RAFAEL LIMA DE CARVALHO
TIAGO DA SILVA ALMEIDA
MARCELO LISBOA ROCHA

Introdução às

Metaheurísticas

EDUFT
UNIVERSIDADE FEDERAL DO TOCANTINS

RAFAEL LIMA DE CARVALHO
TIAGO DA SILVA ALMEIDA
MARCELO LISBOA ROCHA

Introdução às Metaheurísticas

EDUFT
UNIVERSIDADE FEDERAL DO TOCANTINS

Palmas- TO
2020

Universidade Federal do Tocantins

Diagramação e capa: Gráfica Movimento
Arte de capa: Nonono

ABEC BRASIL
Associação Brasileira de Editores Científicos
http://www.abecbrasil.org.br

Dados Internacionais de Catalogação na Publicação – CIP

C331i

Carvalho, Rafael Lima de .
Introdução às Metaheurísticas / Rafael Lima de Carvalho; Tiago da Silva Almeida; Marcelo Lisboa Rocha – Palmas, TO: EDUFT, 2020.
75 p. il. grafs. ; 21 x 29,7 cm.

ISBN 978-65-89119-11-1
Inclui referências ao final.

1. Metaheurística. 2. Algorítmicos genéticos. 3. Colônia de formigas. 4. GRASP. 5. Função Bidimensional. I. Tiago da Silva Almeida. II. Marcelo Lisboa Rocha. III. Título.

CDD – 519

SUMÁRIO

PREFÁCIO

O mundo tal qual conhecemos agora é um reflexo do que avançamos nas diversas áreas do conhecimento tais como filosofia, cultura, ciência e tecnologia. Fato é que na vida do homem sempre existiram problemas a serem resolvidos. Dentro da computação e engenharias não poderia ser diferente. Existem problemas anunciados na década de 1950 que ainda desafiam os matemáticos, engenheiros e cientistas da computação. Ao passo que alguns problemas são resolvidos, outras perguntas a serem desvendadas emergem dessas soluções já encontradas.

Na otimização combinatória são estudados problemas cujo espaço de busca pode ser tão extenso quanto explorar o próprio universo! Naturalmente, em problemas reais de engenharia e computação, o tempo é um recurso finito e, portanto, soluções devem ser encontradas em um tempo aceitável. Neste sentido, na literatura de otimização encontra-se a área de Metaheurísticas. Metaheurísticas são procedimentos que oferecem um meio de guiar o processo de otimização de forma mais inteligente. Desta maneira, propiciam a produção de soluções com qualidade mensurável e em um tempo aceitável de resposta.

Este livro trata de uma introdução a algumas metaheurísticas, a saber: *Simulated Annealing*, Busca Tabu, Algoritmos Genéticos, Colônia de Formigas e GRASP. Focamos em tratar basicamente três problemas: uma função bidimensional multimodal, o Problema do Caixeiro Viajante e Problema de Satisfatibilidade Booleana (SAT). Este livro foi estruturado pensando-se como público-alvo os alunos de graduação de Engenharia, Computação e Matemática Aplicada. Neste sentido, apresentamos os conteúdos utilizando de linguagem acessível, porém incluindo as definições e aspectos técnicos necessários. Portanto, desejamos uma boa leitura e bom aprendizado.

INTRODUÇÃO

A capacidade de resolver problemas é um dos talentos mais desejados na era da informação. Ser um solucionador de problemas, por vezes, não é uma tarefa fácil. Além de conhecimento, é preciso a habilidade de observar todos os detalhes necessários do problema enfrentado. Atualmente, uma onda de informações chega muito rapidamente na vida dos cidadãos da sociedade do conhecimento - a vida hoje é muito acelerada. As opções de fontes de informação e conhecimento, tais como cursos on-line são inúmeras e tornam difícil inclusive a decisão por qual caminho tomar, uma vez que, por mais acelerada que a vida seja nos dias atuais, o tempo de vida ainda é um recurso finito.

Naturalmente, a cada curso feito, um aluno recebe um documento (certificado) declarando que o mesmo realizou o curso proposto. Entretanto, nesta geração do conhecimento, os certificados passam a ter valia apenas nos casos em que o seu portador consiga utilizar o conhecimento, ora certificado, para gerar soluções. Convidamos o leitor a se atentar na seguinte informação: geralmente, na disciplina de sistemas digitais ensina-se sobre o código de *Hamming* (MOON, 2005). Uma das maneiras de checar a paridade de uma sequência de bits consiste em adicionar um bit à quantidade de informações para indicar se a soma dos bits contidos no pacote de dados é par ou ímpar. Portanto, se durante a transmissão de um sinal digital, este vier a sofrer mudanças que impliquem na mudança da paridade, logo o bit de paridade poderá ser útil para detectar a existência de erros na informação transmitida.

Pode ser que, no tempo que se estudou este tipo de conteúdo, a informação do código de *Hamming* lhe foi útil para resolver alguns exercícios. Sobretudo, solicito novamente sua atenção para o seguinte problema trazido por um aluno: existia um conjunto de pessoas que passaria por uma prova em grupo. A prova consistia em colocá-los enfileirados em uma rampa de tal forma que o último da fila conseguisse ver os demais, o penúltimo conseguisse ver os demais , e assim por diante. Sobre a cabeça de cada indivíduo foi colocado um boné com uma das duas cores: branca ou preta. O indivíduo não pode ver a cor do seu próprio boné. O grupo terá sucesso na prova caso, pelo menos indivíduos acertem a cor dos seus respectivos bonés. Antes de serem colocados em fila e terem os bonés colocados em suas cabeças, eles podem se reunir durante 15 minutos para montarem uma estratégia. A Figura 1 exemplifica o problema anunciado. Qual estratégia você sugeriria? Pause sua leitura agora e tente sugerir uma estratégia.

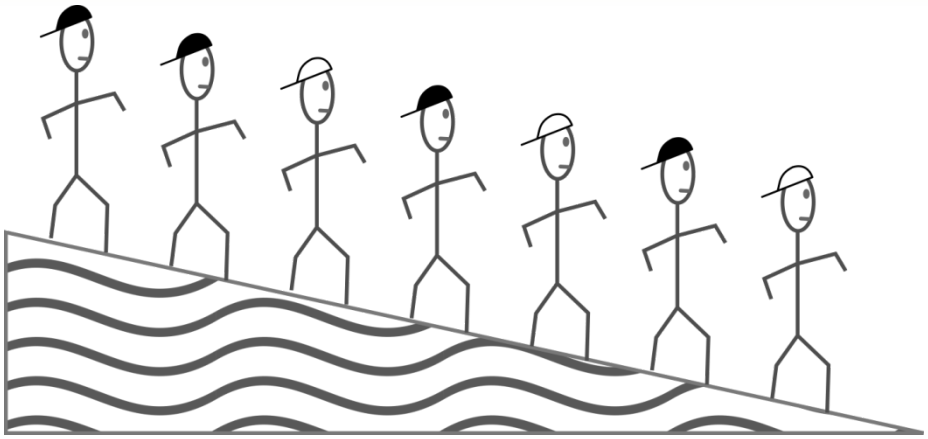

Figura 1 - Exemplo do problema dos bonés.

Acredito que tenha sido natural escolher a estratégia de paridade. Veja, como o último indivíduo da fila pode ver as cores dos bonés dos n-1 restantes, então ele pode dizer em voz alta a paridade dos n-1 e, portanto, o grupo pode escolher uma codificação. Por exemplo, a cor branca poderia indicar um número par de bonés na cor branca, e a cor preta poderia indicar um número ímpar de bonés brancos. Neste caso, o penúltimo faria o cálculo da paridade e com base na cor dita pelo último indivíduo, acertaria com certeza a cor do seu chapéu. O antepenúltimo descontaria a cor falada pelo penúltimo e faria o cálculo da paridade acertando a cor do seu boné e assim por diante. Portanto, o grupo garantiria pelo menos n-1 acertos (visto que o último elemento ainda poderia ter a sorte da cor da paridade ser a cor de seu próprio boné). A questão que pode ser levantada é: e se a informação da paridade não fosse dada antes de anunciar o problema, será que esta teria sido uma das formas relembradas por você, caro leitor?

Este livro tem o objetivo de apresentar, de forma introdutória, os fundamentos das metaheurísticas. Espera-se que o texto ora apresentado possa ser usado em disciplinas de introdução às metaheurísticas em cursos de graduação em Ciência da Computação, Sistemas de Informação, Engenharia da Computação, Engenharia de Produção ou Engenharia Elétrica (SANTIAGO et al., 2020). Apesar de introduzirmos como resolver um problema simples de programação não-linear irrestrita (a única restrição é o limite do domínio), mantemos o foco deste livro em dois principais problemas combinatórios: o problema SAT, que consiste em encontrar uma atribuição de valores booleanos tais que levem uma dada função booleana para o valor VERDADEIRO; além do Problema do Caixeiro Viajante (usaremos a sigla TSP, do inglês, *Travelling Salesman Problem*).

Para se ter uma ideia da classe de problemas que SAT e TSP representam, vamos utilizar um exemplo, a princípio pequeno, de uma instância do SAT. Imagine que tenhamos uma função booleana na forma normal conjuntiva:

$$f(\boldsymbol{x}) = (x_{12} \vee x_{31} \vee \overline{x_1}) \wedge \ldots \wedge (x_{99} \vee \overline{x_{100}})$$

tal que **x** é uma cadeia de valores VERDADEIROS ou FALSO de acordo com sua posição (assumindo que a posição inicie de 1) e que tenha tamanho igual a 100, ou seja, há 100 variáveis booleanas esperando seus valores serem atribuídos a VERDADEIRO ou FALSO. Um fato interessante a ser notado neste problema é que, uma vez que se tenha qualquer atribuição x, é fácil

e computacionalmente rápido verificar se a função retorna verdadeiro ou falso. Mas quando, o problema SAT pede para encontrar uma atribuição de 100 valores VERDADEIRO ou FALSO, tal que torne a função verdadeira, a facilidade e rapidez computacional já não são fatores presentes.

Tome-se como base que os valores booleanos podem ser abstraídos para valores de bit. Ou seja, *bit* =1 indica o valor VERDADEIRO, bit=0 indica o valor FALSO. Não é difícil ver que, se fosse possível enumerar todas as soluções, estas partiriam de um número binário de 100 posições, todas iguais a zero (FALSO) até o último número, com todas as entradas iguais a um (VERDADEIRO), o qual seria igual a 2^{100}-1. Em Michalewicz e Fogel (2013), os autores fazem uma simulação interessante colocada da seguinte maneira: se tivéssemos um computador capaz de testar 1000 cadeias por segundo e teletransportássemos este computador para iniciar esta computação no início do próprio tempo, ou seja, aproximadamente 15 bilhões de anos atrás, no ponto do Big Bang, este algoritmo teria examinado pouco mais de 1% de todas as possibilidades até o momento presente!

Adicionalmente, o problema do caixeiro viajante também é considerado neste trabalho. Informalmente, este problema pode ser anunciado da seguinte maneira: imagine que um caixeiro viajante, dado um conjunto de cidades, precise visitar cada uma delas, partindo de uma cidade, de forma que cada cidade seja visitada apenas uma vez, ao final ele retorne para a cidade de onde iniciou e que faça este tour com custo mínimo (por simplicidade, vamos considerar que todos os custos envolvidos no trajeto sejam abstraídos em um único número associado à ligação de uma cidade *i* a uma cidade *j*). É interessante observar que, ao utilizarmos um grafo simétrico (simétrico aqui indica que o custo de viajar da cidade *i* para *j* é o mesmo de *j* para *i*) uma representação natural de uma possível solução consiste em uma lista contendo uma ordem de visitação das cidades. A Figura 2 mostra um modelo de grafo simétrico que pode ser usado como uma abstração de uma instância para o TSP.

Assim, quando se seleciona um tour, por exemplo (2 – 6 – 3 - ...1 – 2, ilustrado na Figura 2) é fácil calcular o custo total do tour (apenas soma-se o custo associado a cada aresta). Na Figura 2, o tour selecionado tem um custo de 68,5. O problema surge quando se deseja saber o tour de custo mínimo. Imagine que para que se haja esta garantia de tour mínimo, seja necessário buscar exaustivamente por cada solução possível para este problema. Desta forma, se considerarmos todo o espaço de busca deste problema, veremos facilmente que para um problema com *n* cidades, o espaço de busca consiste em consultar *n*! possibilidades. Isso se deve ao fato de que há *n*! maneiras distintas de permutar *n* números. Ainda que se removam os tours repetidos (veja que um tour de *n* cidades pode ser rotacionado de maneira a gerar a mesma solução *n* vezes), isto daria ainda um número igual a (*n*-1)!/2, o que continua um grande número. Para se ter ideia de quanto isso é grande, uma instância de 20 cidades tem 60.822.550.204.416.000 possíveis soluções (para ilustrar, uma instância do SAT com 20 variáveis teria 1.048.576 soluções□). Maiores detalhes sobre os problemas TSP e SAT podem ser vistos em Goldbarg e Luna (2000).

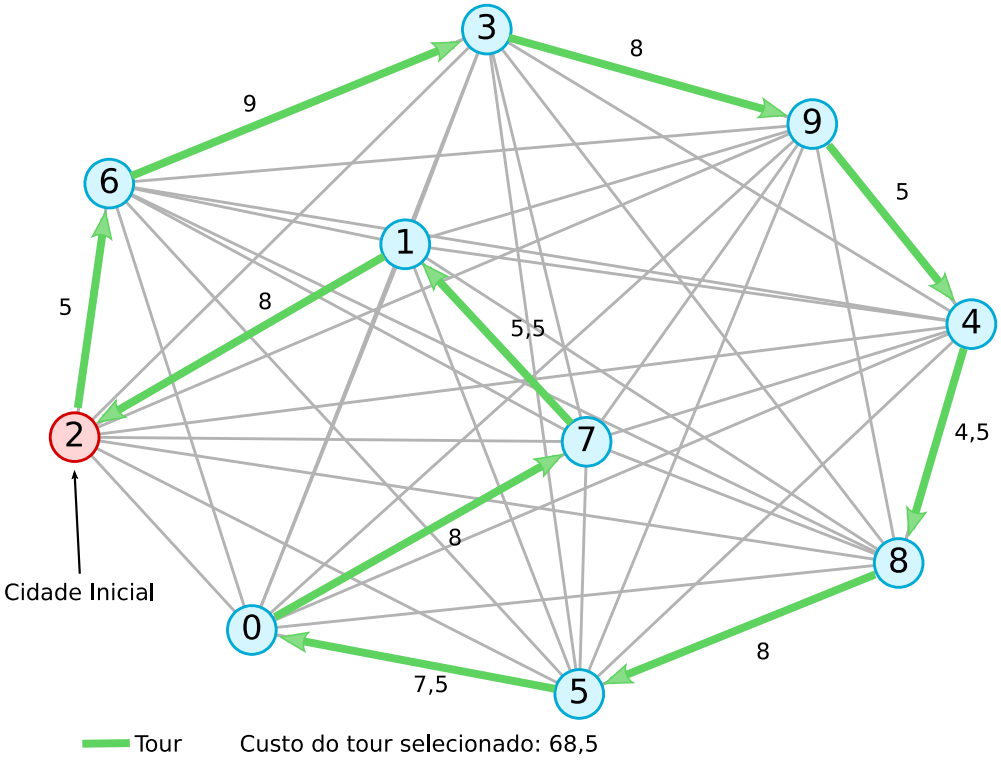

Figura 2 - Uma amostra de uma instância do TSP simétrico.

Neste sentido, este livro se propõe em apresentar as metaheurísticas à luz dos problemas mencionados anteriormente. Desta forma, o Capítulo 2. Conceitos Básicos reúne conteúdos que abordam sobre objetivo, funções de avaliação, definições de vizinhança e ótimos locais e globais. O Capítulo 3. Heurísticas apresenta um conjunto de heurísticas específicas para os problemas SAT e TSP. No Capítulo 4. Recozimento Simulado é apresentada a metaheurística de recozimento simulado, que se utiliza de elementos da estatística mecânica como método de otimização. O Capítulo 5. Busca tabu continua apresentando os métodos capazes de escapar de ótimos locais com um método que possui lembrança de movimentos que se tornam proibidos em certos momentos da busca. Em seguida, o Capítulo 6. Algoritmos Genéticos apresenta uma abordagem que simula processos evolutivos como forma de otimização. Já o Capítulo 7. Colônia de Formigas apresenta uma metaheurística inspirada em fenômenos naturais, mais especificamente na observação do comportamento de uma população de formigas ao saírem de sua colônia para encontrar comida de forma otimizada, apresentando um comportamento coletivo inteligente. Finalmente, no Capítulo 8. GRASP é apresentada a metaheurística GRASP, comumente aplicada em problemas de otimização combinatória, sendo um método iterativo que tem duas fases distintas, sendo uma fase construtiva, gananciosa e adaptativa que cria uma solução do início e uma fase de busca local, que visa melhorar a qualidade dessa solução, onde, a cada iteração, a melhor solução produzida é armazenada.

REFERÊNCIAS

GOLDBARG, M. C.; LUNA, H. P. **Otimização combinatória e programação linear: modelos e algoritmos**. Editora Campus, 3ª Edição, 2000.

MICHALEWICZ, Zbigniew; FOGEL, David B. **How to solve it: modern heuristics**. Springer Science & Business Media, 2013.

MOON, T. K. **Error correction coding: Mathematical Methods and Algorithms**. John Wiley and Sons, 2005.

SANTIAGO, O. L. (Org.); CORONA, C. C. (Org.) ; SILVA NETO, A. J. (Org.) ; VERDEGAY, J. L. (Org.). **Computational Intelligence in Emerging Techonologies for Engeneering Applications**. 1. ed. Cham - Swtizerland: Springer, 2020.

2. CONCEITOS BÁSICOS

Na abordagem algorítmica de resolução de problemas há três aspectos a serem considerados:

1. **Representação**: codifica soluções candidatas e alternativas para manipulação;
2. **Objetivo**: descreve o propósito a ser alcançado; e
3. **Função de Avaliação**: retorna um valor específico que indica a qualidade de uma solução particular, dada a representação.

Portanto, nesta primeira parte dos conceitos básicos cobriremos os três aspectos acima elencados.

2.1. REPRESENTAÇÃO E OBJETIVO

Ao lidar com a otimização de funções com múltiplos máximos locais, tais como $f(x)=x\,sen(10\pi x)+2$,tal que $-1\leq x\leq 2$, cujo gráfico pode ser visto na Figura 3, é necessário estabelecer uma representação. Como sabemos, ainda que se escolha utilizar um tipo de dado decimal, este ainda será limitado pelo tamanho de bits máximo alocado a ele pelo sistema computacional. Em geral, estipula-se um valor de incremento que define o fator de quantização. Em algoritmos que se utilizam de estados discretos tais como as representações binárias (por exemplo os algoritmos genéticos e busca tabu), ao se trabalhar com a maximização de $f(x)$ acima, define-se um número de bits para ser usado na representação e assim pode-se calcular o erro. Para o domínio [-1,2], se escolhermos n=2 bits, teremos 2^2=4 partes iguais do domínio e, portanto, teríamos a seguinte correspondência: 00→ -1,01→ -0.25,10→0.5 e 11→1.25. O valor do erro máximo de quantização pode ser calculado como o valor total do domínio dividido pelo maior número binário possível dentro da quantidade de bits. No caso do domínio considerado acima, este valor seria $(2-(-1))/2^n$. Para n=10, a resolução seria de 0,002929688.

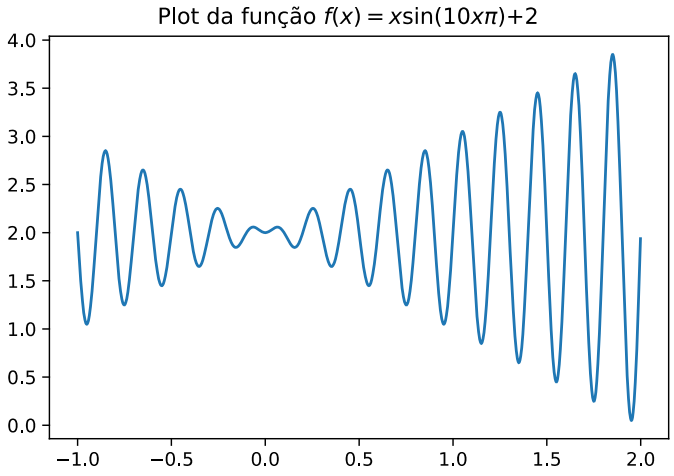

Figura 3 - Gráfico de uma função multimodal.

Perceba ainda que função a ser otimizada ainda necessita receber um valor real pertencente ao domínio especificado. Portanto, é necessário converter a cadeia binária de *n* bits de volta ao número. Conforme descrito por Yu e Gen (2010), este procedimento pode ser alcançado pela seguinte equação:

$$x = x_{min} + (x_{max} - x_{min}) \times \frac{\sum_{i=0}^{n-1} a_i 2^i}{2^n}$$

onde é o valor inferior do domínio (no caso do exemplo dado, este valor é -1) e x_{max} é o valor superior do domínio (no caso do exemplo dado, este valor é 2). Suponha que seja aplicado ao problema envolvendo *f*(x) definido anteriormente, sendo a = "010011", com *n* = 6, então x correspondente seria igual a $x=-1+3\times(1\times2^0+1\times2^1+1\times2^4)/64 = -0{,}109375$.

No problema SAT, uma representação natural de uma solução seria uma cadeia com *n* variáveis booleanas, como se cada caractere desta cadeia representasse um valor para cada uma das variáveis da função lógica avaliada. Neste caso, o tamanho do espaço de busca nesta representação é de 2^n , pois existem 2^n cadeias binárias distintas. Neste caso, cada ponto no espaço de busca é uma solução factível.

Analisando o TSP de *n*-cidades, já consideramos uma possível representação: a permutação dos números naturais 1, ..., *n* onde cada número corresponde a uma cidade a ser visitada em sequência. Usando-se esta representação, o espaço de busca do TSP consiste de todas as possíveis permutações, e neste caso existem diferentes permutações. Entretanto, ao consideramos o TSP simétrico, onde o custo de viajar da cidade *i* para a cidade *j* é o mesmo independente da direção, então não importa se percorremos a lista de cidades da esquerda para a direita ou da direita para a esquerda (pois o tour é o mesmo). Isto significa que o tamanho do espaço de busca pode ser reduzido pela metade. Além disso, o circuito seria o mesmo independentemente de qual cidade fosse escolhida para iniciar o tour, ou seja, reduzimos o espaço de busca por um fator de *n* Assim, o tamanho real do espaço de busca pode ser reduzido de *n!* para (*n*-1)!/2. Sobretudo, ainda é um número enorme, mesmo para instâncias moderadas do TSP. Lembre-se que esta redução só é possível no caso do TSP simétrico.

A escolha da representação é um aspecto crucial na proposta de solução de um problema. Observe o exemplo a seguir, proposto por Michalewicz e Fogel (2013). Suponha que existam seis fósforos em uma mesa e a tarefa é construir quatro triângulos equiláteros onde o tamanho de cada lado é igual ao tamanho de um fósforo. É fácil construir dois triângulos usando cinco fósforos (veja a Figura 4), mas, a partir desta solução é difícil incrementá-la de forma a alcançar os quatro triângulos, especialmente quando resta apenas um fósforo. Aqui, perceba que a representação está dificultando encontrar a solução, pois observando a figura, você pode vê-la no espaço 2D (ou seja, os fósforos são colocados em um plano).

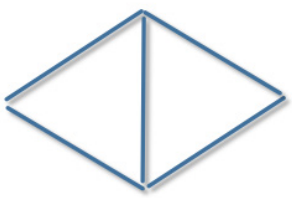

Figura 4 - Usando 5 (cinco) fósforos e formando dois triângulos, ficando de fora um fósforo.

Para encontrar uma solução adequada é necessário observar o espaço de soluções em três dimensões (Figura 5). Se você iniciar no espaço de busca incorreto, possivelmente nunca chegará à resposta correta.

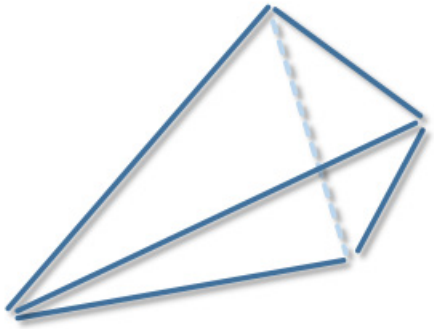

Figura 5 - Quatro triângulos equiláteros usando seis palitos de fósforo.

2.2. FUNÇÕES DE AVALIAÇÃO E O PROBLEMA DE BUSCA

Uma função de avaliação geralmente é um mapeamento dos pontos (possíveis soluções) do domínio do problema, sob a representação escolhida, para um conjunto de números (números reais, por exemplo). Aqui, cada elemento do espaço de possíveis soluções recebe um valor numérico que indica uma medida de qualidade. A função de avaliação, ou função objetivo, permite realizar comparações entre soluções e guiar o processo de busca (otimização).

Suponha que você queira descobrir uma boa solução para um TSP. O objetivo do TSP é minimizar a soma das distâncias entre cada uma das cidades ao longo da rota, ao mesmo tempo em que satisfaz as restrições do problema (visitar uma cidade apenas uma vez). Uma função de avaliação poderia mapear cada tour à soma das distâncias percorridas. Neste caso, poderíamos comparar rotas soluções e não somente dizer se uma é melhor que a outra, mas diz exatamente o quão melhor uma solução é da outra.

Em problemas do mundo real, geralmente é necessário fazer uma parte chamada modelagem matemática. A modelagem consiste em encontrar um modelo, o mais adequado possível, do problema correspondente. Em geral, um modelo matemático implica em representar matematicamente a função objetivo, além de especificar as restrições como funções lineares ou não-lineares. Por exemplo, para o TSP, consideramos que a função objetivo consiste em usar a distância de uma solução candidata como uma função de avaliação. Isto corresponde diretamente ao objetivo de minimizar o total da distância percorrida.

Mas nem sempre é possível extrair funções de avaliação diretamente do objetivo do problema abordado. Para o problema do SAT, onde o objetivo é tornar verdadeiro um grupo de expressões booleanas, cada solução aproximada retorna FALSO, e isto não nos dá qualquer informação útil sobre como melhorar uma solução candidata, ou como construir uma solução alternativa a partir de uma dada. Nestes casos, busca-se adotar alguma função de avaliação substituta que será apropriada para a tarefa dada, a representação que escolhermos conjuntamente com operadores que permitam saltar de uma solução para outra (com algum direcionamento de possível melhora).

Em geral, usaremos $\Gamma \subseteq S$ para indicar o espaço de busca factível dentro do espaço de busca total (). No caso do SAT e TSP, discutidos anteriormente, é natural que $\Gamma = S$, visto que todos os pontos do espaço de busca são soluções factíveis, e assim qualquer lista de cidades a serem visitadas é viável de ser percorrida (assumindo que o TSP opera em um grafo completo, ou seja, há caminhos de cada cidade para todas as outras cidades).

Considere um espaço de busca juntamente com sua parte factível $\Gamma \subseteq S$, o problema de busca consiste em encontrar $x \in \Gamma$ tal que

$$eval(x) \leq eval(y)$$

para todo $y \in \Gamma$, onde *eval*(*x*) é a função de avaliação ou qualidade de uma solução. Perceba que aqui nós estamos usando uma função de avaliação para a qual soluções que retornem valores menores são consideradas melhores (ou seja, um problema de minimização).

Repare que do jeito que foi colocada, a definição de busca não possui relação com o objetivo do problema. Ou seja, esta mesma definição poderia ser facilmente utilizada para descrever TSP ou SAT ou mesmo um problema de Programação Não-linear. A busca por si só não sabe qual o problema você está resolvendo! Tudo que ela sabe é a informação que você providencia na função de avaliação, a representação que você usa e a maneira que você amostra possíveis soluções. Se sua função de avaliação não corresponder ao objetivo do seu problema, você estará buscando pela resposta certa para o problema errado!

O ponto *x* que satisfaz a condição acima é chamado de ótimo *global*. Encontrar tal solução para um problema pode ser muito difícil. De fato, isto é verdadeiro para os exemplos de problemas trabalhados durante a aula (ou seja, SAT e TSP). No entanto, há situações em que encontrar a melhor solução é mais fácil quando a busca se concentra em um subconjunto relativamente pequeno. A melhor solução dentro deste subconjunto é chamada de ótimo local.

2.3. VIZINHANÇAS E ÓTIMO LOCAL

Se nos concentrarmos em uma região do espaço de busca que é de alguma forma "próxima" a algum ponto particular no espaço, podemos descrever esta região como *vizinhança* daquele ponto. Graficamente, considere um espaço de busca abstrato S junto com um único ponto $x \in S$. A intuição é que a vizinhança *N(x)* de *x* é um conjunto de todos os pontos do espaço de busca que estão de alguma maneira "próximos" ao ponto *x*.

Nós podemos definir a proximidade entre pontos no espaço de busca de muitas maneiras diferentes. Pode-se então, definir uma função de distância que opere no domínio das soluções e dada uma solução, o subconjunto de vizinhos são todos os pontos que estão a uma distância ϵ, por exemplo. Neste caso, chama-se ϵ-*dist(x)* o conjunto de vizinhos da solução x que estão a uma distância máxima de ϵ. No caso do problema SAT a distância entre duas cadeias binárias pode ser definida como a distância de *Hamming*, que mede o número de posições com valores distintos.

Podemos definir um mapeamento *m* no espaço de busca *S* de forma que:

$$m: S \rightarrow 2^S,$$

e este mapeamento define a vizinhança para qualquer ponto $x \in S$. Por exemplo, nós podemos definir um mapeamento do tipo *2-swap* para o TSP de tal forma que gere um novo conjunto de soluções potenciais a partir de qualquer solução potencial *x*. Cada solução que pode ser gerada pela troca de duas cidades em um tour escolhido pode ser dita que está na vizinhança daquele tour sob o operador *2-swap*. Em particular, uma solução *x* (MICHALEWICZ e FOGEL, 2013):

15 - 3 - 11- 19 - 17 - 2 -… - 6

possui $\frac{n(n-1)}{2}$ vizinhos. Estes incluem:

15 - 17 - 11- 19 - 3 - 2 -... – 6	Trocando a segunda com a quinta
2 - 3 - 11- 19 - 17 - 15 -... - 6	Trocando a primeira com a sexta
15 - 3 - 6- 19 - 17 - 2 -... - 11	Trocando terceira com a última

Para o problema SAT, um mapeamento possível seria o *-flip* (SZEIDER, 2009), onde *k* é um número inteiro positivo. No caso do exemplo considerando de *k=1*, então dada uma cadeia *x*, a vizinhança *1-flip* é dada por todas as cadeias tal que pelo menos 1 bit tem seu valor trocado. Neste caso, se *x* possuir tamanho 10, o tamanho da vizinhança será de 10 outras soluções.

Com a noção de vizinhança definida, pode-se agora discutir o conceito de ótimo local. Uma solução potencial $x \in \Gamma$ é um ótimo local em relação a uma vizinhança , se e somente se:

$$eval(x) \leq eval(y)$$

para todo $y \in N(x)$ (novamente assumindo um critério de minimização). Em geral é relativamente simples checar se uma dada solução é um ótimo local quando o tamanho da vizinhança é muito pequeno, ou quando sabemos alguma informação sobre a derivada da função de avaliação.

2.4. SUBIDA DE ENCOSTA

Métodos de subida de encosta são baseados na noção de melhoramento. Durante cada iteração da busca, um novo ponto é selecionado da vizinhança da solução atual. Se este novo ponto possuir um melhor valor à luz da função de avaliação, a solução atual é atualizada para o novo ponto encontrado. Caso contrário, uma outra solução dentro da vizinhança é selecionada e compara-se novamente com a solução atual. O método é finalizado quando não há mais o que melhorar dentro da vizinhança explorada ou quando esgotar o tempo destinado à busca.

Pelo exposto, percebe-se que os métodos de *subida de encosta* garantem apenas encontrar ótimos locais. A qualidade desta solução é diretamente relacionada ao ponto de partida para o qual se seleciona a vizinhança. Portanto, para se obter melhores resultados, em geral, é necessário executar os métodos de *subida de encosta* a partir de uma variedade de pontos iniciais. Caso um destes pontos inclua a solução global na vizinhança, a expectativa é que este método consiga criar um caminho até esta solução. Os pontos iniciais podem partir de forma aleatória ou a partir de espaços bem determinados do domínio (YU e GEN, 2010).

De acordo com Michalewicz e Fogel (2013), as versões de subida de encosta se diferem principalmente na maneira que uma nova solução é selecionada para comparação com a solução

atual. Uma versão de um simples algoritmo de *Subida de Encosta Iterativo* é ilustrado na Figura 6 (chamado subida íngreme). Inicialmente, todos os possíveis vizinhos da solução atual são considerados, e então o que retorna o melhor valor de $eval(v_n)$ é selecionado para competir com a cadeia atual v_c. Caso $eval(v_c)$ seja pior que $eval(v_n)$, então a nova cadeia v_n se torna a cadeia atual. Caso contrário, nenhuma melhora local é possível, ou seja o algoritmo alcançou um ótimo local (ou global). Neste caso, a próxima iteração do algoritmo é executada com uma nova cadeia selecionada aleatoriamente.

1: $t \leftarrow 0$
2: Inicialize a variável *best*
3: **repeat**
4: $local \leftarrow FALSE$
5: $\mathbf{v}_c \leftarrow$ solução aleatória
6: **repeat**
7: Selecione $\mathbf{v}_n$ de $N(\mathbf{v}_c)$ tal que $eval(\mathbf{v}_n)$ seja a melhor da vizinhança $N(\mathbf{v}_c)$
8: **if** $eval(\mathbf{v}_n)$ é melhor que $eval(\mathbf{v}_c)$ **then**
9: $\mathbf{v}_c \leftarrow \mathbf{v}_n$
10: **else**
11: $local \leftarrow FALSE$
12: **end if**
13: **until** local=TRUE
14: Atualize o valor de *best*, caso $\mathbf{v}_c$ seja melhor
15: $t \leftarrow t + 1$
16: **until** t = MAX

Figura 6 – Algoritmo *Subida de Encosta Iterativo* **adaptado de Michalewicz e Fogel (2013).**

De acordo com Engelbrecht (2007), as técnicas de busca devem realizar um balanceamento entre a quantidade de esforço para tarefas de intensificação (*exploiting*) e exploração (*exploration*). *Exploiting* consiste na intensificação das melhores soluções encontradas até o momento. *Exploring* consiste na exploração mais global do espaço de busca. Pelo exposto, técnicas de *subida de encosta* são mais voltadas para o processo de *exploiting,* enquanto que uma busca puramente aleatória explora o espaço de buscas de maneira mais ampla (assumindo que pontos são sorteados de maneira uniforme), porém não intensifica a busca em nenhuma região, ainda que promissora.

REFERÊNCIAS

MICHALEWICZ, Zbigniew; FOGEL, David B. **How to solve it: modern heuristics**. Springer Science & Business Media, 2013.

ENGELBRECHT, Andries P. **Computational intelligence: an introduction**. John Wiley & Sons, 2007.

SZEIDER, Stefan. The parameterized complexity of k-flip local search for SAT and MAX SAT. In: **International Conference on Theory and Applications of Satisfiability Testing**. Springer, Berlin, Heidelberg, 2009. p. 276-283.

YU, X.; GEN, M. **Introduction to Evolutionary Algorithms**. Springer London, 2010. 433 p.

3. HEURÍSTICAS

Heurísticas são métodos produtores de soluções baseadas em alguma ideia especializada em um problema. Conforme anunciado na introdução, estudaremos neste livro algumas metaheurísticas, ou seja, meta algoritmos que podem ser adaptados a praticamente qualquer problema de otimização. Sobretudo, neste capítulo serão apresentadas algumas heurísticas específicas para o problema SAT e o TSP, conforme já vínhamos estudando no decorrer do livro. As soluções heurísticas podem ser usadas na exploração da vizinhança ou para prover melhores pontos de partida do que apenas soluções aleatórias.

3.1. MÉTODOS HEURÍSTICOS PARA O TSP

Para efeitos de organização, separamos os métodos aqui reunidos em dois tipos: Métodos Construtivos e Métodos de Busca Local.

3.1.1. MÉTODOS CONSTRUTIVOS

Closest Neighbor Method

De acordo com Yu e Gen (2010), é possível construir uma solução para o TSP ao considerar a cidade mais próxima da cidade atual, através dos seguintes passos:

1. Comece de uma cidade aleatória como cidade atual;
2. Selecione a cidade mais próxima da cidade atual que não foi visitada. Marque a cidade recém selecionada como a cidade atual e a atual anterior como “visitada”;
3. Repita o passo 2 até que todas as cidades estejam visitadas.
4. Inclua a rota da última cidade selecionada para a primeira selecionada.

A aplicação desta heurística pode ser ilustrada na Figura 7. A cidade 1 foi escolhida como inicial. As setas indicam as escolhas para composição da solução inicial.

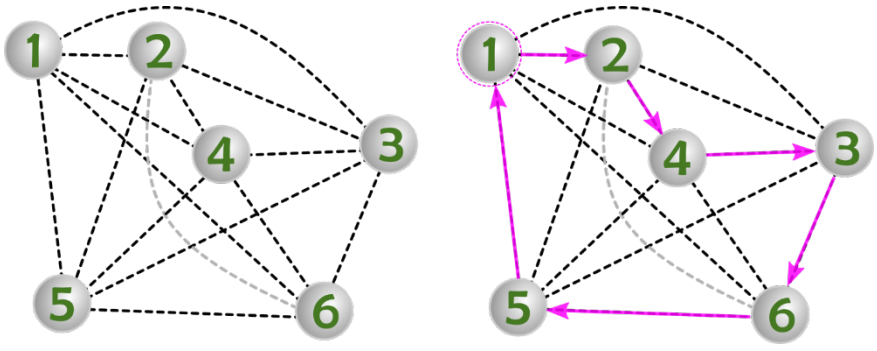

Figura 7 - Exemplo de aplicação do Algoritmo *Closest Neighbor Construction*.

Closest Insertion Method

Ainda segundo Yu e Gen (2010), outro método para construir uma solução para o TSP é partindo-se de um ciclo, iterativamente incluir uma nova cidade no ciclo de maneira que aumente a distância minimamente até obter um ciclo Hamiltoniano para o problema. Os passos para este procedimento são dados a seguir:

1. Use algum método para seleção de construção para construir um ciclo de três cidades. Por exemplo, se usarmos o *Closest Neighbor Method* e partirmos da cidade 5, teremos um resultado inicial conforme mostrado na Figura 8, ciclo (5-4-2-5);

2. Selecione a cidade mais próxima (ainda não visitada) do ciclo atual. Na Figura 8, a cidade 1 foi a selecionada por possuir a menor distância com o ciclo;

3. Suponha que haja *m* arestas no ciclo atual, há *m* formas de como incluir o novo nó no ciclo. No caso do exemplo da Figura 8, temos três novos ciclos a se considerar: (5-4-2-1-5), (5-4-1-2-5) e (5-1-2-4-5). Selecione o ciclo que acrescente o menor valor à nova solução;

4. Repita os passos 2 e 3 até que todas as cidades tenham sido incluídas no ciclo.

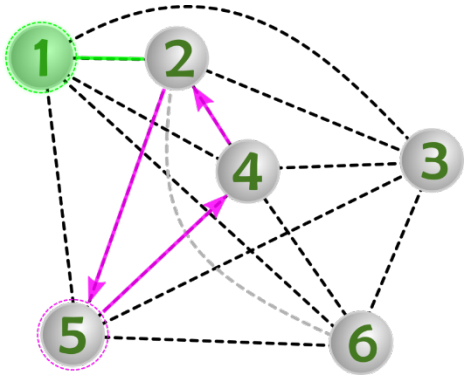

Figura 8 - Um ciclo inicial e seleção da próxima cidade do ciclo formado pelo *Closest Insertion Method*.

3.1.2. MÉTODOS DE BUSCA LOCAL

Em Croes (1958) foi proposto um método simples, porém muito eficaz, de busca local chamado *2-opt,* uma abreviação para*"local optimal by 2-edge perturbation"* (YU e GEN, 2010). Este método consiste em substituir duas arestas não adjacentes (ou seja, não compartilham cidades na solução corrente) e adiciona as arestas necessárias para transformar a solução atual em uma nova solução. A Figura 9 mostra um exemplo de um passo, ou seja, uma perturbação feita pelo procedimento *2-opt*.

Pela Figura 9, as arestas *(w, z)* e *(x, y)* foram trocadas de forma que ao serem excluídas, formam-se as arestas *(w, x)* e *(y, z)*. Cada movimento de troca, como o ilustrado na Figura 9, é conhecido como *2-interchange* (MICHALEWICZ e FOGEL, 2013). Portanto, chamamos vizinhança *2-opt* ao conjunto de soluções geradas a partir de uma solução corrente, onde se realizam todos os movimentos de *2-interchange*. Uma chamada ao *2-opt* se encerra quando nenhuma

das soluções em sua vizinhança apresenta melhora. Neste sentido, para que o algoritmo possa explorar várias áreas do domínio, é preciso chamar o procedimento várias vezes, de forma que a cada chamada, uma solução inicial diferente seja passada ao método.

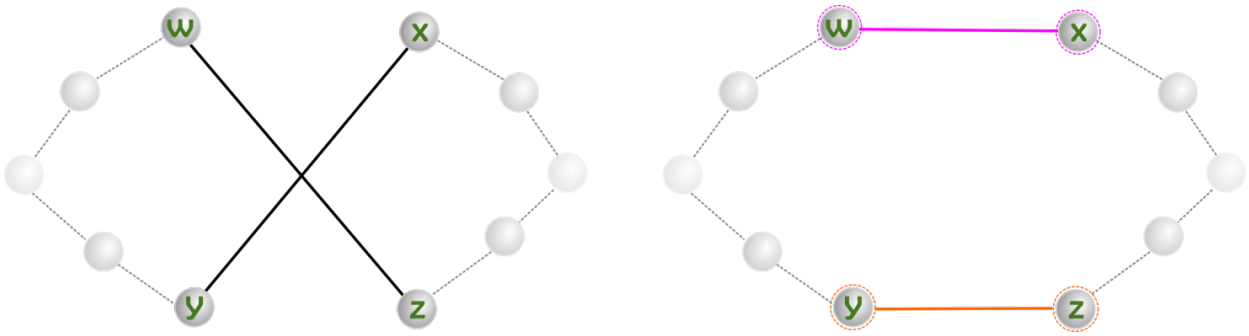

Figura 9 - Exemplo de funcionamento do método 2-interchange.

Naturalmente, pode-se pensar na extensão do *2-opt*, em lugar de considerar apenas duas arestas não adjacentes, pode ser possível considerar três. O método *3-opt* foi sugerido por Lin (1965). Com um conjunto de três arestas, sete novas soluções são possíveis, porém três delas pertencem ao conjunto gerado pela *2-opt*. Devido ao maior número de soluções geradas em uma vizinhança *3-opt*, por exemplo, uma alternativa é ignorar as soluções *2-opt* geradas no procedimento. Em geral, os resultados da *3-opt* são melhores do que aqueles da *2-opt* (YU e GEN, 2010).

Poderíamos então generalizar os procedimentos acima para um procedimento *k-opt*, onde *k* é a quantidade de arestas que seriam consideradas para haver a recomposição de novas arestas. O problema é que o tamanho da vizinhança aumenta exponencialmente em função de *k*. Lin e Kernighan (1973) pensaram neste problema e criaram uma solução adaptativa, de forma que o valor de *k* mude de uma interação para outra (MICHALEWICZ e FOGEL, 2013). Mais detalhes sobre esta última solução podem ser encontrados em Lin e Kernighan (1973).

3.2. MÉTODOS HEURÍSTICOS PARA O SAT

Pode-se considerar um método construtivo para o problema SAT. Por exemplo, começamos atribuindo um valor booleano para cada variável, sob a orientação de uma heurística para guiar o processo de tomada de decisão, que poderia ser: para cada variável de 1 a *n*, em qualquer ordem, atribua o valor verdadeiro que resulte em tornar verdadeiro o maior número de cláusulas atualmente com o valor falso. Dando empate, escolha a melhor entre as melhores de forma aleatória. Veja o exemplo apresentado por Michalewicz e Fogel (2013): $\overline{x_1} \wedge (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_1 \vee x_4)$.

Se a solução fizer com que a variável x_1=VERDADEIRO, o algoritmo guloso como o apresentado vai optar por esta solução, pois fazendo-se isso, três cláusulas terão seus valores avaliados para VERDADEIRO. Sobretudo, verifica-se que a busca se encerrará, pois qualquer valor que se atribua às variáveis x_2,x_3,x_4 não terá qualquer efeito (ou seja, não mudará o valor da primeira cláusula $\overline{x_1}$), enquanto x_1=VERDADEIRO. Os autores ainda argumentam que ainda que fosse possível criar um algoritmo construtivo guloso para o SAT, muito provavelmente não funcionaria para todas as instâncias do problema.

Um dos procedimentos de busca local de maior sucesso no SAT foi proposto por Selman, Levesque e Mitchel (1992), conhecido como GSAT. Trata-se de um método de busca local guloso, é apresentado na Figura 10. O procedimento GSAT inicia com uma cadeia de valores booleanos para as variáveis atribuídos de forma aleatória. Prossegue então por trocar (*flip*) o valor booleano da variável tal que produza o maior aumento no número total de cláusulas satisfeitas. Estas trocas são feitas até que se encontre uma atribuição tal que torne a função booleana verdadeira, ou que alcance o número máximo de trocas possíveis, variável controlada pelo valor de MAX-FLIPS. Este processo é então repetido até no máximo MAX-TRIES vezes. Perceba, no entanto, que para cada tentativa (laço de repetição envolvendo MAX-TRIES), uma nova solução aleatória é gerada. Então, tenta-se explorar a vizinhança daquela solução, levando para outras soluções para terem suas vizinhanças exploradas, enquanto durar MAX-FLIPS.

```
1:  for i = 1 ... MAX_TRIES do
2:      T ← uma string aleatória de valores booleanos
3:      for j = 1 ... MAX_FLIPS do
4:          if f(T) = TRUE then
5:              Retorne T
6:          else
7:              p ← uma variável tal que seu valor trocado retorna o maior au-
                mento no número de cláusulas satisfeitas de f(T)
8:              T ← flip(T, p)
9:          end if
10:     end for
11:     Retorne "Solução não encontrada"
12: end for
```

Figura 10 - Procedimento GSAT proposto por (SELMAN, LEVESQUE e MITCHEL, 1992).

REFERÊNCIAS

CROES, Georges A. A method for solving traveling-salesman problems. **Operations research**, v. 6, n. 6, p. 791-812, 1958.

LIN, Shen. Computer solutions of the traveling salesman problem. **Bell System Technical Journal**, v. 44, n. 10, p. 2245-2269, 1965.

LIN, Shen; KERNIGHAN, Brian W. An effective heuristic algorithm for the traveling-salesman problem. **Operations research**, v. 21, n. 2, p. 498-516, 1973.

MICHALEWICZ, Zbigniew; FOGEL, David B. **How to solve it: modern heuristics**. Springer Science & Business Media, 2013.

SELMAN, Bart; LEVESQUE, H.; MITCHELL, David. A new method for solving hard satisfiability problems. **Proceedings of the Tenth National Confer-ence on Artificial Intelligence**, p. 440-446, 1992.

YU, Xinjie; GEN, Mitsuo. **Introduction to evolutionary algorithms**. Springer Science & Business Media, 2010.

4. RECOZIMENTO SIMULADO

Pelo exposto até o momento, vimos que os métodos apresentados podem ficar presos a ótimos locais. Estratégias que sempre executam melhoras, tais como o método de *subida de encosta,* só podem oferecer respostas diferentes caso sejam inicializadas em vários pontos distintos, por várias vezes. Este capítulo trata de um método chamado Recozimento Simulado (*Simulated Annealing* - SA) que possui um mecanismo para escapar de ótimos locais, por aceitar soluções piores, com certa probabilidade, para atualizar o ponto de referência corrente durante a busca.

O nome recozimento simulado remete ao processo de aquecimento de materiais (principalmente metais) e seu resfriamento controlado de forma que, ao aquecer, as partículas se agitam mantendo o sistema com muita energia. Cada uma destas partículas pode ser vista como soluções caminhando pelo espaço de busca. Em temperaturas altas, esta busca é mais exploratória, ou seja, contempla mais estados distintos. Conforme a temperatura do sistema vai diminuindo, ou seja, o material vai esfriando, estas partículas já não se agitam tanto. Neste sentido, a etapa de exploração do espaço de busca vai diminuindo e dando espaço para intensificação de estados mais locais, buscando-se melhorias apenas nas vizinhanças onde o estado se encontra. Este processo se conduz até que o sistema tenha convergido para o completo resfriamento, e consequentemente, seu estado final. Esta observação foi simulada pelo algoritmo de Metropolis *et. al.* (1953). Sobretudo, em Kirkpatrick *et al*. (1983) foi observada a relação entre esta simulação e o processo de otimização.

Enquanto que um procedimento de busca local convencional geralmente finaliza até que a vizinhança tenha sido explorada (ou seja, não haja mais possibilidade de melhora no contexto da pesquisa local), no processo de Recozimento Simulado existe uma variável de controle T relacionada à temperatura do sistema, a qual inicia com valores altos. Após aplicar uma perturbação, ou seja, uma investigação em algum ponto da vizinhança da solução atual, o método escolhe se aceita ou não a nova solução. Esta função de aceitação depende de uma probabilidade que envolve o parâmetro T. A probabilidade de aceitar uma solução pior do que a solução atual é mais alta, caso a temperatura esteja elevada. Perceba que este simples mecanismo permite um detalhe importante: a possibilidade de escapar de ótimos locais. Ao permitir a aceitação de uma solução pior do que a que se está investigando no momento, isso dá a possibilidade de levar a busca para áreas possivelmente mais promissoras.

Resgatando o procedimento de **subida de encosta** apresentado na Figura 6, no Capítulo 2, percebemos que é necessário visitar toda a vizinhança do ponto atual. Para chegar a um algoritmo de Recozimento Simulado, propõem-se então as seguintes primeiras duas modificações (MICHALEWICZ e FOGEL, 2013):

A) Em vez de checar por todas as soluções pertencentes à vizinhança do ponto atual v_c e selecionar a melhor, selecione apenas um ponto, v_n, desta vizinhança;

B) Aceite este novo ponto, ou seja, $v_c \leftarrow v_n$, com uma probabilidade que dependa do mérito relativo destes dois pontos, ou seja, a diferença entre os valores retornados pela função de avaliação para os dois pontos considerados.

Após realizar as modificações propostas, chegamos a um procedimento chamado *Subida de Encosta Estocástico* (Figura 11):

1: $t \leftarrow 0$
2: $\mathbf{v}_c \leftarrow$ solução aleatória
3: **repeat**
4: Selecione $\mathbf{v}_n$ de $N(\mathbf{v}_c)$ de forma aleatória
5: Aceite $\mathbf{v}_c \leftarrow \mathbf{v}_n$ com probabilidade $\frac{1}{1+e^{\frac{eval(v_c)-eval(v_n)}{T}}}$
6: $t \leftarrow t + 1$
7: **until** t = MAX

Figura 11 - Subida de Encosta Estocástico.

O primeiro ponto a respeito deste algoritmo é que a fórmula de aceitação está considerando uma função de avaliação para maximização. Ela possui apenas um laço de repetição, que investigará a área de busca durante *MAX* iterações, para uma única temperatura *T*. Perceba que não é necessário reiniciar a busca em pontos diferentes. O novo ponto selecionado é "aceito" com uma determinada probabilidade. Esta probabilidade depende de dois parâmetros: o primeiro é a diferença entre as avaliações e o segundo é *T*. Se consideramos que o parâmetro *T* é fixo, então a probabilidade de aceitação depende de quão discrepante esta solução está da solução atual.

Vamos supor que a diferença entre a solução atual e a nova solução é de 13 unidades de melhora. Se colocarmos este valor na equação de probabilidade do algoritmo, variando-se o parâmetro *T*, teremos um gráfico semelhante ao da Figura 12. Perceba que em temperaturas baixas, a probabilidade de aceitação se aproxima do máximo (100%). Note, contudo, que conforme *T* se aproxima do valor 100 (ou seja, uma alta temperatura) a decisão se torna quase que aleatória.

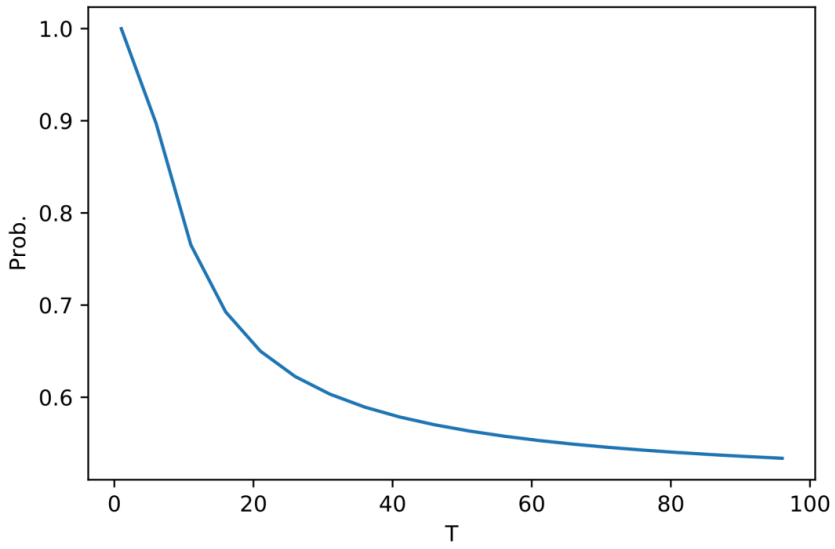

Figura 12 - Gráfico da probabilidade, variando-se o parâmetro T, quando o movimento é de melhora.

Analisando outro cenário, onde há piora, suponha que a solução atual tem valor de 130 unidades, enquanto que a nova solução tem valor de 110 unidades, ou seja, diferença de -20 pontos para maximização (solução pior do que a atual). O gráfico da temperatura neste cenário é dado na Figura 13. Perceba agora que, em baixos valores de *T*, a probabilidade de aceitação deste tipo de movimento é quase nula. Enquanto que, para temperaturas altas, a decisão novamente é semelhante a uma busca aleatória.

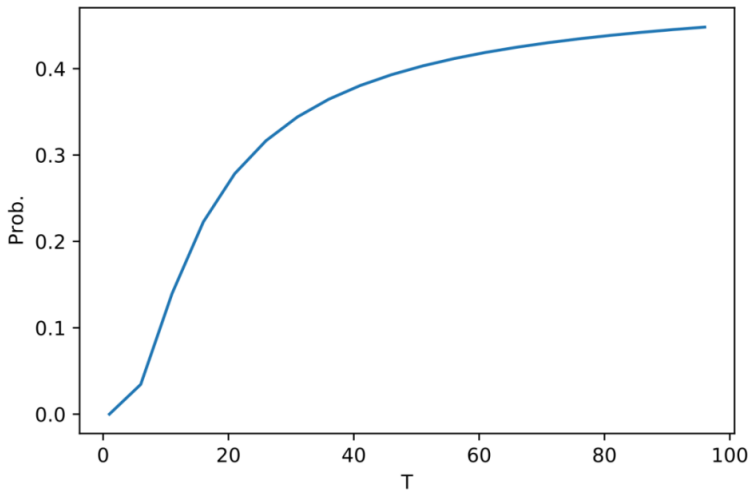

Figura 13 - Gráfico da temperatura em um cenário de movimento de piora.

Ao fixar o valor da temperatura, vejamos o que ocorre com a probabilidade em relação a soluções próximas. Suponha que a solução atual seja 110 unidades de avaliação. Suponha ainda que as soluções vizinhas possuam avaliações entre 80 e 150. Fixando a temperatura em *T=10*, a Figura 14 ilustra o comportamento da probabilidade. Perceba que a função de probabilidade é baixa quando a função de avaliação está distante (menor) do que a solução corrente. Conforme as soluções vizinhas possuam valores de avaliação maiores do que a solução atual, a probabilidade de aceitação aumenta gradativamente.

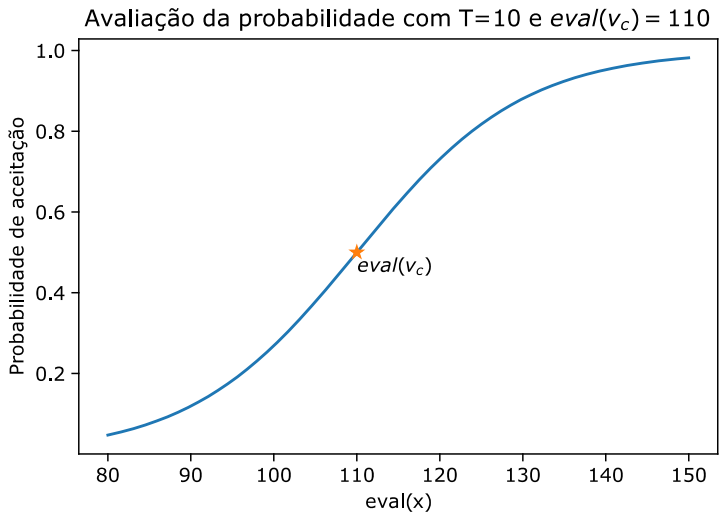

Figura 14 Avaliação da probabilidade com a temperatura T=10.

Por outro lado, se diminuirmos a temperatura para *T=0,4*, perceba que a função de probabilidade fica bem mais restrita, só aceitando soluções melhores do que a solução atual. A Figura 15 ilustra esta situação.

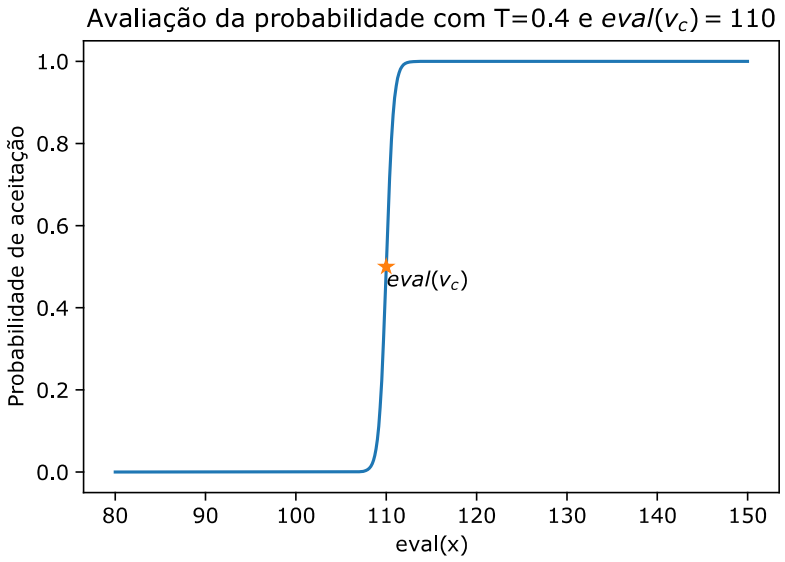

Figura 15 - Avaliação da probabilidade com T=0.4.

Como foi colocado no início deste capítulo, no processo de Recozimento Simulado o parâmetro T é atualizado, iniciando de um valor alto até um valor mais baixo. Ou seja, em temperaturas altas a busca é semelhante à aleatória, sobretudo conforme a temperatura vai "esfriando", o comportamento do algoritmo se aproxima do procedimento de Subida de Encosta visto no Capítulo 2. O algoritmo completo pode ser visto na Figura 16.

1: $t \leftarrow 0$
2: Inicialize T com uma temperatura alta
3: $\mathbf{v}_c \leftarrow$ solução aleatória
4: **repeat**
5: **repeat**
6: Selecione $\mathbf{v}_n$ de $N(\mathbf{v}_c)$ de forma aleatória
7: **if** $eval(\mathbf{v}_n) > eval(\mathbf{v}_c)$ **then**
8: $\mathbf{v}_c \leftarrow \mathbf{v}_n$
9: **else**
10: **if** $random[0,1) < e^{\frac{eval(v_c)-eval(v_n)}{T}}$ **then**
11: $\mathbf{v}_c \leftarrow \mathbf{v}_n$
12: **end if**
13: **end if**
14: **until** (Parada de busca na temperatura T)
15: $T \leftarrow g(T,t)$
16: $t \leftarrow t+1$
17: **until** Critério de finalização

Figura 16 - Procedimento Recozimento Simulado.

A partir do algoritmo definido, ainda é preciso adaptá-lo para servir como otimizador. Para isso, é necessário responder a algumas questões estipuladas no capítulo 2. O que de fato é a representação de uma solução? Como construir o conjunto de vizinhos de uma solução? Qual a função custo a ser otimizada? Qual espaço de busca e como determinar o espaço factível de busca?

As respostas a estas questões dependem unicamente do problema a ser resolvido. Mas suponha que tenham sido resolvidas. Olhando ainda para o algoritmo proposto, outras questões são relevantes tais como: A) o que é uma temperatura "alta"? Ou como determinar esta temperatura inicial? Na atualização da temperatura, o algoritmo faz uso de uma função de esfriamento $g(T,t)$, entretanto, que tipo de função deve ser usada? O que é terminar a busca em um determinado valor de temperatura (laço mais interno)? E, além disso, o que é saber que o sistema está resfriado, ou seja, como determinar o critério de finalização? Ilustraremos algumas respostas para estas perguntas nas seções que seguem ao aplicarmos o Recozimento Simulado a alguns problemas.

4.1. RECOZIMENTO SIMULADO APLICADO A UM PROBLEMA SIMPLES

Para mostrar a eficiência do SA em sair de mínimos locais, vamos aplica-lo inicialmente a uma função multimodal 2D, $f(x) = xsen(10x\pi) + 2$, cujo gráfico pode ser visto na Figura 17. Buscaremos pela solução no intervalo [-1,2].

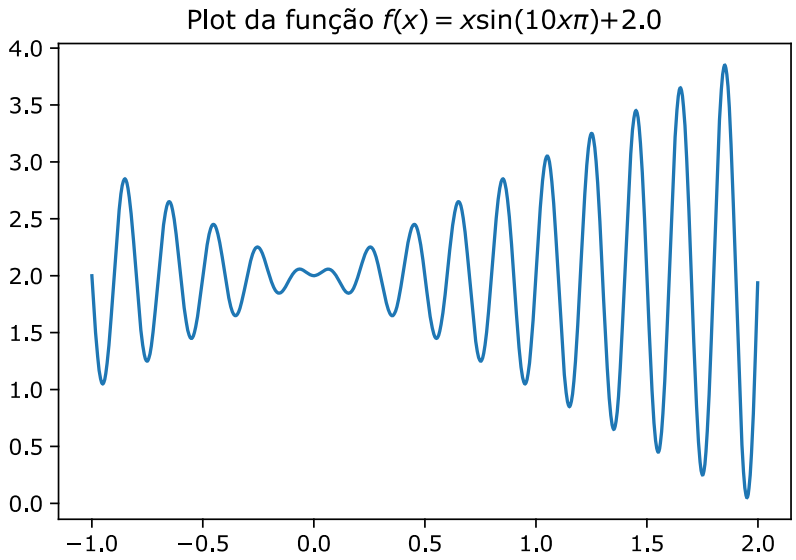

Figura 17 Gráfico da função a ser otimizada.

Para mostrar a eficiência do SA, vamos propor um algoritmo que tenha apenas uma descida de temperatura. Ou seja, gera apenas uma solução inicial, e a partir desta, explora a área utilizando o mecanismo estipulado até o momento. Neste caso, iremos usar uma função de resfriamento com decaimento exponencial. Além disso, vamos estipular que para cada nível de temperatura,

a solução seja explorada por no máximo 10 vezes localmente. Ou seja, dentro daquela temperatura, 10 outras soluções serão geradas a partir do ponto corrente. O algoritmo proposto pode ser visto na Figura 18.

1: $v_c \leftarrow$ um número aleatório no intervalo $[-1, 2]$
2: $best = v_c$
3: $j \leftarrow 0$
4: **repeat**
5: $T = T_{max}.e^{-j.r}$
6: $v_n =$
7: **for** $k = 1$ até MAX **do**
8: $v_n \leftarrow$ um número aleatório no intervalo $[vc - \epsilon, vc + \epsilon]$
9: $\Delta \leftarrow f(v_c) - f(v_n)$
10: **if** $random[0, 1) < e^{-\frac{\Delta}{T}}$ **then**
11: $v_c \leftarrow v_n$
12: **end if**
13: Atualize $best$
14: **end for**
15: $j \leftarrow j + 1$
16: **until** T ultrapasse T_{min}

Figura 18 - Protótipo de um SA para otimizar f(x).

Ao visualizar o algoritmo proposto na Figura 18, percebe-se que os seguintes parâmetros devem ser definidos: T_{max},T_{min},ϵ *e* r. A definição destes parâmetros é dependente do problema a ser resolvido. Vamos executar o algoritmo acima para os seguintes valores de parâmetros T_{max}=*10*,T_{min}=*0,01*,ϵ=*0,5 e* r=*0,5*. A Figura 19 mostra o resultado de uma execução do algoritmo proposto. Perceba que apesar da solução inicial estar perto, houve uma exploração do espaço de busca. Nesta execução, a melhor solução encontrada também condiz com o máximo global, entretanto, isso não é garantido pelo algoritmo.

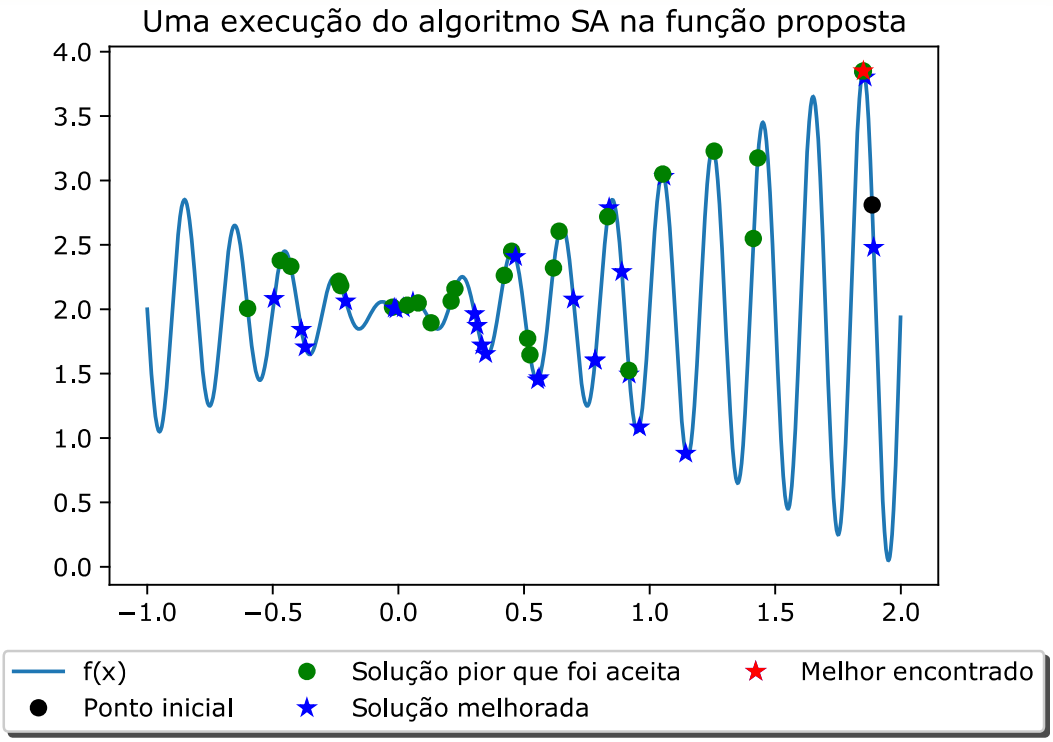

Figura 19 – Gráfico representando uma execução do SA abordado.

4.2. RECOZIMENTO SIMULADO APLICADO AO SAT

O problema SAT tem como entrada uma função booleana com n variáveis. Portanto, busca-se uma cadeia com *n* valores booleanos, que quando seus valores são atribuídos às respectivas variáveis, a equação se torna verdadeira. No modelo aqui exposto, nossa função objetivo retorna a quantidade de cláusulas satisfeitas pela cadeia de booleanos. Portanto, um modelo de SA para este problema pode ser visto na Figura 20. Ao examinar o algoritmo, percebemos que existem dois laços principais. O primeiro controla uma quantidade de tentativas globais, controladas pela variável . Esta variável controla quantas vezes o algoritmo de busca será executado, ou seja, quantas vezes o sistema de recozimento será executado para soluções aleatórias.

1: $global_t \leftarrow 0$
2: **repeat**
3: $\mathbf{v} \leftarrow$ uma string aleatória de valores V e F
4: $j \leftarrow 0$
5: **repeat**
6: $T = T_{max}.e^{-j.r}$
7: **for** $k = 1$ até $len(\mathbf{v})$ **do**
8: **if** $eval(\mathbf{v}) =$MAX-CLAUSES **then**
9: retorne $\mathbf{v}$
10: **end if**
11: $v_n \leftarrow flip(\mathbf{v}, k)$
12: $\delta \leftarrow eval(v_n) - eval(\mathbf{v})$
13: **if** $random[0, 1) < (1 + e^{-\frac{\delta}{T}})^{-1}$ **then**
14: $\mathbf{v} \leftarrow v_n$
15: **end if**
16: **end for**
17: $j \leftarrow j + 1$
18: **until** T ultrapasse T_{min}
19: $global_t \leftarrow global_t + 1$
20: **until** $global_t$ alcance o valor MAX-ITERATIONS

Figura 20 - SA aplicado ao problema MAX-SAT.

No laço mais interno, temos o esquema de resfriamento e busca pela vizinhança. A linha 6 apresenta o sistema de redução da temperatura. Esta atualização depende da temperatura máxima, do número da iteração durante o processo de busca (controlado pela variável *j*) e por um fator de decaimento (*r*). Quanto mais o processo avança, menos se retira do valor da temperatura. Para se ter uma noção de como o valor da temperatura é alterado pela função de resfriamento, a Figura 21 apresenta o gráfico da função para alguns valores do parâmetro *r*.

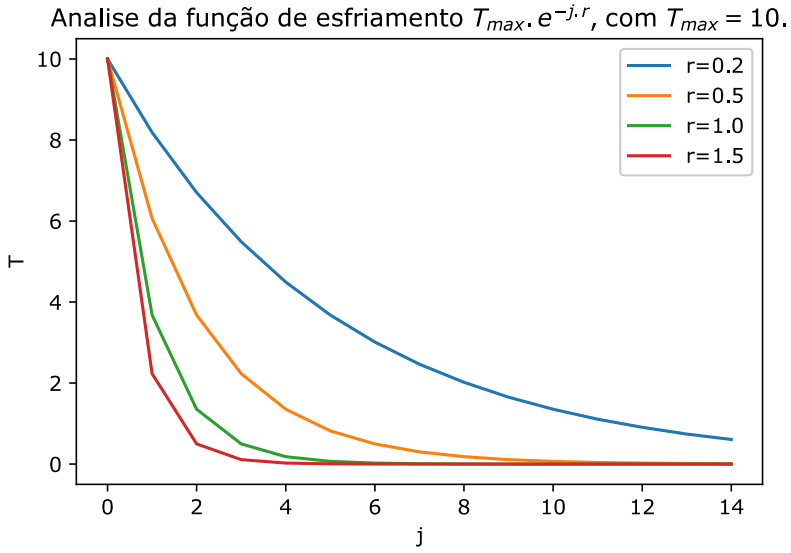

Figura 21 Análise da função de esfriamento.

Perceba que, quanto maior o valor de *r* mais forte é o decaimento dos valores. Ou seja, pode ser que o parâmetro *r* poderia ser alterado conforme o índice global fosse se aproximando do valor máximo de iterações globais. Outro detalhe do algoritmo exibido na Figura 20 é que ele só retorna se encontrar de fato uma solução para o problema. É interessante também colocar que a vizinhança de *1-flip* é toda explorada, e que a escolha de se mover para a próxima solução é dada pela probabilidade que depende da diferença entre o número de cláusulas satisfeitas pela nova solução e pela solução atual.

De acordo com Michalewicz e Fogel (2013), os autores de Spears (1996) usaram os seguintes parâmetros em uma versão do SA-SAT: $T_{max}=0,3$, $T_{min}=0,01$ e sobre o parâmetro *r*, utilizaram uma função inversa do produto do número de variáveis do problema e o número da tentativa atual. Perceba que o ajuste dos parâmetros impacta diretamente no comportamento do algoritmo. Ao diminuir o parâmetro *r* (ou aumentar o intervalo da temperatura), o número de tentativas independentes é também reduzido, sobretudo se explora mais outras atribuições aleatórias. Por outro lado, ao aumentar o parâmetro (ou diminuir o intervalo da temperatura), as buscas pelas vizinhanças se intensificam.

4.3. RECOZIMENTO SIMULADO APLICADO AO TSP

Apesar de muitas variantes do Recozimento Simulado terem sido aplicadas ao TSP, uma variante padrão pode ser baseada no algoritmo proposto na Figura 16, colocando apenas que v_c é um tour e $eval(v_c)$ devolve a soma das arestas envolvidas nesta solução. De acordo com Michalewicz e Fogel (2013), as diferenças entre implementações de SA para o TSP são:

1. **Os métodos para gerar a solução inicial**: podemos iniciar de uma solução aleatória ou a partir de uma solução produzida por algum algoritmo de busca local.
2. **A definição de vizinhança**, **Seleção de um vizinho** e **Métodos de resfriamento**:
3. **Condição de terminação**: o número de passos em cada temperatura pode ser proporcional ao tamanho da vizinhança;
4. **Existência pós-processamento**: é possível definir tamanhos de vizinhanças diferentes e incluir uma busca local para explorar os picos (visto que no SA não há garantias que isso é feito devido às diferentes possibilidades de resfriamento).

REFERÊNCIAS

KIRKPATRICK, S., GELATT, C.D. and VECCHI, M.P., 1983. *Optimization by simulated annealing*. science, 220(4598), pp.671-680.

METROPOLIS, N., ROSENBLUTH, A.W., ROSENBLUTH, M.N., Teller, A.H. and Teller, E., 1953. *Equation of state calculations by fast computing machines*. The journal of chemical physics, 21(6), pp.1087-1092.

MICHALEWICZ, Zbigniew; FOGEL, David B. **How to solve it: modern heuristics**. Springer Science & Business Media, 2013.

SPEARS, William M. Simulated annealing for hard satisfiability problems. **Cliques, Coloring, and Satisfiability**, v. 26, p. 533-558, 1993.

5. BUSCA TABU

A busca tabu, assim como o Recozimento Simulador, é baseada em busca em vizinhança e que também possui mecanismos para escapar de ótimos locais, porém de forma determinística fazendo uso de processos de memorização (RAYWARD-SMITH, 1996). O mecanismo de memorização geralmente envolve guardar soluções vistas anteriormente por algum período de tempo e, portanto, não podem compor os próximos movimentos exploratórios. Esta memória é uma lista implementada com estruturas de dados eficientes e é conhecida como lista 'tabu', ou seja, movimentos proibidos que perduram certo período de tempo (FONSECA et. al., 2015). Para Glover (1995), a Busca Tabu é uma metaheurística que guia uma técnica heurística de busca local para explorar o espaço de solução além da otimalidade local.

5.1. BUSCA TABU APLICADA AO PROBLEMA SAT

Para entendermos melhor os conceitos envolvidos na Busca Tabu, assuma que, igualmente aos demais algoritmos para o SAT, nossa solução seja representada por uma cadeia de booleanos. Vamos considerar que o tamanho desta cadeia seja 10, ou seja, nossa equação booleana envolve cláusulas com 10 variáveis. Suponha que tenhamos uma solução $x=[x_1,x_2,...,x_{10}]$ com valores aleatórios tais como exibidos a seguir:

x =	F_1	V_2	V_3	V_4	V_5	F_6	F_7	V_8	V_9	V_{10}

Suponha ainda que nossa função de avaliação seja uma soma das cláusulas tornadas verdadeiras na equação (ou poderia ser uma soma ponderada por pesos que estariam relacionados ao quantitativo de variáveis em determinada cláusula). Vamos supor que a cláusula tenha valor *f(x)*=30. Se estipularmos a vizinhança do tipo *1-flip*, teríamos então 10 (dez) outras soluções vizinhas. Naturalmente, selecionamos a melhor solução na vizinhança, *x'*, segundo o índice *f(x)*. Até este ponto, a busca segue a mesma ideia que o procedimento de Subida de Encosta já visto no Capítulo 2. Vamos supor que tenha sido obtida da operação de inversão do primeiro *bit* com *f(x')* = 35.

O que faremos agora é usar uma estrutura de memória para nos lembrarmos de qual bit e quando este bit foi modificado. Esta memória deve possuir um parâmetro de quantas vezes a informação deve ficar lá. Supondo que seja *K=5*, podemos criar uma estrutura de memória do tipo vetor, sempre que um bit for trocado, colocamos este número de lembranças na mesma posição correspondente ao bit. A cada nova iteração, este número é diminuído permitindo que no máximo até *K* rodadas, aquele bit possa ser novamente trocado. O que estamos fazendo na prática é impedir que determinado bit tenha seu valor trocado por pelo menos *K* rodadas. Neste caso, uma entrada *i* da memória poderá ser interpretada como "*o i-ésimo bit foi trocado* há *K-j iterações*".

x =	F_1	V_2	V_3	V_4	V_5	F_6	F_7	V_8	V_9	V_{10}
x_1	V_1	V_2	V_3	V_4	V_5	F_6	F_7	V_8	V_9	V_{10}
M	5	0	0	0	0	0	0	0	0	0

Nota de implementação
Conforme argumentado em Michalewicz e Fogel (1996), que a implementação proposta de guardar o valor K e ir decrementando pode custar até n operações de leitura e escrita, onde n é o número de variáveis consideradas. Na implementação de fato, é possível guardar um contador global de iterações no lugar de K. Neste caso, considerando a variável t a iteração atual, todo bit i tal que $t - M[i] > K$ está disponível para ter seu valor trocado (ou seja, deve ser "esquecido" da memória). Esta maneira requer que apenas uma operação de escrita seja necessária (a do bit que foi trocado, ganhando o valor da iteração atual t). Principalmente, para as explicações a seguir a maneira apresentada inicialmente será a adotada.

Suponha que após 4 (quatro) outras iterações consistindo em selecionar as melhores soluções das vizinhanças (note que pode ser que tenha navegado por soluções eventualmente piores, razão pela qual a busca tabu consegue sair de ótimos locais), a memória seja esta:

M =	1	2	0	5	0	0	0	0	3	4

Neste caso podemos concluir que: os bits 3, 5, 6, 7 e 8 podem ter seus valores trocados a qualquer momento. A troca do bit na posição 1 (um) é tabu por pelo menos uma iteração, o bit 2 por pelo menos 2 iterações e assim por diante. O bit 4 foi o que acabou de ser trocado (lembre-se que para este exemplo K=5). É fácil verificar que a simples equação K-$M[i]$, para todo $M[i]\neq 0$ nos oferece a informação de quantas iterações atrás a mudança no bit i foi feita. Por exemplo, considerando o bit na posição 9, sabemos que ele foi trocado há 5-3=2 iterações. Neste caso, o estado atual é dado por:

X_c	V_1	F_2	V_3	F_4	V_5	F_6	F_7	V_8	F_9	F_{10}	$f(x_c) = 40$
	1	2	0	5	0	0	0	0	3	4	

Vamos expandir este modelo para incluir a possível vizinhança desta solução.

Tabela 1 - Exemplo de memória, soluções vizinhas e soluções permitidas.

x_c	V_1	F_2	V_3	F_4	V_5	F_6	F_7	V_8	F_9	F_{10}	$f(x\_c) = 40$
M =	1	2	0	5	0	0	0	0	3	4	
$x_c^{(1)}$	**F_1**	F_2	V_3	F_4	V_5	F_6	F_7	V_8	F_9	F_{10}	
$x_c^{(2)}$	V_1	**V_2**	V_3	F_4	V_5	F_6	F_7	V_8	F_9	F_{10}	
$x_c^{(3)}$	V_1	F_2	**F_3**	F_4	V_5	F_6	F_7	V_8	F_9	F_{10}	$f(x\_c) = 30$
$x_c^{(4)}$	V_1	F_2	V_3	**V_4**	V_5	F_6	F_7	V_8	F_9	F_{10}	
$x_c^{(5)}$	V_1	F_2	V_3	F_4	**F_5**	F_6	F_7	V_8	F_9	F_{10}	$f(x\_c) = 33$
$x_c^{(6)}$	V_1	F_2	V_3	F_4	V_5	**V_6**	F_7	V_8	F_9	F_{10}	$f(x\_c) = 29$
$x_c^{(7)}$	V_1	F_2	V_3	F_4	V_5	F_6	**V_7**	V_8	F_9	F_{10}	$f(x\_c) = 30$
$x_c^{(8)}$	V_1	F_2	V_3	F_4	V_5	F_6	F_7	**F_8**	F_9	F_{10}	**$f(x\_c) = 38$**
$x_c^{(9)}$	V_1	F_2	V_3	F_4	V_5	F_6	F_7	V_8	**V_9**	F_{10}	
$x_c^{(10)}$	V_1	F_2	V_3	F_4	V_5	F_6	F_7	V_8	F_9	**V_{10}**	

Pelo exposto na Tabela 1, que apesar de listarmos todos os vizinhos gerados pela operação de 1-flip, apenas as soluções que trocam os bits nas posições 3, 5, 6, 7 e 8 é que estão disponíveis. Veja então que a melhor solução deve ser escolhida entre este subconjunto. Perceba ainda que a melhor solução entre elas é a $x_c^{(8)}$ cujo valor da função objetivo é de 38 unidades (pior que a solução atual cuja função objetivo é igual a 40). Após esta operação, os conteúdos da memória se modificam de forma a diminuir 1 (uma) unidade em cada entrada diferente de zero, se tornando:

x_c	V_1	F_2	V_3	F_4	V_5	F_6	F_7	F_8	F_9	F_{10}	$f(x_c) = 38$
M =	0	1	0	4	0	0	0	5	2	3	

Agora suponha que $f(x_c^{(4)}) = 55$, mesmo que $x_c^{(4)}$ esteja proibida neste movimento. Seria tentador quebrar a regra e considerar esta solução. Para tornar o processo de busca mais flexível, a busca tabu considera soluções de toda a vizinhança e em situações usuais opera conforme explicado até o momento. Portanto, ao encontrar uma solução muito melhor, a solução tabu pode ser escolhida como a próxima solução. Esta quebra de protocolo é chamada de *critério de aspiração* (MICHALEWICZ e FOGEL, 1996).

Até o momento, foi apresentada a memória de curto prazo ou *recency-based memory*. Entretanto, para que a busca tabu consiga empregar o conceito de diversidade outros mecanismos podem ser utilizados. Um destes mecanismos consiste em uma memória de longo prazo. Por exemplo, pode-se estipular um horizonte h para que cada entrada desta memória corresponda

ao número de visitas ou operações feitas no decorrer de uma janela de tamanho temporal *h*. No caso do SAT, um valor *j* associado ao *bit i* desta memória pode ser interpretado como "o bit *i* foi trocado *j* vezes durante as últimas *h* iterações".

Assim, a memória de longo prazo pode guardar informações úteis como "quais os bits que foram menos trocados", ou seja, espaço de busca menos explorado. Suponha uma situação em que todas as soluções disponíveis na vizinhança da solução atual levem a valores inferiores da função objetivo. Neste caso, uma escolha poderia ser guiada optando-se por soluções com bits com menor frequência. Geralmente este cálculo inclui uma função de penalidade, criando-se uma função objetivo adaptada para fazer este tipo de decisão.

Para ilustrar, considere o seguinte cenário proposto por Michalewicz e Fogel (1996): suponha que o valor $f(x_c) = 35$, para a solução atual x_c. As soluções permitidas são $x_c^{(2)}$, $x_c^{(3)}$ e $x_c^{(7)}$, cujos valores das funções objetivo são $f(x_c^{(2)}) = 30$, $f(x_c^{(3)}) = 33$ e $f(x_c^{(7)}) = 31$, e que não haja solução maior que a globalmente encontrada até este ponto da busca, ou seja, o critério de aspiração não pode ser aplicado. Neste caso, a função objetivo adaptada para julgar este critério poderia ser dada por $f'(x^{(i)}) = f(x^{(i)}) - 0{,}7LTM[i]$, onde *LTM*[*i*] é a entrada do bit *i* para a memória de longo prazo. Para este exemplo, vamos supor que o LTM tenha os seguintes valores: $LTM[2] = 7, LTM[3] = 11\ e\ LTM[7] = 1$. Portanto, teríamos os seguintes valores para a função objetivo adaptada: $f'(x_c^{(2)}) = 30 - 0{,}7 * 7 = 25{,}1$, $f'(x_c^{(3)}) = 33 - 0{,}7 * 11 = 25{,}3$ e $f'(x_c^{(7)}) = 31 - 0{,}7 * 1 =$ 30,3, ou seja, $x_c^{(7)}$ seria selecionada como próximo passo da busca.

De acordo com Fogel, outras opções podem ser incorporadas:

1. *Aspiration by search direction*: além de armazenar apenas os movimentos recentes, armazenar também se estes movimentos geraram alguma melhora na função objetivo;

2. *Aspiration by influence*: influência pode ser medida como a quantidade de mudança da nova solução em termos de distância entre a solução antiga e a nova ou a mudança na factibilidade da solução (em casos de problemas com restrição). Uma solução candidata terá mais influência caso possuir um passo mais "longo" em relação à solução atual;

3. *Aspiration by default*: se for necessário aplicar o critério de aspiração, então que se escolha o movimento mais "antigo", segundo a memória de longo prazo.

Um algoritmo de busca tabu para o problema SAT, envolvendo os conceitos aqui apresentados é ilustrado na Figura 22.

1: Seja n o número de variáveis do problema SAT, K o tamanho a memória de curto prazo e h o horizonte da memória de longo prazo
2: Inicialize os vetores $M[n]$ e $LTM[n]$
3: $x \leftarrow$ string aleatória de booleanos
4: $best \leftarrow x$
5: **repeat**
6: $N \leftarrow$ vizinhança 1-flip
7: $Tabu(N, M) \leftarrow$ subconjunto de soluções Tabu
8: $NoTabu(N, M) \leftarrow$ subconjunto de soluções disponíveis
9: $best_N \leftarrow$ melhor solução pertencente à vizinhança $NoTabu(N, M)$
10: $best_T \leftarrow$ melhor solução pertencente à vizinhança $Tabu(N, M)$
11: **if** $f(best_N) > f(x)$ **then**
12: $x \leftarrow best_N$
13: **else**
14: **if** $f(best_T) > f(best)$ **then**
15: $x \leftarrow best_T$
16: **else**
17: $x \leftarrow$ escolha usando algum critério que leve em conta LTM
18: **end if**
19: **end if**
20: Atualize M, LTM e $best$
21: **until** x não receba mais atualizações

Figura 22 - Um algoritmo de Busca Tabu para o problema SAT.

5.2. BUSCA TABU APLICADA AO TSP

Apresenta-se agora um esquema de solução para o problema do TSP usando a Busca Tabu. Por simplicidade, vamos considerar os seguintes itens:

1. **Representação**: a representação de um tour é simplesmente uma lista sem repetição dos índices das cidades e que o modelo é de um grafo completo (ou seja, há caminhos entre todas as cidades).

2. **Vizinhança**: a vizinhança estabelecida para este exemplo é a *2-swap*, onde duas cidades podem ser trocadas em um tour.

Diante das hipóteses estabelecidas anteriormente, vamos considerar um tour de seis cidades, neste caso um exemplo de representação de solução seria x = (2,3,1,6,4,5). Como já relatado anteriormente, a quantidade de vizinhos de uma rota x é $|N(x)| = \frac{6\times5}{2} = 15$. Neste caso bidimensional, podemos utilizar uma matriz triangular superior para representação da memória de curto prazo, conforme ilustra a Figura 23. Assim, assumindo k = 4 lembranças, por exemplo, cada célula $M[i][j]$ com $i < j$ gravaria quando foi a última vez que a cidade i foi trocada pela cidade j, conforme ilustra a Figura 24. Perceba nesta figura que a cidade troca entre a cidade 1 e a cidade 4 acabou de ocorrer, ou seja, foi efeito da última atualização da busca. Isto significa que se uma solução era x = (2,3,1,6,4,5), pelo estado da memória de curto prazo exibido na Figura 24, a nova solução passou a ser x'= (2,3,4,6,1,5).

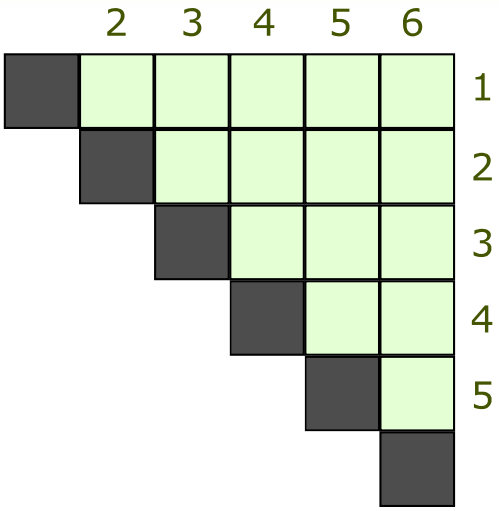

Figura 23 - Esquema de memória de curto prazo.

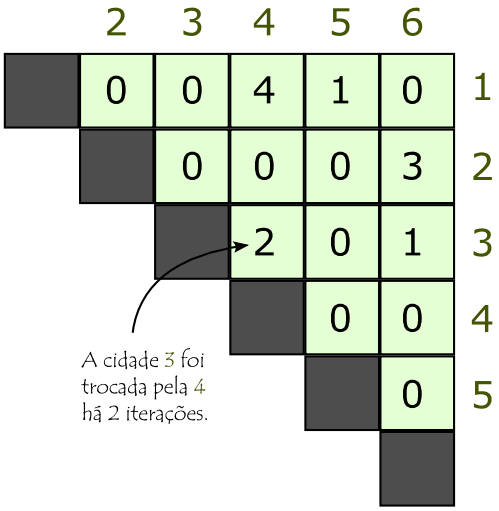

Figura 24 - Considerando *k = 4*, como fica a memória após algumas execuções da busca.

A mesma estrutura pode ser utilizada para a memória de longo prazo. Para um melhor aproveitamento, a matriz completa pode ser utilizada para representar ambas as memórias. A parte de baixo, porém, teria uma interpretação de índices diferente. Ou seja, enquanto a memória de curto prazo é indexada de forma $M[i][j]$ com $i < j$, a memória de longo prazo é referida como $M[j][i]$ com $j < i$ e diz respeito a quantas vezes a cidade i foi trocada pela j no decorrer do horizonte escolhido. A Figura 25 mostra o esquema completo.

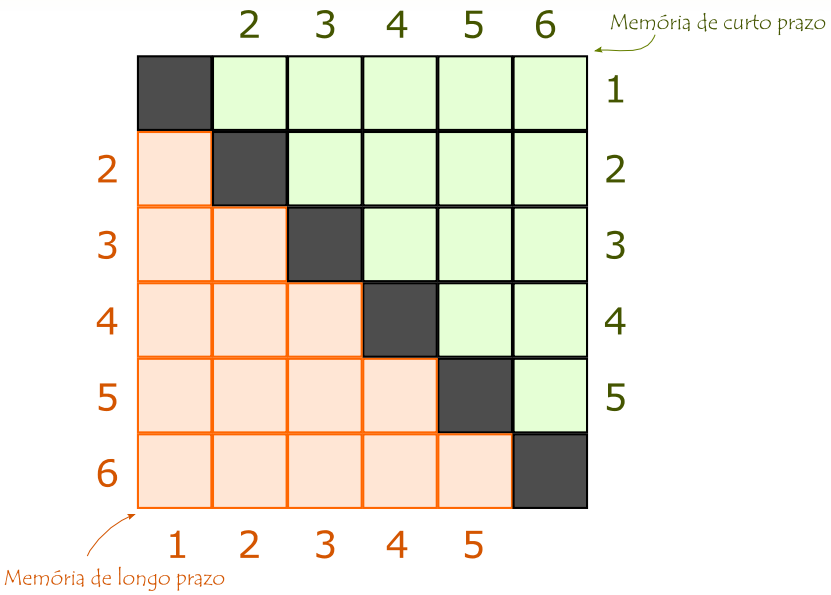

Figura 25 - Esquema de memória de curto e longo prazo.

A Figura 26 mostra um pseudocódigo do algoritmo de uma busca tabu aplicada ao problema de TSP. O algoritmo apresentado na figura apresenta o mecanismo de gerar tentativas. Ou seja, a busca vai ser chamada por um número *TOTAL* de vezes. Para cada tentativa, a busca tabu tentará melhorar ao máximo a solução *x*, através da exploração de sua vizinhança mantendo as estruturas apresentadas. Repare na linha 18 que a solução é escolhida segundo algum critério da memória de longo prazo, que está também codificada na mesma matriz de memória *M*. Vale ressaltar que este algoritmo mantém pelo menos duas variáveis chamadas *best_l* e *best_g* que significam o melhor global encontrado pela busca até o momento e o melhor local de uma tentativa.

Outro aspecto do algoritmo é que ele seleciona a vizinhança usando a operação padrão *2-swap*. Contudo, é natural que outras formas de exploração de vizinhança possam ser utilizadas. Em Knox (1996), é proposta uma busca tabu em que a vizinhança é explorada por apenas um número máximo de iterações além de utilizar a heurística *2-interchange* e fazer um tour ser tabu se ambas as arestas resultantes do *2-interchange* estiverem na lista tabu. Tentativas mais modernas incluem hibridização de métodos como SA e busca tabu conforme apresentado em Lin *et. al.* (2016).

1: Seja n o número de cidades da instância TSP, K o tamanho a memória de curto prazo e h o horizonte da memória de longo prazo, $best_g$ a variável que controla melhor global
2: **for** $trial = 0$ to $TOTAL$ **do**
3: Inicialize a matriz de memória $M[n][n]$
4: $x \leftarrow$ permutação aleatória de booleanos
5: $best_l \leftarrow x$
6: **repeat**
7: $N \leftarrow$ vizinhança 2-swap(x)
8: $Tabu(N, M) \leftarrow$ subconjunto de soluções Tabu
9: $NoTabu(N, M) \leftarrow$ subconjunto de soluções disponíveis
10: $best_N \leftarrow$ melhor solução pertencente à vizinhança $NoTabu(N, M)$
11: $best_T \leftarrow$ melhor solução pertencente à vizinhança $Tabu(N, M)$
12: **if** $f(best_N) > f(x)$ **then**
13: $x \leftarrow best_N$
14: **else**
15: **if** $f(best_T) > f(best)$ **then**
16: $x \leftarrow best_T$
17: **else**
18: $x \leftarrow$ escolha usando algum critério que leve em conta LTM
19: **end if**
20: **end if**
21: Atualize M, $best_l$ e $best_g$
22: **until** x não receba mais atualizações
23: **end for**

Figura 26 - Busca tabu aplicada ao TSP.

REFERÊNCIAS

FONSECA, F. A., ROCHA, M. L., PRATA, D. N., MOREIRA, P. L. A Decision-Making Technique for Financial Grant Allocation to Research Projects. International Proceedings of Economics Development and Research, v. 85, p. 119-124, 2015. Disponível em: http://www.ipedr.com/vol85/014-R011.pdf

GLOVER, Fred. **Tabu search fundamentals and uses**. Boulder: Graduate School of Business, University of Colorado, 1995.

KNOX, John. Tabu search performance on the symmetric traveling salesman problem. **Computers & Operations Research**, v. 21, n. 8, p. 867-876, 1994.

LIN, Yu; BIAN, Zheyong; LIU, Xiang. Developing a dynamic neighborhood structure for an adaptive hybrid simulated annealing–tabu search algorithm to solve the symmetrical traveling salesman problem. **Applied Soft Computing**, v. 49, p. 937-952, 2016.

MICHALEWICZ, Zbigniew; FOGEL, David B. **How to solve it: modern heuristics**. Springer Science & Business Media, 2013.

RAYWARD-SMITH, Victor J. et al. **Modern heuristic search methods**. Wiley, 1996.

6. ALGORITMOS GENÉTICOS

Algoritmos Genéticos (AG) são modelos de otimização estocásticos baseados em busca por um determinado espaço de solução, utilizando a analogia biológica para serem descritos e implementados, sendo proposto inicialmente no trabalho seminal de Holland (1975) e largamente divulgado e utilizado a partir do trabalho de Goldberg (1989). Por analogia biológica, entende-se a teoria da evolução de Charles Darwin, ou seja, em um grande espaço de busca (espaço de soluções possíveis para um determinado problema), os indivíduos melhores adaptados (melhores soluções candidatas) são capazes de se reproduzir (combinar soluções pré-existentes) e passar o seu material genético para as próximas gerações (possível solução para o problema que será considerada nas próximas iterações do algoritmo) (SIVANANDAM e DEEPA, 2008; YU e GEN, 2010).

Assim, algumas definições biológicas também devem ser consideradas na utilização de AGs na resolução de problemas. De modo geral, as espécies evoluem por meio de variações aleatórias (estocásticas) via processos de mutações, recombinação, seguidos por seleção natural. Por meio da seleção natural as espécies mais adaptadas ao ambiente tendem a sobreviver e se reproduzir, propagando seu material genético (HOLLAND, 1975).

Todos os organismos vivos constituem-se de células, e cada célula contém um conjunto de um ou mais c*romossomos* que são cadeias de DNA (*DeoxyriboNucleic Acid* / ácido desoxirribonucleico). Além disso, um cromossomo pode ser conceitualmente divido em *genes* que são blocos funcionais de DNA, cada um codificando uma partícula de proteína. As diferentes possibilidades que um gene pode assumir são chamadas de *alelos*. Cada gene está localizado em uma particular posição no cromossomo.

Há ainda muitos organismos que possuem múltiplos cromossomos em cada célula. A coleção completa do material genético é chamada de *genoma*. Já o *genótipo* é o conjunto de genes contidos em um genoma. Dois indivíduos que possuem genomas idênticos, possuem o mesmo genótipo.

Durante a reprodução ocorre a *recombinação* (ou *cruzamento*). Em cada indivíduo pai e mãe, genes são trocados entre cada par de cromossomos para formar um *gameta* (um único cromossomo). Então, os gametas provenientes de dois pais são usados para criar um conjunto completo de novos cromossomos na reprodução haploide, genes são trocados entre os cromossomos dos pais. Adicionalmente, novas gerações são sujeitas a *mutações*, operação em que as unidades elementares do DNA são trocadas nos pais, para gerar cromossomos filhos.

Por fim, há também o *fitness*, que é a probabilidade de que o organismo viva e possa reproduzir. Também pode ser uma função do número de indivíduos filhos (prole) que o organismo possui. Na resolução de problemas mais complexos e multiobjetivos, a função de *fitness* pode ser uma função de avaliação e ponderação das funções objetivo (SIVANANDAM e DEEPA, 2008; YU e GEN, 2010).

Todas definições biológicas são importantes porque ajudam a definir o problema em questão e a modelar o AG de maneira apropriada. A maior parte dos métodos (algoritmos) para resolução de problemas chamados de AG possuem no mínimo os seguintes elementos em comum:

- População de cromossomos (obtida aleatoriamente, a partir de uma ou um conjunto de soluções iniciais, ou mesmo uma combinação de ambas);
- Seleção de indivíduos de melhor qualidade de acordo com o *fitness* de cada indivíduo;
- Cruzamento para produzir uma nova geração (prole);
- Mutação aleatória da nova prole.

Os cromossomos em uma população são tipicamente cadeias de bits, podendo ser também sequências de valores contínuos ou discretos. No caso de cadeias de bits, os alelos têm dois valores possíveis: 0 ou 1. Cada cromossomo pode ser visto como um ponto no espaço de busca de soluções candidatas. Logo, os AGs processam populações de cromossomos, substituindo sucessivamente uma população por outra, onde a cada cromossomo é atribuído um valor de *fitness* na população em questão. O fitness de um cromossomo depende do quão bem aquele cromossomo resolve o problema em questão.

De acordo com Golberg (1989), um AG básico possui a seguinte estrutura:

- Dados:
    - Um problema claramente definido para ser resolvido;
    - Cadeias de bits (ou valores contínuos) representando soluções candidatas.
- Algoritmo:

1. Inicie com uma população gerada aleatoriamente de n cromossomos de l bits (soluções candidatas);
2. Calcule o *fitness* $f(x)$ de cada cromossomo x presente na população;
3. Repita os seguintes passos até que indivíduos tenham sido criados:

    A) Selecione um par de cromossomos pertencentes à população atual. A probabilidade de seleção deve ser uma função crescente do *fitness*. No processo de seleção, o mesmo cromossomo pode ser selecionado para ser pai mais de uma vez;

    B) Com probabilidade p_c (probabilidade de cruzamento ou taxa de cruzamento) execute o cruzamento a partir de um ponto aleatoriamente escolhido (escolhido com probabilidade uniforme ou não) para formar dois indivíduos filhos. Se não ocorrer cruzamento, os dois indivíduos gerados serão cópias exatas dos pais (cruzamento em um único ponto versus cruzamento multiponto);

    C) Execute mutação nos dois indivíduos filhos com probabilidade p_m (probabilidade de mutação ou taxa de mutação) e coloque os cromossomos resultantes na nova população. Se $m \geq n$, membros antigos (ou mesmo novos membros) serão descartados aleatoriamente.

    D) Substitua a população atual pela nova população;

    E) Volte ao passo 2.

Cada iteração é chamada de geração. Uma vez que uma nova população é criada, o processo entra em novo ciclo no qual os indivíduos são sujeitos aos mesmos operadores, e continua até que algum critério de parada seja satisfeito.

Um cromossomo de baixa qualidade pode sobreviver ao processo pelo acaso, contudo, se o valor de fitness de sua prole permanecer baixo, os genes irão perecer nas gerações subsequentes. Por outro lado, alguns dos indivíduos de genes não promissores podem se mostrar uteis quando incorporados em diferentes cromossomos por meio do cruzamento, dando a eles uma segunda chance. O processo oferece uma flexibilidade que seria impossível em um processo mais determinístico (SIVANANDAM e DEEPA, 2008).

Existe uma ampla gama de implementações e variações do AGs que são usados em um grande número de modelos e problemas científicos e de engenharia, tais como:

a) Programação automática – AGs são usados para evoluir programas computacionais para tarefas específicas, e para projetar outras estruturas computacionais, como autômatos e redes classificadoras;

b) *Machine Learning* – AGs têm sido usados em muitas aplicações, incluindo tarefas de classificação e predição. AGs também têm sido usados para evoluir aspectos de sistemas de aprendizado de máquinas, tais como pesos de redes neurais artificiais, sistemas de predição simbólica e sensores para robôs;

c) Economia – AGs têm sido usados para modelar processos de inovação e de mercados econômicos, como, por exemplo, mercado de ações;

d) Sistemas imunológicos – AGs têm sido usados para modelar vários aspectos de sistemas imunológicos naturais, incluindo mutações somáticas durante o tempo de vida de um indivíduo e a descoberta de famílias multigenes durante evoluções;

e) Ecologia – AGs têm sido usados para modelar fenômenos ecológicos como coevolução hospedeiro-parasita e simbioses;

f) Genética populacional – AGs têm sido usados para estudar questões como “Sob quais condições a ida de um gene para recombinação será viável?”;

g) Evolução e aprendizado – AGs têm sido usados para estudar como aprendizagem individual e evolução das espécies afetam uma a outra;

h) Sistemas sociais – AGs têm sido usados para estudar aspectos evolucionários de sistemas sociais, tais como evolução de comportamento social em colônias de insetos e, de forma geral, a evolução da cooperação e da comunicação em sistemas multiagentes;

i) Otimização – AGs têm sido usados em uma ampla variedade de tarefas de otimização, incluindo otimização numérica e problemas de otimização combinatória, como por exemplo, layout de circuitos.

Assim, de acordo com Michalewicz (1996), algumas vantagens do AGs são:

a) Otimizam com parâmetros discretos ou contínuos;

b) Não necessitam de cálculos de derivadas e podem trabalhar em espaços descontínuos e não-convexos;

c) Buscam simultaneamente pelo mínimo, a partir de uma amostragem ampla da superfície de custo;

d) Lidam com grandes números de parâmetros;

e) São adequados para computadores paralelos. Otimizam parâmetros com funções de custo altamente complexas;

f) Podem escapar de mínimos locais pobres;

g) Conseguem prover uma lista de parâmetro ótimos, não apenas uma única solução;

h) Pode codificar os parâmetros, de tal forma que a otimização seja feita com os parâmetros codificados;

i) Trabalham com dados numericamente gerados, dados experimentais ou funções analíticas.

6.1. COMPONENTES DE UM AG

De acordo com Goldberg (1989), pode-se definir um algoritmo genético da seguinte forma:

$$AG = (N,P,F,\Theta,\Omega,\Psi)$$

onde P é uma população de N indivíduos, $P = (S_1,S_2,...,S_n)$. Cada indivíduo $S_i, i = 1,...,N$, é um conjunto de valores inteiros de comprimento n que representa uma solução do problema. F representa a função de fitness que retorna um valor positivo e real na avaliação de cada indivíduo. Θ é um operador de seleção de pais que escolhe r indivíduos de P. Ω é um conjunto de operadores genéticos incluindo o crossover, denominado c, e a mutação, denominado M ou qualquer outro operador específico que produza s filhos a partir de r pais, tal como,

$$\Omega = \{\Omega_C, \Omega_M, \dots\}: \{p_1, p_2, \dots p_r\} \rightarrow \{f_1, f_2, \dots f_s\}$$

Ψ é o operador de remoção que retira s indivíduos selecionados na população P, permitindo que s filhos sejam adicionados à nova população $P_{(t+1)}$, como,

$$P_{t+1} = P_t - \Psi(P_t) + \{f_1, f_2, \dots, f_s\}$$

A função de custo é uma superfície com uma grande quantidade de picos e vales no espaço de parâmetros, como um mapa topográfico, como o exemplo da função Rastrigin mostrado na Figura 27. Os vales são o objetivo em uma busca por custo mínimo e os topos são uma busca por custo máximo. Técnicas tradicionais de otimização falham em encontrar os máximos globais, também chamados de *long's peak global*, a menos que iniciem a busca na vizinhança imediata dos picos. Todos os métodos que necessitam de gradiente da função de custo falharão com dados discretos (GOLDBARG e LUNA, 2005).

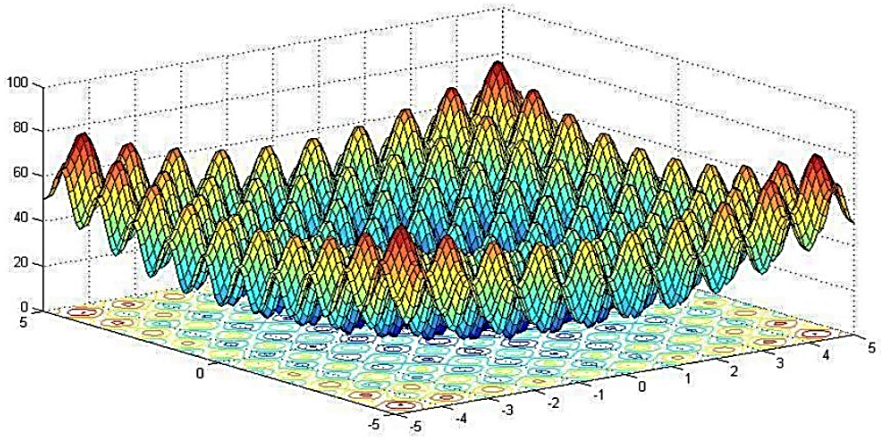

Figura 27 - Função Rastrigin com inúmeros ótimos locais.

A função de custo gera uma saída a partir de um conjunto de parâmetros de entrada (cromossomos). Portanto, é fundamental a escolha de uma função de custo apropriada e dos parâmetros adequados a serem utilizados. Os vetores de parâmetros que serão utilizados são vetores n_{par}-dimensionais, da forma,

$$cromossomo = (p_1, p_2, \ldots, p_{npar})$$

Em funções de custo bidimensionais a busca pelo valor máximo necessita de dois parâmetros de entrada ($n_{par} = 2$), *cromossomo = (x,y)*. Cada cromossomo tem o custo determinado avaliando a função de custo *f* em $p_1, p_2, \ldots, p_{npar}$:

$$custo = f(cromossomo) = f(p_1, p_2, \ldots, p_{npar})$$

Quando buscamos um vale, a função de custo é escrita como *f*(*x*,*y*) = -*elevação*, para que o problema seja tratado como um problema de minimização. Quando conhecemos a expressão analítica da função de custo, os parâmetros são as variáveis da função, caso contrário, a escolha dos parâmetros é baseada em estimativas empíricas, ou em tentativas.

Muitos problemas de otimização necessitam de limites ou restrições (>,<,≥,≤). Variáveis restritas podem ser transformadas em variáveis irrestritas, permitindo que um problema de otimização restrito possa ser tratado como irrestrito, através de um procedimento chamado penalização (BARBOSA e LEMONGE, 2008; MICHALEWICZ e SCHOENAUER, 1996).

Parâmetros dependentes representam problemas especiais para algoritmos de otimização. Na literatura de AGs, a interação entre parâmetros é chamada de epistasia. Entretanto, AGs desempenham adequadamente quando a epistasia for média ou alta (YU e GEN, 2010).

Quando mais de uma saída precisa ser considerada em uma função de custo, as saídas envolvidas possuem ordens de grandeza diferentes e são igualmente importantes na formação do custo.

6.2. REPRESENTAÇÃO DOS PARÂMETROS

O AG binário trabalha com um espaço de parâmetros finito (discreto), mas usualmente muito grande. Esta característica torna os AGs ideais para otimizar um custo que é devido a parâmetros que podem assumir um número finito de valores. Se um parâmetro é contínuo, precisa ser quantizado, como:

1. O intervalo contínuo é dividido em níveis de quantização iguais;
2. Qualquer valor que caia dentro de um dos níveis é quantizado para o valor médio, alto ou baixo daquele nível.

Em geral, escolher o valor médio é mais indicado, porque neste caso, o maior erro possível corresponderá à metade do nível. Escolher, o valor mínimo ou o valor máximo do nível permitirá um erro máximo igual ao nível de quantização.

Para exemplificar um AG básico e discreto, consideremos o TSP. Suponha que haja um vendedor que gostaria de começar em sua cidade natal e visitar outras cidades para promoção de mercadorias. Ele gostaria de visitar todas as outras cidades uma vez e retornar à sua cidade natal. O problema é como organizar a sequência de visitas dessas cidades para que a distância total / despesas de viagem sejam minimizadas. O TSP possui diversas aplicações práticas, como o problema de roteamento de veículos, o design de circuitos VLSI (*Very Large-Scale Integration*), o design da placa de circuito impresso, etc. (KUBAT, 2015).

Considere um circuito com 20 cidades, dispostas nas seguintes coordenadas no plano cartesiano (*x,y*): (60, 200); (180, 200); (80, 180); (140, 180); (20, 160); (100, 160); (200, 160); (140, 140); (40, 120); (100, 120); (180, 100); (60, 80); (120, 80); (180, 60); (20, 40); (100, 40); (200, 40); (20, 20); (60, 20); (160, 20). Para facilitar a visualização os pontos são dispostos no gráfico da Figura 28.

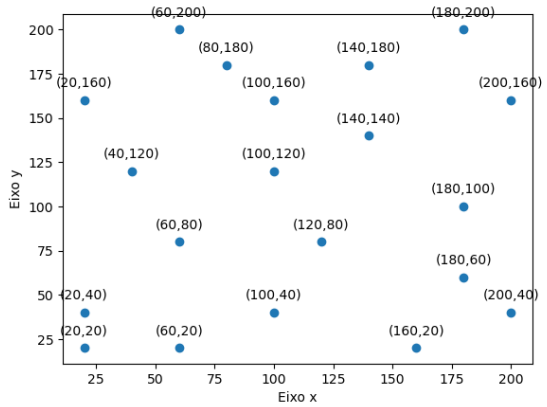

Figura 28 - Disposição das cidades no plano cartesiano mostrando.

A cidade ou ponto inicial não é relevante, já que poderia ser qualquer um dos 20 pontos, uma vez que se trata de um exemplo. Neste caso, o objeto é encontrar um tour mínimo entre as cidades. A todo o momento o AG precisará calcular a distância euclidiana entre duas cidades como candidatas a terem um roteiro entre elas, como:

$$D = \sqrt{|(x_i - x_{i+1})|^2 + |(y_i - y_{i+1})|^2}$$

onde x_i representa a coordenada x da cidade atual i, e y_i a coordenada y da cidade atual i. Logo, x_{i+1} e y_{i+1} representam as coordenadas da cidade de destino.

6.3. POPULAÇÃO INICIAL

O AG binário inicia com um grande número de indivíduos, ou cromossomos, chamado de população inicial ou I_{pop}. O I_{pop} tem n_{Ipop} cromossomos e é uma matriz de [n_{ipop} x n_{bits}], cujos elementos são zeros e uns gerados aleatoriamente a partir de.

$$I_{pop} = round\{random(n_{ipop}, n_{bits})\}$$

ou seja, cada linha da matriz é um cromossomo e cada cromossomo corresponde a valores discretos de longitude e latitude.

A exemplo do TSP, a população inicial é representada por objetos que são a sequência de pontos no tour, em que um indivíduo poderia ser: [(200,160), (40,120), (180,200), (100,40), (200,40), (60,200), (100,160), (80,180), (140,180), (60,20), (20,160), (100,120), (120,80), (180,60), (160,20), (20,40), (60,80), (140,140), (20,20), (180,100)], o qual pode ser observado na Figura 29.

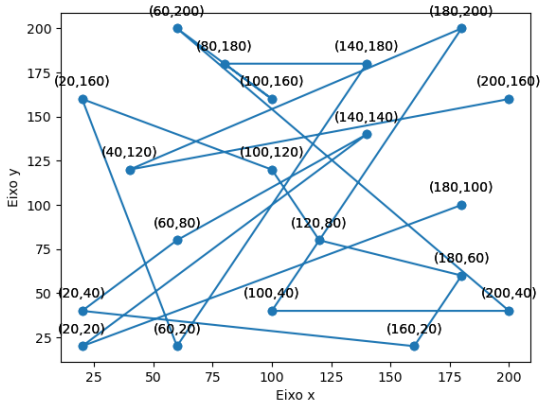

Figura 29 - Exemplo de um cromossomo na população inicial.

Uma representação alternativa para o problema seria usar uma codificação binária para representar números de cidades. Para um TSP de n-cidade, precisamos de $\lceil \log_2 n \rceil$ números binários para representar a cidade, onde $\lceil \ \rceil$ é a função teto (que arredonda para o primeiro inteiro acima). Um exemplo de código binário de um TSP de seis cidades é (000 010 001 100 011 101), o que significa um tour na sequência 1−3−2−5−4−6−1. Os operadores de variação padrão poderiam destruir a factibilidade da prole. Portanto, são necessárias técnicas de reparo extras. Geralmente, todo gene, por exemplo, cidade 3 (010), tem n possibilidades. Assim, para um tour pela cidade n, o domínio de operação do código binário é n^n (KUBAT, 2015).

6.4. SELEÇÃO NATURAL

A população inicial é em geral muito grande para ser inteiramente utilizada ao longo dos passos iterativos do AG. Uma porção de cromossomos de custo elevado descartada por meio de uma seleção natural. Apenas os $n_{pop} \leq n_{ipop}$ melhores membros da população são mantidos para serem operados por cada iteração do AG, e os demais são descartados. A seleção natural ocorre em cada geração do algoritmo. Dos n_{pop} cromossomos em uma geração, apenas os n_{good} superiores irão sobreviver para serem combinados (cruzados). Os n_{bad} cromossomos inferiores são descartados para que haja lugar para os cromossomos da futura geração.

A sobrevivência de um indivíduo é determinada probabilisticamente. Essa forma de seleção também é chamada de *seleção proporcional*. Temos F_i denotando o *fitness* do i-ésimo cromossomo e temos $F = \sum_i F_i$ sendo o somatório dos valores de *fitness* de toda a população que podem ser arranjados ao longo do intervalo (0,*F*]. O sobrevivente é modelado por um número aleatório gerado como $r \in (0, F]$, ou seja, o número no subintervalo determinado por *r* aponta para o sobrevivente. Assim, cromossomos com valores de *fitness* mais baixos são mais prováveis de serem eliminados, já os cromossomos com valores maiores podem ser selecionados como sobreviventes mais de uma vez (KUBAT, 2015; SIVANANDAM; DEEPA, 2008; YU; GEN, 2010).

Além da seleção proporcional também é possível implementar a *seleção por ranking*. A seleção por ranking pode ser implementada em duas condições. A primeira é que, às vezes, não podemos obter o valor exato de *fitness*, mas podemos obter o ranking, portanto, não podemos usar a seleção proporcional diretamente. A segunda é que podemos usar o ranking para ajustar efetivamente a pressão seletiva. Assim, o ranking é classificar os indivíduos primeiro e usar essas classificações para garantir a probabilidade de serem selecionados. Em seguida, a seleção proporcional pode ser usada (YU; GEN, 2010).

O ranking pode ser linear e existem vários métodos de classificação linear. O primeiro método é classificar o indivíduo da seguinte maneira: o melhor indivíduo tem classificação 0 e o pior tem *popsize*-1. Em seguida, podemos atribuir a probabilidade de ser selecionado p_i para o indivíduo *i* da seguinte maneira:

$$p_i = \frac{\alpha + \frac{rank_i}{popsize - 1}(\beta - \alpha)}{popsize}$$

onde α e β são parâmetros para controlar a pressão seletiva e é a classificação do indivíduo *i*. A classificação do melhor indivíduo é 0 e seu número esperado na seguinte seleção proporcional é α. A classificação do pior indivíduo é *popsize*-1 e seu número esperado na seguinte seleção proporcional é β.

Precisamos garantir que todas as probabilidades de serem selecionadas correspondam a adicionar 1. Portanto, podemos deduzir que α e β devem satisfazer $\alpha + \beta = 2$. O maior número de α é 2, o que significa que o pior indivíduo não será selecionado. E também podemos deduzir que o número esperado de um indivíduo com aptidão média é 1. Portanto, o número esperado do melhor indivíduo não é mais do que o dobro do número médio.

Um segundo método é classificar os indivíduos de maneira semelhante. O melhor tem classificação 1 e o pior tem classificação *popsize*. Em seguida, podemos atribuir a probabilidade de seleção p_i para o indivíduo *i* da seguinte maneira:

$$p_i = q - (rank_i - 1)r$$

onde *q* é o parâmetro para controlar a pressão seletiva, *r* é o parâmetro para garantir que a soma da probabilidade de ser selecionado seja 1 e $rank_i$ é a classificação do indivíduo *i*. Podemos deduzir que *q* e *r* devem satisfazer à equação:

$$q = \frac{r(popsize-1)}{2} + \frac{1}{popsize}.$$

Se $r = 0$, então $q = 1/popsize$. De acordo com (1), isso significa que todo indivíduo tem a mesma probabilidade de ser selecionado $1/popsize$, o que reflete a pressão seletiva mínima.

Se $r = \frac{2}{popsize(popsize-1)}$, então $q = 2/popsize$. Isso significa que o pior indivíduo não será selecionado, o que reflete a pressão seletiva máxima. Não é difícil obter o resultado de que a probabilidade de ser selecionado dos indivíduos na população atual diminua de *q* para $\left(\frac{2}{popsize} - q\right)$ linearmente.

Outra alternativa à classificação linear (a depender da modelagem e do problema em questão), a probabilidade de ser selecionado na classificação não-linear é a função não-linear da classificação. Também existem muitas implementações para a classificação não-linear. Uma função de densidade da distribuição geométrica para construir a probabilidade de ser selecionado é:

$$p_i = \alpha(1-\alpha)^{popsize-rank_i}$$

onde $0 < \alpha < 1$ é o parâmetro para controlar a pressão seletiva. Se *popsize* for muito grande, a soma da probabilidade é aproximadamente 1. O melhor indivíduo tem ranking no valor de *popsize*, o que aumenta a probabilidade de ser selecionado por *α*. O pior indivíduo tem a rank 1, o que aumenta a probabilidade de ser selecionado por $\alpha(1-\alpha)^{popsize-1} \approx 0$. Geralmente, a classificação não-linear tem uma pressão seletiva mais forte que a classificação linear, que é como a relação entre escala linear e escala não-linear (YU; GEN, 2010).

Os métodos de seleção discutidos até o momento consideram informações globais, isto é, o valor relativo de adequação ou a classificação. Às vezes, apenas informações locais, ou seja, a melhor em um pequeno grupo, estão disponíveis. Então a seleção de torneios é bastante útil. Para implementar a seleção de torneios, precisamos escolher *k* indivíduos aleatoriamente com a substituição e comparar os valores de *fitness* desses *k* indivíduos, que é o torneio. O melhor ganha o torneio e é selecionado para o cruzamento. O processo é repetido até que *popsize* indivíduos tenham sido selecionados.

O *k* é chamado de tamanho do torneio, que controla a pressão seletiva. Se $k = 1$, isso indica uma amostra aleatória na população sem pressão seletiva. Mas se $k = popsize$, o pool de cruzamento conterá apenas o melhor indivíduo da população atual. A seleção de torneio mais frequen-

temente usada é binária, em que $k = 2$. As principais características da seleção por torneio podem ser resumidas da seguinte forma:

- A seleção por torneio usa apenas informações locais;
- A seleção por torneio é muito fácil de implementar e sua complexidade de tempo é pequena;
- A seleção por torneio pode ser facilmente implementada em um ambiente paralelo.

Essas propriedades tornam a seleção por torneio bastante útil em algumas situações, como otimização multiobjetivo. Mas a seleção por torneio também sofre com o viés de seleção, o que significa que o melhor não será selecionado se for muito azarado e vice-versa (YU; GEN, 2010).

6.5. OPERADOR DE CRUZAMENTO

Uma estratégia trivial de seleção de pais é selecionar aleatoriamente pares de indivíduos na população dentro do intervalo [1,*popsize*], onde *popsize* é o número de indivíduos na população. Entretanto, esta técnica falha ao faz fazer jus aos indivíduos com valores de fitness mais altos, que provavelmente são mais atrativos do que outros. Uma maneira simples de refletir isso é ordenar os indivíduos em uma ordem decrescente de aptidão e, em seguida, emparelhar os vizinhos.

Geralmente, atribuiremos a probabilidade de cruzamento p_c, chamada de "taxa de cruzamento", para controlar a probabilidade de realizar um cruzamento. Para dois indivíduos selecionados, atribuímos um ponto entre 1 e l - 1 aleatoriamente, onde l é o comprimento do cromossomo. Isso significa gerar um número inteiro aleatório no intervalo [1,l-1]. Os genes após o ponto são alterados entre os pais e os cromossomos resultantes são descendentes dos pais. Chamamos esse operador de ponto único de cruzamento, ilustrado na Figura 30 (KUBAT, 2015; SIVANANDAM; DEEPA, 2008; YU; GEN, 2010).

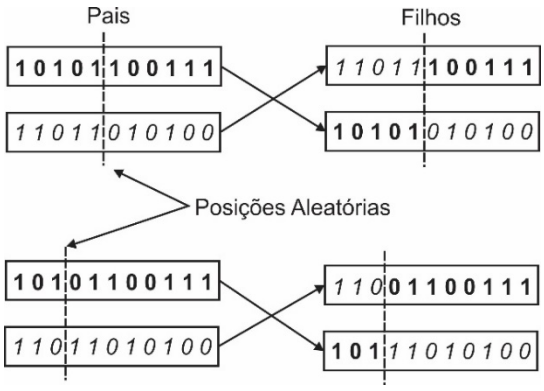

Figura 30 - Exemplo de cruzamento entre dois indivíduos binários utilizando um único ponto de corte.

Ainda, outra estratégia seria selecionar os pares probabilisticamente. O indivíduo de melhor ordenado escolhe seu parceiro usando o mecanismo empregado no jogo de sobrevivência. O mesmo é feito para o segundo indivíduo com a classificação mais alta, depois para o terceiro e assim por diante, até que a nova população atinja o tamanho necessário. Assim, é

provável que indivíduos melhores (embora não seja garantido) acasalem com outros indivíduos fortes. Às vezes, o parceiro terá um valor baixo (devido à seleção probabilística), mas isso gera a diversidade que dá ao sistema a oportunidade de preservar pedaços valiosos de cromossomos que só têm a má sorte de serem incorporados a cromossomos de baixa qualidade.

Uma das deficiências dessa abordagem é que um organismo muito bom pode ser substituído por filhos de menor valor e genes úteis podem desaparecer. Para impedir que isso aconteça, alguns algoritmos copiam as melhores amostras para a nova geração junto com seus descentes. Por exemplo, o algoritmo pode inserir diretamente na nova geração os 20% melhores sobreviventes e, em seguida, criar os 80% restantes aplicando os operadores de recombinação e mutação aos melhores 95% dos indivíduos, ignorando totalmente os 5% inferiores. Dessa maneira, não apenas os melhores espécimes viverão por mais tempo (até se tornarão "imortais"), mas o algoritmo também se livrará de alguns espécimes muito fracos que sobreviveram por mero acaso.

Em muitas aplicações, o operador de cruzamento é aplicado apenas a uma certa porcentagem de indivíduos. Por exemplo, se 50 pares foram selecionados para cruzamento e se a probabilidade de cruzamento tiver sido definida pelo usuário do AG como 80%, apenas 40 pares estarão sujeitos ao cruzamento, e os 10 restantes serão copiados para a próxima geração.

No exemplo do TSP, para uma população de 100 indivíduos, usando o método de seleção proporcional e uma taxa de cruzamento de 80%, o impacto no desempenho do algoritmo acaba se tornando ruim, como pode ser observado no gráfico da Figura 31. Comparativamente, com uma taxa de 20%, os resultados são melhores, conforme a Figura 6. No caso de $p_c = 0{,}2$, a distância final do tour (solução) é de 894,36, contra uma distância final de 1330,47 para $p_c = 0{,}8$, o que reflete uma degradação de 48% da solução.

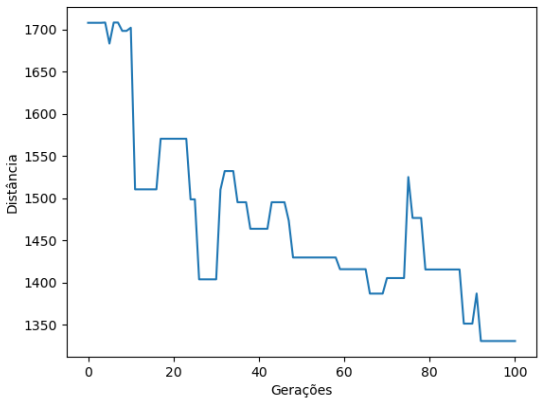

Figura 31 - Evolução do AG ao longo de 100 gerações para o TSP com uma taxa de cruzamento de 80%.

No caso de AGs contínuos e da mesma forma que nos AG binários, dois cromossomos pais são escolhidos e a prole será alguma combinação destes pais. Há muitas heurísticas para cruzamento em AGs de parâmetros contínuos. A mais frequentemente utilizada busca mimetizar as técnicas adotadas em AGs, de forma mais próxima possível.

O cruzamento em um AG contínuo, também chamado de *cruzamento aritmético*, pode ser realizado na escolha de um parâmetro aleatório no primeiro par de cromossomos pais para ser o ponto de cruzamento, tal como,

$$\alpha = round(rand \times n_{par})\text{C}$$

Sejam os cromossomos pais p_1 e p_2,

$$p_1 = [p_{m1}, p_{m2}, \ldots, p_{m\alpha}, \ldots, p_{mnpar}]$$

$$p_2 = [p_{p1}, p_{p2}, \ldots, p_{p\alpha}, \ldots, p_{pnpar}]$$

em que os subscritos *m* e *p* referem-se, respectivamente, a pai e mãe.

Os parâmetros selecionados são combinados para formar novos parâmetros que irão constituir parte dos cromossomos filhos:

$$p_{novo2} = p_{p\alpha} - \beta[p_{m\alpha} - p_{p\alpha}]$$

onde também é um valor aleatório entre 0 e 1 (SIVANANDAM; DEEPA, 2008; YU; GEN, 2010). O cruzamento é então completado com o resto do cromossomo,

$$f_1 = [p_{m1}, p_{m2}, \ldots, p_{novo1}, \ldots, p_{pnpar}]$$

$$f_2 = [p_{p1}, p_{p2}, \ldots, p_{novo2}, \ldots, p_{mnpar}]$$

6.6. OPERADOR DE MUTAÇÃO

A operação de mutação corrompe a informação genética herdada. Na prática, isso é feito ao trocar uma pequena porcentagem dos bits, no sentido de que o valor 0 de um bit é alterado para 1 ou o contrário. A porcentagem concreta (a frequência das mutações) é um parâmetro definido pelo usuário. Suponha que esse parâmetro exija que $p_m = 0{,}001$ dos bits sejam afetados. O método gerará, para cada bit, um número inteiro aleatório do intervalo [1,1000]. Se o número inteiro for igual a 1, o valor do bit será alterado, caso contrário, não será.

A Figura 32 ilustra o processo de cruzamento de duas maneiras distantes. Na primeira são escolhidos dois pontos de inversão de bits diferentes, e as posições entre os pontos são invertidas. No segundo caso, somente um ponto é escolhido para realizar a inversão.

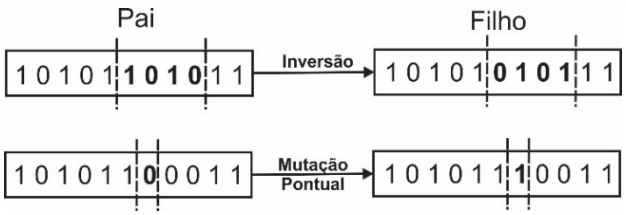

Figura 32 - Exemplo do processo de mutação em individuo binário, utilizando a inversão de bits multipontos na parte superior e único ponto na parte inferior da imagem.

Pensando um pouco sobre a frequência de mutações, de um lado, mutações muito raras dificilmente terão efeito. Mas por outro lado, a frequência de mutação muito alta interromperia a busca genética, danificando muitos cromossomos. Se a frequência se aproximar de 50%, cada novo cromossomo se comportará como uma sequência de bits gerada aleatoriamente; o AG então degenera para um gerador de números aleatórios.

O operador de mutação serve a um propósito diferente do operador de cruzamento. No cruzamento de um ponto, nenhuma informação nova é criada, apenas as subsequências existentes são trocadas. A mutação introduz uma nova reviravolta, anteriormente ausente na população (KUBAT, 2015).

Considerando mais uma vez o problema de TSP, e considerando duas taxas diferentes de mutação, $p_m = 0{,}001$ e $p_m = 0{,}01$, temos respectivamente as distâncias total e final do tour, 957,73 e 894,36, respectivamente, o que demostra que uma taxa inferior a 1% faz com que muitos indivíduos sejam apenas copiados para as próximas gerações, degenerando a solução em torno de 7%.

6.7. CONVERGÊNCIA E DEGENERAÇÃO PREMATURA

Uma simples implementação do AG será interrompida após um número predefinido de gerações. Uma versão mais sofisticada acompanhará o maior valor de fitness alcançado até o momento e encerrará a busca quando esse valor não melhorar mais. O fato de o valor da fitness ter atingido um patamar, pode não garantir que uma solução foi encontrada. Em vez disso, a busca pode ter atingido o estágio chamado *degeneração prematura*. Suponha que uma busca tenha atingido a seguinte população:

0 1 0 0 0

0 1 0 0 1

0 1 0 0 0

0 1 0 0 0

A recombinação não nos levará a lugar algum. Se os dois últimos cromossomos (idênticos) acasalarem, a prole será apenas cópia dos pais. Se os dois primeiros estiverem recombinados, o cruzamento de 1 ponto somente trocará o bit mais à direita, operação que também não cria um novo cromossomo. A única maneira de causar uma mudança é usar a mutação. Alterando os bits apropriados, a mutação pode reacender a busca. Por exemplo, isso acontecerá após a mutação do

terceiro bit no primeiro cromossomo e do quarto bit (da esquerda) do último cromossomo. Infelizmente, as mutações são raras e esperar que isso aconteça pode ser impraticável. Para todos os fins práticos, degeneração prematura significa que a busca ficou paralisada (KUBAT, 2015).

Uma maneira simples de identificar essa situação é calcular a similaridade média entre pares de cromossomos contando o número de bits que têm o mesmo valor nas duas sequências. Por exemplo, a semelhança entre [0 0 1 0 0] e [0 1 1 0 0] será 4 (quatro bits são iguais) e a semelhança entre [0 1 0 1 0] e [1 0 1 0 1] será 0.

Uma vez detectada uma queda na similaridade média de cromossomo para cromossomo, o sistema precisa reagir. Assim, as gerações avançadas serão marcadas por populações onde a maioria das amostras já está próxima do máximo. Esse tipo de degeneração certamente não será considerado prematuro. No entanto, a situação é diferente se o melhor cromossomo puder ser muito diferente da solução. Nesse evento, temos que aumentar a diversidade.

Várias estratégias podem ser usadas. A mais simples apenas inserirá na população atual um ou mais indivíduos aleatórios criados recentemente. Uma abordagem mais sofisticada executará o AG em duas ou mais populações em paralelo. Então, em intervalos aleatórios, ou sempre que houver suspeita de degeneração prematura, será permitido que uma amostra de uma população escolha seu parceiro em outra população. Ao implementar essa técnica, o programador deve decidir em qual população colocar a prole.

Atenção especial deve ser dada ao tamanho da população. Geralmente, embora nem sempre, o tamanho é mantido constante durante toda a busca genética. Como regra geral, populações menores precisarão de muitas gerações para alcançar uma boa solução - a menos que tenham degenerado prematuramente. Populações muito grandes podem ser robustas contra a degeneração, mas podem incorrer em custos computacionais impraticáveis (KUBAT, 2015).

Considerando o impacto da degeneração prematura para o TSP, as Figuras 34 e 35 apresentam a evolução da solução ao longo de 100 e 500 gerações, com exatamente os mesmos parâmetros (($p_c = 0{,}2$ e $p_m = 0{,}01$). Analisando os dois gráficos, podemos observar que próximo de 70 gerações (Figura 34), o AG não consegue encontrar uma melhor solução e que, antes de 200 gerações e após 400 gerações, o AG também não consegue encontrar melhores soluções (Figura 35). Evidentemente, a natureza do problema deve definir onde está o nível aceitável para solução do problema. Para o caso da Figura 33, o a solução encontrada foi uma distância final de 894,36 contra 871,11 para a solução da Figura 34, o que representa uma degeneração de somente 2%.

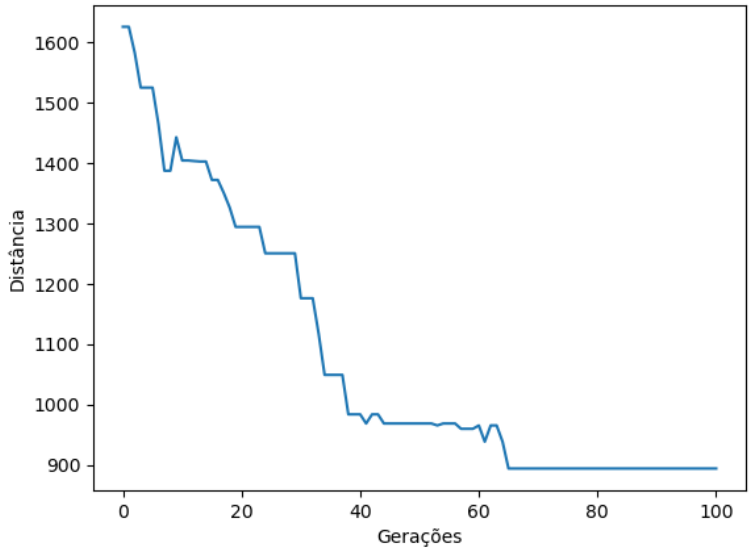

Figura 33 - Valor da função objetivo ao longo de 100 gerações, usando taxa de cruzamento de 20% e taxa de mutação de 0,1%.

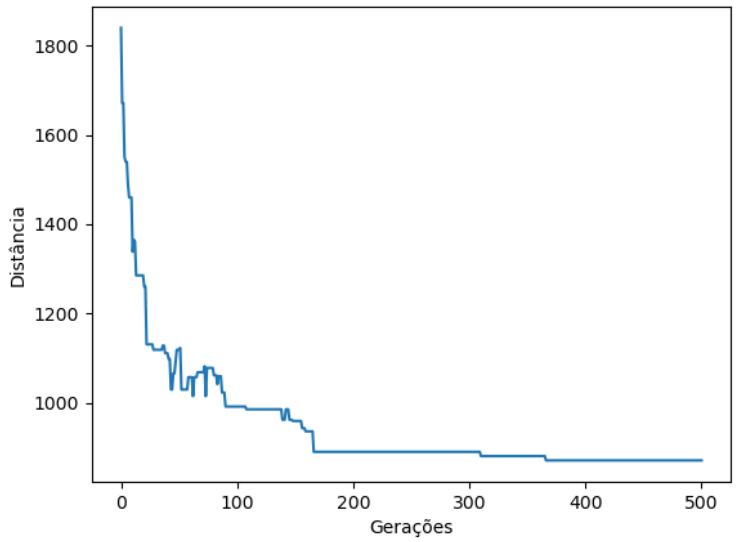

Figura 34 - Valor da função objetivo ao longo de 500 gerações, taxa de cruzamento de 20% e taxa de mutação de 0,1%.

Contudo, devido à natureza estocástica do AG, a solução pode apresentar um ou mais trajetos equivocados entre duas cidades, mas mesmo assim apresentando boa solução no tour entre todas as cidades. Existem diversas maneiras esses trajetos ruins entre dois pontos (cidades) podem ser evitados. Uma maneira de resolução para o TSP com 20 cidades seria a utilização de 500 gerações para garantir a qualidade da solução obtida, a qual pode ser visualizada na Figura 35.

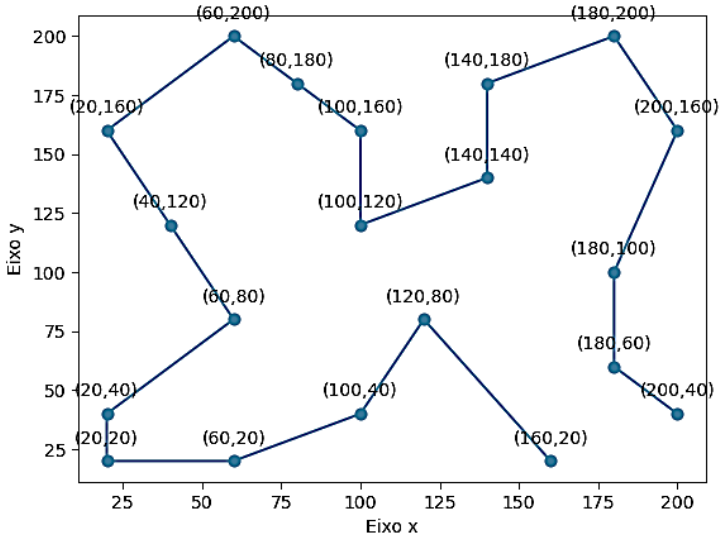

Figura 35 - Solução encontrada para o TSP com 500 gerações, taxa de cruzamento de 20% e taxa de mutação de 0.1%.

REFERÊNCIAS

BARBOSA, H. J. and LEMONGE, A. C., **"An adaptive penalty method for genetic algorithms in constrained optimization problems"**, Frontiers in Evolutionary Robotics, v. 34, 2008.

GOLDBARG, M. C. e LUNA, H. P. L. **Otimização Combinatória e Programação Linear**. Rio de Janeiro, RJ, Brazil: Campus, 2005.

GOLDBERG, D. E. **Genetic Algorithms in Search, Optimization and Machine Learning**. Addison-Wesley, 1989.

HOLLAND, J. H. **Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence**. University of Michigan Press, 1975.

KUBAT, M. **An Introduction to Machine Learning**. Springer International Publishing, 2015.

MICHALEWICZ, Z. **Genetic algorithms + data structures = evolution programs**. Springer-Verlag, 3rd Ed., 1996.

MICHALEWICZ, Z. and SCHOENAUER, M., **"Evolutionary algorithms for constrained parameter optimization problems"**, Evolutionary computation, v. 4, n. 1, pp. 1–32, 1996.

SIVANANDAM, S. N. and DEEPA, S. N. **Introduction to Genetic Algorithms**. New York, USA: Springer, 2008.

YU, X. and GEN, M. **Introduction to Evolutionary Algorithms**. Springer London, 2010.

7. COLÔNIA DE FORMIGAS

Vimos no Capítulo 4 sobre como o processo de simulação do recozimento de materiais pode ser adaptado para se tornar um método de otimização. Outrossim, no Capítulo 6, foi também apresentado um otimizador que imita a evolução de estruturas biológicas. Neste capítulo, apresentamos a um algoritmo da área de computação natural. A computação natural é uma área de pesquisa que investiga processos que ocorrem na natureza para o desenvolvimento de novos algoritmos otimizadores que sejam capazes de serem aplicados em problemas reais.

Na natureza, há um fenômeno chamado inteligência de enxames, ou inteligência coletiva, que diz respeito a indivíduos que apresentam um comportamento de inteligência capaz de resolver problemas complexos, que se manifesta através do comportamento social (FACELI et al., 2011).

Estes agentes são capazes de interagir entre eles e o ambiente em que estão inseridos. Ao observar a natureza, os enxames de insetos são capazes de alcançar comportamentos altamente complexos, partindo-se de indivíduos limitados. Estes indivíduos trabalham de forma auto-organizada, ou seja, não existe um sistema global de controle. As tarefas são alcançadas a partir da troca de informação entre os indivíduos que seguem regras simplificadas. Entre as tarefas que estes enxames podem executar estão a construção de estruturas complexas para viver, prevenção contra predadores e exploração do ambiente para encontrar comida, por exemplo.

De acordo com Millonas (1994) *apud* (FACELI et al., 2011), os sistemas baseados em inteligência coletiva seguem cinco princípios:

1. *Proximidade*: indivíduos de uma população devem interagir entre si;
2. *Qualidade*: indivíduos devem ser capazes de analisar a interação entre si e com o ambiente;
3. *Diversidade*: capacidade de um sistema reagir a ações inesperadas;
4. *Estabilidade*: os indivíduos não podem modificar seu comportamento em resposta a qualquer modificação no ambiente;
5. *Adaptabilidade*: os indivíduos devem ser capazes de se adaptar às mudanças do ambiente e da população.

7.1. FUNDAMENTOS DA OTIMIZAÇÃO POR COLÔNIA DE FORMIGAS

O comportamento social complexo demonstrado por colônias de insetos, tais como formigas, abelhas e cupins, intrigaram os humanos a ponto de ser estudado por estes. As formigas reais usam um tipo específico de comunicação indireta entre indivíduos chamado comunicação por feromônios. A partir destes estudos, surgiram os primeiros algoritmos modelando o complexo comportamento de busca por alimentos. Tal atividade complexa emerge do comportamento

coletivo de muitos indivíduos de baixa complexidade. No contexto de comportamento coletivo, insetos sociais são basicamente compostos de agentes do tipo estímulo-resposta e, portanto, com base nas informações capturadas (ou percebidas) do ambiente local, um indivíduo executa uma ação básica e simples com base na informação percebida (ENGELBRECHT, 2007). Estas ações simples, quando combinadas em escala global, permitem alcançar uma ação mais complexa.

No processo de procura por alimentos, as formigas mostram um comportamento interessante: encontram o caminho mais curto entre a fonte de alimento descoberta e a entrada do ninho. Portanto, vale indagar como é que elas alcançam tal feito na ausência de um mecanismo de visão central? De acordo com Gambardella, Taillard e Dorigo (1997) no início da procura por alimento, as formigas se espalham de maneira aleatória ou caótica. No entanto, assim que uma fonte de comida é localizada, este padrão de comportamento se modifica para uma forma mais organizada, onde um número cada vez maior de formigas passa a seguir o mesmo caminho até a fonte de alimento. Conforme elucidado por Engelbrecht (2007), este comportamento emergente é um resultado de um mecanismo de recrutamento onde as formigas que localizaram a fonte de alimento influenciam outras formigas na mesma direção desta fonte. A Figura 36 ilustra este comportamento.

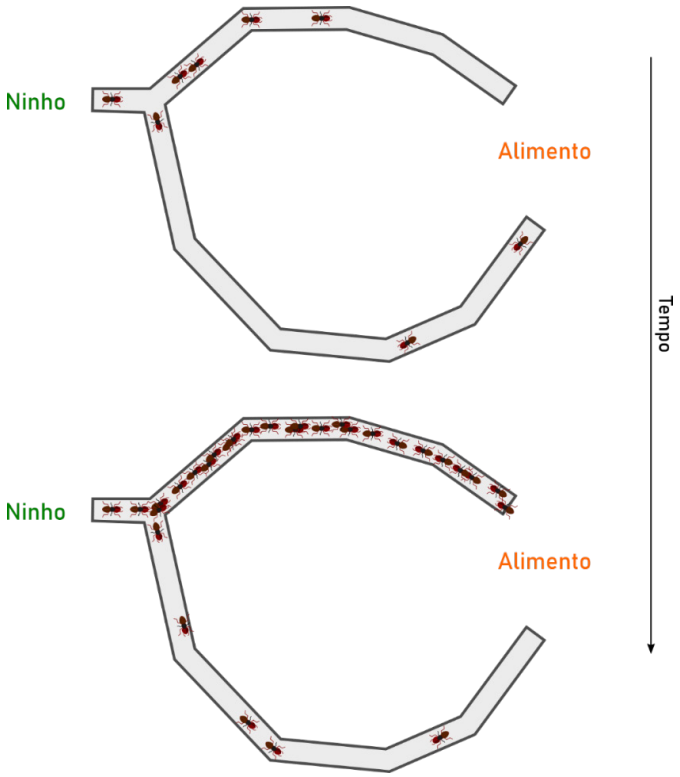

Figura 36 - Ilustração do comportamento das formigas.

A forma de realizar o recrutamento pode ser feita de contato direto ou comunicação indireta, onde esta última ocorre através de trilhas de feromônio. Quando uma formiga localiza uma fonte de alimento, ela carrega um item deste alimento para o ninho e vai depositando feromônio no caminho. As formigas então decidem que caminho seguir, levando em conta as concentrações de feromônio encontradas nos diferentes caminhos existentes. Caminhos com mais feromônio possuem maiores chances de serem selecionados. Com mais formigas trilhando certo caminho, depósitos adicionais de feromônio são efetuados por elas, o que atrai mais formigas a seguirem este caminho.

De acordo com Dorigo e Stützle (2004) este comportamento de trilha de feromônio foi investigado por diversos pesquisadores. Entre as pesquisas mais interessantes, destaca-se a realizada por Gross *et al*. (1989) citado por Dorigo e Stützle (2004), que utilizaram uma ponte dupla conectando um ninho de formigas de uma espécie Argentina *I. humilis* e uma fonte de alimento. Em um primeiro experimento, as pontes criadas eram de tamanhos iguais. O que ocorreu foi que, no princípio, as formigas escolhiam os caminhos sem demonstrar qualquer preferência (aleatório). Após algum tempo, as formigas optaram veementemente por apenas uma ponte. No início, não havia feromônio e, portanto, a escolha era aleatória. Devido à flutuação aleatória, eventualmente, um pouco mais de formigas tomou uma ponte específica. Devido a uma destas pontes possuir mais feromônio do que a outra, as formigas posteriores decidem então pela que possui mais feromônio.

No segundo experimento, uma das duas pontes tinha o dobro de tamanho da outra. Após algumas experimentações, a maioria delas mostrou que as formigas optavam pelo caminho mais curto. Assim como no primeiro experimento, a princípio as formigas escolhiam as pontes de forma aleatória. Porém, as formigas que tomaram o caminho mais curto foram (obviamente) as primeiras a chegar à fonte de alimento e consequentemente as primeiras a espalharem feromônios pelo caminho mais curto de volta ao ninho. No caminho entre o ninho e o alimento agora passa a ser escolhida a ponte de menor tamanho, pois apresentará um rastro de feromônio mais forte, o que explica o efeito explicado na Figura 36.

A *Ant Colony Optimization* (ACO) ou Otimização por Colônia de Formigas descrita em Dorigo, Caro e Gambardella (1999) explora este mecanismo em uma forma estruturada para lidar com problemas de otimização. Em um problema de otimização que faça uso de grafos, por exemplo, e tem sua estrutura de distância tais como o TSP, a utilização do ACO parece ser diretamente adequada. Assim, o ACO funciona como um conjunto de agentes computacionais concorrentes e assíncronos que se movimentam entre os estados do problema, que correspondem às soluções parciais no espaço de busca (FACELLI *et al.*, 2011).

De maneira semelhante ao que foi exemplificado na Figura 36, cada formiga constrói iterativamente uma solução para o problema se baseando em duas informações (ou parâmetros) conhecidas como trilha e atratividade. Todavia, conforme explica Dorigo e Stützle (2004), se apenas este mecanismo for utilizado, por exemplo, em um problema de encontrar o menor caminho em um grafo, é possível que a formiga acabe presa em loop. Portanto, as seguintes considerações devem ser feitas (DORIGO e STÜTZLE, 2004):

1) **Tomada de decisão probabilística e construção da solução**. As formigas podem trabalhar de dois modos: *forward* e *backward*. Elas estão em modo *forward* quando estão se movimentando do ninho em direção à comida e em modo *backward*, caso contrário. Uma vez que uma formiga alcança seu destino, ela troca o modo de trabalho. Assim, formigas no modo *forward* constroem uma solução através de escolhas probabilísticas para o próximo nó a se mover entre aqueles na vizinhança do nó do grafo em que estão localizadas. A escolha probabilística é enviesada pelas trilhas de feromônios depositavas previamente no grafo por outras formigas. Formigas caminhando em modo *forward* não depositam feromônios enquanto se movem;

2) **Atualização de feromônio**: O uso de uma memória explícita permite a uma formiga refazer o caminho que ela seguiu enquanto buscava o nó destino. Além disso, as formigas melhoram o desempenho do sistema por implementar a eliminação de laço de repetição. Na prática,

antes de iniciar o retorno, as formigas eliminam os laços de repetição do caminho. Quando movimentam de volta para o ninho, as formigas deixam feromônios nos arcos usados na travessia;

3) **Atualização de feromônios baseada na qualidade da solução encontrada**. As formigas memorizam os nós que visitam durante o caminho do ninho até o alimento (nó destino, neste caso), assim como os custos dos arcos das soluções que eles geram. Assim, as formigas usam esta informação para saber como modular a quantidade de feromônio que depositam enquanto estão retornando para o ninho. Tornar a função de atualização de feromônio dependente da qualidade da solução gerada pode ajudar a direcionar futuras formigas a tomar o caminho que leva a melhores soluções;

4) **Evaporação de feromônio**. Em colônias reais, a intensidade de feromônio diminui conforme o tempo passa. Na ACO a evaporação é simulada por aplicar uma regra de evaporação propriamente escolhida. Pode, por exemplo, diminuir de forma constante. A evaporação reduz a influência de feromônios depositados em estágios mais antigos da busca, onde formigas artificiais podem ter construído soluções de baixa qualidade. Desta forma, este mecanismo ajuda a escapar de ótimos locais.

7.2. ACO APLICADO AO TSP

Em Lopes, Rodrigues e Steiner (2013), os autores apresentam um algoritmo genérico para a representação da ACO, o qual é exibido na Figura 37.

```
1: Inicialize os parâmetros;
2: Inicialize as trilhas de feromônio;
3: while Não alcançar a condição de parada do
4:     Construa soluções com as formigas;
5:     Aplique busca local (opcional)
6:     Atualize os Feromônios;
7: end while
```

Figura 37 - Visão geral da ACO de acordo com Lopes, Rodrigues e Steiner (2013).

Pelo exposto no início deste capítulo, percebe-se que a ACO se utiliza de um mecanismo que mescla as decisões probabilísticas com um sistema guloso de decisão. Uma função de decisão gulosa define através de heurísticas a importância de uma determinada parte a ser incorporada na construção da solução. Como ela decide através de heurísticas, pode-se então afirmar que é orientada ao problema em questão. No caso do TSP, a ordem da incorporação das cidades na solução não é relevante, pois se espera ao final apenas uma sequência de visitações. No caso do ACO, as formigas irão basear suas decisões em uma função probabilística que combina duas informações: o valor da função gulosa e o valor de feromônio acumulado no trecho da decisão, no momento da decisão (note que aqui, o tempo também é importante, por conta da evaporação do feromônio).

Conforme é afirmado em Lopes, Rodrigues e Steiner (2013), o valor da função heurística para o problema do TSP pode ser dado por:

$$\eta_{ij} = 1/d_{ij}$$

No TSP, o grafo utilizado pelas formigas para construção da solução é o mesmo que representa a instância do problema. A cada vez que uma formiga for se mover do vértice que ela se encontra no momento do tempo, ela se move para uma cidade vizinha que não esteja no tour de sua memória. Quando não restar mais cidades a serem visitadas, a formiga então inclui a aresta de retorno para sua cidade de origem. Para este problema, define-se então a variável τ_{ij} como sendo a quantidade de feromônio depositada no arco que liga a origem *i* ao destino *j*. Portanto, a informação combinada a ser avaliada pela formiga é dada por:

$$\tau_{ij}^{\alpha} \times \eta_{ij}^{\beta}$$

onde α e β são parâmetros da ACO e permitem regular a importância dada tanto ao feromônio quanto ao peso da aresta (*i,j*). Assim, sendo o número de formigas artificiais igual ao número de cidades da instância do problema, coloca-se cada formiga em uma cidade distinta. Sendo o número de formigas maior que o número de cidades, é interessante garantir que cada cidade terá pelo menos uma formiga posicionada (LOPES, RODRIGUES, STEINER, 2013).

Assim, de maneira paralela, cada formiga inicia sua jornada de construção da solução (tour). A cada passo, estando na cidade *i*, a formiga *k* escolhe qual cidade *j* irá compor o próximo passo da solução utilizando-se a seguinte probabilidade:

$$p_{ij}^{k} = \begin{cases} \dfrac{\tau_{ij}^{\alpha} \times \eta_{ij}^{\beta}}{\sum_{h \in N(i,s_k)} \tau_{ih}^{\alpha} \times \eta_{ih}^{\beta}} & se\ j \in N(i,s_k) \\ 0 & se\ j \notin N(i,s_k) \end{cases}$$

onde $N(i,s_k)$ é composto pelas cidades permitidas de serem selecionadas a partir do nó *i*, levando-se em consideração o tour já construído até o momento (s_k).

Ainda de acordo com Lopes, Rodrigues e Steiner (2013), os parâmetros de um algoritmo de ACO incluem:

1. ρ: taxa de evaporação do feromônio;
2. *MAX_IT*: número máximo de iterações do algoritmo;
3. τ_0: quantidade de feromônio a ser atribuída para as arestas na inicialização do algoritmo;
4. *m*: número de formigas artificias utilizadas pelo algoritmo.

Ao observar o algoritmo geral estipulado na Figura 37, é notório que os itens essenciais do algoritmo principal, são dois:

1. Construa soluções com as formigas;
2. Atualize os feromônios;

O procedimento *1. Construa as soluções com as formigas* é proposto na Figura 38.

1: $s_k[j] \leftarrow 0$ para $k = 1, \ldots, m$
2: **for** $k = 1$ até m **do**
3: **while** $|s_k| \neq |G|$ **do**
4: Selecione a cidade j com probabilidade p_{ij}^k;
5: $s_k \leftarrow s_k \cup \{j\}$
6: $i \leftarrow j$ (Mover a formiga para a cidade j)
7: **end while**
8: Mova a formiga k para a cidade de origem
9: $L_k \leftarrow f(s_k)$ (Cálculo do custo da solução s_k construída pela formiga k)
10: **end for**

Figura 38 - Procedimento para gerar as soluções construtivas adaptado de Lopes, Rodrigues e Steiner (2013).

Uma vez que as formigas finalizam uma iteração do procedimento exposto na Figura 38, uma iteração da atualização da matriz de feromônios deve ocorrer. Este processo simula a formiga viajar de volta da última cidade (alimento) até o ninho (sua cidade de origem). Esta atualização é feita utilizando-se a seguinte equação:

$$\tau_{ij} = (1 - \rho)\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k}, \forall i, j \in G$$

onde

$$\Delta\tau_{ij}^{k} = \begin{cases} 1/L_k & se\ (i,j) \in s_k \\ 0 & caso\ contrário \end{cases}$$

onde L_k é o custo da solução calculado na linha 9 do algoritmo exibido na Figura 38, que está associado à função objetivo do TSP.

Conforme elenca (LOPES, RODRIGUES, STEINER, 2013), os critérios de parada do algoritmo podem ser:

1. Atingir um número máximo de iterações (variável MAXIT);
2. "Estagnação": este fenômeno está relacionado com a convergência da maioria das formigas para uma mesma solução. Neste ponto, as atualizações não farão mais modificações significativas para que outros espaços possam ser explorados.

7.3. ACO APLICADO AO SAT

Aplicaremos o ACO na variação do SAT chamada de MAX-SAT. Apenas para lembrar, esta variação do SAT possui uma função objetivo que tenta maximizar o número de cláusulas que se tornam VERDADEIRO. Considerando que a instância a ser resolvida possua *n* variáveis, então toda solução para este problema consiste em uma cadeia $X = \{x_1,x_2,\ldots,x_n\}$, tal que $x_i \in \{TRUE, FALSE\}$. A função objetivo pode ser a simples minimização da contagem do número de cláusulas que se tornam verdadeiras, como também pode ser a soma dos pesos das cláusulas não satisfeitas (para o caso de se atribuir pesos às cláusulas).

Para cada par de atribuição $<x_i,v_i>$, sendo x_i a variável correspondente à posição i e $v_i \in \{TRUE, FALSE\}$, o valor atribuído a ela no grafo de exploração das soluções, cada um dos dois estados da variável é armazenado em uma matriz de feromônio $\tau_{(n\times2)}$. Isto significa que uma entrada $<x_i,v_i>$ possui a informação do desejo de ser inclusa na solução sendo construída. Os feromônios são atualizados no fim de cada iteração de maneira que preserve a informação sobre soluções de mais alta qualidade encontradas nas iterações anteriores. Esta atualização passa por duas fases (ALBA e NAKIB, 2013):

1. Aplicação do coeficiente de evaporação para cada entrada da matriz τ

$$\tau \leftarrow \tau \times (1-\rho)$$

2. Atualização da entrada $\tau_{<xi,vi>}$ da matriz de feromônio com relação à uma solução *T*

$$\tau_{<x_i,v_i>} \leftarrow \tau_{<x_i,v_i>} + \frac{1}{f(T)} + \frac{1}{f(T_{best})}$$

sendo a função custo utilizada.

Assim como no caso do TSP, a decisão probabilística de que caminho tomar por cada formiga se baseia também em duas informações: o feromônio, que acabamos de exibir, e uma informação heurística. Para o problema do MAX-SAT, usaremos uma matriz $\eta_{<xi,vi>}$. Seja $f(T \cup < x_i, v_i >)$ o custo da solução T ao adicionar a atribuição $<x_i,v_i>$. A informação heurística pode então ser definida como:

$$\eta_{<x_i,v_i>} = f(T \cup < x_i, v_i >) - \mathrm{f(T)}$$

Assim, a probabilidade da formiga escolher a atribuição $<x_i,TRUE>$ é dada por:

$$P_{<x_i,TRUE>} = \frac{\tau^{\alpha}_{<x_i,TRUE>} \times \eta^{\beta}_{<x_i,TRUE>}}{\sum_{v_k \in \{TRUE,FALSE\}} \tau^{\alpha}_{<x_i,V_k>} \times \eta^{\beta}_{<x_i,v_k>}}$$

onde α e β são os parâmetros de controle para balancear a importância entre o feromônio e o valor da heurística, respectivamente.

1: Initizalize a matriz τ com o valor padrão de feromônios τ_0
2: $T_{best} \leftarrow$ solução vazia
3: $ITER \leftarrow 0$
4: **while** $ITER < MAX\_TRIALS$ **do**
5: $T \leftarrow$ solução vazia
6: **while** Não convergiu **do**
7: **for** $k = 1 \ldots m$ **do**
8: Escolha uma variável x_i aleatoriamente
9: Escolha a atribuição v_i
10: Calcule o custo de incluir $< x_i, v_i >$
11: **end for**
12: $BestAnt < e_i, v_{e_i} > \leftarrow$ formiga com o melhor custo benefício
13: $T \leftarrow T \quad \cup < e_i, v_{e_i} >$
14: **end while**
15: $T_{best} \leftarrow min(T_{best}, T)$
16: Atualize τ
17: $ITER \leftarrow ITER + 1$
18: **end while**

Figura 39 - Algoritmo ACO para o problema MAX-SAT adaptado de (ALBA, NAKIB, 2013).

O algoritmo proposto por Alba e Nakib (2013) é apresentado na Figura 39. Ele inicia de uma solução vazia e as formigas vão construindo a solução colaborativamente em cada iteração. Note, porém, que a escolha do valor v_i é feita utilizando a probabilidade discutida anteriormente. Ao final, o algoritmo retorna a melhor solução encontrada dentro de MAX-TRIALS.

REFERÊNCIAS

ALBA, Enrique; NAKIB, Amir. **Metaheuristics for dynamic optimization**. Berlin: Springer, 2013.

DORIGO, Marco; CARO, Gianni Di; GAMBARDELLA, Luca M. Ant algorithms for discrete optimization. **Artificial life**, v. 5, n. 2, p. 137-172, 1999.

DORIGO, Marco; STÜTZLE, Thomas: **Ant Colony Optimization.** MIT Press, Cambridge, 2004.

ENGELBRECHT, Andries P. **Computational intelligence: an introduction**. John Wiley & Sons, 2007.

FACELI, Katti et al. Inteligência Artificial: Uma abordagem de aprendizado de máquina. 2011.

GAMBARDELLA, L. M.; TAILLARD, E.; DORIGO, M. **Ant colonies for the QAP**. Technical Report 4-97, IDSIA, Lugano, Switzerland, 1997.

GRASSÉ, Plerre-P. La reconstruction du nid et les coordinations interindividuelles chez Bellicositermes natalensis et Cubitermes sp. la théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs. **Insectes sociaux**, v. 6, n. 1, p. 41-80, 1959.

GOSS, Simon et al. Self-organized shortcuts in the Argentine ant. **Naturwissenschaften**, v. 76, n. 12, p. 579-581, 1989.

LOPES, Heitor Silvério; RODRIGUES, L. C.; STEINER, Maria Teresinha Arns. Meta-heurısticas em pesquisa operacional. **Omnipax, Curitiba, PR**, v. 1, 2013.

MILLONAS, M. M. (1994). Swarms, phase transitions, and collective intelligence. In C.G. Langton (Ed.), Artificial Life III, pp. 417-445. Reading, MA: Addison-Wesley.

8. GRASP

A metaheurística GRASP (*Greedy Randomized Adaptative Search Procedures*), foi desenvolvida por Feo e Resende (1995) como uma tentativa de obter bons resultados para problemas difíceis de otimização combinatória. Como apresentado em Feo e Resende (1995), atualmente esta metaheurística de construção e melhoria possui um grande destaque na literatura devido aos bons resultados obtidos nas aplicações em problemas de otimização combinatória, pela facilidade de paralelização e principalmente por sua flexibilidade (SIQUEIRA *et. al*., 2016).

O GRASP é basicamente um procedimento iterativo, onde cada iteração consiste de duas fases: uma fase de construção de uma solução inicial de forma gulosa, aleatória e adaptativa; e uma fase de busca local, que objetiva melhorar a solução obtida, explorando sua vizinhança.

8.1. FUNDAMENTOS DO GRASP

GRASP (*Greedy Randomized Adaptive Search Procedure* – Procedimento de Busca Gulosa Adaptativa Aleatorizada) é uma metaheurística multipartida, proposta por Feo e Resende (1995), em que cada iteração é composta de uma fase de construção e de uma fase de aprimoramento. Durante o processo, a solução incumbente (melhor solução até o momento) é armazenada e atualizada sempre que a fase de aprimoramento resulta em uma solução melhor. Ao final de um número estabelecido de iterações, o algoritmo retorna a melhor solução encontrada.

A fase de construção gera uma solução viável para o problema através de um procedimento parcialmente guloso e parcialmente aleatório. A cada etapa da construção a seleção dos elementos que vão compor a solução é feita aleatoriamente em um subconjunto dos melhores elementos candidatos, denominada Lista Restrita de Candidatos (LRC). A fase de aprimoramento é uma busca local.

A Figura 40 apresenta o pseudo-código de uma versão tradicional do GRASP pra um problema de minimização (FEO e RESENDE, 1995). Neste algoritmo, o critério de parada adotado é ativado quando se atinge um determinado número de iterações (*MaxIter*) sem melhora da melhor solução que é definida pelo usuário (linha 2). A função *f*(S) retorna o custo da solução *S* e *S** é a solução incumbente. Este algoritmo tem duas fases principais, onde na primeira, executa-se a fase de construção (linha 3), que procura construir uma solução viável e de qualidade. Logo em seguida, na segunda fase, é executada a fase de busca local (linha 4), procurando refinar a solução inicial. Após a fase de construção e o refinamento da solução (busca local), é verificado se a solução encontrada é a melhor até o momento (linha 5). Em caso positivo, atualiza-se a melhor solução encontrada (linha 6), e finalmente a melhor solução encontrada também é atualizada (linha 7).

```
1: f(S*) ← +∞;
2: for i = 1...MaxIter do
3:    Aplicar o procedimento de construção para obter uma solução viável S;
4:    Aplicar busca local em S gerando uma nova solução S';
5:    if f(S') < f(S*) then
6:       S* ←S';
7:       f(S*) ←f(S');
8:    end if
9: end for
```

Figura 40 - Pseudo-código do algoritmo GRASP tradicional.

No pseudo-código da Figura 40, cada iteração é um procedimento independente, não havendo nenhum tipo de memória entre as iterações no decorrer da execução. Já no Capítulo 3 são apresentados diversos métodos construtivos e de busca local aplicados ao TSP e ao SAT.

Porém, muitas técnicas podem ser agregadas ao GRASP tradicional (hibridismo), no intuito de aproveitar de alguma forma os resultados de iterações anteriores para tentar encontrar melhores soluções. Dentre essas técnicas, pode-se citar reconexão de caminhos (*path-relinking*), métodos de pesquisa em vizinhança variável (VND/VNS) e quaisquer procedimentos de aprimoramento sobre soluções já encontradas (FESTA e RESENDE, 2008). Além disso, outros elementos podem ser incorporados para aprimorar o desempenho do GRASP, como o uso de diferentes algoritmos de construção ou mesmo outras metaheurísticas: busca em vizinhança variada (ou VNS), busca tabu, busca local iterada (ILS), etc.

A seguir, serão descritas como são realizadas as fases de construção e busca local do GRASP.

8.2. FASE DE CONSTRUÇÃO

Na fase de construção, a solução viável é iterativamente construída, elemento por elemento. A heurística é adaptativa porque os benefícios associados com cada elemento são atualizados a cada iteração da fase de construção para refletir as mudanças ocorridas pela seleção de elementos anteriores. A parte aleatória corresponde à forma de escolha dos melhores candidatos da lista. Conforme especificado na Figura 41, cada iteração da fase de construção é composta por três etapas:

- Adaptação ou recálculo da função gulosa para os elementos ainda não pertencentes à solução (linha 3);
- Construção da Lista Restrita de Candidatos (LRC), a qual contém um conjunto reduzido de elementos candidatos a pertencer à solução (linha 4);
- Escolha aleatória de um elemento da LRC (linha 5) e inclusão de elemento na solução (linha 6).

Ressalta-se que o tamanho da LRC é controlado por um parâmetro $\alpha \in [0, 1]$, onde para $\alpha = 0$ tem-se um comportamento puramente guloso do algoritmo e para $\alpha = 1$, um comportamento aleatório. Os elementos (*Elementos*) que farão parte da LRC em *AtualizaLRC*, devem possuir

valores menores ou iguais a *Min(Elementos)* + α_x *(Max(Elementos) - Min(Elementos))*. A componente probabilística do método é devido à escolha aleatória de um elemento da LRC. Este procedimento permite que diferentes soluções de boa qualidade sejam geradas.

1: $S \leftarrow \{\}$;
2: **do**
3: $\quad Elementos \leftarrow FunçãoGulosaAdaptativa()$;
4: $\quad LRC \leftarrow AtualizaLRC(Elementos)$;
5: $\quad S_p \leftarrow SelecionaElemento(LRC)$;
6: $\quad S \leftarrow S \cup \{S_p\}$;
7: **while** S não está viável **or** $Elementos \neq \varnothing$

Figura 41 - Pseudo-código do algoritmo da fase de construção do GRASP.

8.3. FASE DE BUSCA LOCAL

Em Alvarenga e Rocha (2006) é explicado que os métodos de busca local em problemas de otimização constituem uma família de técnicas baseadas na noção de vizinhança, ou seja, são métodos que percorrem o espaço de pesquisa passando, iterativamente, de uma solução para outra que seja sua vizinha.

No GRASP, faz-se interessante aplicar o processo de busca local a cada solução produzida na fase construtiva com intuito de melhorá-la. A fase de busca local aproveita a solução inicial da fase de construção e explora a vizinhança ao redor desta solução. Se um melhoramento é encontrado, a solução corrente é atualizada e novamente a vizinhança ao redor da nova solução é pesquisada. O processo se repete até nenhum melhoramento ser encontrado. É preciso ter cuidado em escolher uma vizinhança apropriada. Também é preciso usar estruturas de dados eficientes para acelerar o processo de busca local. Por fim, é importante ter uma boa solução inicial. A descrição da fase de busca local se encontra na Figura 42.

1: $S' \leftarrow S$;
2: $BL = \{x \in N(S') \mid f(x) < f(S')\}$;
3: **while** |BL| > 0 **do**
4: $\quad$ *Selecione o melhor* $x \in BL$;
5: $\quad S' \leftarrow x$;
6: $\quad BL = \{x \in N(S') \mid f(x) < f(S')\}$;
7: **end while**

Figura 42 - Pseudo-código do algoritmo da fase de busca local do GRASP.

De acordo com o pseudo-código da Figura 42, verifica-se que na linha 1, a solução da fase construtiva (*S*) é a solução inicial da fase de busca local (*S′*). Na linha 2, forma-se o conjunto *BL* de soluções candidatas *x* que pertencem à vizinhança de *S′* (representada por *N*(*S′*)), que possuam melhor valor da função objetivo. Já da linha 3 a 7, tem-se o *loop* principal que é realizado enquanto houver pelo menos uma solução vizinha candidata (|*BL*| > 0). Na linha 4, escolhe-se a melhor solução candidata *x* pertencente à *BL* e na linha 5 atualiza a solução *S′*.

Finalmente na linha 6, busca-se por novas soluções candidatas, conforme feito na linha 2. Na seção 2.5 deste material, encontra-se como determinar a vizinhança de diversos problemas.

8.4. MELHORIAS AO GRASP BÁSICO

Como é possível observar, o GRASP básico é de simples implementação (por exemplo, utilizar os métodos construtivos e de busca local descritos no Capítulo 3) e requer a configuração de apenas dois parâmetros (*MaxIter* e α). Outra característica interessante do GRASP é a facilidade com que se pode ter uma implementação paralela do mesmo (RESENDE, 2001). Neste caso, cada elemento processador pode ser inicializado com sua própria instância do GRASP, dos dados de entrada e uma sequência independente de números aleatórios (com sementes diferentes). Não obstante, cada instância do GRASP pode ter uma combinação de método construtivo e busca local diferentes. Assim sendo, as iterações do GRASP são executadas em paralelo com apenas uma variável global para armazenar a melhor solução encontrada em todos os processadores. Os últimos avanços e extensões aplicados ao GRASP podem ser verificados em Resende e Ribeiro (2019). A seguir serão apresentadas algumas melhorias que podem ser aplicadas ao GRASP.

8.4.1. PATH-RELINKING

Uma das maneiras de se melhorar a qualidade das soluções obtidas pelo GRASP básico é através do uso de *path-relinking* (reconexão por caminhos). A técnica de *path-relinking* foi proposta originalmente em (GLOVER, 1996) como uma estratégia de intensificação, explorando trajetórias que conectavam soluções de elite obtidas por busca tabu ou *scatter search* (GLOVER, 1996; GLOVER, LAGUNA e MARTÍ, 2000). Neste contexto, na busca por melhores soluções são gerados e explorados caminhos no espaço de soluções, partindo de uma ou mais soluções de elite e levando a outras soluções de elite. Isto é alcançado selecionando-se movimentos que introduzem atributos das soluções guia na solução corrente. Esta técnica pode ser vista como uma estratégia que busca incorporar atributos das soluções de boa qualidade, favorecendo a seleção de movimentos que os contenham (FERONE, FESTA e RESENDE, 2016).

De acordo com Aiex (2002) essa abordagem é denominada de *path-relinking* porque quaisquer duas soluções visitadas por um algoritmo de busca estão ligadas por uma sequência de movimentos. Na Figura 44 são mostrados dois caminhos hipotéticos, que ligam uma solução inicial a uma solução guia através de uma sequência de movimentos (para um problema de minimização, sem perda de generalidade). A linha pontilhada mostra soluções visitadas por um algoritmo de acordo com a metaheuristica GRASP e a linha contínua mostra as soluções visitadas pela *path-relinking*. Observa-se que as trajetórias seguidas pelas duas estratégias são diferentes. Isso ocorre principalmente porque em cada iteração do algoritmo GRASP, os movimentos são escolhidos de forma “gulosa”, isto é, escolhe o movimento não proibido que minimize localmente o valor da função objetivo. Durante a *path-relinking*, o objetivo principal é incorporar atributos da solução guia à solução inicial, melhorando assim o valor da função objetivo.

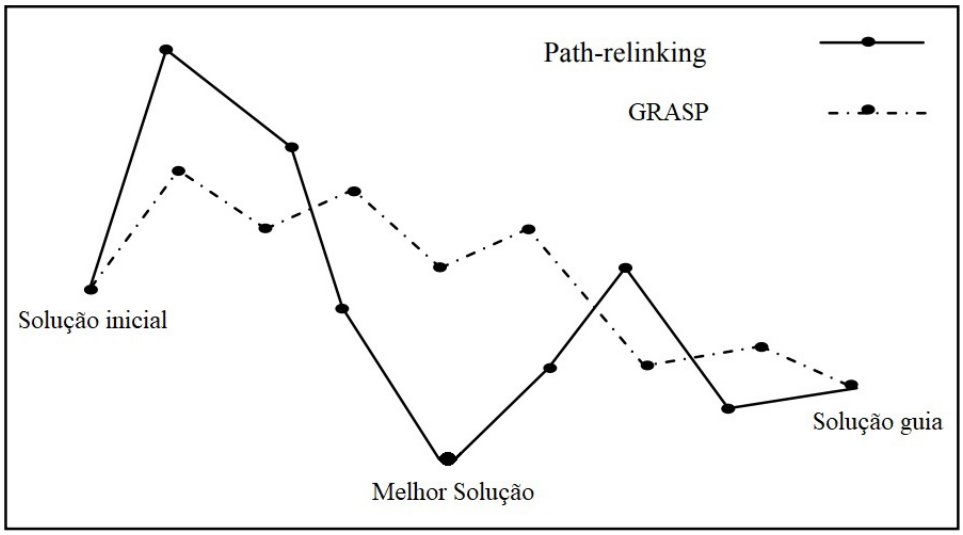

Figura 43 – Caminhos e soluções visitadas por Path-relinking e pela metaheuristica GRASP.

Como apresentado na Figura 43 o objetivo de realizar a *path-relinking* no contexto da metaheurística GRASP é obter soluções não visitadas pela trajetória original, que melhoram a função objetivo, possibilitando encontrar soluções melhores do que o GRASP trabalhando sozinho.

Diversas estratégias de implementação são investigadas com detalhes por Resende e Ribeiro (2005). De acordo com Resende e Ribeiro (2005) as duas estratégias básicas de aplicação de *path-relinking* em procedimentos GRASP são as seguintes:

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

A aplicação da técnica de *path-relinking* como uma estratégia de intensificação a cada ótimo local é mais eficaz do que empregá-la como um procedimento de pós-otimização (RESENDE e RIBEIRO, 2005; ROCHA et. al., 2009). Neste contexto, a *path-relinking* é aplicada a pares (x_1, x_2) de soluções, onde x_1 é uma solução localmente ótima obtida após o uso do procedimento de busca local e x_2 é uma solução selecionada aleatoriamente de um conjunto formado por um número limitado, *MaxElite*, de soluções de elite encontradas ao longo da execução do GRASP. Este conjunto está inicialmente vazio. Cada solução obtida pela busca local é considerada como uma candidata a ser inserida no conjunto de elite, se ela é diferente de todas as outras soluções que estão atualmente neste conjunto. Se o conjunto de elite já possui *MaxElite* soluções e a candidata é melhor que a pior solução existente no conjunto, então a primeira substitui a última. Se o conjunto de elite ainda não está completo, a solução candidata é simplesmente inserida.

De acordo com Rosseti (2003) o algoritmo inicia computando a diferença simétrica $\Delta(x_1, x_2)$ entre x_1 *e* x_2, resultando no conjunto de movimentos que devem ser aplicados a uma delas (a solução inicial identificada como x_1) para alcançar a outra (a solução guia identificada como x_2), indicados a cada passo pelo símbolo $\rightarrow$. A partir da solução inicial, o melhor movimento ainda não executado de $\Delta(x_1, x_2)$ é aplicado à solução corrente, até que a solução guia seja atingida, como demonstrado na Figura 44. A melhor solução encontrada ao longo desta trajetória é con-

siderada como candidata à inserção no conjunto de elite e a melhor solução globalmente já encontrada é atualizada. Diversas alternativas têm sido consideradas e combinadas em implementações recentes (AIEX e RESENDE, 2005):

- Não aplicar *path-relinking* a cada iteração GRASP, mas sim apenas periodicamente;
- Explorar duas trajetórias potencialmente diferentes, usando primeiramente x_1 como solução inicial e posteriormente x_2;
- Explorar apenas uma tr ajetória, usando ou x_1 ou x_2 como solução inicial;
- Não percorrer a trajetória completa de x_1 até x_2, mas sim apenas parte dela (*path-relinking* truncada).

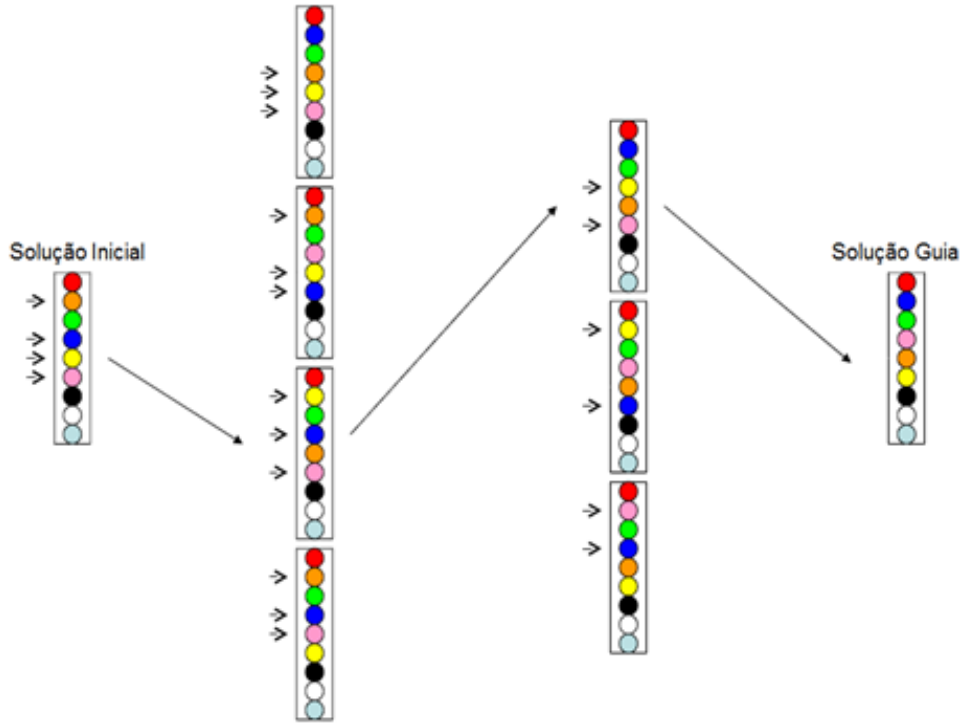

Figura 44 - Exemplo do Path-relinking no contexto da metaheuristica GRASP.

8.4.2. GRASP REATIVO

O parâmetro de controle α da LRC é basicamente o único parâmetro a ser calibrado em uma implementação prática de um GRASP. Em Prais e Ribeiro (2000a), foi discutido o efeito da escolha do valor de α em termos de qualidade e diversidade da solução durante a fase de construção e como isso afeta o resultado do GRASP. Numa abordagem GRASP reativa (DELAMAIRE et. al, 1998), esse α é auto ajustado de acordo com a qualidade das soluções encontradas anteriormente. O pseudocódigo do algoritmo GRASP reativo implementando esse esquema está descrito na Figura 45 e é explicado a seguir.

Em vez de usar um valor fixo para o parâmetro α, que determina quais elementos são colocados na LRC a cada iteração da fase de construção, o procedimento seleciona aleatoriamente esse valor α de um conjunto discreto $A = \{\alpha_1, ..., \alpha_m\}$ contendo *m* valores aceitáveis pre-

determinados (linha 2). O uso de valores diferentes de α em diferentes iterações permite criar LRCs diferentes, possivelmente levando à construção de soluções diferentes que nunca seriam construídas se um único valor fixo de α fosse usado (linha 6). Seja p_i denotar a probabilidade associada à escolha de α_i, para $i = 1,\ldots m$. Inicialmente, $p_i = 1 / m$, $i = 1,\ldots, m$, correspondendo a uma distribuição uniforme (linha 3). Em seguida, essas probabilidades são atualizadas periodicamente usando as informações coletadas durante a pesquisa (linhas 13 a 18). Diferentes estratégias para esta atualização podem ser exploradas.

Em qualquer iteração GRASP, seja A_i o valor médio das soluções obtidas com $\alpha = \alpha_i$ na fase de construção (linha 15). A distribuição de probabilidade é atualizada periodicamente a cada iteração do período de atualização da seguinte maneira. Calcule primeiro $q_i = (1 / Ai)^{\delta}$ para $i = 1,\ldots, m$ linha 16) e atualize os novos valores das probabilidades normalizando o q_i como $p_i = q_i / (\Sigma_j q_j)$ (linha 17). Observe que quanto menor o A_i, maior o p_i correspondente. Consequentemente, no próximo bloco de iterações (*PeriodoAtualizacao*), os valores de α que levam a melhores soluções têm maiores probabilidades de serem selecionados e serão mais frequentemente usados na fase de construção. O expoente δ pode ser usado e explorado para atenuar diferentemente os valores atualizados das probabilidades. Em implementações de referência (PRAIS e RIBEIRO, 2000a) o *PeriodoAtualizacao* = 100 ou 200, δ = 10, m = 10 e A = {0,05; 0,1; 0,15; 0,2; 0,25; 0,3; 0,35; 0,4; 0,45; 0,5}.

Como citado em (PRAIS e RIBEIRO, 2000a) a implementação de um GRASP reativo que, leva em conta o histórico do processo, onde não é necessário ajustar o parâmetro que regula o tamanho da lista de candidatos restritos na fase construtiva, é crucial para o sucesso do processo. Desta forma, o componente reativo do GRASP foi comprovado como sendo muito robusto e provou ser um ativo muito valioso sobre a calibração dispendiosa do parâmetro de qualidade α, fornecendo na média soluções superiores ao GRASP básico em uma série de problemas práticos e teóricos de relevância (PRAIS e RIBEIRO, 2000b).

1: $f(S^*) \leftarrow +\infty$;
2: $A = \{\alpha_1, ..., \alpha_m\}$;
3: $p_i \leftarrow 1/m, i = 1, ..., m$;
4: $sum_i \leftarrow 0, i = 1, ..., m$;
5: **for** i = 1...*MaxIter* **do**
6: *Selecione aleatoriamente* $\alpha = \alpha_i$ de A utilizando as probabilidades $p_i, i = 1, ..., m$;
7: Aplicar o procedimento de construção para obter uma solução viável S;
8: Aplicar busca local em S gerando uma nova solução S';
9: **if** $f(S') < f(S^*)$ **then**
10: $S^* \leftarrow S'$;
11: $MelhorValor \leftarrow f(S')$;
12: **end if**
13: $sum_i \leftarrow sum_i + f(S')$;
14: **if** $mod(i, PeriodoAtualizacao) == 0$ **then**
15: $A_i \leftarrow sum_i / n_i, i = 1, ..., m$;
16: $q_i \leftarrow (f(S^*)/A_i)^{\delta}, i = 1, ..., m$;
17: $p_i \leftarrow q_i / (\sum_{j=1}^{m} q_j), i = 1, ..., m$;
18: **end if**
19: **end for**

Figura 45 - Pseudo-código do algoritmo GRASP Reativo.

REFERÊNCIAS

AIEX, R. M. **Uma investigação experimental da distribuição de probabilidade do tempo de solução em heurísticas GRASP e sua aplicação na análise de implementações paralelas**. Tese de D.Sc., PUC-Rio, Rio de Janeiro, RJ, Brasil, 2002.

AIEX, R. M. and RESENDE, M. G. C. **Parallel strategies for GRASP with path-relinking**. In: Metaheuristics: Progress as Real Problem Solvers, T. Ibaraki, K. Nonobe and M. Yagiura, (Eds.), Springer, pp. 301-331, 2005.

ALVARENGA, F. V. e ROCHA, M. L. **Uma Meta-Heurística GRASP para o problema da árvore geradora de custo mínimo com grupamentos utilizando Grafos Fuzzy**. INFOCOMP Journal of Computer Science, [S.l.], v. 5, n. 1, p. 66-75, mar. 2006. ISSN 1982-3363. Disponível em: <http://www.dcc.ufla.br/infocomp/index.php/INFOCOMP/article/view/124>

DELMAIRE, H., DIAZ, J. A., FERNANDEZ, E. and ORTEGA, M. **Reactive GRASP and tabu search based heuristics for the single source capacitated plant location problem**. INFOR: Information Systems and Operational Research, 37(3):194–225, 1999.

FEO, T. A. and RESENDE, M. G. C. **Greedy randomized adaptive search procedures**. Journal of global optimization 6.2, pp. 109-133, 1995.

FERONE, D., FESTA, P. and RESENDE, M. G. C. **Hybridizations of GRASP with path--relinking for the far from most string problem**. International Transactions in Operational Research, vol. 23, pp. 481-506, 2016

FESTA, P. and RESENDE, M. G. C. **Hybrid GRASP heuristics**. In "Global optimization: Theoretical foundations and applications", A. Abraham, A.-E. Hassanien, and P. Siarry (Eds.), Studies in Computational Intelligence, Springer-Verlag, 2008.

FESTA, P. and RESENDE, M. G. C. **An annotated bibliography of GRASP, Part I: Algorithms**. International Transactions in Operational Research, vol. 16, pp. 1-24, 2009.

GLOVER, F. **Tabu search and adaptive memory programing – Advances, applications and challenges**. In: Barr, R.; Helgason, R.; Kennington, J., editors, Interfaces in Computer Science and Operations Research, p. 1-75. Kluwer, 1996.

GLOVER, F., LAGUNA, M. and MARTÍ, R. **Fundamentals of scatter-search and path-relinking**. Control and Cybernetics, 39:635-684, 2000.

PRAIS, M. and RIBEIRO, C. C. **Reactive GRASP: An application to a matrix decomposition problem in TDMA traffic assignment**. Journal on Computing, 12(3):164–176, 2000a.

PRAIS, M. and RIBEIRO, C. C. **Parameter variation in GRASP procedures**. Investigacion Operativa, 9:1–20, 2000b.

RESENDE, M. G. C. **Greedy randomized adaptive search procedures (GRASP)**. In Encyclopedia of Optimization, C. Floudas and P.M. Pardalos, eds., Kluwer Academic Press, vol. 2, pp. 373-382, 2001.

RESENDE, M. G. C. and RIBEIRO, C. C. **GRASP with path-relinking: Recent advances and applications**. In Metaheuristics: Progress as Real Problem Solvers, T. Ibaraki, K. Nonobe and M. Yagiura, (Eds.), Springer, pp. 29-63, 2005.

RESENDE, M. G. C. and RIBEIRO, C. C. **"Greedy randomized adaptive search procedures: Advances and extensions"**, Handbook of Metaheuristics (M. Gendreau e J.-Y. Potvin, editores), Springer, 2019, 3a edição, 169-220.

ROCHA, M. L., BHAYA, A., MONTENEGRO, F. M. T. e MACULAN, N. **Uma Heurística GRASP com Reconexão por Caminhos para o Problema de Árvore de Steiner Euclidiana Tridimensional**. XLI Simpósio Brasileiro de Pesquisa Operacional, pp. 2205-2217, 2009. Disponível em: http://www.din.uem.br/sbpo/sbpo2009/artigos/56155.pdf

ROSSETI, I. C. M. **Estratégias sequenciais e paralelas de GRASP com reconexão por caminhos para o problema de síntese de redes a 2-caminhos**. Tese de D.Sc., PUC - Rio, Rio de Janeiro, Brasil, 2003.

SIQUEIRA, V. S., SILVA, F. J. E. L., SILVA, E. N., SILVA, R. V. S. and ROCHA, M. L. **Implementation of the Metaheuristic GRASP Applied to the School Bus Routing Problem**. International Journal of e-Education, e-Business, e-Management and e-Learning, v. 6, p. 137-145, 2016. Disponível em: http://www.ijeeee.org/vol6/400-BMG042.pdf

EDUFT

UNIVERSIDADE FEDERAL DO TOCANTINS