = 11$
2	5	1	$s_{21}^5 = 7 + 9$ - $2 = 14$
2	6	1	$s_{21}^6 = 2 + 1$ - $2 = 1^*$

Como o menor custo de inserção é s_{21}^6, então a cidade 6 deve ser inserida entre as cidades 2 e 1. Logo, a subrota corrente passa a ser: $s = (1 \rightarrow 3 \rightarrow 2 \rightarrow 6)$. A distância associada a esta subrota é: $d(s) = d(s)_{\text{anterior}} + s_{21}^6 = 8 + 1 = 9$.

iv) Passo 4: Das cidades ainda não visitadas (4 e 5), calculemos o custo de inserção entre todas as cidades i e j da subrota corrente. A tabela a seguir mostra os custos de inserção.

i	k	j	$s_{ij}^k = d_{ik} + d_{kj} - d_{ij}$
1	4	3	$s_{13}^4 = 4 + 3$ - $1 = 6^*$
1	5	3	$s_{13}^5 = 9 + 8$ - $1 = 16$
3	4	2	$s_{32}^4 = 3 + 9$ - $5 = 7$
3	5	2	$s_{32}^5 = 8 + 7$ - $5 = 7$
2	4	6	$s_{26}^4 = 9 + 5$ - $2 = 12$
2	5	6	$s_{26}^5 = 7 + 2$ - $2 = 7$
6	4	1	$s_{61}^4 = 5 + 4$ - $1 = 8$
6	5	1	$s_{61}^5 = 2 + 9$ - $1 = 10$

Como o menor custo de inserção é s^4_{13}, então a cidade 4 deve ser inserida entre as cidades 1 e 3. Logo, a subrota corrente passa a ser: $s = (1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 6)$. A distância associada a esta subrota é: $d(s) = d(s)_{\text{anterior}} + s^4_{13} = 9 + 6 = 15$.

v) Passo 5: A única cidade ainda não visitada é a cidade 5. A tabela a seguir mostra os custos de inserção desta cidade entre todas as arestas da subrota corrente.

i	k	j	$s^k_{ij} = d_{ik} + d_{kj} - d_{ij}$
1	5	4	$s^5_{14} = 9 + 2$ - $4 = 7^*$
4	5	3	$s^5_{43} = 2 + 8$ - $3 = 7$
3	5	2	$s^5_{32} = 8 + 7$ - $5 = 10$
2	5	6	$s^5_{26} = 7 + 2$ - $2 = 7$
6	5	1	$s^5_{61} = 2 + 9$ - $1 = 10$

Como o menor custo de inserção é s^5_{14}, então a cidade 5 deve ser inserida entre as cidades 1 e 4. Logo, a rota resultante é: $s = (1 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 6)$. A distância associada a esta rota é: $d(s) = d(s)_{\text{anterior}} + s^5_{14} = 15 + 7 = 22$.

3 Heurísticas de Refinamento

As heurísticas de refinamento em problemas de otimização, também chamadas de técnicas de busca local, constituem uma família de técnicas baseadas na noção de vizinhança. Mais especificamente, seja S o espaço de pesquisa de um problema de otimização e f a função objetivo a minimizar. A função N, a qual depende da estrutura do problema tratado, associa a cada solução $s \in S$, sua vizinhança $N(S) \subseteq S$. Cada solução $s' \in N(s)$ é chamada de vizinho de s. Denomina-se *movimento* a modificação m que transforma uma solução s em outra, s', que esteja em sua vizinhança. Representa-se esta operação por $s' \leftarrow s \oplus m$.

Em linhas gerais, esta classe de heurísticas parte de uma solução inicial qualquer (a qual pode ser obtida por uma heurística construtiva ou então gerada aleatoriamente) e caminha, a cada iteração, de vizinho para vizinho de acordo com a definição de vizinhança adotada.

Conforme [34], um método de busca local pode ser visto como um procedimento que percorre um caminho em um grafo não-orientado $G = (S, E)$, onde S representa o conjunto de soluções s do problema e E o conjunto de arestas (s, s'), com $s' \in N(s)$.

A definição de vizinhança é crucial em uma heurística de refinamento. De uma solução s do espaço de soluções deve ser sempre possível atingir qualquer outra solução em um número finito de passos, utilizando um determinado tipo ou tipos de movimentos. Por exemplo, considere no problema da mochila as soluções $s^{(1)} = (01001)^t$ e $s^{(2)} = (11010)^t$ do espaço de soluções. Com o movimento $m = \{\text{trocar o valor de um bit}\}$ é possível navegar no espaço de soluções do problema de $s^{(1)}$ a $s^{(2)}$. De fato, com esse movimento m podemos percorrer

o seguinte caminho: $s^{(1)} = (01001)^t \rightarrow (11001)^t \rightarrow (11011)^t \rightarrow (11010)^t = s^{(2)}$. No entanto, se definíssimos o movimento m como sendo a troca de dois bits simultaneamente, jamais conseguiríamos de $s^{(1)}$ chegar a $s^{(2)}$. Desta forma, a exploração do espaço de soluções ficaria prejudicada e, eventualmente, a solução ótima poderia não ser alcançada.

Em muitos problemas combinatórios é também difícil até mesmo encontrar uma solução viável. Nessas situações, pode ser uma má idéia caminhar apenas no espaço das soluções viáveis do problema considerado. Para tais problemas, o espaço de busca pode incluir soluções inviáveis, obtidas a partir do relaxamento de algumas restrições do problema original. Para tanto, basta acrescentar à função de avaliação componentes que penalizam violações às restrições. Um exemplo típico é o problema de programação de horários de cursos universitários (*Course Timetabling Problem*). A restrição principal deste problema requer que as aulas dadas pelo mesmo professor para turmas distintas não se sobreponham, isto é, que não sejam realizadas no mesmo horário. Considerando como movimento m a mudança das aulas de um curso de um horário para outro, dificilmente geraríamos quadros de horários sem situações de sobreposição. Relaxando-se estas restrições e penalizando-as na função de avaliação, torna-se muito mais eficiente a exploração do espaço de busca [32].

Exemplifiquemos, agora, como gerar diferentes estruturas de vizinhança. Para tanto, consideremos o Problema do Caixeiro Viajante, para o qual representamos uma solução s por um vetor de n posições, sendo que em cada posição i tem-se a cidade s_i.

Com o movimento m de troca de posição entre duas cidades definimos a estrutura de vizinhança $N^{(T)}$. Assim, $s = (s_1\ s_2\ s_3\ \cdots\ s_n)^t$ tem como vizinhos em $N^{(T)}(s)$ as seguintes soluções: $s^{(1)} = (\mathbf{s_2}\ \mathbf{s_1}\ s_3\ \cdots\ s_n)^t$, $s^{(2)} = (\mathbf{s_3}\ s_2\ \mathbf{s_1}\ \cdots\ s_n)^t$, $\cdots$, $s^{(n-1)} = (\mathbf{s_n}\ s_2\ s_3\ \cdots\ \mathbf{s_1})^t$, $s^{(n)} = (s_1\ \mathbf{s_3}\ \mathbf{s_2}\ \cdots\ s_n)^t$, $\cdots$, $s^{(n\times(n-1)/2)} = (s_1\ s_2\ s_3\ \cdots\ \mathbf{s_n}\ \mathbf{s_{n-1}})^t$.

Por outro lado, considerando como movimento m a realocação de uma cidade de uma posição na seqüência de visita para outra, definimos a estrutura de vizinhança $N^{(R)}$. Nesta estrutura, são vizinhos de $s = (s_1\ s_2\ s_3\ \cdots\ s_{n-1}\ s_n)^t$ as seguintes soluções: $s^{(1)} = (s_2\ \mathbf{s_1}\ s_3\ \cdots\ s_{n-1}\ s_n)^t$, $s^{(2)} = (s_2\ s_3\ \mathbf{s_1}\ \cdots\ s_{n-1}\ s_n)^t$, $\cdots$, $s^{(n-2)} = (s_2\ s_3\ \cdots\ s_{n-1}\ \mathbf{s_1}\ s_n)^t$, $s^{(n-1)} = (s_2\ s_3\ \cdots\ s_{n-1}\ s_n\ \mathbf{s_1})^t$, $s^{(n)} = (s_1\ s_3\ \mathbf{s_2}\ \cdots\ s_{n-1}\ s_n)^t$, $s^{(n+1)} = (s_1\ s_3\ s_4\ \mathbf{s_2}\ \cdots\ s_{n-1}\ s_n)^t$, $\cdots$, $s^{(2n-4)} = (s_1\ s_3\ s_4\ \cdots\ s_{n-1}\ \mathbf{s_2}\ s_n)^t$, $s^{(2n-3)} = (s_1\ s_3\ s_4\ \cdots\ s_{n-1}\ s_n\ \mathbf{s_2})^t$, $\cdots$, $s^{(2n-5)} = (s_1\ \mathbf{s_3}\ s_2\ s_4\ \cdots\ s_{n-1}\ s_n)^t$, $\cdots$, $s^{(n-2)^2\times(n-1)} = (s_1\ s_2\ s_3\ s_4\ \cdots\ \mathbf{s_n}\ s_{n-1})^t$.

Poderíamos, também, definir como vizinhança de uma solução s o conjunto de vizinhos gerados tanto por movimentos de troca quanto por movimentos de realocação, isto é, $N(s) = N^{(T)}(s) \cup N^{(R)}(s)$.

Há outros movimentos mais elaborados, tal como o movimento *k-Or*, com $k \geq 2$, que consiste em realocar um bloco contíguo de k cidades em outra posição da seqüência. Por exemplo, considerando a solução $s = (4\ 3\ 1\ 2)^t$ e blocos de tamanho $k = 2$, teríamos os seguintes vizinhos para s: $s_1^{'} = (1\ 4\ 3\ 2)^t$, $s_2^{'} = (4\ 2\ 3\ 1)^t$, $s_3^{'} = (4\ 1\ 2\ 3)^t$. Neste exemplo, o primeiro vizinho é gerado pela inserção do bloco (4 3) entre as cidades 1 e 2, o segundo vizinho, pela

inserção do bloco (3 1) entre as cidades 2 e 4 e, finalmente, o terceiro vizinho, pela inserção do bloco (1 2) entre as cidades 4 e 3.

Nas seções 3.1 a 3.4 apresentamos heurísticas clássicas de refinamento, enquanto na Seção 3.5 é apresentada uma heurística de refinamento mais sofisticada, que explora o espaço de soluções de um problema de otimização por meio de trocas sistemáticas de vizinhanças.

3.1 Método da Descida/Subida (Descent/Uphill Method)

A idéia desta técnica é partir de uma solução inicial qualquer e a cada passo analisar todos os seus possíveis vizinhos, movendo somente para aquele que representar uma melhora no valor atual da função de avaliação. Pelo fato de analisar **todos** os vizinhos e escolher o melhor, esta técnica é comumente referenciada na literatura inglesa por *Best Improvement Method* (BI), ou *Steepest Descent method* (SD). O método se encerra quando um ótimo local é encontrado.

A Figura 3 mostra o pseudocódigo do Método de Descida aplicado à minimização de uma função de avaliação f a partir de uma solução inicial conhecida s e considerando a busca em uma dada vizinhança $N(.)$.

procedimento *Descida*($f(.)$,$N(.)$,s);
1 $V = \{s' \in N(s) \mid f(s') < f(s)\}$;
2 enquanto ($|V| > 0$) faça
3 Selecione $s' \in V$, onde $s' = \arg\min\{f(s') \mid s' \in V\}$;
4 $s \leftarrow s'$;
5 $V = \{s' \in N(s) \mid f(s') < f(s)\}$;
6 fim-enquanto;
7 Retorne s;
fim *Descida*;

Figura 3: Método da Descida

Para o problema da mochila, representemos uma solução s por um vetor binário de n posições e consideremos como movimento m a troca do valor de um bit. Assim, a vizinhança de uma solução s, e se escreve $N(s)$, é o conjunto de todos os vizinhos s' que diferem de s pelo valor de um bit. Formalmente, representamos $N(s) = \{s' : s' \leftarrow s \oplus m\}$, onde m significa a troca do valor de um bit. É necessário, agora, definir uma função de avaliação para guiar a busca no espaço de soluções do problema. Considerando que se deseja maximizar o valor de retorno trazido pela utilização de cada item, há duas possibilidades para a escolha dessa função.

A primeira possibilidade é considerar a exploração apenas no espaço de soluções viáveis. Neste caso, a função de avaliação coincide com a própria função objetivo do problema, isto é:

$$f(s) = \sum_{j=1}^{n} p_j s_j \tag{2}$$

com s satisfazendo à condição de que $\sum_{j=1}^{n} w_j s_j \leq b$.

Outra possibilidade é permitir a geração de soluções inviáveis. Neste caso, uma função de avaliação apropriada seria:

$$f(s) = \sum_{j=1}^{n} p_j s_j - \alpha \times \max\{0, \sum_{j=1}^{n} w_j s_j - b\} \tag{3}$$

sendo α uma penalidade, por exemplo, $\alpha = \sum_{j=1}^{n} p_j = 15$.

Observe que o objetivo da segunda parcela desta última função de avaliação é penalizar a colocação na mochila de objetos que ultrapassam sua capacidade. Como a função de avaliação f deve ser maximizada, o sinal desta segunda parcela é negativo de forma a não incentivar a realização de movimentos que gerem soluções inviáveis. O valor de α deve ser suficientemente grande para atender a este objetivo.

Apliquemos esta heurística à instância do problema dado, considerando a possibilidade de geração de soluções infactíveis.

Passo 0 : Seja uma solução inicial qualquer, por exemplo:
$s = (01010)^t$
$f(s) = 6$
Peso corrente da mochila = 14

Passo 1 : Devemos, agora, analisar todos os vizinhos de s e calcular a função de avaliação deles por meio da função (3).

Vizinhos de s	Peso total dos vizinhos de s	Benefício total dos vizinhos de s	$f(s')$
$(\mathbf{1}1010)^t$	18	8	8
$(0\mathbf{0}010)^t$	9	4	4
$(01\mathbf{1}10)^t$	21	9	9
$(010\mathbf{0}0)^t$	5	2	2
$(01011)^t$	20	10	10

Melhor vizinho: $s' = (01011)^t$
$f(s') = 10$
Como s' é melhor que s, pois $f(s') > f(s)$, então $s \leftarrow s'$, isto é, a nova solução corrente passa a ser:
$s = (01011)^t$

Passo 2 : Determinemos, agora, o melhor vizinho de $s = (01011)^t$:

Vizinhos de s	Peso total dos vizinhos de s	Benefício total dos vizinhos de s	$f(s')$
$(\mathbf{1}1011)^t$	24	12	-3
$(0\mathbf{0}011)^t$	15	8	8
$(011\mathbf{1}1)^t$	27	13	-47
$(0100\mathbf{1})^t$	11	6	6
$(0101\mathbf{0})^t$	14	6	6

Melhor vizinho: $s' = (00011)^t$
$f(s') = 8$
Como $f(s')$ é pior que $f(s)$, pois $f(s') < f(s)$, então PARE. A solução anterior é um ótimo local, isto é, o método da subida retorna $s^* = (01011)^t$, com $f(s^*) = 10$ como solução final.

No exemplo anterior, caso fosse utilizada a função de avaliação (2), as soluções anteriores que ultrapassaram a capacidade da mochila, isto é, $(\mathbf{1}1011)^t$) e $(011\mathbf{1}1)^t$, sequer seriam avaliadas!

É importante observar que diferentes soluções iniciais conduzem, na maioria das vezes, a diferentes soluções finais. A Figura 4, em que s indica um ponto de partida e s^* um ótimo local encontrado a partir da aplicação do Método da Descida a s, ilustra esta situação.

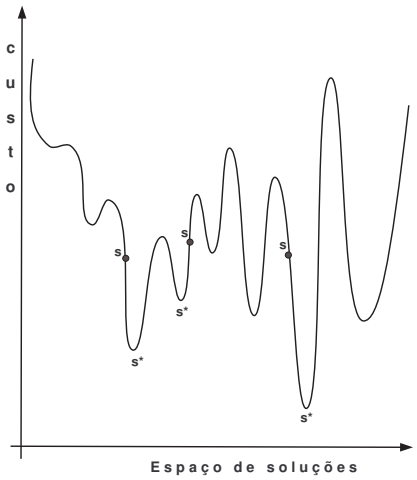


Figura 4: Representação esquemática do funcionamento do Método da Descida

Uma outra função de avaliação que poderia ser considerada em substituição à da Eq. (3) é a seguinte:

$$f(s) = \sum_{j=1}^{n} p_j s_j - \alpha \times f_1(s) \tag{4}$$

$$\text{sendo } f_1(s) = \begin{cases} 1 & \text{se } \sum_{j=1}^{n} w_j s_j - b > 0 \\ 0 & \text{se } \sum_{j=1}^{n} w_j s_j - b \leq 0 \end{cases}$$

e α uma penalidade, definida como antes, por exemplo, $\alpha = \sum_{j=1}^{n} p_j = 15$.

Nesta formulação não se faz diferença entre o nível de inviabilidade, pois qualquer que seja o excesso de peso na mochila, a penalização é a mesma. Esta modelagem pode dificultar a exploração do espaço de soluções, pois conduz a regiões planas, ditas platôs, regiões nas quais as heurísticas têm dificuldade para escapar. Em [34] os autores argumentam que uma forma comum de evitar esta topologia no espaço de busca é adicionar componentes à função de avaliação de forma a discriminar soluções que teriam o mesmo valor da função custo original. Assim, no exemplo mencionado da mochila, das duas funções de avaliação apresentadas, a mais adequada para guiar a busca é a da Eq. (3).

3.2 Método de Primeira Melhora

O método de descida/subida requer a exploração de toda a vizinhança. Um método alternativo, que evita esta pesquisa exaustiva e muito utilizado na literatura é o Método de Primeira Melhora, ou *First Improvement Method* (FI), na nomenclatura inglesa. Nesta variante, interrompe-se a exploração da vizinhança quando um vizinho melhor é encontrado. Como toda a vizinhança é explorada ao não se encontrar um vizinho melhor, então, tal como no método da Descida/Subida, este método fica preso no primeiro ótimo local encontrado.

É desejável neste método que a ordem de exploração das soluções vizinhas seja alterada a cada passo; do contrário, privilegia-se apenas um caminho determinístico no espaço de soluções. Por exemplo, em um problema de programação de tripulações, imagine que um movimento consista em realocar uma tarefa de uma tripulação para outra tripulação. Se a cada passo, a ordem de exploração das soluções vizinhas começar com a movimentação da primeira tarefa do primeiro tripulante para o segundo tripulante, depois para o terceiro, a seguir para o quarto e assim sucessivamente, então os tripulantes iniciais serão privilegiados. Para evitar isso, uma alternativa é utilizar uma seqüência diferente de tripulações a cada iteração do método. Assim, na primeira iteração do método poder-se-ia usar uma seqüência de tripulantes, por exemplo, (1,2,3,4, $\cdots$,n); outra seqüência diferente na segunda iteração, por exemplo, (7,4,1,9, $\cdots$) e assim por diante. Observe que, desta forma, diferentes ótimos locais podem ser encontrados se forem realizadas várias aplicações do método. Esse resultado está coerente com o fato de que um caminho de busca diferente pode ser encontrado

a cada aplicação do método, conduzindo a um ótimo local diferente, em função da ordem de exploração da vizinhança.

3.3 Método de Descida/Subida Randômica

Como dito anteriormente, o Método de Descida/Subida requer a exploração de toda a vizinhança. Outro método alternativo, que evita esta pesquisa exaustiva é o Método de Descida/Subida Randômica (*Random Descent/Uphill Method*). Ele consiste em analisar um vizinho qualquer e o aceitar somente se ele for estritamente melhor que a solução corrente; não o sendo, a solução corrente permanece inalterada e outro vizinho é gerado. O procedimento é interrompido após um número fixo de iterações sem melhora no valor da melhor solução obtida até então.

Como neste método não é feita a exploração de toda a vizinhança da solução corrente, não há garantia de que a solução final seja um ótimo local com relação à vizinhança explorada.

Na Figura 5 mostra-se o pseudocódigo do Método de Descida Randômica aplicado ao refinamento de uma solução s em um problema de minimização de uma função $f(.)$, utilizando uma estrutura de vizinhança $N(.)$. Nesta figura, *IterMax* representa o número máximo de iterações sem melhora no valor da solução corrente, e é um parâmetro a ser calibrado.

```
procedimento DescidaRandomica(f(.),N(.),IterMax,s);
1  Iter ← 0;      {Contador de iterações sem melhora }
2  enquanto (Iter < IterMax) faça
3      Iter ← Iter + 1;
4      Selecione aleatoriamente s' ∈ N(s);
5      se (f(s') < f(s)) então
6          Iter ← 0;
7          s ← s' ;
8      fim-se;
9  fim-enquanto;
10 Retorne s;
fim DescidaRandomica;
```

Figura 5: Método de Descida Randômica

3.4 Método Não Ascendente/Descendente Randômico

O método Não Ascendente Randômico (RNA, *Randomized Non-Ascendent method*) (respectivamente, Método Não Descendente Randômico - RND, *Randomized non-descendent method*) é uma variante do método de descida randômica (respectivamente, método da subida), diferindo dele por aceitar o vizinho gerado aleatoriamente se ele for melhor ou igual à solução corrente. Este método pára,

também, após um número fixo de iterações sem melhora no valor da melhor solução produzida.

Por este método é possível navegar pelo espaço de busca por *movimentos laterais* [56]. Assim, ele tem condições de percorrer caminhos de descida/subida que passam por regiões planas. Ou seja, se a busca chega em um região dessas, o método tem condições de mover-se nela e sair dela por meio de uma solução diferente daquela que a ela chegou.

O método RNA/RND é, portanto, um procedimento que explora o espaço de soluções combinando movimentos de descida/subida com movimentos laterais. Tal como no método de descida randômica, não há garantia de que a solução final seja um ótimo local.

3.5 Descida em Vizinhança Variável

O Método de Descida em Vizinhança Variável (*Variable Neighborhood Descent*, VND), proposto por Nenad Mladenović e Pierre Hansen [42], é um método de refinamento que consiste em explorar o espaço de soluções por meio de trocas sistemáticas de estruturas de vizinhança, aceitando somente soluções de melhora da solução corrente e retornando à primeira estrutura quando uma solução melhor é encontrada.

Segundo os autores, o método VND baseia-se em três princípios básicos:

- Um ótimo local com relação a uma dada estrutura de vizinhança não corresponde necessariamente a um ótimo local com relação a uma outra estrutura de vizinhança;
- Um ótimo global corresponde a um ótimo local para todas as estruturas de vizinhança;
- Para muitos problemas, ótimos locais com relação a uma ou mais estruturas de vizinhança são relativamente próximas.

Ainda de acordo os autores, o último princípio, de natureza empírica, indica que um ótimo local freqüentemente fornece algum tipo de informação sobre o ótimo global. Este é o caso em que os ótimos local e global compartilham muitas variáveis com o mesmo valor, o que sugere uma investigação sistemática da vizinhança de um ótimo local até a obtenção de uma nova solução de melhor valor.

O pseudocódigo deste método, em que se considera o refinamento de uma solução s utilizando uma função de avaliação f a ser minimizada e um conjunto de r diferentes vizinhanças $N = \{N^{(1)}, N^{(2)}, \cdots, N^{(r)}\}$, é apresentado pela Figura 6.

A ordem de exploração das r vizinhanças $(N^{(1)}, N^{(2)}, \cdots, N^{(r)})$ deve ser calibrada por meio de um projeto de experimentos (*Design of Experiments – DoE*) ou por ferramentas específicas para esse fim, como o *Irace* [37]. Entretanto, é também muito comum ordenar as vizinhanças de acordo com a sua complexidade

```
procedimento VND(f(.),N(.),r,s)
1  Seja r o número de estruturas diferentes de vizinhança;
2  k ← 1;                {Tipo de estrutura de vizinhança corrente}
3  enquanto (k ≤ r) faça
4      Encontre o melhor vizinho s' ∈ N^(k)(s);
5      se (f(s') < f(s))
6          então
7              s ← s';
8              k ← 1;
9          senão
10             k ← k + 1;
11     fim-se;
12 fim-enquanto;
13 Retorne s;
fim VND;
```

Figura 6: Algoritmo *VND*

de exploração. Isto é, vizinhanças que usam movimentos de menor complexidade devem ser exploradas antes daquelas que utilizam movimentos de maior complexidade, procedimento que é frequentemente utilizado na literatura [31].

Dependendo do problema abordado, a busca pelo melhor vizinho (linha 4 da Figura 6) pode ser muito custosa computacionalmente. Nesta situação é comum fazer a busca pela primeira solução de melhora (vide o método *First Improvement* da Seção 3.2, à página 14). Outra alternativa é considerar a exploração apenas em um certo percentual da vizinhança, isto é, analisar apenas um subconjunto V da vizinhança da solução corrente, isto é, $V \subseteq N^{(k)}(s)$, e retornar o melhor vizinho dessa vizinhança.

Uma maneira de implementar essa última estratégia para um problema de minimização está ilustrada na Figura 7. Nessa figura, $RDmax_k$ é o número máximo de vizinhos analisados na k-ésima vizinhança, que pode ser estimado por $RDmax_k = Perc_k \times SizeNeighborhood_k$, sendo $Perc_k$ o percentual a ser analisado da k-ésima vizinhança e $SizeNeighborhood_k$ o tamanho dessa vizinhança. Dessa forma, $Perc_k$ é um parâmetro do método; por exemplo, $Perc = 0{,}01$ indica que a busca local é feita apenas em 1% da k-ésima vizinhança.

O parâmetro $RDmax$ na Figura 7 deve ter seu valor fixado em um número inferior ao de vizinhos da solução corrente; do contrário, se o método estiver em um ótimo local, seus vizinhos podem ser visitados mais de uma vez. Ao contrário do VND tradicional, vide Algoritmo 6, o VND-RD não garante que a solução final seja ótima com relação às vizinhanças usadas para explorar o espaço de soluções do problema.

```
procedimento VND-RD(f(.), N(.), r, RDmax_k, s)
1  Seja r o número de estruturas diferentes de vizinhança;
2  Seja RDmax_k o número máximo de iterações na k-ésima vizinhança;
3  k ← 1;              {Tipo de estrutura de vizinhança corrente}
4  enquanto (k ≤ r) faça
5      iterRD ← 1;
6      s' ← s;
7      enquanto (iterRD ≤ RDmax_k) faça
8          Selecione, aleatoriamente, um vizinho s'' ∈ N^(k)(s');
9          se (f(s'') < f(s')) então
10             s' ← s'';
11         fim-se;
12         iterRD ← iterRD + 1;
13     fim-enquanto;
14     se (f(s') < f(s))
15         então
16             s ← s';
17             k ← 1;
18         senão
19             k ← k + 1;
20     fim-se;
21 fim-enquanto;
22 Retorne s;
fim VND-RD;
```

Figura 7: Algoritmo *VND usando descida randômica*

4 Meta-heurísticas

As meta-heurísticas são procedimentos destinados a encontrar uma boa solução, eventualmente a ótima, consistindo na aplicação, em cada passo, de uma heurística subordinada, a qual tem que ser modelada para cada problema específico [52].

Contrariamente às heurísticas convencionais, as meta-heurísticas são de caráter geral e providas de mecanismos para evitar que se fique preso em ótimos locais possivelmente distantes dos ótimos globais.

As meta-heurísticas diferenciam-se entre si basicamente pelo mecanismo usado para sair das armadilhas dos ótimos locais. Elas se dividem em duas categorias, de acordo com o princípio usado para explorar o espaço de soluções: busca local (ou trajetória) e busca populacional.

Nas meta-heurísticas baseadas em busca local, a exploração do espaço de soluções é feita por meio de movimentos, os quais são aplicados a cada passo sobre a solução corrente, gerando outra solução promissora em sua vizinhança. Busca Tabu, *Simulated Annealing*, Busca em Vizinhança Variável (*Variable Neighborhood Search*) e *Iterated Local Search* são exemplos de métodos que se enqua-

dram nesta categoria.

Os métodos baseados em busca populacional, por sua vez, consistem em manter um conjunto de boas soluções e combiná-las de forma a tentar produzir soluções ainda melhores. Exemplos clássicos de procedimentos desta categoria são os Algoritmos Genéticos, os Algoritmos Meméticos e o Algoritmo Colônia de Formigas.

Apresentamos, a seguir, algumas das principais meta-heurísticas referenciadas na literatura.

4.1 *Multi-Start*

A meta-heurística *Multi-Start* [40] consiste em fazer amostragens do espaço de soluções, aplicando a cada solução gerada um procedimento de refinamento. As amostras são obtidas por meio da geração de soluções aleatórias. Com este procedimento, há uma diversificação no espaço de busca, possibilitando não ficar preso em ótimos locais. A grande vantagem do método é que ele é de fácil implementação.

Na Figura 8 apresenta-se o pseudocódigo de um procedimento *Multi-Start* básico para um problema de minimização. Um número máximo de iterações ou um tempo máximo de processamento é normalmente utilizado como critério de parada.

```
procedimento MultiStart(f(.),N(.),CriterioParada, s)
1   f* ← ∞;            {Valor associado a s* }
2   enquanto (Critério de parada não atendido) faça
3       s ← ConstruaSolucao();   {Gere uma solução s do espaço de soluções}
4       s ← BuscaLocal(s);       {Aplique um procedimento de melhora em s}
5       se (f(s) < f(s*)) então
6           s* ← s;
7           f* ← f(s);
8       fim-se;
9   fim-enquanto;
10  s ← s*;
11  Retorne s;
fim MultiStart;
```

Figura 8: Pseudocódigo da meta-heurística *Multi-Start*

Uma variação comum no procedimento *Multi-Start* consiste em partir de uma solução inicial gerada por um procedimento construtivo guloso. Assim, na Figura 8 cria-se uma linha 0 e substitui-se a linha 1, tal como se segue:

```
0   s* ← ConstruaSolucaoGulosa(); {Melhor solução até então}
1'  f* ← f(s*);                   {Valor associado a s* }
```

O procedimento *ContruaSolucaoGulosa*() encontra-se descrito na Figura 1, à página 3 destas notas de aula.

4.2 *Simulated Annealing*

Trata-se de uma técnica de busca local probabilística, proposta originalmente por Kirkpatrick *et al.* [35], que se fundamenta em uma analogia com a termodinâmica, ao simular o resfriamento de um conjunto de átomos aquecidos, operação conhecida como recozimento [12].

Esta técnica começa sua busca a partir de uma solução inicial qualquer. O procedimento principal consiste em um *loop* que gera aleatoriamente, em cada iteração, um único vizinho s' da solução corrente s.

Considerando um problema de minimização, seja Δ a variação de valor da função objetivo ao mover-se para uma solução vizinha candidata, isto é, $\Delta = f(s') - f(s)$. O método aceita o movimento e a solução vizinha passa a ser a nova solução corrente se $\Delta < 0$. Caso $\Delta \geq 0$ a solução vizinha candidata também poderá ser aceita, mas neste caso, com uma probabilidade $e^{-\Delta/T}$, onde T é um parâmetro do método, chamado de *temperatura* e que regula a probabilidade de se aceitar soluções de pior custo.

A temperatura T assume, inicialmente, um valor elevado T_0. Após um número fixo de iterações (o qual representa o número de iterações necessárias para o sistema atingir o equilíbrio térmico em uma dada temperatura), a temperatura é gradativamente diminuída por uma razão de resfriamento α, tal que $T_k \leftarrow \alpha \times T_{k-1}$, sendo $0 < \alpha < 1$. Com esse procedimento, dá-se, no início uma chance maior para não ficar preso em mínimos locais e, à medida que T aproxima-se de zero, o algoritmo comporta-se como o método de descida, uma vez que diminui a probabilidade de se aceitar movimentos de piora $\left(T \to 0 \Longrightarrow e^{-\Delta/T} \to 0\right)$

A Figura 9 mostra a influência da variação da temperatura na função de probabilidade.

Para melhor entendimento desta função de probabilidade, considera-se que a variação de energia Δ é a mesma durante toda a busca, no caso, Δ é fixado em uma unidade. Observe que no início do processo, quando a temperatura é elevada, a função de probabilidade assume valores próximos à unidade, enquanto que no final do processo, quando a temperatura se aproxima de zero, o valor da função de probabilidade também se aproxima de zero.

O procedimento é interrompido quando a temperatura chega a um valor próximo de zero e nenhuma solução de piora da solução corrente é mais aceita, isto é, quando o sistema está estável. A solução obtida quando o sistema encontra-se nesta situação evidencia o encontro de um ótimo local.

Os parâmetros de controle do procedimento são a razão de resfriamento α, o número de iterações para cada temperatura ($SAmax$) e a temperatura inicial T_0.

Apresenta-se, pela Figura 10, o algoritmo *Simulated Annealing* básico aplicado a um problema de minimização.

Observamos que no caso de o problema ser de maximização, as seguintes modificações devem ser feitas na Figura 10: Na linha 9, considerar que $\Delta > 0$;

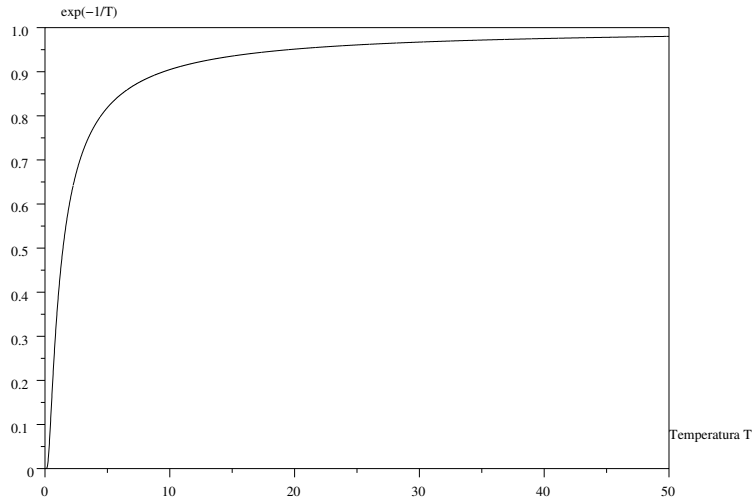


Figura 9: Comportamento da função de probabilidade

na linha 12, substituir pelo teste $(f(s') > f(s^\star))$ e, finalmente, na linha 15, substituir por $(x < e^{\Delta/T})$.

Dependendo do processo de resfriamento, pode ser mostrada a convergência do método a uma solução que seja globalmente ótima [12]. Para tal, a temperatura na iteração k do método, dada por T_k, deve ser calculada com base na expressão (5):

$$T_k = \frac{c}{\ln(1+k)} \tag{5}$$

em que c é da ordem do valor do ótimo local mais profundo (no caso de o problema ser de minimização) ou mais elevado (no caso de o problema ser de maximização). A convergência é garantida quando $k \leftarrow \infty$.

Tal resultado, entretanto, é de utilidade prática restrita, uma vez que o resfriamento é muito lento, requerendo um número proibitivo de iterações do método.

Há várias outras formas de fazer o resfriamento, além do geométrico. Uma alternativa é fazer o decaimento da temperatura por meio da expressão (6):

$$T_k = \frac{T_{k-1}}{1+\gamma\sqrt{T_{k-1}}} \;\; \forall k \geq 1 \tag{6}$$

em que T_k representa a temperatura na iteração k do método, isto é, na k-ésima vez em que há alteração no valor da temperatura e γ uma constante tal que $0 < \gamma < 1$. Valores de γ próximos a zero indicam resfriamento muito lento.

Outra alternativa, usada em [43], consiste em fazer o resfriamento por meio da expressão:

```
procedimento SA(f(.),N(.),α,SAmax,T0,s)
1  s* ← s;                 {Melhor solução obtida até então}
2  IterT ← 0;              {Número de iterações na temperatura T}
3  T ← T0;                 {Temperatura corrente}
4  enquanto (T > 0) faça
5      enquanto (IterT < SAmax) faça
6          IterT ← IterT + 1;
7          Gere um vizinho qualquer s' ∈ N(s);
8          Δ = f(s') − f(s);
9          se (Δ < 0)
10             então
11                 s ← s';
12                 se (f(s') < f(s*)) então s* ← s';
13             senão
14                 Tome x ∈ [0,1];
15                 se (x < e^(−Δ/T)) então s ← s';
16         fim-se;
17     fim-enquanto;
18     T ← α × T;
19     IterT ← 0;
20 fim-enquanto;
21 s ← s*;
22 Retorne s;
fim SA;
```

Figura 10: Algoritmo *Simulated Annealing*

$$T_k = \begin{cases} \beta T_{k-1} & \text{se } k = 1 \\ \dfrac{T_{k-1}}{1 + \gamma T_{k-1}} & \text{se } k \geq 2 \end{cases} \tag{7}$$

sendo $\gamma = \frac{T_0 - T_{k-1}}{(k-1)T_0 T_{k-1}}$, T_0 a temperatura inicial, T_k a temperatura na k-ésima iteração e β um parâmetro para corrigir a imperfeição do resfriamento ($0 < \beta < 1$). Igualmente, valores de γ próximos a zero indicam resfriamento muito lento.

Algoritmos baseados em *SA* normalmente incluem reaquecimento, seguido de novo resfriamento, quando a quantidade de movimentos consecutivamente rejeitados é alta [12]. É comum, também, trabalhar nas temperaturas mais altas com uma taxa de resfriamento menor e aumentá-la quando a temperatura reduzir-se.

A estimação do número máximo de iterações em uma dada temperatura, isto é, $SAmax$, normalmente é feita em função das dimensões do problema tratado. Por exemplo, em um problema de programação de horários em escolas (*school timetabling*), envolvendo n turmas, m professores e p horários reservados para a realização das aulas, o valor de $SAmax$ pode ser estimado em $SAmax =$

$k \times p \times m \times n$, sendo k uma constante a determinar. Já em um problema de programação de tripulações (*crew scheduling*) envolvendo *ntrip* tripulantes e *ntarefas* tarefas, $SAmax$ é estimado em $SAmax = k \times ntrip \times ntarefas$.

Há pelo menos duas prescrições para a determinação autoadaptativa da temperatura inicial: por simulação ou pelo custo das soluções.

A primeira delas consiste em determinar a temperatura inicial por simulação. Por este mecanismo, parte-se de uma solução s e de uma temperatura de partida. A seguir, contam-se quantos vizinhos desta solução s são aceitos em $SAmax$ iterações do método nesta temperatura. Caso esse número de vizinhos aceitos seja elevado, algo como 95% dos vizinhos, então retorna-se a temperatura corrente como a temperatura inicial para o processo de refinamento pelo método. Caso o número de vizinhos aceitos não atinja o valor mínimo requerido, aumenta-se a temperatura segundo uma certa taxa, por exemplo 100% (o que equivale a dobrar a temperatura), e repete-se a contagem do número de vizinhos aceitos naquela temperatura. O procedimento prossegue até que se obtenha o número mínimo de vizinhos aceitos. A temperatura na qual esta condição ocorre representa a temperatura inicial para o método *Simulated Annealing*. A Figura 11 mostra o pseudocódigo para determinar a temperatura inicial por este método, considerando um problema de minimização. Nesta figura, β é a taxa de aumento da temperatura (por exemplo, $\beta = 2$), γ é a taxa mínima de aceitação de soluções vizinhas (por exemplo, $\gamma = 0{,}95$) e T_0 é uma temperatura de partida para o método, que pode ser o valor da função de avaliação da solução s usada para determinar a temperatura, que pode ser uma solução aleatória s, isto é, $T_0 = f(s)$. Esse método tem a vantagem de determinar a temperatura inicial mais apropriada para cada problema-teste, não se dependendo da solução inicial. Experimente observar que se a solução inicial tiver função de avaliação alta, então a temperatura inicial será baixa. Por outro lado, se a solução inicial for mais próxima de um ótimo global, isto é, mais baixa, então a temperatura inicial será alta.

Outra prescrição para determinar a temperatura inicial consiste em partir de uma dada solução e gerar todos os seus possíveis vizinhos ou uma fração destes. Para cada um desses vizinhos, calcular o respectivo custo segundo a função de avaliação considerada. Repetir este procedimento para outros pontos iniciais, já que dependendo da solução inicial o custo das soluções vizinhas pode ser diferente. O maior custo encontrado é uma estimativa para a temperatura inicial.

Em teoria, a temperatura final deve ser zero. Entretanto, na prática é suficiente chegar a uma temperatura próxima de zero devido à precisão limitada da implementação computacional [58]. Um valor típico é tomar $T_f = 0{,}001$. Alternativamente, pode-se identificar o congelamento do sistema quando a taxa de aceitação de movimentos ficar abaixo de um valor predeterminado.

Observa-se, finalmente, como regra geral, que os parâmetros mais adequados para uma dada aplicação do algoritmo só podem ser estabelecidos por experimentação [58]. Ferramentas específicas para este fim estão disponíveis na literatura, entre eles o pacote *irace* [37].

```
procedimento TemperaturaInicial(f(.),N(.),β,γ,SAmax,T0,s)
1  T ← T0;            {Temperatura corrente}
2  Continua ← TRUE;
3  enquanto (Continua) faça
4      Aceitos ← 0; {Número de vizinhos aceitos na temperatura T}
5      para IterT = 1 até SAmax faça
6          Gere um vizinho qualquer s' ∈ N(s);
7          Δ = f(s') − f(s);
8          se (Δ < 0)
9              então
10                 Aceitos ← Aceitos + 1;
11             senão
12                 Escolha um número aleatório x ∈ [0,1];
13                 se (x < e^(−Δ/T)) então Aceitos ← Aceitos + 1;
14         fim-se;
15     fim-para;
16     se (Aceitos ≥ γ × SAmax)
17         então Continua ← FALSE;
18         senão T ← β × T;
19     fim-se;
20 fim-enquanto;
21 Retorne T;
fim TemperaturaInicial;
```

Figura 11: Determinação autoadaptativa da temperatura inicial

4.3 Busca Tabu

Descrevemos, a seguir, de forma resumida, os princípios básicos da Busca Tabu - BT (*Tabu Search*, em inglês, ou Pesquisa Tabu, como referenciado em Portugal), técnica originada nos trabalhos independentes de Fred Glover [15] e Pierre Hansen [27]. Referenciamos a [15, 16, 17, 23, 20, 21, 8, 33] para um detalhamento mais aprofundado do método.

A Busca Tabu é um método de busca local que consiste em explorar o espaço de soluções movendo-se de uma solução para outra que seja seu melhor vizinho. Esta estratégia, juntamente com uma estrutura de memória para armazenar as soluções geradas (ou características destas) permite que a busca não fique presa em um ótimo local.

Mais especificamente, começando com uma solução inicial s_0, um algoritmo BT explora, a cada iteração, um subconjunto V da vizinhança $N(s)$ da solução corrente s. O membro s' de V com melhor valor nesta região segundo a função $f(.)$ torna-se a nova solução corrente mesmo que s' seja pior que s, isto é, que $f(s') > f(s)$ para um problema de minimização.

O critério de escolha do melhor vizinho é utilizado para não ficar preso em um ótimo local. Esta estratégia, entretanto, pode fazer com que o algoritmo cicle,

isto é, que retorne a uma mesma sequência de soluções já geradas anteriormente.

De forma a evitar que isto ocorra, existe uma lista tabu T, a qual é uma lista de movimentos proibidos. A lista tabu clássica contém os movimentos reversos aos últimos $|T|$ movimentos realizados (sendo $|T|$ um parâmetro do método) e funciona como uma fila de tamanho fixo, isto é, quando um novo movimento é adicionado à lista, o mais antigo sai. Assim, na exploração do subconjunto V da vizinhança $N(s)$ da solução corrente s, ficam excluídos da busca os vizinhos s' que são obtidos de s por movimentos m que constam na lista tabu.

A lista tabu se, por um lado, reduz o risco de ciclagem (uma vez que ela garante o não retorno, por $|T|$ iterações, a uma solução já visitada anteriormente); por outro, também pode proibir movimentos para soluções que ainda não foram visitadas [8]. Assim, existe também uma *função de aspiração*, que é um mecanismo que retira, sob certas circunstâncias, o *status* tabu de um movimento. Mais precisamente, para cada possível valor v da função objetivo existe um nível de aspiração $A(v)$: uma solução s' em V pode ser gerada se $f(s') < A(f(s))$, mesmo que o movimento m esteja na lista tabu. A função de aspiração A é tal que, para cada valor v da função objetivo, retorna outro valor $A(v)$, que representa o valor que o algoritmo aspira ao chegar de v. Um exemplo simples de aplicação desta idéia é considerar $A(f(s)) = f(s^\star)$ onde $s^\star$ é a melhor solução encontrada até então. Neste caso, aceita-se um movimento tabu somente se ele conduzir a um vizinho melhor que $s^\star$. Esta é a chamada aspiração por objetivo. Esse critério se fundamenta no fato de que soluções melhores que a solução $s^\star$ corrente, ainda que geradas por movimentos tabu, não foram visitadas anteriormente, evidenciando que a lista de movimentos tabu pode impedir não somente o retorno a uma solução já gerada anteriormente mas também a outras soluções ainda não geradas.

Duas regras são normalmente utilizadas de forma a interromper o procedimento. Pela primeira, pára-se quando é atingido um certo número máximo de iterações sem melhora no valor da melhor solução. Pela segunda, quando o valor da melhor solução chega a um limite inferior conhecido (ou próximo dele). Esse segundo critério evita a execução desnecessária do algoritmo quando uma solução ótima é encontrada ou quando uma solução é julgada suficientemente boa.

Os parâmetros principais de controle do método de Busca Tabu são a cardinalidade $|T|$ da lista tabu, a função de aspiração A, a cardinalidade do conjunto V de soluções vizinhas testadas em cada iteração e $BTmax$, o número máximo de iterações sem melhora no valor da melhor solução.

Apresenta-se, pela Figura 12, o pseudocódigo de um algoritmo de Busca Tabu básico para o caso de minimização. Neste procedimento, $f_{\min}$ é o valor mínimo conhecido da função f, informação esta que em alguns casos está disponível.

É comum em métodos de Busca Tabu incluir estratégias de intensificação, as quais têm por objetivo concentrar a pesquisa em determinadas regiões consideradas promissoras. Uma estratégia típica é retornar a uma solução já visitada para explorar sua vizinhança de forma mais efetiva. Outra estratégia consiste em incorporar atributos das melhores soluções já encontradas durante o pro-

procedimento $BT(f(.),N(.),A(.),|V|,f_{\min},|T|,BTmax,s)$
1 $s^\star \leftarrow s$; {Melhor solução obtida até então}
2 $Iter \leftarrow 0$; {Contador do número de iterações}
3 $MelhorIter \leftarrow 0$; {Iteração mais recente que forneceu $s^\star$}
4 $T \leftarrow \emptyset$; {Lista Tabu}
5 Inicialize a função de aspiração A;
6 enquanto $(f(s) > f_{\min}$ e $Iter - MelhorIter \leq BTmax)$ faça
7 $Iter \leftarrow Iter + 1$;
8 Seja $s' \leftarrow s \oplus m$ o melhor elemento de $V \subseteq N(s)$ tal que
o movimento m não seja tabu $(m \notin T)$ ou
s' atenda a condição de aspiração $(f(s') < A(f(s)))$;
9 Atualize a lista tabu T;
10 $s \leftarrow s'$;
11 se $(f(s) < f(s^\star))$ então
12 $s^\star \leftarrow s$;
13 $MelhorIter \leftarrow Iter$;
14 fim-se;
15 Atualize a função de aspiração A;
16 fim-enquanto;
17 $s \leftarrow s^\star$;
18 Retorne s;
fim BT;

Figura 12: Algoritmo de Busca Tabu

gresso da pesquisa e estimular componentes destas soluções a tornar parte da solução corrente. Nesse caso, são consideradas livres no procedimento de busca local apenas as componentes não associadas às boas soluções, permanecendo as demais componentes fixas. Um critério de término, tal como um número fixo de iterações, é utilizado para encerrar o período de intensificação. Na Seção 5.1, página 59, detalha-se um procedimento de intensificação, a Reconexão por Caminhos, mecanismo que é comumente associado a implementações de Busca Tabu.

Métodos baseados em Busca Tabu podem incluir, também, estratégias de diversificação. O objetivo destas estratégias, que tipicamente utilizam uma memória de longo prazo, é redirecionar a pesquisa para regiões ainda não suficientemente exploradas do espaço de soluções. Estas estratégias procuram, ao contrário das estratégias de intensificação, gerar soluções que têm atributos significativamente diferentes daqueles encontrados nas melhores soluções obtidas. A diversificação, em geral, é utilizada somente em determinadas situações, como, por exemplo, quando dada uma solução s, não existem movimentos m de melhora para ela, indicando que o algoritmo já exauriu a análise naquela região. Para escapar desta região, a idéia é estabelecer uma penalidade $w(s, m)$ para uso desses movimentos. Um número fixo de iterações sem melhora no valor da solução ótima corrente é, em geral, utilizado para acionar estas estratégias. Na

Seção 5.3, página 65, detalha-se um procedimento de diversificação, a Relaxação Adaptativa.

Métodos de Busca Tabu podem incluir, também, listas tabu dinâmicas [6, 55], muitas das quais atualizadas de acordo com o progresso da pesquisa [4, 3, 2]. A grande vantagem de se usar uma lista tabu de tamanho dinâmico é que se minimiza a possibilidade de ocorrência de ciclagem. Em [23] os autores resolvem um problema de roteamento de veículos por meio de Busca Tabu utilizando uma lista tabu dinâmica que varia no intervalo $[t_{\min}, t_{\max}]$, sendo $t_{\min} = 0{,}9n$ e $t_{\max} = 1{,}1n$, com n representando o número de cidades da instância considerada. Nesta aplicação, depois que o tamanho da lista é escolhido aleatoriamente no intervalo $[t_{\min}, t_{\max}]$, ele é mantido constante por $2t_{\max}$ iterações. A idéia por trás da utilização da lista dinâmica é que, se com um dado tamanho de lista há ciclagem, então aumentando-se ou diminuindo-se esse tamanho haverá alteração da quantidade de movimentos tabu e, assim, diferentes soluções vizinhas poderão ser geradas. Com esta possibilidade de mudança de trajetória no espaço de busca, a ocorrência de ciclagem fica reduzida.

Aspectos de convergência do método são estudados em [19] e [26], entre outras referências.

De forma a ilustrar o método de Busca Tabu, apliquemos este método heurístico à instância do problema da mochila apresentado no início destas notas de aula. Consideremos uma lista tabu T de cardinalidade $|T| = 1$ e como atributo tabu a posição do bit alterado. Utilizemos o critério de aspiração por objetivo global, isto é, um movimento tabu somente será realizado se a solução produzida melhorar a melhor solução gerada até então. O critério de parada será $BTmax = 1$, isto é, apenas uma iteração sem melhora.

Passo 0 : Seja uma solução inicial qualquer, por exemplo:
$s = (01010)^t$
$f(s) = 6$
Peso corrente da mochila = 14
Lista tabu = $T = \emptyset$
Melhor solução até então: $s^* = (01010)^t$ e $f(s^*) = 6$
$Iter = 0$; $MelhorIter = 0$;

Passo 1 : Devemos, agora, analisar todos os vizinhos de s e calcular a função de avaliação deles por meio da expressão (3), definida à página 12.

Vizinhos de s	Peso total dos vizinhos de s	Benefício total dos vizinhos de s	$f(s')$
$(\mathbf{1}1010)^t$	18	8	8
$(0\mathbf{0}010)^t$	9	4	4
$(011\mathbf{1}0)^t$	21	9	9
$(0100\mathbf{0})^t$	5	2	2
$(0101\mathbf{1})^t$	20	10	10

Melhor vizinho: $s' = (01011)^t$, com $f(s') = 10$.
Como s' é o melhor vizinho de s, então $s \leftarrow s'$, isto é, a nova solução corrente passa a ser: $s = (01011)^t$.
Lista tabu = $T = \{5\}$ (indicando que o bit da quinta posição não pode ser modificado, a não ser que o critério de aspiração seja satisfeito).
Melhor solução até então: $s^* = (01011)^t$ e $f(s^*) = 10$ (pois $f(s') > f(s^*)$).
$Iter = 1$; $MelhorIter = 1$.
Como $Iter - MelhorIter = 1 - 1 = 0 \not> BTmax = 1$, então o procedimento de exploração do espaço de soluções deve continuar.

Passo 2 : Determinemos, agora, o melhor vizinho de $s = (01011)^t$:

Vizinhos de s	Peso total dos vizinhos de s	Benefício total dos vizinhos de s	$f(s')$
$(\mathbf{1}1011)^t$	24	12	-3
$(0\mathbf{0}011)^t$	15	8	8
$(011\mathbf{1}1)^t$	27	13	-47
$(0100\mathbf{1})^t$	11	6	6
$(0101\mathbf{0})^t$	14	6	6

Melhor vizinho: $s' = (00011)^t$, com $f(s') = 8$.
Como s' é o melhor vizinho de s, então $s \leftarrow s'$ (mesmo sendo $f(s')$ pior que $f(s)$), isto é, a nova solução corrente passa a ser: $s = (00011)^t$.
Lista tabu = $T = \{2\}$ (observa-se que, como a cardinalidade da lista tabu foi fixada em uma unidade, isto é, $|T| = 1$, então o movimento proibido anterior sai e entra o novo movimento proibido, isto é, o bit da segunda posição não pode ser modificado, a não ser que o critério de aspiração seja satisfeito).
Melhor solução até então: $s^* = (01011)^t$ e $f(s^*) = 10$.
$Iter = 2$; $MelhorIter = 1$.
Como $Iter - MelhorIter = 2 - 1 = 1 \not> BTmax = 1$, então o procedimento BT continua.

Passo 3 : Determinemos, agora, o melhor vizinho de $s = (00011)^t$:

Vizinhos de s	Peso total dos vizinhos de s	Benefício total dos vizinhos de s	$f(s')$
$(\mathbf{1}0011)^t$	19	10	10
$(0\mathbf{1}011)^t$	20	10	10
$(001\mathbf{1}1)^t$	22	11	11
$(0000\mathbf{1})^t$	6	4	4
$(0001\mathbf{0})^t$	9	4	4

Melhor vizinho: $s' = (00111)^t$, com $f(s') = 11$.
Como s' é o melhor vizinho de s, então $s \leftarrow s'$, isto é, a nova solução

corrente passa a ser: $s = (00111)^t$.
Lista tabu = $T = \{3\}$ (indicando que o bit da terceira posição não pode ser modificado, a não ser que o critério de aspiração seja satisfeito).
Melhor solução até então: $s^* = (00111)^t$ e $f(s^*) = 11$ (pois $f(s') > f(s^*)$).
$Iter = 3$; $MelhorIter = 3$.
Como $Iter - MelhorIter = 3 - 3 = 0 \not> BTmax = 1$, então prossegue-se com o procedimento de exploração do espaço de soluções.

Passo 4 : Determinemos, agora, o melhor vizinho de $s = (00111)^t$:

Vizinhos de s	Peso total dos vizinhos de s	Benefício total dos vizinhos de s	$f(s')$
$(\mathbf{1}0111)^t$	24	13	-2
$(0\mathbf{1}111)^t$	25	13	-17
$(00\mathbf{0}11)^t$	15	8	8
$(001\mathbf{0}1)^t$	13	7	7
$(0011\mathbf{0})^t$	16	7	7

Observe que o vizinho com o melhor valor para a função de avaliação é $s' = (00011)^t$, com $f(s') = 8$, mas esta solução é tabu, uma vez que o bit da terceira posição está na lista tabu. Como o critério de aspiração desta solução não é satisfeito, pois $f(s') = 8 \not> f(s^*) = 11$, esta solução não é aceita. Desta forma, toma-se o melhor vizinho não tabu, a saber:
Melhor vizinho: $s' = (00101)^t$, com $f(s') = 7$ (Em caso de empate, como é o caso, a solução vizinha escolhida é aquela que satisfaz a um determinado critério, como por exemplo, aquela associada ao menor peso ou menor índice).
Como s' é o melhor vizinho de s (mesmo sendo de piora), então $s \leftarrow s'$, isto é, a nova solução corrente passa a ser: $s = (00101)^t$.
Lista tabu = $T = \{4\}$ (indicando que o bit da quarta posição não pode ser modificado, a não ser que o critério de aspiração seja satisfeito).
Melhor solução até então: $s^* = (00111)^t$ e $f(s^*) = 11$.
$Iter = 4$; $MelhorIter = 3$.
Como $Iter - MelhorIter = 4 - 3 = 1 \not> BTmax = 1$, então a busca prossegue.

Passo 5 : Determinemos, agora, o melhor vizinho de $s = (00101)^t$:

Vizinhos de s	Peso total dos vizinhos de s	Benefício total dos vizinhos de s	$f(s')$
$(\mathbf{1}0101)^t$	17	9	9
$(0\mathbf{1}101)^t$	18	9	9
$(00\mathbf{0}01)^t$	6	4	4
$(001\mathbf{1}1)^t$	23	11	11
$(0010\mathbf{0})^t$	7	3	3

Observe que o vizinho com o melhor valor para a função de avaliação é $s' = (00111)^t$, com $f(s') = 11$. Entretanto, esta solução é tabu, uma vez que o bit da quarta posição está na lista tabu. Como o critério de aspiração desta solução não é satisfeito, pois $f(s') = 11 \not> f(s^*) = 11$, esta solução não é aceita. Desta forma, toma-se o melhor vizinho não tabu, a saber (já aplicado um critério de desempate):
Melhor vizinho: $s' = (10101)^t$, com $f(s') = 9$.
Desta forma, a nova solução corrente passa a ser: $s = (10101)^t$, com $f(s) = 9$.
Lista tabu = $T = \{1\}$ (indicando que o bit da primeira posição não pode ser modificado, a não ser que o critério de aspiração seja satisfeito)
Melhor solução até então: $s^* = (00111)^t$ e $f(s^*) = 11$.
$Iter = 5$; $MelhorIter = 3$;
Como $Iter - MelhorIter = 5 - 3 = 2 > BTmax = 1$, então PARE. O método de Busca Tabu retorna, então, $s^* = (00111)^t$ como solução final, com valor $f(s^*) = 11$.

A aplicação deste método mostra que a solução final obtida é melhor que aquela encontrada com a aplicação do método da subida, apresentado à página 11. Isto foi possível devido à aceitação de movimentos de piora, no caso, com o passo 2 do método. Esta estratégia possibilitou escapar de um ótimo local, atingindo uma outra região do espaço de soluções na qual uma solução melhor foi encontrada.

Da forma como apresentado, a consulta se um movimento é tabu ou não em uma lista T tem complexidade $O(|T|)$, no pior caso. Essa consulta pode ser reduzida para complexidade constante, $O(1)$, utilizando-se um vetor como estrutura de dados. Seja, então, T um vetor com tantas posições quantos forem os objetos candidatos a serem carregados na mochila, isto é, $T = (T_1, T_2, \cdots, T_j, \cdots, T_n)$. Esse vetor T é chamado de *duração tabu* e cada célula T_j dele armazena a iteração até a qual o movimento de alterar o valor de seu bit da posição j está proibido. Exemplo: se considerarmos uma duração tabu fixa, com valor *duracao*=3, a iteração atual igual a *iter*=1 e o movimento proibido é o de alterar o j-ésimo bit, então o vetor *duração tabu* dever ser atualizado como segue: $T = (0, 0, \cdots, iter + duracao, \cdots, 0$, isto é, $T = (0, 0, \cdots, 1 + 3, \cdots, 0)$ $= T = (0, 0, \cdots, 4, \cdots, 0)$. Desta forma, o movimento de alterar o j-ésimo bit permanecerá tabu ativo até a quarta iteração do método, ou seja, ele é tabu se $iter \leq T_j$, sendo *iter* a iteração atual. Assim, há uma única atualização no vetor *duração tabu* a cada iteração do método, no caso, apenas na célula T_j.

No caso do problema do caixeiro viajante, seja o movimento proibido o de realizar a troca das posições i e j de visita entre as cidades s_i e s_j. Neste caso, a lista tabu T conteria os movimentos proibidos (s_i, s_j) e seria atualizado da seguinte forma: $T \leftarrow T \cup \{(s_i, s_j)\}$. Tal como no problema da mochila, a complexidade de consulta à essa lista é $O(|T|)$. Essa complexidade pode ser reduzida para $O(1)$ utilizando-se uma **matriz** de *duração tabu*. Neste caso, cada célula (i,j) deve armazenar a iteração atual *iter* e a duração de proibição estipulada. Por exemplo, se a iteração atual é *iter*=1 e a duração estipulada

é *duracao*=3, então a célula T_{ij} deverá armazenar o valor 4 (= *iter* + *duracao* = 1 + 3), para indicar que a troca das cidades s_i e s_j que estão nas posições i e j de visita está proibida até a quarta iteração do método. Isto é, esse movimento de troca é tabu se $iter \leq T_{ij}$, sendo *iter* a iteração atual. Como trocar i com j é a mesma coisa que trocar j com i, então basta utilizar uma matriz triangular, p.ex., triangular superior, na qual apenas os elementos T_{ij}, com $i > j$ são atualizados e os demais elementos $i < j$ permanecem nulos, isto é, $T_{ij} = 0 \; \forall i < j$.

A Figura 4.3 ilustra uma matriz T de duração tabu na qual a célula T_{23} armazena a iteraçao até a qual está proibido o movimento de troca entre as cidades s_2 e s_3 que estão nas posições $i = 2$ e $j = 3$, respectivamente. Ou seja, nesse exemplo, considerando que a iteração atual é *iter*=1 e *duracao*=3, o movimento de troca envolvendo as cidades s_2 e s_3 permanece tabu até a iteração 4. Após a quarta iteração, ele deixa de ser tabu.

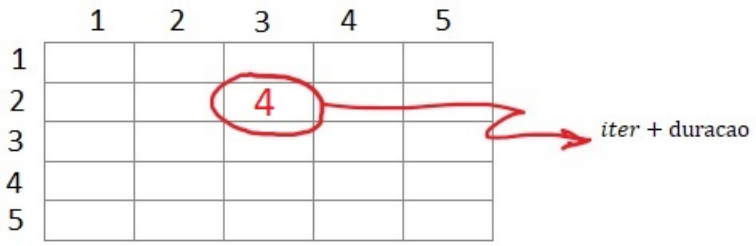


Figura 13: Matriz de Duração Tabu para o PCV

Ainda considerando o PCV, ao invés de considerar uma matriz de duração tabu envolvendo o movimento de troca de posições, poderíamos armazenar apenas um atributo tabu desse movimento para proibir a troca. P.ex., se o movimento a ser proibido é o de trocar as posições i e j, poderíamos considerar como atributo tabu a proibição de alterar a cidade que está na posição i. Neste caso, teríamos um **vetor** T de duração tabu que armazena na célula T_i a iteração até a qual estão proibidas trocas que envolvam a cidade que está nessa posição. Observe que esse vetor de duração tabu é mais restritivo do que a matriz de duração tabu. De fato, enquanto uma matriz de duração tabu impede somente as trocas envolvendo as cidades i e j, um vetor de duração tabu impede qualquer outra troca que envolva a cidade que está na posição i. Em situações como essa, a duração tabu deve ser reduzida porque essa metodologia impede que muitas soluções vizinhas da solução atual sejam alcançadas. Um estudo bem completo sobre regras de proibição e limitantes para a duração tabu em problemas envolvendo permutação, como é o caso do PCV, é detalhado em [53]. Recomenda-se, como leitura complementar, o trabalho [57] para uma visão mais ampla de aplicações, regras de proibição, limitantes para a duração tabu, memória de longo prazo, etc., em métodos de Busca Tabu.

4.4 GRASP

GRASP (*Greedy Randomized Adaptive Search Procedure* - Procedimento de busca adaptativa gulosa e randômica) é um método iterativo, proposto em [13, 50], que consiste de duas fases: uma fase de construção, na qual uma solução é gerada, elemento a elemento, e de uma fase de busca local, na qual um ótimo local na vizinhança da solução construída é pesquisado. A melhor solução encontrada ao longo de todas as iterações GRASP realizadas é retornada como resultado. O pseudocódigo descrito pela Figura 14 ilustra um procedimento GRASP para um problema de minimização.

```
procedimento GRASP(f(.),g(.),N(.),GRASPmax,s)
1  f* ← ∞;
2  para (Iter = 1, 2, ..., GRASPmax) faça
3      Construcao(g(.),α,s);
4      BuscaLocal(f(.),N(.),s);
5      se (f(s) < f*) então
6          s* ← s;
7          f* ← f(s);
8      fim-se;
9  fim-para;
10 s ← s*;
11 Retorne s;
fim GRASP
```

Figura 14: Algoritmo GRASP

Na fase de contrução, uma solução é iterativamente construída, elemento por elemento. A cada iteração desta fase, os próximos elementos candidatos a serem incluídos na solução são colocados em uma lista C de candidatos, seguindo um critério de ordenação pré-determinado. Este processo de seleção é baseado em uma função adaptativa gulosa $g : C \mapsto \Re$, que estima o benefício da seleção de cada um dos elementos. A heurística é dita adaptativa porque os benefícios associados com a escolha de cada elemento são atualizados em cada iteração da fase de construção para refletir as mudanças oriundas da seleção do elemento anterior. A componente probabilística do procedimento reside no fato de que cada elemento é selecionado de forma aleatória a partir de um subconjunto restrito formado pelos melhores elementos que compõem a lista de candidatos. Este subconjunto recebe o nome de lista de candidatos restrita (LCR). Esta técnica de escolha permite que diferentes soluções sejam geradas em cada iteração GRASP. O pseudocódigo representado pela Figura 15, onde $\alpha \in [0,1]$ é um parâmetro do método, descreve a fase de construção GRASP.

Observamos que o parâmetro α controla o nível de gulosidade e aleatoriedade do procedimento *Construcao*. Um valor $\alpha = 0$ faz gerar soluções puramente gulosas, enquanto $\alpha = 1$ faz produzir soluções totalmente aleatórias.

Assim como em muitas técnicas determinísticas, as soluções geradas pela

```
procedimento Construcao(g(.),α,s);
1  s ← ∅;
2  Inicialize o conjunto C de candidatos;
3  enquanto (C ≠ ∅) faça
4      g(t_min) = min{g(t) | t ∈ C};
5      g(t_max) = max{g(t) | t ∈ C};
6      LCR = {t ∈ C | g(t) ≤ g(t_min) + α(g(t_max) − g(t_min))};
7      Selecione, aleatoriamente, um elemento t ∈ LCR;
8      s ← s ∪ {t};
9      Atualize o conjunto C de candidatos;
10 fim-enquanto;
11 Retorne s;
fim Construcao;
```

Figura 15: Fase de construção de um algoritmo GRASP

fase de construção do GRASP provavelmente não são localmente ótimas com respeito à definição de vizinhança adotada. Daí a importância da fase de busca local, a qual objetiva melhorar a solução construída. A Figura 16 descreve o pseudo-código de um procedimento básico de busca local com respeito a uma certa vizinhança $N(.)$ de s para um problema de minimização.

```
procedimento BuscaLocal(f(.),N(.),s);
1  V = {s' ∈ N(s) | f(s') < f(s)};
2  enquanto (|V| > 0) faça
3      Selecione s' ∈ V;
4      s ← s' ;
5      V = {s' ∈ N(s) | f(s') < f(s)};
6  fim-enquanto;
7  Retorne s;
fim BuscaLocal;
```

Figura 16: Fase de Busca Local de um algoritmo GRASP

A eficiência da busca local depende da qualidade da solução construída. O procedimento de construção tem então um papel importante na busca local, uma vez que as soluções construídas constituem bons pontos de partida para a busca local, permitindo assim acelerá-la.

O parâmetro α, que determina o tamanho da lista de candidatos restrita, é basicamente o único parâmetro a ser ajustado na implementação de um procedimento GRASP. Em [13] discute-se o efeito do valor de α na qualidade da solução e na diversidade das soluções geradas durante a fase de construção. Valores de α que levam a uma lista de candidatos restrita de tamanho muito limitado (ou seja, valor de α próximo da escolha gulosa) implicam em soluções finais de qualidade muito próxima àquela obtida de forma puramente gulosa, ob-

tidas com um baixo esforço computacional. Em contrapartida, provocam uma baixa diversidade de soluções construídas. Já uma escolha de α próxima da seleção puramente aleatória leva a uma grande diversidade de soluções construídas mas, por outro lado, muitas das soluções construídas são de qualidade inferior, tornando mais lento o processo de busca local.

O procedimento GRASP procura, portanto, conjugar bons aspectos dos algoritmos puramente gulosos, com aqueles dos procedimentos aleatórios de construção de soluções.

Procedimentos GRASP mais sofisticados incluem estratégias adaptativas para o parâmetro α. O ajuste deste parâmetro ao longo das iterações GRASP, por critérios que levam em consideração os resultados obtidos nas iterações anteriores, produz soluções melhores do que aquelas obtidas considerando-o fixo [45, 46, 47]. Recomenda-se o livro [49] para conhecimento de avanços recentes deste método.

4.5 Busca em Vizinhança Variável

A Busca em Vizinhança Variável, ou Método de Pesquisa em Vizinhança Variável (*Variable Neighborhood Search*, VNS), proposta por Nenad Mladenović e Pierre Hansen [42], é um método de busca local que consiste em explorar o espaço de soluções por meio de trocas sistemáticas de estruturas de vizinhança. Contrariamente à outras meta-heurísticas baseadas em métodos de busca local, o método VNS não segue uma trajetória, mas sim explora vizinhanças gradativamente mais "distantes" da solução corrente e focaliza a busca em torno de uma nova solução se e somente se um movimento de melhora é realizado. O método inclui, também, um procedimento de busca local a ser aplicado sobre a solução corrente. Esta rotina de busca local também pode usar diferentes estruturas de vizinhança. Na sua versão original, o método VNS faz uso do método VND (descrito na Seção 3.5, à página 16) para fazer a busca local.

O pseudocódigo do método é apresentado pela Figura 17. Detalhes adicionais deste método podem ser encontrados em [42, 28, 29, 30].

Neste algoritmo, parte-se de uma solução inicial qualquer e a cada iteração seleciona-se aleatoriamente um vizinho s' dentro da vizinhança $N^{(k)}(s)$ da solução s corrente. Esse vizinho é então submetido a um procedimento de busca local. Se a solução ótima local, s'', for melhor que a solução s corrente, a busca continua de s'' recomeçando da primeira estrutura de vizinhança $N^{(1)}(s)$. Caso contrário, continua-se a busca a partir da próxima estrutura de vizinhança $N^{(k+1)}(s)$. Este procedimento é encerrado quando uma condição de parada for atingida, tal como o tempo máximo permitido de CPU, o número máximo de iterações ou número máximo de iterações consecutivas entre dois melhoramentos. A solução s' é gerada aleatoriamente no passo 7 de forma a evitar ciclagem, situação que pode ocorrer se alguma regra determinística for usada.

Se a busca local realizada no passo 8 for convencional (isto é, utilizar apenas um tipo de movimento), o método recebe a denominação *Basic Variable Neighborhood Search* [30]. Se, por outro lado, essa busca local for a Descida em Vizinhança Variável (VND), descrita na Seção 3.5, então o método recebe a

```
procedimento VNS()
1  Seja s_0 uma solução inicial;
2  Seja r o número de estruturas diferentes de vizinhança;
3  s ← s_0;              {Solução corrente}
4  enquanto (Critério de parada não for satisfeito) faça
5      k ← 1;            {Tipo de estrutura de vizinhança corrente}
6      enquanto (k ≤ r) faça
7          Gere um vizinho qualquer s' ∈ N^(k)(s);
8          s'' ← BuscaLocal(s');
9          se (f(s'') < f(s))
10             então
11                 s ← s'';
12                 k ← 1;
13             senão
14                 k ← k + 1;
15         fim-se;
16     fim-enquanto;
17 fim-enquanto;
18 Retorne s;
fim VNS;
```

Figura 17: Algoritmo *VNS*

denominação *General Variable Neighborhood Search* (GVNS) [30]. Neste último caso, as vizinhanças usadas no algoritmo principal do VNS e na busca local podem ser diferentes.

4.6 *Iterated Local Search*

O método *Iterated Local Search* (ILS) [38, 39] é baseado na idéia de que um procedimento de busca local pode ser melhorado gerando-se novas soluções de partida, as quais são obtidas por meio de perturbações na solução ótima local.

Para aplicar um algoritmo ILS, quatro componentes têm que ser especificadas: (a) Procedimento $GeraSolucaoInicial()$, que gera uma solução inicial s_0 para o problema; (b) Procedimento $BuscaLocal$, que retorna uma solução possivelmente melhorada s''; (c) Procedimento $Perturbacao$, que modifica a solução corrente s guiando a uma solução intermediária s' e (d) Procedimento $CriterioAceitacao$, que decide de qual solução a próxima perturbação será aplicada.

Na Figura 18 mostra-se o pseudocódigo de um algoritmo ILS básico.

O sucesso do ILS é centrado no conjunto de amostragem de ótimos locais, juntamente com a escolha do método de busca local, das perturbações e do critério de aceitação. Em princípio, qualquer método de busca local pode ser usado, mas o desempenho do ILS com respeito à qualidade da solução final e a velocidade de convergência depende fortemente do método escolhido. Normal-

```
procedimento ILS
1  s_0 ← GeraSolucaoInicial();
2  s ← BuscaLocal(s_0);
3  enquanto (os critérios de parada não estiverem satisfeitos) faça
4      s' ← Perturbacao(histórico,s);
5      s'' ← BuscaLocal(s');
6      s ← CriterioAceitacao(s, s'',histórico);
8  fim-enquanto;
fim ILS;
```

Figura 18: Algoritmo *Iterated Local Search*

mente um método de descida é usado, mas também é possível aplicar algoritmos mais sofisticados, tais como Busca Tabu ou outras meta-heurísticas.

A intensidade da perturbação deve ser forte o suficiente para permitir escapar do ótimo local corrente e permitir explorar diferentes regiões. Ao mesmo tempo, ela precisa ser fraca o suficiente para guardar características do ótimo local corrente.

O critério de aceitação é usado para decidir de qual solução se continuará a exploração, bem como qual será a perturbação a ser aplicada. Um aspecto importante do critério de aceitação e da perturbação é que eles induzem aos procedimentos de intensificação e diversificação. A intensificação consiste em permanecer na região do espaço onde a busca se encontra, procurando explorá-la de forma mais efetiva; enquanto a diversificação consiste em se deslocar para outras regiões do espaço de soluções. A intensificação da busca no entorno da melhor solução encontrada é obtida, por exemplo, pela aplicação de "pequenas" perturbações sobre ela. A diversificação, por sua vez, pode ser realizada aceitando-se quaisquer soluções s'' e aplicando "grandes" perturbações na solução ótima local.

Um critério de aceitação comumente utilizado é mover-se para o ótimo local s'' somente se ele for melhor que o ótimo local corrente s, isto é, somente se $f(s'') < f(s)$ em um problema de minimização, ou se $f(s'') > f(s)$ em um problema de maximização.

A Figura 19, à página 37, ilustra o funcionamento do método ILS em um problema de minimização. A partir de um ótimo local s, é feita uma perturbação que guia a uma solução intermediária s'. Após a aplicação de um método de busca local a s' é produzido um novo ótimo local s''. Considerando como critério de aceitação o fato de $f(s'')$ ser melhor que $f(s)$, então a busca prossegue a partir de s''.

Para definir o que seria uma perturbação no Problema do Caixeiro Viajante, consideremos uma estrutura de vizinhança que utilize movimentos de troca de posição de duas cidades para gerar vizinhos. Uma perturbação poderia ser dividida em vários níveis de intensidade. Assim, por exemplo, uma perturbação de nível 1 poderia consistir na realização de duas trocas aleatórias. A perturbação de nível 2 consistiria na execução de três trocas aleatórias sobre uma mesma

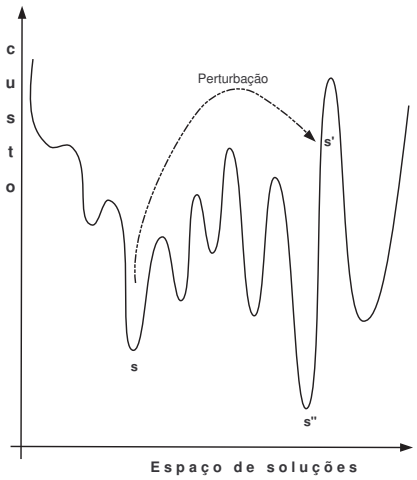


Figura 19: Representação esquemática do funcionamento do ILS

solução e assim sucessivamente. O algoritmo então funcionaria da seguinte maneira: Para cada nível de perturbação seria realizada uma busca local, a qual, se bem sucedida, faria retornar ao nível mínimo de perturbação; caso contrário, seriam realizadas mais algumas buscas locais no mesmo nível de perturbação. Em caso de insucesso destas buscas locais, o nível de perturbação seria gradativamente aumentado. O método se encerraria após um certo número de iterações sem melhora ou quando um tempo limite fosse atingido.

Para o problema da mochila 0-1 tratado nestas notas, em que se considera como movimento a troca no valor de um bit, uma perturbação de nível 1 poderia ser a troca no valor de dois bits simultaneamente, a perturbação de nível 2 seria a troca no valor de três bits simultaneamente e assim sucessivamente. Igualmente ao caso anterior, para cada nível de perturbação (*nperturb*) seriam executadas várias buscas locais, digamos *nbuscas*. Esta estratégia se justifica uma vez que sendo a perturbação aleatória, poderiam ser produzidos diferentes ótimos locais em função da solução intermediária gerada em um mesmo nível de perturbação.

4.7 *Guided Local Search*

Guided Local Search ou Busca Local Guiada é uma meta-heurística proposta por Christos Voudouris [60]. Ela consiste em promover modificações na função custo para evitar que se fique preso em ótimos locais. Essas modificações consistem, basicamente, em valorar menos atributos dos ótimos locais encontrados em iterações anteriores e, assim, estimular a busca em outras regiões do espaço de soluções. Por exemplo, no PCV, um arco (i,j) de uma solução vizinha não recebe o valor d_{ij} como custo direto e, sim, o valor $d_{ij} + \lambda \times p_{ij}$, sendo λ um parâmetro e p_{ij} um valor que mensura o número de vezes que o arco (i,j) apareceu ao longo da busca.

4.8 Algoritmos Genéticos

4.8.1 Descrição genérica do método

Trata-se de uma metaheurística que se fundamenta em uma analogia com processos naturais de evolução, nos quais, dada uma população, os indivíduos com características genéticas melhores têm maiores chances de sobrevivência e de produzirem filhos cada vez mais aptos, enquanto indivíduos menos aptos tendem a desaparecer. Foram propostos por John Holland nos anos 70 [25]. Referenciamos a [25, 48] para um melhor detalhamento do método.

Nos Algoritmos Genéticos (AGs), cada cromossomo (indivíduo da população) está associado a uma solução do problema e cada gene está associado a uma componente da solução. Um alelo, por sua vez, está associado a um possível valor que cada componente da solução pode assumir. Um mecanismo de reprodução, baseado em processos evolutivos, é aplicado sobre a população com o objetivo de explorar o espaço de busca e encontrar melhores soluções para o problema. Cada indivíduo é avaliado por uma certa função de aptidão, a qual mensura seu grau de adaptação ao meio. Quanto maior o valor da função de aptidão, mais o indivíduo está adaptado ao meio.

Um Algoritmo Genético inicia sua busca com uma população $\{s_1^0, s_2^0, \ldots, s_n^0\}$, normalmente aleatoriamente escolhida, chamada de população no tempo 0.

O procedimento principal consiste em um *loop* que cria uma população $\{s_1^{t+1}, s_2^{t+1}, \ldots, s_n^{t+1}\}$ no tempo $t+1$ a partir de uma população do tempo t. Para atingir esse objetivo, os indivíduos da população do tempo t passam por uma fase de reprodução, a qual consiste em selecionar indivíduos para operações de recombinação e/ou mutação.

Há várias formas de selecionar indíviduos para o processo de reprodução. Uma dessas formas é a *Binary Tournament Selection*. Nesse processo de seleção, dois indivíduos são selecionados aleatoriamente e aquele que tiver o maior valor para a função de aptidão é escolhido para ser o primeiro pai. O segundo pai é escolhido de forma análoga. Outra forma de selecionar os pais para o processo de seleção é escolhê-los aleatoriamente.

Na operação de recombinação, os genes de dois cromossomos pais são combinados de forma a gerar cromossomos filhos (normalmente dois), de sorte que para cada cromossomo filho há um conjunto de genes de cada um dos cromossomos pais.

Por sua vez, a operação de mutação consiste em alterar aleatoriamente uma parte dos genes de cada cromossomo (componentes da solução).

Ambas as operações são realizadas com uma certa probabilidade. A operação de recombinação é realizada normalmente com uma probabilidade mais elevada (por exemplo, 80%) e a de mutação, com uma baixa probabilidade (de 1 a 2%, em geral).

Gerada a nova população do tempo $t+1$, define-se a população sobrevivente, isto é, as n soluções que integrarão a nova população. A população sobrevivente é definida pela aptidão dos indivíduos. Os critérios comumente usados para escolher os cromossomos sobreviventes são os seguintes: 1) aleatório; 2) roleta (no

qual a chance de sobrevivência de cada cromossomo é proporcional ao seu nível de aptidão) e 3) misto (isto é, uma combinação dos dois critérios anteriores). Em qualquer um desses critérios admite-se, portanto, a sobrevivência de indivíduos menos aptos. Isto é feito tendo-se por objetivo não ficar preso em ótimos locais.

A Figura 20 mostra a estrutura de um Algoritmo Genético básico.

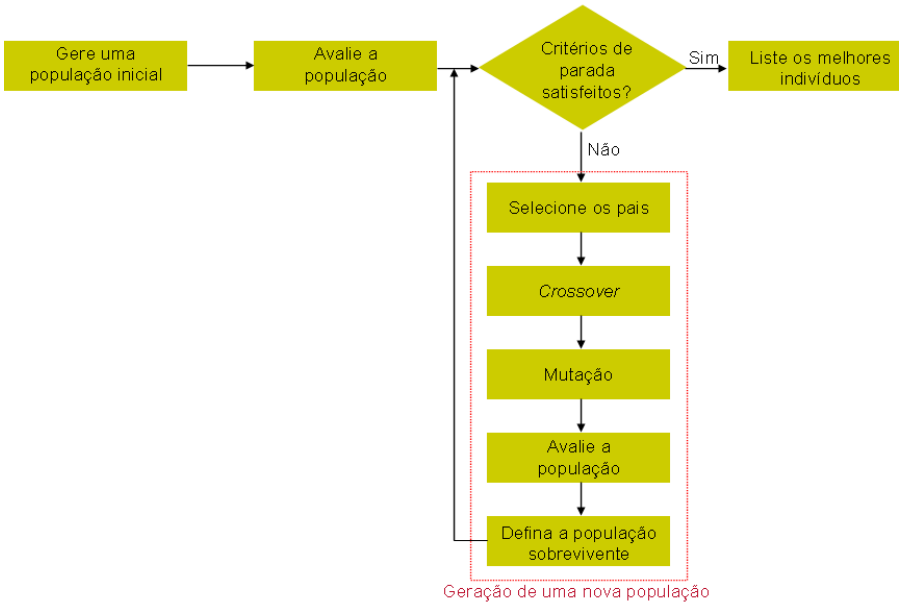


Figura 20: Estrutura de um Algoritmo Genético básico

Os critérios de parada mais comuns são: quando um certo número de gerações é atingido, quando não há melhora após um certo número de iterações ou quando o desvio padrão da população é inferior a um certo valor (situação que evidencia uma homogeneidade da população).

Os parâmetros principais de controle do método são: o tamanho da população, a probabilidade da operação *crossover* (p_{cross}), a probabilidade de mutação (p_{mut}), o número de gerações e o número de iterações sem melhora.

O pseudocódigo de um Algoritmo Genético básico está descrito na Figura 21.

4.8.2 Representação genética de soluções

Normalmente uma solução de um problema está associado a um cromossomo **p** representado na forma de um vetor (ou lista) com m posições, isto é, $p = (x_1, x_2, \ldots, x_m)$, sendo que cada componente x_i representa um gene (ou uma variável da solução).

procedimento *AG*
1 $t \leftarrow 0$;
2 Gere a população inicial $P(t)$;
3 Avalie $P(t)$;
4 enquanto (os critérios de parada não estiverem satisfeitos) faça
5 $t \leftarrow t + 1$;
6 Gere $P(t)$ a partir de $P(t-1)$;
7 Avalie $P(t)$;
8 Defina a população sobrevivente;
9 fim-enquanto;
fim *AG*;

Figura 21: Algoritmo Genético

Dentre os tipos de representação de um cromossomo, os mais conhecidos são: representação binária zero-um e a representação por inteiros.

A representação binária é a clássica no contexto dos AGs. Contudo, existem aplicações para as quais é mais conveniente o uso de representações por inteiros.

A seguir, alguns exemplos ilustram os dois tipos de representação:

1. **Representação Binária**:

   Na representação binária, uma solução do problema é representada por um vetor (ou lista) de 0's e 1's.

   Para ilustrar esta representação, considere o seguinte exemplo extraído de [41], no qual deseja-se maximizar a função:

   $$f(x) = |11 * num(x) - 150|,$$

   em que $num(x)$ fornece o número de "1s" do vetor cromossomo.

   Por exemplo, se $x = (101001)$ então $num(x) = 3$.

   Considere, agora, a representação binária de dois cromossomos compostos de 30 genes:

   $$p_1 = (101010101010101010101010101010)$$
   $$p_2 = (110000001100001110000000000000)$$

   Então $f(p_1) = |11 \times num(p_1) - 150| = |11 \times 15 - 150| = 15$ e $f(p_2) = |11 \times num(p_2) - 150| = |11 \times 7 - 150| = 73$.

   Observe que o cromossomo que fornece um ponto de máximo local é dado por:

$$p_0 = (00000000000000000000000000000000), \text{ com } f(p_0) = 150$$

Já o máximo global de f é fornecido pelo cromossomo:

$$p^* = (11111111111111111111111111111111), \text{ com } f(p^*) = 180$$

2. **Representação por Inteiros**:

   Na representação por inteiros, uma solução do problema é representada por um vetor (ou lista) de números inteiros. Por exemplo, considerando o PCV, o cromossomo $p = (1\ 3\ 6\ 5\ 4\ 2\ 7\ 9\ 8)$ pode representar exatamente a ordem de visita do PCV, ou seja, a rota (**tour**) t do PCV é idêntica a p:

   $$t = \{1\ 3\ 6\ 5\ 4\ 2\ 7\ 9\ 8\}$$

   Em um problema de programação de horários, um cromossomo pode ser uma matriz de números inteiros, em que cada componente x_{ij} representa o número da turma para a qual o professor i ministra aula no horário j.

4.8.3 Operador *crossover* clássico

A idéia do operador *crossover* clássico é a de efetuar cruzamentos entre dois ou mais cromossomos pais e formar cromossomos filhos (*offsprings*) a partir da união de segmentos de genes de cada pai.

Inicialmente são feitos **cortes** aleatórios nos pais. Por exemplo, considere dois pais e um ponto de corte realizado na parte central dos cromossomos pais.

$$p_1 = (1\ 0\ 1 \mid 0\ 1\ 0) = (p_1^1 \mid p_1^2)$$
$$p_2 = (1\ 1\ 1 \mid 0\ 0\ 0) = (p_1^2 \mid p_2^2)$$
$$\uparrow$$
$$\text{ponto de corte}$$

A partir dos **cortes**, são gerados dois filhos, cada qual formado a partir da reunião de partes de cada um dos pais.

$$O_1 = (p_1^1 \mid p_2^2) = (1\ 0\ 1 \mid 0\ 0\ 0)$$
$$O_2 = (p_2^1 \mid p_1^2) = (1\ 1\ 1 \mid 0\ 1\ 0)$$

4.8.4 Operador mutação clássico

Operadores mutação consistem em alterar um ou mais genes de um cromossomo. Para ilustrar um operador mutação clássico, seja o cromossomo p de 7 genes, dado por: $p = (x_1\ x_2\ x_3\ \ldots\ x_7) = (1\ 0\ 1\ 0\ 1\ 0\ 1)$.

Considerando como mutação a alteração de um gene de um valor 1 para um valor 0 ou vice-versa, então o cromossomo $p' = (1\ \mathbf{1}\ 1\ 0\ 1\ 0\ 1)$ representa uma mutação de p, no qual o gene x_2 foi alterado do valor 0 para o valor 1.

4.8.5 Operadores *crossover* para o PCV

No caso específico do PCV, a representação por inteiros é considerada mais adequada que a representação binária; primeiro por melhor se relacionar com uma solução (rota) do PCV, que consiste de uma lista de n números inteiros (cidades) e segundo porque a representação por inteiros favorece a construção de cromossomos que representem soluções viáveis para o PCV.

Infelizmente, o simples uso de operadores clássicos conduzem freqüentemente a soluções inviáveis, sendo necessário, nesses casos, incorporar regras adicionais para viabilizar tais soluções.

Para ilustrar este fato, considere dois cromossomos pais:

$$p_1 = (1\ 2\ 3 \mid 4\ 5\ 6) \text{ associado à rota } t_1 = \{1\ 2\ 3\ 4\ 5\ 6\ 1\}$$
$$p_2 = (3\ 5\ 6 \mid 2\ 1\ 4) \text{ associado à rota } t_1 = \{1\ 4\ 3\ 5\ 6\ 2\ 1\}$$

Neste caso, são gerados dois cromossomos filhos O_1 e O_2:

$$O_1 = (p_1^1 \mid p_2^2) = (1\ 2\ 3 \mid 2\ 1\ 4)$$
$$O_2 = (p_2^1 \mid p_1^2) = (3\ 5\ 6 \mid 4\ 5\ 6)$$

As rotas associadas aos cromossomos filhos O_1 e O_2 são inviáveis porque em O_1 foi gerada uma rota contendo apenas as cidades 1, 2, 3 e 4 repetindo as visitas às cidades $i = 1$ e $i = 2$. Igualmente, em O_2 foram repetidas visitas em $i = 5$ e $i = 6$ e não foram visitadas as cidades $i = 1$ e $i = 2$.

A seguir, são mostrados alguns operadores *crossover* típicos para o Problema do Caixeiro Viajante, os quais geram filhos viáveis.

1. Operador *Crossover* PMX:

   O operador *Partial Mapped Crossover* (PMX), proposto em [24], trabalha da seguinte forma: considere dois cromossomos pais p_1 e p_2. Selecione dois cortes em p_1 e p_2 aleatoriamente. No caso de serem gerados dois cromossomos filhos O_1 e O_2, os genes localizados entre os dois cortes de p_1 e p_2 são herdados integralmente por O_2 e O_1, respectivamente, preservando a ordem e a posição de cada cidade.

   Para ilustrar seu funcionamento considere o seguinte exemplo:

   $$p_1 = (1\ 2\ 3 \mid 4\ 5\ 6\ 7 \mid 8\ 9)$$
   $$p_2 = (4\ 2\ 6 \mid 1\ 8\ 5\ 9 \mid 3\ 7)$$

   Então:

   $$O_1 = (\text{X}\ \text{X}\ \text{X} \mid 1\ 8\ 5\ 9 \mid \text{X}\ \text{X})$$
   $$O_2 = (\text{X}\ \text{X}\ \text{X} \mid 4\ 5\ 6\ 7 \mid \text{X}\ \text{X})$$

A seguir tenta-se preencher cada componente X de O_1 pela componente correspondente de p_1 e as de O_2 com p_2, caso não formem uma rota ilegal para o PCV.

$$\begin{array}{c} \downarrow \\ O_1 = (\text{X } 2\ 3 \mid 1\ 8\ 5\ 9 \mid \text{X X}) \\ O_2 = (\text{X } 2\ \text{X} \mid 4\ 5\ 6\ 7 \mid 3\ \text{X}) \end{array}$$

Considere agora o primeiro X de O_1 (que deveria ser $X = 1$, mas forneceria uma rota ilegal, já que esta cidade já se encontra presente na rota). Como $X = 1 \in p_1$ não é possível, a cidade correspondente à posição em que está a cidade 1 de p_1 no segundo pai p_2, é a cidade 4. Assim, esta cidade é alocada no primeiro X de O_1.

$$\begin{array}{c} \text{X} \\ \| \\ p_1 = (1 \ldots \\ \downarrow \\ p_2 = (4 \ldots \end{array}$$

Obtém-se com isto:

$$O_1 = (4\ 2\ 3 \mid 1\ 8\ 5\ 9 \mid \text{X X})$$

Repete-se o procedimento para o próximo $X \in O_1$ ($X = 8 \in p_1$)

$$\begin{array}{c} \text{X} \\ \| \\ p_1 = (\ldots \mid \ldots \mid 8 \ldots) \\ \downarrow \\ p_2 = (\ldots \mid \ldots \mid 3 \ldots) \end{array}$$

Mas $X = 3$ já está presente em O_1, então toma-se o n° de p_2 associado ao n° 3 de p_1.

$$\begin{array}{c} \text{X} \\ \| \\ p_1 = (\ldots\ 3 \mid \ldots \\ \downarrow \\ p_2 = (\ldots\ 6 \mid \ldots \end{array}$$

Como $X = 6$ ainda não está presente em O_1, substitui-se X por 6 em O_1, obtendo-se:

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\quad \text{X} \\
\qquad\qquad\qquad\qquad\qquad\quad \| \\
p_1 = (\ldots \mid \ldots \mid \ldots\ 9) \\
\qquad\qquad\qquad\qquad\qquad\quad \downarrow \\
p_2 = (\ldots \mid \ldots \mid \ldots\ 7)
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\quad \downarrow \\
O_1 = (4\ 2\ 3 \mid 1\ 8\ 5\ 9 \mid 6\ \text{X})
\end{array}
$$

Repete-se o procedimento para o próximo $X \in O_1$ ($X = 9 \in p_1$)

Como $X = 7$ ainda não está presente em O_1, substitui-se X por 7 em O_1, obtendo-se:

$$\boxed{O_1 = (4\ 2\ 3 \mid 1\ 8\ 5\ 9 \mid 6\ 7)}$$

Analogamente, preenchem-se os X's de O_2, obtendo-se ao seu final:

$$\boxed{O_2 = (1\ 2\ 8 \mid 4\ 5\ 6\ 7 \mid 3\ 9)}$$

O operador PMX explora a conveniência ou não de se reproduzir cromossomos **localmente idênticos** a um cromossomo pai. Esta propriedade é útil, por exemplo, quando se têm rotas já localmente otimizadas.

2. Operador *crossover* OX:

   Proposto em [7], o operador *Order Crossover* (OX) constrói um *offspring* (cromossomo filho) escolhendo uma subseqüência de uma rota associado a um cromossomo pai p_1 e preservando a ordem relativa das cidades do outro cromossomo pai p_2.

   Exemplo: Considere dois cromossomos pais:

$$
\begin{array}{l}
p_1 = (1\ 2\ 3 \mid 4\ 5\ 6\ 7 \mid 8\ 9) \\
p_2 = (4\ 5\ 2 \mid 1\ 8\ 7\ 6 \mid 9\ 3)
\end{array}
$$

   Os filhos O_1 e O_2 herdam a faixa entre os dois cortes de p_1 e p_2, respectivamente.

$$
\begin{array}{l}
O_1 = (\text{X}\ \text{X}\ \text{X} \mid 4\ 5\ 6\ 7 \mid \text{X}\ \text{X}) \\
O_2 = (\text{X}\ \text{X}\ \text{X} \mid 1\ 8\ 7\ 6 \mid \text{X}\ \text{X})
\end{array}
$$

Agora, partindo-se do segundo corte de um pai p_2, copia-se em uma lista as **n** cidades de p_2. A seguir, remove-se desta lista as cidades contidas entre os dois cortes do outro pai (p_1). A subseqüência resultante é enxertada no filho O_1 associado a p_1 a partir do segundo corte seguindo a ordem da subseqüência.

Assim, considerando o exemplo, partindo-se de p_2 tem-se a lista:

$$9-3-4-5-2-1-8-7-6$$

Removem-se 4, 5, 6 e 7 desta lista, uma vez que estas cidades já foram visitadas, resultando na subseqüência: $9-3-2-1-8$. Esta subseqüência é enxertada em O_1, nesta ordem, a partir do seu segundo corte.

$$O_1 = (\text{X X X} \mid 4\ 5\ 6\ 7 \mid \text{X X})$$

ou

$$\boxed{O_1 = (2\ 1\ 8 \mid 4\ 5\ 6\ 7 \mid 9\ 3)}$$

Analogamente, partindo de p_1, tem-se a lista $8-9-1-2-3-4-5-6-7$. Removendo-se 1, 8, 7 e 6 tem-se a subseqüência: $9-2-3-4-5$, a qual é enxertada em O_2 a partir do segundo corte, obtendo-se:

$$\boxed{O_2 = (3\ 4\ 5 \mid 1\ 8\ 7\ 6 \mid 9\ 2)}$$

O operador OX prioriza a ordem das cidades e não suas posições na rota. A posição das cidades na rota não é importante no PCV e sim, a ordem de visitas, ou seja, no PCV é importante saber quem são os vizinhos de uma dada cidade i.

A irrelevância das posições na rota pode ser vista no exemplo a seguir:

$$T_1 : 5-3-1-2-4$$
$$T_2 : 3-1-2-4-5$$

Em T_1 e T_2 as cidades aparecem em posições diferentes, mas tanto a rota T_1 quanto a rota T_2 representam a mesma solução.

3. Operador *crossover* CX:

   Proposto em [44], o operador *Cycle Crossover* (CX) preserva a posição absoluta das cidades nos cromossomos pais.

   Para ilustrá-lo, considere o seguinte exemplo com dois cromossomos pais:

$$p_1 = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9)$$
$$p_2 = (4\ 1\ 2\ 8\ 7\ 6\ 9\ 3\ 5)$$

O primeiro filho O_1 é obtido tomando-se, inicialmente, a primeira cidade de p_1, obtendo-se:

$$O_1 = (1\ X\ X\ X\ X\ X\ X\ X\ X)$$

Com a 1ª posição de O_1 já preenchida, então o elemento da 1ª posição de p_2 igual a 4 não pode ser também alocado à 1ª posição de O_1. Portanto, o elemento da 1ª posição de p_2, igual a 4, é herdado pelo filho O_1 na posição que ele ocupa em p_1. Ou seja, o n° 4 está na 4ª posição de p_1, logo estará também na 4ª posição de O_1.

$$O_1 = (1\ X\ X\ 4\ X\ X\ X\ X\ X)$$

A seguir, estando o elemento da 4ª posição do outro pai $p_2 = 8$, e o inteiro 8 em p_1 (na oitava posição), aloca-se 8 na oitava posição de O_1.

$$O_1 = (1\ X\ X\ 4\ X\ X\ X\ 8\ X)$$

Seguindo esse procedimento, tem-se:

$$O_1 = (1\ 2\ 3\ 4\ X\ X\ X\ 8\ X)$$

O último inteiro alocado foi 2, mas na 2ª posição em p_2 está o inteiro 1 que já está em O_1. Neste caso, diz-se que foi completado um ciclo.

A partir daí, as cidades restantes de O_1 são preenchidas do outro pai (p_2), obtendo-se:

$$O_1 = (1\ 2\ 3\ 4\ 7\ 6\ 9\ 8\ 5)$$

De forma semelhante, tomando inicialmente a 1ª cidade do segundo cromossomo pai p_2, começa-se a construir o segundo cromossomo filho O_2, que ao final será da forma:

$$O_2 = (4\ 1\ 2\ 8\ 5\ 6\ 7\ 3\ 9)$$

Observe que o operador **Cycle Crossover** sempre preserva a posição de elementos de um outro cromossomo pai.

4. Operador *crossover* ERX:

   O operador *Edge Recombination* (ERX) foi desenvolvido especialmente para o PCV, já que prioriza o fator **adjacência**. Outra característica principal do ERX é a de que um cromossomo filho deve ser construído sempre que possível a partir das arestas presentes em ambos os pais. Em [61], os autores observaram que em média 95% das arestas dos pais são transferidas para os filhos, reduzindo consideravelmente o percentual de arestas selecionadas aleatoriamente.

   No ERX, o significativo número de arestas transferidas de cromossomos pais para cromossomos filhos é uma conseqüência do seguinte procedimento:

   Criar uma lista de arestas de ambos os pais. A lista de arestas produz para cada cidade, todas as outras cidades a ela conectadas em pelo menos um dos cromossomos pais. Isto significa que para cada cidade, existem no mínimo duas e no máximo quatro cidades conectadas.

   Exemplo: Considere duas rotas do PCV, ou, equivalentemente, na representação por caminhos, dois cromossomos.

$$p_1 = (1\ 2\ 3\ 4\ 5\ 6)$$
$$p_2 = (3\ 4\ 1\ 6\ 2\ 5)$$

   A lista de arestas será:

cidade 1: (1, 2), (6, 1), (4, 1)
cidade 2: (1, 2), (2, 3), (6, 2), (2, 5)
cidade 3: (2, 3), (3, 4), (5, 3)
cidade 4: (3, 4), (4, 5), (4, 1)
cidade 5: (4, 5), (5, 6), (2, 5), (5, 3)
cidade 6: (5, 6), (6, 1), (6, 2)

   Neste caso supõe-se que $(i, j) = (j, i)$.

   Construção do Cromossomo Filho:

   O operador ERX pode, a partir de dois cromossomos pais, gerar um ou dois filhos. A versão para gerar um filho é a seguinte:

   Selecione a cidade inicial de um dos pais (no caso, cidade 1 ou cidade 3). A cidade inicial é àquela associada ao menor número de arestas, conforme será justificado mais adiante. Como no exemplo considerado houve empate, escolhe-se arbitrariamente a cidade 1.

A cidade 1 está diretamente ligada às cidades 2, 4 e 6. A próxima cidade no cromossomo filho será dentre estas cidades, aquela que possui o menor número de arestas (no exemplo, escolhe-se a cidade 4). A rota parcial será, então: $1 - 4 - \ldots$ ou $O_1 = (1\ 4\ \mathrm{X}\ \mathrm{X}\ \mathrm{X}\ \mathrm{X})$.

Dentre as cidades conectadas diretamente à cidade 4, seleciona-se aquela que possui o menor número de arestas e que ainda não esteja na rota parcial O_1. Caso não existam candidatos, a seleção da próxima aresta é feita aleatoriamente dentre as cidades não pertencentes à rota parcial. No exemplo, os vizinhos de 4 são: 3, 5 e 1, mas a cidade 1 já está na rota, então escolhe-se entre 3 e 5. Como a cidade 3 possui menos arestas, ela é selecionada, obtendo:

$$O_1 = (1\ 4\ 3\ \mathrm{X}\ \mathrm{X}\ \mathrm{X})$$

Seguindo o mesmo procedimento, escolhe-se a cidade 2:

$$O_1 = (1\ 4\ 3\ 2\ \mathrm{X}\ \mathrm{X})$$

A próxima cidade a ser escolhida é a 6:

$$O_1 = (1\ 4\ 3\ 2\ 6\ \mathrm{X})$$

Finalmente, a cidade 5 é incluida na rota:

$$O_1 = (1\ 4\ 3\ 2\ 6\ 5)$$

Observe que o cromossomo filho O_1 foi, neste exemplo, construído inteiramente de arestas de um dos pais sem a necessidade de gerar arestas aleatoriamente. Isso, em parte, se deve ao critério da escolha da próxima cidade.

Escolhendo a cidade com o menor número de arestas na lista de arestas, a tendência é obter uma rota com arestas herdadas de um dos pais.

A idéia de iniciar com cidades com poucas arestas, deixando-se para o final as cidades com o número maior de arestas (vizinhos), é que o risco de ter que gerar uma aresta aleatoriamente cresce apenas no final do procedimento, devido ao maior número de cidades já incluídas na rota parcial.

Uma variante do ERX, chamada EERX (*Enhanced Edge Recombination*) prioriza as arestas comuns aos dois pais, para ser herdada pelo cromossomo filho. Para distinguir estes casos, basta notar que se uma cidade possui **duas arestas** então, necessariamente, as duas serão comuns aos dois pais. Se uma cidade possui **três arestas**, uma destas arestas será comum aos dois pais e, finalmente, se uma cidade possui **quatro arestas** então não existe nenhuma aresta em comum.

4.9 *Scatter Search*

Scatter Search ou Busca Dispersa é um método de busca populacional que consiste em construir soluções pela combinação de outras soluções, tendo sua origem em estratégias originalmente propostas no contexto de programação inteira.

A Busca Dispersa é projetada para trabalhar com um conjunto de soluções, denominado *Conjunto de Referência*, o qual contém boas soluções obtidas ao longo da busca preguessa. O conceito de uma solução de *boa* qualidade vai além do seu valor propriamente dito da função de avaliação e inclui critérios especiais tais como diversidade. O método gera combinações de soluções de referência para criar novas soluções do espaço de busca.

O método é organizado para (1) capturar informações não contidas separadamente nas soluções originais, (2) levar vantagem de métodos de solução heurística auxiliares (para avaliar as combinações produzidas e gerar novas soluções) e (3) fazer uso de estratégias específicas ao invés de estratégias aleatórias de busca. A Busca Dispersa envolve basicamente cinco procedimentos:

1. Um procedimento de *Diversificação*, para gerar um conjunto de soluções diversificadas, usando uma ou mais soluções aleatórias como entrada;

2. Um procedimento de *Refinamento*, para transformar uma solução em uma ou mais soluções melhoradas;

3. Um procedimento de *Atualização do Conjunto de Referência*, para construir e manter o *Conjunto de Referência*, o qual contém as *nbest* melhores soluções encontradas (sendo o valor de *nbest* tipicamente pequeno, por exemplo, não mais que 20 [22]). Vários critérios alternativos podem ser usados para incluir ou remover soluções do conjunto de referência.

4. Um procedimento de *Geração de Subconjuntos* para operar o conjunto de referência e escolher um subconjunto de suas soluções como base para criar combinações de soluções. O procedimento de geração de subconjuntos mais comum consiste em gerar todos os pares de soluções de referência, isto é, todos os subconjuntos de tamanho 2;

5. Um procedimento de *Combinação de Soluções* para transformar um dado subconjunto de soluções produzidas pelo procedimento *Geração de Subconjuntos* em uma ou mais soluções combinadas. O procedimento de combinação é análogo ao operador *crossover* de Algoritmos Genéticos, embora ele seja capaz de combinar mais do que duas soluções.

O procedimento da Figura 22, que descreve o método, inicia com a criação de um conjunto de referência (*RefSet*). Para isso, o procedimento *Diversificação* é usado para construir um conjunto P de soluções diversificadas. A cardinalidade de P, *PSize*, é tipicamente 10 vezes o tamanho de *RefSet* [22]. Inicialmente, o conjunto de referência *RefSet* consiste de *nbest* soluções distintas e bastante diversas retiradas de P. As soluções em *RefSet* são ordenadas de acordo com sua qualidade, sendo a melhor colocada na primeira posição da lista. A busca

procedimento *ScatterSearch*
1 $P \leftarrow \emptyset$;
Use o procedimento *Diversificação* para construir uma solução x;
Se $x \notin P$ então adicione x a P, isto é, $P \leftarrow P \cup \{x\}$; caso contrário, descarte x;
Repita este procedimento até que $|P| = PSize$;
Construa $RefSet = \{x^1, \cdots, x^{nbest}\}$, com $nbest$ soluções diversificadas de P;
2 Avalie as soluções em *RefSet* e ordene-as de acordo com a função de avaliação;
(Considere x^1 a melhor solução e x^{nbest} a pior)
$NewSolutions \leftarrow TRUE$;
enquanto (*NewSolutions*) faça
3 Gere *NewSubsets*, isto é, todos os pares de soluções de *RefSet*
desde que haja pelo menos uma nova solução, isto é, que $NewSolutions = TRUE$;
$NewSolutions \leftarrow FALSE$;
enquanto ($NewSubsets \neq \emptyset$) faça
4 Selecione o próximo subconjunto s em *NewSubsets*;
5 Aplique *Combinação de Soluções* a s para obter uma ou mais soluções x;
se ($x \notin RefSet$ e $f(x) < f(x^{nbest})$ então
6 $x^{nbest} \leftarrow x$ e reordene *RefSet*;
7 $NewSolutions \leftarrow TRUE$;
fim-se
8 Remova s de *NewSubsets*;
fim-enquanto;
fim-enquanto;
fim *ScatterSearch*;

Figura 22: Algoritmo *Scatter Search*

é então iniciada atribuindo-se o valor *TRUE* à variável booleana *NewSolutions*. No passo 3, *NewSubsets* é construído e *NewSolutions* é trocado para *FALSE*. Considerando subconjuntos de tamanho 2, por exemplo, a cardinalidade de *NewSubsets* (relativa ao conjunto de referência inicial) é dada por $(nbest^2 - nbest)/2$, que corresponde a todos os possíveis pares de soluções em *RefSet*. Os pares em *NewSubsets* são selecionados um por vez em ordem lexicográfica e o procedimento *Combinação de Soluções* é aplicado para gerar uma ou mais soluções no passo 5. Se uma nova solução criada melhora a pior solução do conjunto de referência *RefSet* corrente, então ela a substitui e *RefSet* é reordenado no passo 6. O *flag NewSolutions* é alterado para *TRUE* e o subconjunto s que foi combinado é removido de *NewSubsets* nos passos 7 e 8, respectivamente.

4.10 Colônia de Formigas

A meta-heurística Otimização por Colônia de Formigas, ou simplesmente Colônia de Formigas (*Ant Colony Optimization Metaheuristic - ACO*), tem sua origem na tese de doutorado de Marco Dorigo [9], publicada mais tarde em [10].

O método simula o comportamento de um conjunto de agentes (formigas) que se cooperam para resolver um problema de otimização. A cooperação entre as formigas se dá por meio do feromônio depositado por cada formiga ao se deslocar no espaço de busca, permitindo que esse rastro possa ser usado como informação por outras formigas.

De acordo com [11], informalmente, o comportamento das formigas em um

algoritmo ACO pode ser resumido como segue. Uma colônia de formigas move de forma concorrente e assíncrona construindo caminhos no espaço de busca. Elas movem aplicando uma política de decisão local estocástica, que faz uso das trilhas de feromônio e informações heurísticas. Ao se moverem, as formigas constroem novas soluções para o problema de otimização. Construída uma solução, ou durante a construção de uma solução, a formiga avalia a solução (parcial ou completa) e deposita uma trilha de feromônio apenas nas componentes ou conexões usadas durante o caminho. Esta informação de feromônio é usada para direcionar a busca das outras formigas.

Além das atividades das formigas, um algoritmo ACO inclui dois procedimentos adicionais: evaporação da trilha de feromônio e ações *daemon*, sendo esta última componente opcional. A evaporação de feromônio é o processo pelo qual o feromônio depositado pelas formigas decresce ao longo do tempo. Do ponto de vista prático, a evaporação de feromônio é necessária para evitar uma convergência prematura do algoritmo em uma região subótima. Este procedimento implementa uma forma útil de *esquecimento*, favorecendo a exploração de novas regiões do espaço de busca. As ações *daemon* podem ser usadas para implementar ações centralizadas, as quais não seriam realizadas pelas formigas tomadas isoladamente. Como exemplos destas ações podemos citar a ativação de um procedimento de busca local ou uma coleção de informações globais que podem ser usadas para decidir se é útil ou não depositar feromônio adicional para guiar a busca sob uma perspectiva não local. Como um exemplo prático, o *daemon* pode observar o caminho encontrado por cada formiga da colônia e escolher para depositar uma quantidade extra de feromônio apenas nas componentes usadas pela formiga que construiu a melhor solução. Atualizações de feromônio realizadas por *daemon* são chamadas *atualizações de feromônio off-line*.

Apresenta-se, pela Figura 23, uma implementação básica da meta-heurística Colônia de Formigas aplicada ao Problema do Caixeiro Viajante. Nesta implementação, m é a quantidade de formigas, Q é a quantidade de feromônio depositada por uma formiga após concluir uma rota, Γ_0 é a quantidade inicial de feromônio em cada arco, d_{ij} é a distância entre as cidades i e j, Γ_{ij} é a quantidade de feromônio em cada arco (i, j), $\Delta\Gamma_{ij}^k$ é a quantidade de feromônio depositada por cada formiga k em cada arco (i, j), ρ é a taxa de evaporação de feromônio e $\Delta\Gamma$ é a quantidade de feromônio depositada por todas as formigas no arco (i, j).

Para a obtenção de uma rota para uma formiga (passo 3(b) da Figura 23), o algoritmo pressupõe que a formiga lembra-se das cidades já visitadas. Estando na cidade i, a formiga k escolhe a cidade j, dentre as ainda não visitadas, com uma probabilidade dada pela expressão (8):

$$p_{ij}^k = \frac{[\Gamma_{ij}]^\alpha \times [\eta_{ij}]^\beta}{\sum_{l \in \aleph_i^k} [\Gamma_{il}]^\alpha \times [\eta_{il}]^\beta} \qquad \forall j \in \aleph_i^k \tag{8}$$

em que $\aleph_i^k$ representa o conjunto de cidades ainda não visitadas pela formiga

procedimento *ColoniaFormigas*
1 Seja Q e Γ_0 constantes;
  $f^* \leftarrow \infty$;
2 Faça $\Delta\Gamma_{ij} \leftarrow 0$ e $\Gamma_{ij} \leftarrow \Gamma_0$ para todo arco (i,j);
3 Para (cada formiga $k = 1, \cdots, m$) faça
  (a) Selecione a cidade inicial para a k-ésima formiga
  (b) Obtenha uma rota R^k para cada formiga k
  (c) Seja L^k o comprimento da rota R^k
  (d) se ($L^k < f^*$) então $s^* \leftarrow R^k$ e $f^* \leftarrow f(s^*)$
  (e) Calcule a quantidade de rastro deixado pela formiga k:
    se (arco (i,j) pertence à rota R^k)
      então $\Delta\Gamma_{ij}^k \leftarrow d_{ij} \times Q/L^k$
      senão $\Delta\Gamma_{ij}^k \leftarrow 0$
    fim-se
  (f) Faça $\Delta\Gamma_{ij} \leftarrow \Delta\Gamma_{ij} + \Delta\Gamma_{ij}^k$
4 fim-para;
5 Faça $\Gamma_{ij} \leftarrow (1-\rho) \times \Gamma_{ij} + \Delta\Gamma_{ij} \quad \forall\,(i,j)$
6 se (a melhor rota s^* não foi alterada nas últimas *IterMax* iterações)
7 então *PARE*: s^* é a melhor solução
8 senão Retorne ao Passo 3
9 fim-se
fim *ColoniaFormigas*;

Figura 23: Algoritmo *Colônia de Formigas*

k, $\eta_{ij} = 1/d_{ij}$ é uma informação heurística disponível *a priori* e α e β são dois parâmetros que determinam a influência relativa da trilha de feromônio e da informação heurística, respectivamente. Se $\alpha = 0$ então a probabilidade de seleção é proporcional à $[\eta_{ij}]^\beta$ e as cidades mais próximas têm maiores chances de serem escolhidas. Neste caso, o algoritmo comporta-se como um método guloso estocástico (diz-se estocástico porque as formigas são inicialmente aleatoriamente distribuídas entre o conjunto de cidades e, assim, há múltiplos pontos de partida). Se $\beta = 0$, então somente a utilização de feromônio tem influência, induzindo a uma estagnação rápida da busca, isto é, todas as formigas seguindo o mesmo caminho e construindo as mesmas soluções.

Observe, no passo 3(e) da Figura 23, que a quantidade de feromônio depositada em cada arco (i,j) visitado por uma formiga é proporcional ao comprimento do arco, uma vez que a quantidade unitária de feromônio (Q/L^k) depositada pela formiga k é multiplicada pelo comprimento d_{ij} do arco. Assume-se, também, ao contrário do caso real, que a formiga só deposita o feromônio após concluir a rota e não durante o percurso. Nas implementações apresentadas em [11], considera-se apenas $\Delta\Gamma_{ij}^k \leftarrow 1/L^k$, desprezando-se o comprimento de cada arco e a quantidade Q de feromônio depositada por cada formiga após concluir uma rota.

Uma extensão interessante do algoritmo ACO apresentado na Figura 23

consiste em substituir o passo 5 pela expressão (9):

$$\Gamma_{ij} \leftarrow (1-\rho) \times \Gamma_{ij} + \rho \times \Delta\Gamma_{ij}^{best} \quad \forall\,(i,j) \tag{9}$$

em que $\Delta\Gamma_{ij}^{best}$ representa a trilha de feromônio deixada pela formiga que produziu a melhor solução (pode ser tanto a melhor solução de uma iteração envolvendo m formigas ou a melhor solução global). A atualização da trilha de feromônio é feita, neste caso, usando uma forte estratégia elitista, pois somente o feromônio da formiga que produziu a melhor solução é usado para proceder à atualização da trilha de feromônio em todos os arcos.

Uma outra estratégia elitista consiste em ordenar os comprimentos das rotas geradas pelas m formigas em cada iteração e considerar que apenas as formigas associadas às $w-1$ melhores rotas ($w < m$) e à melhor rota global podem depositar feromônio. A r-ésima melhor formiga da colônia contribui com a atualização de feromônio com um peso dado por $\max\{0, w-r\}$ enquanto a formiga que produziu a melhor solução global contribui com um peso w. Nesta situação, o passo 5 da Figura 23 é substituído pela expressão (10):

$$\Gamma_{ij} \leftarrow (1-\rho) \times \Gamma_{ij} + \sum_{r=1}^{w-1} \left((w-r) \times \Delta\Gamma_{ij}^{r}\right) + w \times \Delta\Gamma_{ij}^{gb} \quad \forall\,(i,j) \tag{10}$$

em que $\Delta\Gamma_{ij}^{gb}$ indica a trilha de feromônio deixada pela formiga que produziu a melhor solução global até então.

4.11 Algoritmos Meméticos

A meta-heurística Algoritmos Meméticos (*Memetic Algorithms*) é uma variação de Algoritmos Genéticos, consistindo no refinamento dos indivíduos antes de se submeterem às operações de recombinação e mutação.

4.12 *Annealing* Microcanônico

Trata-se de uma variante do algoritmo *Simulated Annealing*, proposta em [1]. Diferentemente do SA, que baseia-se na simulação dos estados de um sistema físico a temperatura constante, o algoritmo *Annealing* Microcanônico simula a variação dos estados a energia constante. Utiliza, para esse propósito, uma versão do Algoritmo de Creutz [5, 36], o qual introduz uma variável, chamada de *demônio*, para modelar as flutuações de energia.

A partir de uma solução inicial arbitrária s, geram-se novas soluções de forma que $E + D =$ constante, sendo D a energia do demônio ($0 \leq D \leq D_{max}$) e E a energia do sistema ($f(s)$). O procedimento principal consiste em um *loop* que gera, randomicamente, em cada iteração, um único vizinho s' da solução corrente.

Chamando de Δ a variação de energia ao mover-se de s para s', o método aceita o movimento, e a solução vizinha s' passa a ser a nova solução corrente,

se $\Delta \leq 0$, com a condição de que $D - \Delta \leq D_{max}$, isto é, que a energia liberada não supere a capacidade do demônio. Caso $\Delta > 0$ a solução vizinha s' também pode ser aceita, desde que $\Delta < D$, isto é, que a energia necessária possa ser suprida pelo demônio. Em ambos os casos, se houver aceitação, o demônio D recebe (ou libera, respectivamente) a variação de energia envolvida ao mover-se de s para s', isto é, $D \leftarrow D - \Delta$. Desta forma, a energia total, $E + D$, permanece constante.

O demônio assume, inicialmente, um valor D_{max}. Após um certo número de iterações (o qual representa o número de iterações necessárias para o sistema atingir o equilíbrio térmico numa dada configuração de energia), esse valor é gradativamente diminuído até anular-se.

A vantagem de se usar o algoritmo AM é que pode-se estabelecer, com precisão, se o equilíbrio térmico foi atingido [58]. Da física estatística sabe-se que o valor médio do demônio é igual à temperatura ($\bar{D} = T$) e no equilíbrio térmico tem-se $\bar{D}/\sigma(D) = 1$, onde $\sigma(D)$ é o desvio-padrão dos valores do demônio. Entretanto, sob o ponto de vista prático, é custoso avaliar computacionalmente se tal equilíbrio foi atingido. Assim, ao invés de avaliá-lo durante o progresso da pesquisa, prefixa-se um número máximo de iterações em um dado nível de energia, o qual passa a ser um parâmetro de controle do método.

Os parâmetros de controle do procedimento são, pois, a taxa α de diminuição da capacidade do demônio, o número de iterações em um dado nível de energia ($AMmax$), o valor D_i do demônio no início de cada fase e a capacidade inicial do demônio D_{max}.

Apresentamos, pela Figura 24, o pseudocódigo de um algoritmo *Annealing* Microcanônico básico.

```
procedimento AM(f(.),N(.),AMmax,α,D_i,D_max,s)
1  s* ← s;   {Melhor solução obtida até então}
2  enquanto (D_max > 0) faça
3      D ← D_i;
4      para (IterE = 1, 2, ..., AMmax) faça
5          Gere um vizinho qualquer s' ∈ N(s);
6          Δ = f(s') − f(s);
7          se (Δ ≤ 0)
8              então
9                  se (D − Δ ≤ D_max) então
10                     s ← s';
11                     D ← D − Δ;
12                     se (f(s') < f(s*)) então s* ← s';
13                 fim-se;
14             senão
15                 se (D − Δ ≥ 0) então
16                     s ← s';
17                     D ← D − Δ;
18                 fim-se;
19         fim-se;
20     fim-para;
21     D_max ← α × D_max;
22 fim-enquanto;
23 s ← s*;
24 Retorne s;
fim AM;
```

Figura 24: Algoritmo *Annealing* Microcanônico

4.13 Otimização Microcanônica

Trata-se de uma alternativa ao *Annealing* Microcanônico, proposta originalmente em [59]. Referenciamos a [59, 36, 58] para uma descrição mais detalhada do método.

O algoritmo de Otimização Microcanônica (OM), descrito pela Figura 25, consiste de dois procedimentos, os quais são aplicados alternadamente: inicialização e amostragem.

Na fase de inicialização, OM realiza somente movimentos de melhora, guiando a uma configuração de mínimo local. Para tentar escapar desta configuração, executa-se a amostragem, fase na qual um grau extra de liberdade (denominada *demônio*) produz pequenas perturbações na solução corrente. Em cada iteração desta fase, movimentos randômicos são propostos, sendo aceitos apenas aqueles nos quais o demônio é capaz de absorver ou liberar a diferença de custo envolvida. Após a fase de amostragem, uma nova inicialização é realizada e o algoritmo assim prossegue, alternando entre as duas fases, até que uma

```
procedimento OM(f(.),N(.),OMmax,s)
1  s* ← s;                 {Melhor solução obtida até então}
2  Iter ← 0;               {Número de iterações}
3  MelhorIter ← 0;         {Iteração mais recente que forneceu s*}
4  enquanto (Iter − MelhorIter < OMmax) faça
5      Iter ← Iter + 1;
6      InicializacaoOM(f(.),N(.),Imax,s);
7      se (f(s) < f(s*)) então
8          s* ← s;
9          MelhorIter ← Iter;
10     fim-se;
11     AmostragemOM(f(.),N(.),D_i,D_max,Amax,s);
12 fim-enquanto;
13 s ← s*;
14 Retorne s;
fim OM;
```

Figura 25: Algoritmo de Otimização Microcanônica

condição de parada seja satisfeita.

O demônio é definido por dois parâmetros: sua capacidade D_{max} e seu valor inicial D_i. A fase de amostragem gera uma sequência de estados cuja energia é conservada, exceto para pequenas flutuações, as quais são modeladas pelo demônio. Chamando de E_i a energia (custo) da solução obtida na fase de inicialização, de D e E a energia do demônio e da solução, respectivamente, em um dado instante da fase de amostragem, tem-se $E + D = E_i + D_i =$ constante. Portanto, esta fase gera soluções no intervalo $[E_i - D_{max} + D_i,\ E_i + D_i]$.

Os parâmetros principais de controle do algoritmo OM são o número máximo $OMmax$ de iterações consecutivas sem melhora em OM, o número máximo de iterações consecutivas sem melhora na fase de inicialização $Imax$, o número máximo de iterações de amostragem $Amax$, o valor inicial D_i do demônio e a sua capacidade máxima D_{max}.

Apresentamos, pelas figuras 26 e 27, os pseudocódigos das fases de inicialização e amostragem, respectivamente, de um algoritmo de Otimização Microcanônica básico.

```
procedimento InicializacaoOM(f(.),N(.),Imax,s)
1  s* ← s;                 {Melhor solução obtida até então}
2  Iter ← 0;               {Número de iterações}
3  MelhorIter ← 0;         {Iteração mais recente que forneceu s*}
4  enquanto (Iter − MelhorIter < Imax) faça
5      Iter ← Iter + 1;
6      Gere um vizinho qualquer s' ∈ N(s);
7      Δ = f(s') − f(s);
8      se (Δ < 0)
9          então
10             s ← s';
11             se (f(s') < f(s*)) então
12                 s* ← s';
13                 MelhorIter ← Iter;
14             fim-se;
15         senão Ponha Δ na lista dos movs rejeitados;
16     fim-se;
17 fim-enquanto;
18 s ← s*;
19 Retorne s;
fim InicializacaoOM;
```

Figura 26: Fase de inicialização do algoritmo de Otimização Microcanônica

```
procedimento AmostragemOM(f(.),N(.),D_i,D_max,Amax,s)
1  Seja s solução advinda da fase de inicialização;
2  Escolha D_max e D_I da lista de movimentos rejeitados;
3  D ← D_i;        {Valor corrente do demônio}
4  para (Iter = 1, 2, ..., Amax) faça
5      Gere um vizinho qualquer s' ∈ N(s);
6      Δ = f(s') − f(s);
7      se (Δ ≤ 0)
8          então
9              se (D − Δ ≤ D_max) então
10                 s ← s';
11                 D ← D − Δ;
12             fim-se;
13         senão
14             se (D − Δ ≥ 0) então
15                 s ← s';
16                 D ← D − Δ;
17             fim-se;
18     fim-se;
19 fim-para;
20 Retorne s;
fim AmostragemOM;
```

Figura 27: Fase de amostragem do algoritmo de Otimização Microcanônica

5 Técnicas especiais de intensificação e diversificação

5.1 Reconexação por Caminhos

A técnica Reconexão por Caminhos ou Religação de Caminhos, conhecida na literatura inglesa como *Path Relinking*, foi proposta em [18] como uma estratégia de intensificação para explorar trajetórias que conectavam soluções elite obtida pelo método Busca Tabu ou *Scatter Search* [21].

Esta busca por soluções de melhor qualidade consiste em gerar e explorar caminhos no espaço de soluções partindo de uma ou mais soluções elite e levando a outras soluções elite. Para tal finalidade, são selecionados movimentos que introduzem atributos das soluções guia na solução corrente. Desse modo, a Reconexão por Caminhos pode ser vista como uma estratégia que objetiva incorporar atributos de soluções de boa qualidade, favorecendo a seleção de movimentos que as contenham.

A Reconexão por Caminhos pode ser aplicada segundo duas estratégias básicas [54]:

- Reconexão por Caminhos aplicada como uma estratégia de pós-otimização entre todos os pares de soluções elite;
- Reconexão por Caminhos aplicada como uma estratégia de intensificação a cada ótimo local obtido após a fase de busca local.

A aplicação da técnica de Reconexão por Caminhos como um procedimento de intensificação a cada ótimo local é mais eficaz do que empregá-la como um procedimento de pós-otimização [54]. Neste caso, a Reconexão por Caminhos é aplicada a pares (s_1, s_2) de soluções, onde s_1 é a solução corrente obtida após o procedimento de busca local e s_2 é uma solução selecionada aleatoriamente de um conjunto formado por um número limitado, *TamConjElite*, de soluções elite encontradas durante a exploração do espaço de soluções. Este conjunto está, inicialmente, vazio. Cada solução obtida ao final de uma busca local é considerada como uma candidata a ser inserida no conjunto elite, desde que ela seja melhor que a solução de pior qualidade desse conjunto e apresente um percentual mínimo de diferença em relação a cada solução do conjunto elite (*PercDif*). Esta estratégia é adotada para evitar que o conjunto elite contenha soluções muito parecidas. Se o conjunto estiver vazio, a solução é simplesmente inserida no conjunto. Se o conjunto elite já possui *TamConjElite* soluções e a solução corrente é candidata a ser inserida neste conjunto, então esta substitui a solução de pior qualidade.

O algoritmo inicia computando a diferença simétrica $\Delta(s_1, s_2)$ entre s_1 e s_2, resultando no conjunto de movimentos que deve ser aplicado a uma delas, dita solução inicial, para alcançar a outra, dita solução guia. A partir da solução inicial, o melhor movimento ainda não executado de $\Delta(s_1, s_2)$ é aplicado à solução corrente até que a solução guia seja atingida. A melhor solução encontrada ao longo desta trajetória é considerada como candidata à inserção no conjunto

elite e a melhor solução já encontrada é atualizada. Diversas alternativas têm sido consideradas e combinadas em implementações como as de [54]:

- Não aplicar Reconexão por Caminhos a cada iteração, mas sim periodicamente, para não onerar o tempo de execução do algoritmo;
- Explorar duas trajetórias potencialmente diferentes, usando s_1 como solução inicial e depois s_2 no mesmo papel;
- Explorar apenas uma trajetória, usando s_1 ou s_2 como solução inicial e
- Não percorrer a trajetória completa de s_1 até s_2, mas sim apenas parte dela (Reconexão por Caminhos Truncada).

Para a escolha da alternativa mais apropriada, deve-se ter um compromisso entre tempo de processamento e qualidade da solução. Em [51] foi observado que a exploração de duas trajetórias potencialmente diferentes para cada par (s_1, s_2) de soluções consome, aproximadamente, o dobro do tempo de processamento necessário para explorar apenas uma delas, com um ganho marginal muito pequeno em termos de qualidade de solução. Também foi observado pelos autores que, se apenas uma trajetória deve ser investigada, melhores soluções tendem a ser obtidas quando a Reconexão por Caminhos usa como solução inicial a melhor solução dentre s_1 e s_2 (Esta é a chamada Reconexão por Caminhos Regressiva - *Backward Path Relinking*). Como a vizinhança da solução inicial é explorada com muito mais profundidade do que aquela da solução guia, usar como solução inicial a melhor dentre s_1 e s_2 oferece mais chances ao algoritmo de investigar mais detalhadamente a vizinhança da solução mais promissora. Pela mesma razão, as melhores soluções são normalmente encontradas próximas da solução inicial, permitindo que o procedimento de Reconexão por Caminhos seja interrompido após algumas iterações, antes de a solução guia ser alcançada.

Na Figura 28 representa-se o pseudocódigo do procedimento de reconexão por caminhos para um problema de minimização.

A Figura 28 mostra que o algoritmo de Reconexão por Caminhos unidirecional inicia determinando o conjunto de movimentos $\Delta(s,g)$ que será aplicado a s (solução inicial) até chegar a g (solução guia) (linha 3). Cada iteração do procedimento de reconexão por caminhos unidirecional possui os quatro seguintes passos:

- aplicar à solução corrente $\bar{g}$ o melhor movimento do conjunto de movimentos (linha 5), obtendo a solução g'';
- excluir o melhor movimento do conjunto de movimentos ainda possível (linha 6);
- atualizar a solução corrente (linha 7); e
- testar se a solução corrente, $\bar{g}$, é melhor que a melhor solução, g', encontrada ao longo da trajetória aplicada a s para chegar a g. Em caso afirmativo, atribui-se $\bar{g}$ a g' (linha 9). No início do método de reconexão por caminhos, atribui-se a g' a melhor solução dentre s e g (linha 2).

Procedimento `Reconexão-Caminhos`

1: $\bar{g} \leftarrow s$;
2: Atribuir a g' a melhor solução entre s e g;
3: Calcular o conjunto de movimentos possíveis $\Delta(s,g)$;
4: **while** $|\Delta(s,g)| \neq 0$ **do**
5: Atribuir a g'' a melhor solução obtida aplicando o melhor movimento de $\Delta(s,g)$ a $\bar{g}$;
6: Excluir de $\Delta(s,g)$ este movimento;
7: $\bar{g} \leftarrow g''$;
8: **if** $f(\bar{g}) < f(g')$ **then**
9: $g' \leftarrow \bar{g}$;
10: **end if**
11: **end while**
12: Retorne g';

Figura 28: Procedimento de Reconexão por Caminhos

Quando a solução corrente chega a g, o algoritmo de Reconexão por Caminhos se encerra (linha 4 da Figura 28) e retorna a solução g' (linha 12).

A Figura 29 ilustra o funcionamento da Reconexão por Caminhos em um problema de programação de tripulações (*Crew Scheduling Problem*). Nesta figura representa-se uma solução por uma matriz cujos elementos $(1,\ldots,9)$ são tarefas a serem alocadas a jornadas de trabalho. Cada linha da matriz constitue uma jornada de trabalho.

Para computar a diferença $\Delta(s_1,s_2)$ entre as soluções inicial e guia, verifica-se, inicialmente, se cada jornada da solução guia está na solução inicial. Para tornar este passo mais otimizado, pode-se utilizar o seguinte esquema de comparação entre jornadas:

1. Compara-se, inicialmente, a função de avaliação de cada jornada envolvida. Se forem diferentes, conclui-se que as jornadas são diferentes. Caso contrário, passa-se para o passo seguinte;

2. Compara-se o número de tarefas de cada jornada envolvida. Se forem diferentes, conclui-se que as jornadas são diferentes. Caso contrário, analisa-se o passo seguinte;

3. Compara-se tarefa por tarefa de cada jornada envolvida. Se houver alguma tarefa diferente, conclui-se que as jornadas são diferentes; caso contrário, as jornadas são iguais.

Após ser computada a diferença entre as soluções inicial e guia, incorpora-se na solução inicial cada jornada presente na diferença. A seguir é fixada a jornada adicionada e feita a consistência da solução, eliminando-se as tarefas redundantes, confome ilustrado na Figura 29(a). Cada solução intermediária é então submetida a uma busca local mantendo-se fixa a jornada adicionada. O

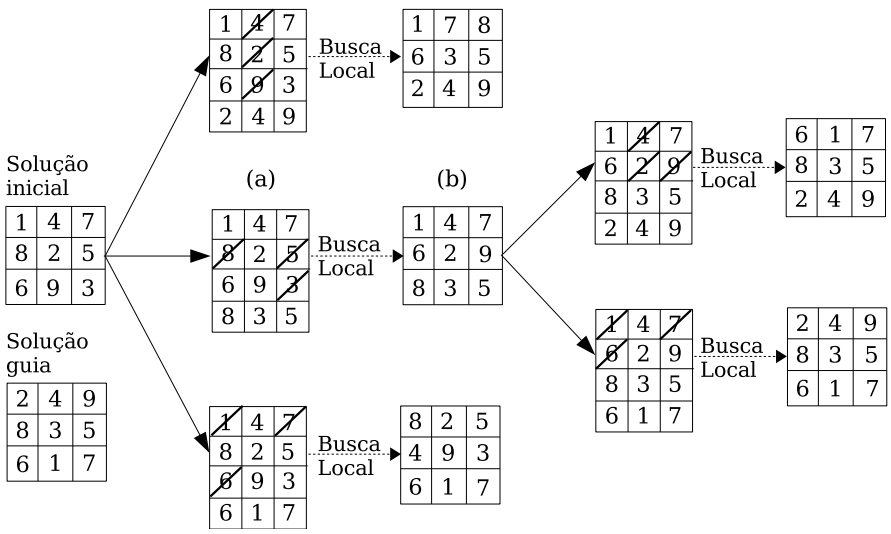


Figura 29: Mecanismo de intensificação Reconexão por Caminhos aplicado ao PPT

procedimento Reconexão por Caminhos prossegue com a melhor solução intermediária (Figura 29(b)). Como a solução intermediária desse exemplo incorporou a segunda jornada da solução guia, então são analisadas a incorporação à solução corrente da primeira e terceira jornadas da solução guia. Excluída a redundância das soluções e aplicada a busca local (mantendo-se fixas as jornadas incorporadas), o procedimento é encerrado, retornando a melhor das soluções intermediárias encontradas.

Ilustremos, agora, a aplicação do procedimento RC em uma instância do PCV. Seja $sC = (1, 3, 5, 2, 4)$ a solução corrente e $sG = (4, 5, 1, 3, 2)$ a solução guia. Vamos supor que $f(sC) = 100$ e $f(sG) = 120$, isto é, que estamos aplicando a Reconexão por Caminhos Regressiva.

Inicialmente, devemos definir qual é o atributo da solução guia que deve ser inserido na solução corrente a cada iteração. Uma opção é definir como atributo a ordem (posição) de visita de uma cidade. Neste caso, a cidade 4 é a primeira a ser visitada na solução guia, a cidade 5 é a segunda e assim por diante.

Assim sendo, vejamos como inserir esses atributos na solução corrente. A primeira iteração da RC consiste em escolher, dentre os cinco possíveis atributos (cidade 4 na primeira posição, cidade 5 na segunda posição, cidade 1 na terceira posição, cidade 3 na terceira posição e cidade 2 na quinta posição), qual deles traz um melhor valor para a função de avaliação.

1. Inserção do primeiro atributo da solução guia: Inserindo a cidade 4 na primeira posição de sC no lugar da cidade 1, então esta última cidade deve ocupar o lugar da cidade 4, isto é, a quinta posição de sC, isto é, $sC^1 = (\mathbf{4}, 3, 5, 2, \mathbf{1})$.

2. Inserção do segundo atributo da solução guia: Inserindo a cidade 5 na segunda posição de sC no lugar da cidade 3, então esta última cidade deve ocupar o lugar da cidade 5, isto é, a terceira posição de sC, isto é, $sC^2 = (1, \mathbf{5}, \mathbf{3}, 2, 4)$.

3. Inserção do terceiro atributo da solução guia: Inserindo a cidade 1 na terceira posição de sC no lugar da cidade 5, então esta última cidade deve ocupar o lugar da cidade 1, isto é, a primeira posição de sC, isto é, $sC^3 = (\mathbf{5}, 3, \mathbf{1}, 2, 4)$.

4. Inserção do quarto atributo da solução guia: Inserindo a cidade 3 na quarta posição de sC no lugar da cidade 2, então esta última cidade deve ocupar o lugar da cidade 3, isto é, a segunda posição de sC, isto é, $sC^4 = (1, \mathbf{2}, 5, \mathbf{3}, 4)$.

5. Inserção do quinto atributo da solução guia: Inserindo a cidade 2 na quinta posição de sC no lugar da cidade 4, então esta última cidade deve ocupar o lugar da cidade 2, isto é, a quarta posição de sC, isto é, $sC^5 = (1, 3, 5, \mathbf{4}, \mathbf{2})$.

Suponhamos que a melhor dessas soluções intermediárias seja sC^4, isto é, o valor de $f(sC^4) = \min\{f(sC^j)\ \forall j = 1, \cdots, 5\}$. Então, sobre essa solução intermediária sC^4 aplicamos um procedimento de busca local com a condição de que a quarta cidade a ser visitada (ou seja, a cidade 3) não seja alterada dessa posição. Essa condição é necessária pois esse é um atributo da solução guia e que deve ser alcançado. Ao aplicar a busca local, a ordem de visita das demais cidades pode ser alterada. Vamos supor que após a busca local sobre sC^4 a solução resultante seja o ótimo local $sC'' = (5, 1, 2, \mathbf{3}, 4)$. Esse ótimo local passa a ser a nova solução corrente, isto é, $sC \leftarrow sC''$.

O próximo passo é definir qual o segundo atributo da solução guia a ser inserido na solução corrente sC. Observe que, agora, só há quatro atributos a serem analisados, a saber: (cidade 4 na primeira posição, cidade 5 na segunda posição, cidade 1 na terceira posição e cidade 2 na quinta posição). Evidentemente, a quarta posição não pode ser modificada e permanece fixa durante todo o procedimento. O atributo escolhido é aquele que produz a solução intermediária com a melhor avaliação e sobre a qual será aplicada busca local.

Prosseguindo com este procedimento, fica claro que em $n - 1$ passos, sendo n o número de cidades, alcança-se a solução guia. Ao incorporar atributos da solução guia na solução corrente em cada passo, o procedimento RC pode gerar soluções ainda melhores.

Apresenta-se na Figura 30 o pseudocódigo dos principais blocos de um algoritmo Busca Tabu com intensificação por Reconexão por Caminhos para um problema de minimização.

Na Figura 30, *IterAtivRC* é um parâmetro que representa o número de iterações sem melhora da Busca Tabu a partir do qual deve-se acionar o procedimento Reconexão por Caminhos e *IteracoesRC* indica quantas vezes o procedimento *RC* será acionado.

Algoritmo `BTRC`

1: Entrada: $f(.), N(.), A(.), |V|, f_{\min}, |T|, BTmax, s, IterAtivRC, IteracoesRC$
2: $s^* \leftarrow s$;
3: $Iter \leftarrow 0$;
4: $MelhorIter \leftarrow 0$;
5: $T \leftarrow \oslash$;
6: $ConjuntoElite \leftarrow \oslash$;
7: $IterSemMelhora \leftarrow 0$;
8: Inicialize a função de aspiração A;
9: **while** Critério de parada não satisfeito **do**
10: $Iter \leftarrow Iter + 1$;
11: Seja $s' \leftarrow s \oplus m$ tal que o movimento m não seja tabu ($m \notin T$) **ou** $f(s') < A(f(s))$;
12: Atualize a lista tabu;
13: $s \leftarrow s'$;
14: **if** s deve pertencer ao conjunto de soluções elite **then**
15: Insira s no $ConjuntoElite$;
16: **end if**
17: **if** ($IterSemMelhora$ **mod** $IterAtivRC < IteracoesRC$) **e** ($Iter \geq IterAtivRC$) **then**
18: Escolha aleatoriamente $g \in ConjuntoElite$ com probabilidade uniforme;
19: Atribua a g' a melhor solução obtida aplicando reconexão por caminhos ao par (s,g);
20: $s \leftarrow g'$;
21: **end if**
22: **if** g' deve pertencer ao conjunto de soluções elite **then**
23: Insira g' no $ConjuntoElite$;
24: **end if**
25: **if** $f(s) < f(s^*)$ **then**
26: $s^* \leftarrow s$;
27: $MelhorIter \leftarrow Iter$;
28: $IterSemMelhora \leftarrow 0$;
29: **else**
30: $IterSemMelhora \leftarrow IterSemMelhora + 1$;
31: **end if**
32: Atualize a função de aspiração A;
33: **end while**
34: $s \leftarrow s^*$;
35: Retorne s;

Figura 30: Algoritmo Busca Tabu com Reconexão por Caminhos

5.2 Princípio da Otimalidade Próxima

Pelo Princípio da Otimalidade Próxima (*Proximate Optimality Principle*), **boas soluções em um nível estão próximas de boas soluções em um nível adjacente** [21].

Uma implementação deste princípio no contexto de uma heurística construtiva consiste em refinar a solução parcial periodicamente durante o processo de construção.

Por exemplo, considerando o Problema do Caixeiro Viajante, imagine que uma solução parcial envolvendo 6 cidades seja $s = (1\ \ 3\ \ 4\ \ 5)$. Então, antes de prosseguir com a escolha das duas cidades ainda não visitadas, a saber, as cidades 2 e 6, esta solução parcial deve ser refinada por um método qualquer, por exemplo, o método da Descida. Assim, a construção só prossegue após o refinamento, que, no caso, pode alterar a ordem de visita das cidades previamente visitadas. Inserida a próxima cidade, é feito outro refinamento e assim sucessivamente. Como o refinamento após cada inserção pode ser custoso computacionalmente, também é comum aplicá-lo a cada *IterPOP* inserções, sendo *IterPOP* um parâmetro para o procedimento. Outra estratégia comumente adotada consiste em prosseguir com o refinamento até que um percentual máximo de inserções seja atingido. Por exemplo, no caso do PCV, far-se-iam as inserções seguidas de refinamento até que 75% das cidades fossem inseridas. Após esse percentual, a construção seguiria sem a fase de refinamento.

5.3 Relaxação Adaptativa

A Relaxação Adaptativa (também conhecida como Oscilação Estratégica) está intimamente ligada às origens da Busca Tabu [21]. Esta técnica provê um meio de se alcançar uma combinação entre intensificação e diversificação. Ela consiste em orientar movimentos em relação a um nível crítico, que pode ser identificado por um estágio de construção ou um intervalo escolhido de valores para uma função. Tal nível crítico (ou fronteira de oscilação) freqüentemente representa um ponto onde o método normalmente seria interrompido. Em vez de parar quando esta fronteira é alcançada, as regras para selecionar movimentos são modificadas para permitir que a região definida pelo nível crítico seja atravessada. Esta abordagem então continua até uma dada profundidade além da fronteira de oscilação. Encontrado este limite, retorna-se na direção oposta, e assim por diante.

A Figura 31 ilustra o comportamento deste procedimento.

O processo de repetidamente abordar e atravessar o nível crítico a partir de diferentes direções cria um comportamento oscilatório, o qual dá o nome ao método. O controle sobre esse comportamento é estabelecido por meio da geração de avaliações e regras modificadas para os movimentos, dependendo da região navegada e da direção da busca. A possibilidade de percorrer uma trajetória já visitada é evitada por mecanismos tabu padrões, tais como aqueles estabelecidos pelas funções da memória de curto prazo.

Em [55] apresenta-se um mecanismo de relaxação adaptativa onde os pesos,

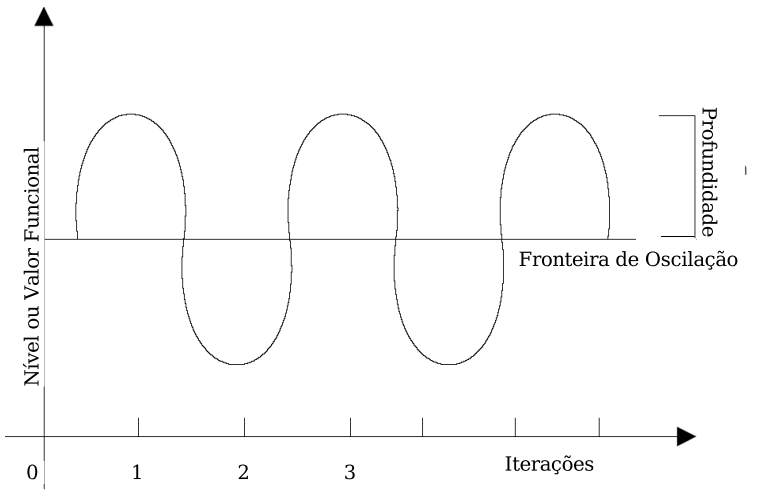


Figura 31: Oscilação Estratégica

para cada fonte de inviabilidade da função de avaliação, são ajustados dinamicamente como proposto por [14]. Para cada fonte de inviabilidade i o peso w_i é multiplicado por um fator α_i que varia de acordo com o seguinte esquema:

1. No início da busca $\alpha_i \leftarrow 1$.
2. A cada k movimentos:
   - se todas as k soluções visitadas são factíveis em relação à inviabilidade i então $\alpha_i \leftarrow \alpha_i / \gamma$;
   - se todas as k soluções visitadas são infactíveis em relação à inviabilidade i então $\alpha_i \leftarrow \alpha_i \times \gamma$;
   - se algumas soluções são factíveis e algumas outras são infactíveis então α_i permanece inalterado.

O valor do parâmetro γ neste esquema é randomicamente selecionado, a cada vez, no intervalo [1,8; 2,2]. Em [14], γ é deterministicamente fixado no valor 2. Em [55] optou-se por randomizar tal valor para evitar que escolhas determinísticas pudessem guiar a busca.

Cada valor de α_i é limitado por duas constantes $\alpha_{i,\,\min}$ e $\alpha_{i,\,\max}$. Isso implica que se α_i assumir um valor superior a $\alpha_{i,\,\max}$, então ele recebe o valor $\alpha_{i,\,\max}$. De maneira semelhante, se α_i assumir um valor inferior a $\alpha_{i,\,\min}$, ele recebe o valor $\alpha_{i,\,\min}$. A limitação do valor de α_i tem como objetivo não perder o referencial de avaliação das soluções, fato que ocorreria após uma longa seqüência de soluções infactíveis ou factíveis, em decorrência de valores muito altos ou muito baixos, respectivamente, para esse parâmetro.

A Figura 32 apresenta o pseudocódigo de um algoritmo Busca Tabu que faz uso de um procedimento de Relaxação Adaptativa.

O método de Busca Tabu implementado interrompe a busca do melhor vizinho da solução corrente em duas situações: (1) quando o vizinho gerado é melhor que a melhor solução gerada até então (linha 11) e (2) quando um vizinho não tabu é melhor que a solução corrente (linha 15). Nesta figura, *IteracaoAtualizacao*() indica quando os pesos dinâmicos devem ser atualizados.

Algoritmo `BTRA`

1: $s \leftarrow$ *GerarSolucaoInicial*();
2: $s^* \leftarrow s$;
3: *ListaTabu* $\leftarrow \emptyset$;
4: **repeat**
5: Selecione um subconjunto V da vizinhança corrente N, com $V \subseteq N$;
6: *melhorMovimento* $\leftarrow$ *movimentoRandomico*(V);
7: *melhorCustoDinamico* $\leftarrow \infty$;
8: **for all** Movimento $m \in V$ **do**
9: **if** ($f(s \oplus m) < f(s^*)$) **then**
10: *melhorMovimento* $\leftarrow m$;
11: **interromper**;
12: **else**
13: **if** ($m \notin$ *ListaTabu*) **then**
14: **if** ($f(s \oplus m) < f(s)$) **then**
15: *melhorMovimento* $\leftarrow m$;
16: **interromper**;
17: **else**
18: **if** (*CustoPorPesosDinamicos*($s \oplus m$) $<$ *melhorCustoDinamico*) **then**
19: *melhorMovimento* $\leftarrow m$;
20: *melhorCustoDinamico* $\leftarrow$ *CustoPorPesosDinamicos*($s \oplus m$);
21: **end if**
22: **end if**
23: **end if**
24: **end if**
25: **end for**
26: $s \leftarrow s \oplus$ *melhorMovimento*;
27: **if** $f(s) < f(s^*)$ **then**
28: $s^* \leftarrow s$;
29: **end if**
30: *AtualizarListaTabu*();
31: **se** *IteracaoAtualizacao*() **então** *AtualizarPesosDinamicos*();
32: **until** Critério de parada seja alcançado;
33: $s \leftarrow s^*$;
34: Retorne s;

Figura 32: Algoritmo de Busca Tabu com Relaxação Adaptativa

Referências

[1] S. Barnard. Stereo matching by hierarchical microcanonical annealing. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, Milan, Italy, 1987.

[2] M.P. Bastos and C.C. Ribeiro. Reactive Tabu Search with Path Relinking for the Steiner Problem in Graphs. In *Proceedings of the Third Metaheuristics International Conference*, pages 31–36, Angra dos Reis, Brazil, 1999.

[3] R. Battiti. Reactive Search: Toward Self-Tuning Heuristics. In V.J. Rayward-Smith, I.H. Osman, C.R. Reeves, and G.D. Smith, editors, *Modern Heuristic Search Methods*, chapter 4, pages 61–83. John Wiley & Sons, New York, 1996.

[4] R. Battiti and G. Tecchiolli. The Reactive Tabu Search. *ORSA Journal of Computing*, 6:126–140, 1994.

[5] M. Creutz. Microcanonical Monte-Carlo Simulation. *Physical Review Letters*, 50:1411–1414, 1983.

[6] F. Dammeyer and S. Voß. Dynamic tabu list management using the reverse elimination method. In P.L. Hammer, editor, *Tabu Search*, volume 41 of *Annals of Operations Research*, pages 31–46. Baltzer Science Publishers, Amsterdan, 1993.

[7] L. Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence - IJCAI-85*, pages 162–164, 1985.

[8] D. de Werra. Tabu Search Techniques: A Tutorial and an Application to Neural Networks. *OR Spektrum*, 11:131–141, 1989.

[9] M. Dorigo. *Optimization, Learning and Natural Algorithms*. Phd thesis, Dipartimento di Elettronica, Politecnico di Milano, Italy, 1992. 140 pp.

[10] M. Dorigo, V. Maniezzo, and A. Colorni. The Ant System: Optimization by a Colony of Cooperating Agents. *IEEE Transactions on Systems, Man, and Cybernetics - Part B*, 26:29–41, 1996.

[11] M. Dorigo and T. Stützle. The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and Advances. In F. Glover and G. A. Kochenberger, editors, *Handbook of Metaheuristics*, chapter 9, pages 251–285. Kluwer Academic Publishers, 2003.

[12] K.A. Dowsland. Simulated Annealing. In C.R. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, Advanced Topics in Computer Science Series, chapter 2, pages 20–69. Blackwell Scientific Publications, London, 1993.

[13] T.A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[14] M. Gendreau, A. Hertz, and G. Laporte. A tabu search heuristic for the vehicle routing problem. *Management Science*, 40:1276–1290, 1994.

[15] F. Glover. Future paths for Integer Programming and links to Artificial Intelligence. *Computers and Operations Research*, 5:553–549, 1986.

[16] F. Glover. Tabu Search: Part I. *ORSA Journal of Computing*, 1:190–206, 1989.

[17] F. Glover. Tabu Search: Part II. *ORSA Journal of Computing*, 2:4–32, 1990.

[18] F. Glover. Tabu Search and adaptive memory programming - advances, applications and challenges. In R. Barr, R. Helgason, and J. Kennington, editors, *Interfaces in Computer Sciences and Operations Research*, pages 1–75. Kluwer Academic Publishers, 1996.

[19] F. Glover and S. Hanafi. Tabu Search and Finite Convergence. *Discrete Applied Mathematics*, 119:3–36, 2002.

[20] F. Glover and M. Laguna. Tabu Search. In C.R. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, Advanced Topics in Computer Science Series, chapter 3, pages 70–150. Blackwell Scientific Publications, London, 1993.

[21] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Boston, 1997.

[22] F. Glover, M. Laguna, and R. Marti. Scatter Search and Path Relinking: Advances and Applications. In F. Glover and G. A. Kochenberger, editors, *Handbook of Metaheuristics*, chapter 1, pages 1–35. Kluwer Academic Publishers, 2003.

[23] F. Glover, E. Taillard, and D. de Werra. A user's guide to tabu search. In P.L. Hammer, editor, *Tabu Search*, volume 41 of *Annals of Operations Research*, pages 3–28. Baltzer Science Publishers, Amsterdan, 1993.

[24] D. Goldberg and R. Lingle. Alleles, Loci and the Traveling Salesman Problem. In *Proceedings of the 1st International Conference on Genetic Algorithms and Their Applications*, pages 154–159, Los Angeles, USA, 1985.

[25] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Berkeley, 1989.

[26] S. Hanafi. On the Convergence of Tabu Search. *Journal of Heuristics*, 7:47–58, 2001.

[27] P. Hansen. The steepest ascent mildest descent heuristic for combinatorial programming. In *Congress on Numerical Methods in Combinatorial Optimization*, Capri, Italy, 1986.

[28] P. Hansen and N. Mladenović. Variable Neighborhood Search: Methods and Recent Applications. In *Proceedings of the Third Metaheuristics International Conference*, pages 275–280, Angra dos Reis, Brazil, 1999.

[29] P. Hansen and N. Mladenović. Variable Neighborhood Search: Principles and Applications. *European Journal of Operational Research*, 130:449–467, 2001.

[30] Pierre Hansen, Nenad Mladenović, Raca Todosijević, and Saïd Hanafi. Variable neighborhood search: basics and variants. *EURO Journal on Computational Optimization*, 5(3):423–454, 08 2017.

[31] Pierre Hansen and Nenad Mladenović. A tutorial on variable neighborhood search. Technical Report G–2003–46, LES CAHIERS DU GERAD, HEC MONTREAL AND GERAD, 2003. Disponível em https://www.cs.uleth.ca/~benkoczi/OR/read/vns-tutorial.pdf e em https://www.gerad.ca/en/papers/G-2003-46.pdf?locale=en.

[32] A. Hertz. Finding a feasible course schedule using tabu search. *Discrete Applied Mathematics*, 35:255–270, 1992.

[33] A. Hertz and D. de Werra. The tabu search metaheuristic: how we used it. *Annals of Mathematics and Artificial Intelligence*, 1:111–121, 1990.

[34] A. Hertz and M. Widmer. Guidelines for the use of meta-heuristics in combinatorial optimization. *European Journal of Operational Research Society*, 151:247–252, 2003.

[35] S. Kirkpatrick, D.C. Gellat, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[36] A. Linhares and J.R.A. Torreão. Microcanonical Optimization Applied to the Traveling Salesman Problem. *International Journal of Modern Physics C*, 9:133–146, 1998.

[37] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[38] H.R. Lourenço, O. Martin, and T. Stützle. Iterated Local Search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 321–353. Kluwer Academic Publishers, Norwell, MA, 2003.

[39] Helena Ramalhinho Lourenço, Olivier C. Martin, and Thomas Stützle. Iterated local search: Framework and applications. In M. Gendreau and J. Y. Potvin, editors, *Handbook of metaheuristics*, volume 272 of *International Series in Operations Research & Management Science*, pages 129–168. Springer, Cham, Switzerland, 3 edition, 2019.

[40] Rafael Martí, Mauricio G. C. Resende, and Celso C. Ribeiro. Multi-start methods for combinatorial optimization. *European Journal of Operational Research*, 226(1):1–8, 2013.

[41] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, Boston, USA, 1996.

[42] N. Mladenović and P. Hansen. Variable Neighborhood Search. *Computers and Operations Research*, 24:1097–1100, 1997.

[43] J. V. Moccellin, M. O. Santos, and M. S. Nagano. Um método heurístico Busca Tabu-Simulated Annealing para Flowshops Permutacionais. In *Anais do XXXIII Simpósio Brasileiro de Pesquisa Operacional*, pages 1088–1095, 2001.

[44] I. M. Oliver, D. J. Smith, and R. C. J. Holland. Study of permutation crossover operators on the traveling salesman problem. In *Proceedings of the second International Conference on Genetic Algorithms*, pages 224––230, Cambridge, MA, USA, 1987.

[45] M. Prais and C.C. Ribeiro. Parameter Variation in GRASP Implementations. In *Proceedings of the Third Metaheuristics International Conference*, pages 375–380, Angra dos Reis, Brazil, 1999.

[46] M. Prais and C.C. Ribeiro. Reactive GRASP: An application to a matrix decomposition problem in TDMA traffic assignment. *INFORMS Journal on Computing*, 12:164–176, 2000.

[47] M. Prais and C.C. Ribeiro. Variação de Parâmetros em Procedimentos GRASP. *Investigación Operativa*, 9:1–20, 2000.

[48] C.R. Reeves. Genetic Algorithms. In C.R. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, Advanced Topics in Computer Science Series, chapter 4, pages 151–196. Blackwell Scientific Publications, 1993.

[49] Maurício G. C. Resende and Celso C. Ribeiro. *Optimization by GRASP: Greedy Randomized Adaptive Search Procedures*. Springer, Cham, Switzerland, 2016.

[50] Maurício G. C. Resende and Celso C. Ribeiro. Greedy Randomized Adaptive Search Procedures: Advances and Extensions. In M. Gendreau and J. Y. Potvin, editors, *Handbook of metaheuristics*, volume 272 of *International Series in Operations Research & Management Science*, pages 169–220. Springer, Cham, Switzerland, 3 edition, 2019.

[51] C. C. Ribeiro, E. Uchoa, and R. F. Werneck. A hybrid GRASP with perturbations for the Steiner problem in graphs. *INFORMS Journal on Computing*, 14:228–246, 2002.

[52] C.C. Ribeiro. Metaheuristics and Applications. In *Advanced School on Artificial Intelligence*, Estoril, Portugal, 1996.

[53] D. P. Ronconi and V. A. Armentano. Busca Tabu para o problema de minimização do tempo total de atraso no problema de flowshop. *Pesquisa Operacional para o Desenvolvimento*, 1:5–62, 2009.

[54] I. C. M. Rosseti. Estratégias sequenciais e paralelas de GRASP com reconexão por caminhos para o problema de síntese de redes a 2-caminhos. Tese de doutorado, Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2003.

[55] A. Schaefer. Tabu search techniques for large high-school timetabling problems. In *Proceedings of the 30th National Conference on Artificial Intelligence*, pages 363–368, 1996.

[56] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 440–446, 1992.

[57] M. Souza. Busca Tabu. In A. Gaspar-Cunha, R. Takahashi, and C. H. Antunes, editors, *Manual de Computação Evolutiva e Meta-heurística*, volume 2, pages 177–202. Editora UFMG, Belo Horizonte, MG, 1 edition, 2013.

[58] J.R.A. Torreão. Inteligência Computacional. Notas de aula, Universidade Federal Fluminense, Niterói, 2004.

[59] J.R.A. Torreão and E. Roe. Microcanonical Optimization Applied to Visual Processing. *Physics Letters A*, 122:377–382, 1980.

[60] C. Voudouris. Guided Local Search for Combinatorial Optimisation Problems. PhD Thesis, Department of Computer Science, University of Essex, 1997.

[61] L. D. Whitley, T. Starkweather, and D'Ann Fuquay. Scheduling Problems and Traveling Salesmen: The Genetic Edge Recombination Operator. In *Proceedings of the Third International Conference on Genetic Algorithms (ICGA-89)*, pages 133–140, George Mason University, Fairfax, Virginia, USA, 1989.