CENTRO DE ENSINO UNIFICADO DE TERESINA – CEUT
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

DANN LUCIANO DE MENEZES

ALGORITMOS GENETICOS NA OTIMIZAÇÃO DE CONSULTAS POR SIMILARIDADE

TERESINA
2010

DANN LUCIANO DE MENEZES

ALGORITMOS GENETICOS NA OTIMIZAÇÃO DE CONSULTAS POR SIMILARIDADE

Monografia, apresentada ao Centro de Ensino Unificado de Teresina – CEUT como um dos pré-requisitos para a obtenção do grau de bacharel de Ciências de Computação

Ronildo Pinheiro de Araújo Moura

TERESINA

2010

DANN LUCIANO DE MENEZES

ALGORITMOS GENETICOS NA OTIMIZAÇÃO DE CONSULTAS POR SIMILARIDADE

Monografia apresentada como exigência parcial para a obtenção do grau de Bacharel em Ciências da Computação, na area de concentração....., à banca examinadora do Curso de Ciências da Computação, do Centro de Ensino Unificado de Teresina – CEUT

Aprovada em _____/_____/_____

BANCA EXAMINADORA

_____

Ronildo Pinheiro de Araújo Moura

CEUT

_____

Nome do componente

Instituição

_____

Nome do componente

Instituição

A todos aqueles que passaram e contribuíram em minha vida.

AGRADECIMENTOS

A Deus, inteligência suprema do Universo, por todas as provas impostas em beneficio de minha evolução;

A família Menezes, em especial a Margarida Luciano de Menezes (Avó) (*In Memory*), por todas as lapadas de "rei" por que estava "chafurdando" ou "malinando" em algo, a Iranilde Luciano de Menezes (Mãe), por todo o apoio que só uma mãe pode oferecer, a Larissa Luciano Amorim, por todas lapadas que peguei por "bulinar" com ela;

Aos meu pais e mães de criação, em especial ao Luis Soares Amorim, por todo o "paitrocinio" e fonte de inspiração, ao Tio Carvalho e Tia Silvana, por todos as despesas que gerei ao longo destes dezesseis anos, ao pai Talles Ramyro (Meguxo) por ter me ensinado tantas coisas importantes, principalmente o significado de amizade;

A todas instituições de ensino que me propuseram a base para o conhecimento adquirido para realizar este trabalho, em especial a Faculdade CEUT onde passei os últimos quatros anos de minha vida;

A todo o corpo docente da Faculdade CEUT que implicaram diretamente para meu crescimento social e profissional, em especial ao coordenador do curso Ricardo Sekeff Moura Budaruiche por todos os "É barão, tá pensando o que?", ao querido mestre professor Chiquinho (vulgo Francisco José de Araujo) por todos os "Um nojento desse, e ainda diz pro pai que estuda. Num sabe "porra" nenhuma", a professora Rosianni Cruz eternizada em meu coração por todos "Nananinanão, pode fazer tudo de novo, que não tá bom", ao professor Ricardo Lira por todos "A pergunta que não quer calar! Diga ae senhor dann luciani de

menezes de araujo moura sekeff"; aos meus dois professores de TCC (vulgo Trabalho Chato pra Caramba =P) professora tia Nelza Farias e o professor Ronildo Pinheiro, por toda paciência que me foi fornecida nesta ultima etapa;
A todos os colaboradores que fazem parte da família CEUT em especial a tia Nagislla por todos os favores oferecidos;
A Hayala Glenda (neguinha preta) por toda sua preocupação e dedicação, principalmente por toda sua compreensão a minha ausência nas noites de sábados que passei acordado brigando com o "@#$%ˆ&*" do compilador gcc;
Por ultimo mas não menos importantes a todos os meu amigos que podem ter certeza que de alguma forma contribuirão para eu ser a pessoa que sou hoje, em especial a todos os presentes a lista abaixo, se não citei o nome de todos foi por causa de espaço, mas podem ter certeza que no meu coração espaço não é problema.

Aurea Cronembeger (Aurinha)
Cicero Bruno (Nego Preto)
Ernesto Cid (José)
Francilio Oliveira (Pantico)
Hayala Glenda (Neguinha)
Laise Cronembeger (Laila)
Lucas Novaes (Bolinha)
Maxsuelen Rodrigues (Max Kutruco)
Pablo Michael (Meu Pupilo)
Pamalla Castro ("i", pontinho do i)
Rodrigo Carvalho (Rods)
Thaiara Barroso (Thai)
Vinícios Bezerra (The Lost)
Joicynara (Nadja)

Bruna (Garrafinha)

Bruna Spíndola (Picareta)

Isabela Vaz (Caloura)

"*Muito Tempo Eu Levei Para Entender Que Nada Sei*"
Dias de Luta, Ira!

RESUMO

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce at erat dolor. Phasellus iaculis lorem id sapien vulputate accumsan. Sed vel erat augue. In hac habitasse platea dictumst. Etiam ac mauris nec arcu blandit facilisis. Mauris fermentum arcu nec libero iaculis malesuada. Phasellus sed sapien justo, non lacinia enim. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Duis tincidunt nulla eget magna placerat in facilisis turpis aliquet. Donec diam justo, volutpat faucibus sagittis id, rutrum quis urna. Curabitur quis odio nunc, sed viverra mauris. Pellentesque dictum ullamcorper vestibulum. Cras pulvinar tellus eget tortor dapibus vel porta nunc pulvinar. Sed pulvinar imperdiet sagittis. Maecenas id mi nunc, vitae ullamcorper elit.

PALAVRAS-CHAVE: Consultas por Similaridade, Algoritmos Genéticos, Otimização

ABSTRACT

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce at erat dolor. Phasellus iaculis lorem id sapien vulputate accumsan. Sed vel erat augue. In hac habitasse platea dictumst. Etiam ac mauris nec arcu blandit facilisis. Mauris fermentum arcu nec libero iaculis malesuada. Phasellus sed sapien justo, non lacinia enim. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Duis tincidunt nulla eget magna placerat in facilisis turpis aliquet. Donec diam justo, volutpat faucibus sagittis id, rutrum quis urna. Curabitur quis odio nunc, sed viverra mauris. Pellentesque dictum ullamcorper vestibulum. Cras pulvinar tellus eget tortor dapibus vel porta nunc pulvinar. Sed pulvinar imperdiet sagittis. Maecenas id mi nunc, vitae ullamcorper elit.

KEYWORDS:

LISTA DE FIGURA

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Os primeiros Sistemas Gerenciadores de Base de Dados (SGBDs) tinham a finalidade de facilitar e agilizar o processo de recuperação exata dos dados. Essa recuperação exata foi e continua sendo uma das necessidades das aplicações que os SGBDs devem suportar. Com o passar do tempo novas necessidades foram surgindo, e a recuperação exata dos dados deixa de ser necessária em algumas aplicações especificas como, sistemas de apoio à decisão dentre outras. Segundo Bueno et al (2005) isso se dá porque essas aplicações possuem uma característica exploratória, onde nem sempre a precisão dos dados a serem utilizados é requisitada.

Outra funcionalidade que as aplicações atuais necessitam é o armazenamento e recuperação de tipos de dados complexos, como imagens, musicas, videos, mapas geograficos, cadeias de proteina e dna, etc. Muitas vezes esses dados são agrupados como dados binários (*Blobs*). As operações de consulta pontuais ou exatas em dados complexos demandam em uma grande quantidade recursos computacionais, pois para cada um conjunto de objetos inseridos no banco de dados são feitas muitas verificações *Byte* a *Byte. M*esmo existindo varias estruturas de dados que minimizam a quantidade de verificações ainda sim uma consulta exata leva um tempo e recursos computacionais demasiados.

Para resolver o problema das aplicações onde a recuperação exata dos dados não são necessarias e para comportar o melhor armazenamento dos dados complexos é que foi criada toda a teoria das consultas por similaridade que será apresentada no capitulo 2.

Entretanto para solucionar o problema das grandes quantidades de comparações realizadas durante as consultas por similaridade é discutido no capítulo 3 neste trabalho a teoria da computação bio-inspirada mas especificamente a teoria dos Algoritmos Genéticos na otimização do tempo e dos recursos computacionais.

No capítulo 4 é exemplificado como foi possivel entrelaçar a teoria das consultas por similaridade com a teoria dos Algoritmos Genéticos para podermos representar e resolver o problema dos recursos computacionais.

são comentados do capítulo 5. Deixando no capítulo 6 as conclusões e os trabalhados futuros. Depois destas seções será apresentada todas as refências utilizadas no implementação desta monografia e *software.*

2 CONSULTAS POR SIMILARIDADE

2.1 Considerações Iniciais

Como já mencionado na introdução os primeiros Sistemas Gerenciadores de Base de Dados (SGBDs) tinham a finalidade de facilitar e agilizar o processo de recuperação exata dos dados. Essa recuperação exata foi e continua sendo uma das necessidades das aplicações que os SGBDs devem suportar. Com o passar do tempo, novas necessidades foram surgindo e a recuperação exata dos dados deixou de ser necessária em algumas aplicações especificas, como sistemas de apoio à decisão, dentre outras. Segundo Bueno et al (2005), isso se dá porque essas aplicações possuem uma característica exploratória. Outra funcionalidade que as aplicações atuais necessitam é o armazenamento e recuperação de tipos de dados complexos, como já foi afirmado acima.

As operações de recuperação não são realizadas em cima dos dados complexos propriamente ditos, e sim de características extraídas quando os dados são inseridos ou alterados. Os dados complexos devem passar antes por extratores de características que codificam as características dos dados complexos em um vetor de características.

Três perguntas nos ajudam e nos guiam no entendimento de consultas por similaridade. Elas são nomeadas, neste trabalho, como perguntas guia e são expostas logo abaixo:

1. Como comparar Dados Complexos?
2. Como expressar consultas sobre Dados Complexos?
3. Como indexar Dados Complexos?

A próxima seção expõe os Domínios Métricos que são a resposta à primeira pergunta guia.

2.2 Domínios Métricos

Um Domínio Métrico como, já mostrado na seção anterior, responde a primeira pergunta guia e através dele podemos comparar Dados Complexos. Domínios Métricos são definidos como $M = (D, d( ))$, onde D é o conjunto de todos os objetos que atendem às propriedades do domínio e $d( )$ é uma função de distância, ou métrica, entre esses objetos.

Dados x, y e $z \in D$, uma função de distância (métrica) $d : D \; X \; D \rightarrow R+$ deve satisfazer as seguintes propriedades:

1. Simetria: d(x, y) = d(y, x);
2. Não Negatividade: $0 \leq d(x, y) < \infty$, d(x, x) = 0;
3. Desigualdade Triangular: $d(x, y) \leq d(x, z) + d(z, y)$;

A propriedade de número 1 garante que a distância entre dois pontos tem que ser igual, independente da ordem em que é calculada. Já a segunda propriedade, diz que a distância entre dois pontos deve pertencer ao intervalo de zero até o infinito, e que a distância entre dois pontos somente é zero se esses forem os mesmos pontos. A terceira propriedade merece um maior destaque, visto que a partir dela é possível diminuir os cálculos de distâncias, em busca por similaridade em Domínios Métricos, como afirma Bueno et al (2005).

Para cada tipo de aplicação uma função de distância deve ser definida, fazendo com que um problema seja delimitado por um Domínio Métrico. Na seção logo abaixo será discutido sobre o numero de dimensões em que um Domínio Métrico pode estar definido.

2.2.1 Domínios Métricos e suas Dimensões

Um domínio métrico pode possuir n dimensões,em que cada uma das dimensões representa uma característica do dado complexo. Por exemplo, se quiséssemos armazenar em um domínio métrico uma imagem com apenas duas características, nos precisaríamos de duas dimensões como mostrado na imagem 2.1. As características utilizadas são relativas e podem

variar dependendo do domínio do problema.

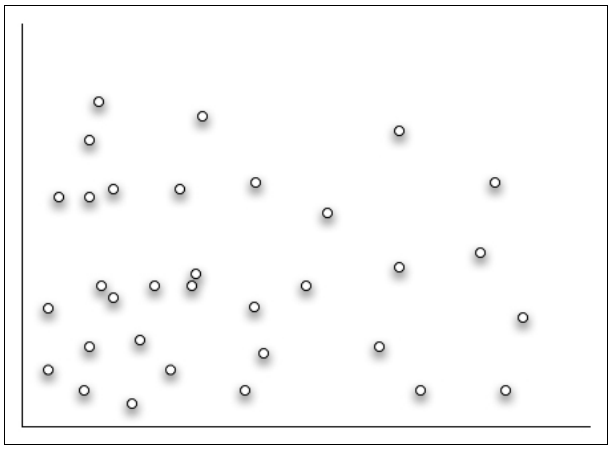


Figura 2.1

A distância entre dois pontos em um domínio métrico, que possui somente suas dimensões mapeadas em pontos, é geralmente calculada pela distância euclidiana, conforme expõe Nadvorny (2005). A relação entre o tamanho de n e a complexidade da função de distância é diretamente proporcional, ou seja, quanto maior for o tamanho de n mais complexa a função de distância se tornará. Como já explicado acima, para cada problema deverá ser criada uma função de distância, que represente a distância entre dois pontos pertencentes a esse domínio. A imagem abaixo representa um domínio métrico com n igual a três dimensões.

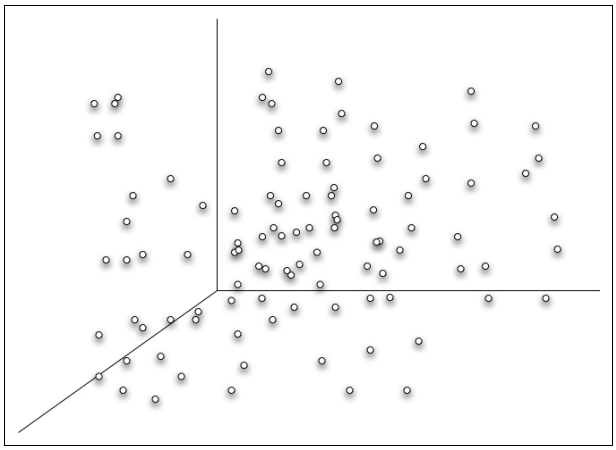


Figura 2.2

O próximo tópico será destinado a responder a segunda pergunta guia.

2.3 Operadores por Similaridade

Os operadores por similaridade, respondendo à segunda pergunta guia, são responsáveis por expressarem consultas nos dados complexos pertencentes a um Domínio Métrico. Existem vários operadores de consulta por similaridade, mas dois se destacam:

- *RQ*(*sq*, *rq*) – Consulta por abrangência ("*Similarity Range Query*"): retorna todos os objetos cujas distâncias até o objeto central da consulta *sq* sejam menores ou iguais ao raio da consulta *rq*;
- *k-NNQ*(*sq*, *k*) – Consulta aos k-vizinhos mais próximos ("*k-Nearest Neighbor Query*"): retorna os k objetos mais próximos do objeto central de consulta *sq*.

Como já explicado, existem outros operadores por similaridade, mas a grande maioria deles derivam dos operadores citados acima. Para melhor compreendermos as consultas por similaridade, precisamos estudar os Métodos de Acesso Métricos, visto que são eles os responsáveis por indexar dados complexos pertencentes a um Domínio Métrico.

2.4 Métodos de Acesso

Os métodos de acesso são responsáveis por indexar os Dados Complexos pertencentes a um determinado Domínio Métrico em uma estrutura de dados. Existem vários métodos de acesso, e eles se classificam conforme descrito logo abaixo:

- Métodos de Acesso Pontual: quando mapeamos o objeto armazenado em pontos do espaço;

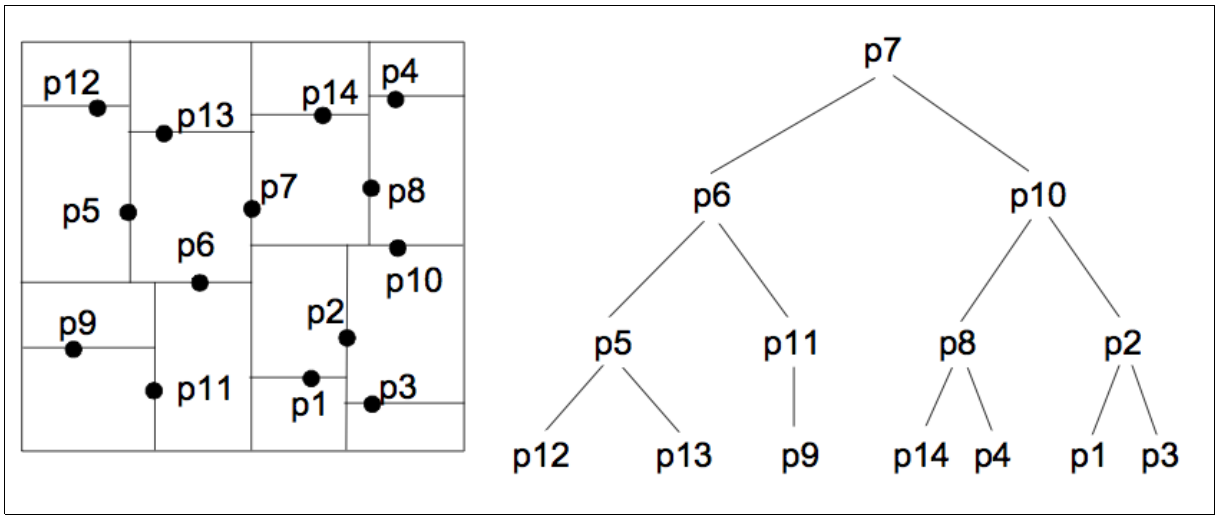


Figura 2.3 (NARDVORNY, 2005 p 16)

- Métodos de Acesso Espacial: nem sempre o mapeamento de objetos em pontos pode ser viável, em dados complexos como dados geográficos que contem polígonos, em geral. Neste caso, os dados indexados são as imagens vetorizadas;

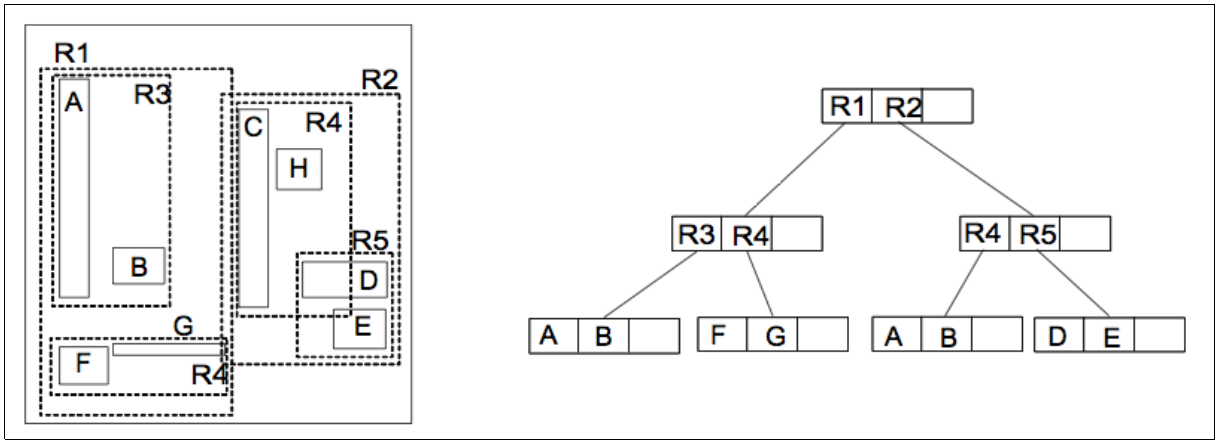


Figura 2.4 (NARDVORNY, 2005 p 18)

- Métodos de Acesso Métrico: os dois métodos anteriores possuem seus objetos mapeados em uma posição absoluta, ou seja, existe um ponto central (normalmente 0) no espaço, onde todos os objetos o utilizam para se referenciar. Então, os métodos de acesso métrico são aplicados nos casos em que não é possível utilizar uma posição absoluta, mas, ainda assim existe uma noção de distância.

2.4.1 Métodos de Acesso Métrico

Conforme expõe Nadvorny (2005), as primeiras estruturas de dados datam de 1973 e são denominadas de árvores Burkhard-Keller. Ao longo do tempo, várias outras árvores foram apresentadas com melhorias significativas comparadas com as primeiras que surgiram. Por exemplo, as árvores VP (*Vantage Point*), GH (*Generalized Hyperplane*). As árvores M, em especial, merece um maior destaque visto que estas possemi características que otimizam as operações não só internas da árvore, mas também operações relacionadas à memória secundária. Outro fator que merece um melhor detalhamento sobre ás arvores M é que elas influenciam as principais estruturas de indexação de dados similares existentes na atualidade.

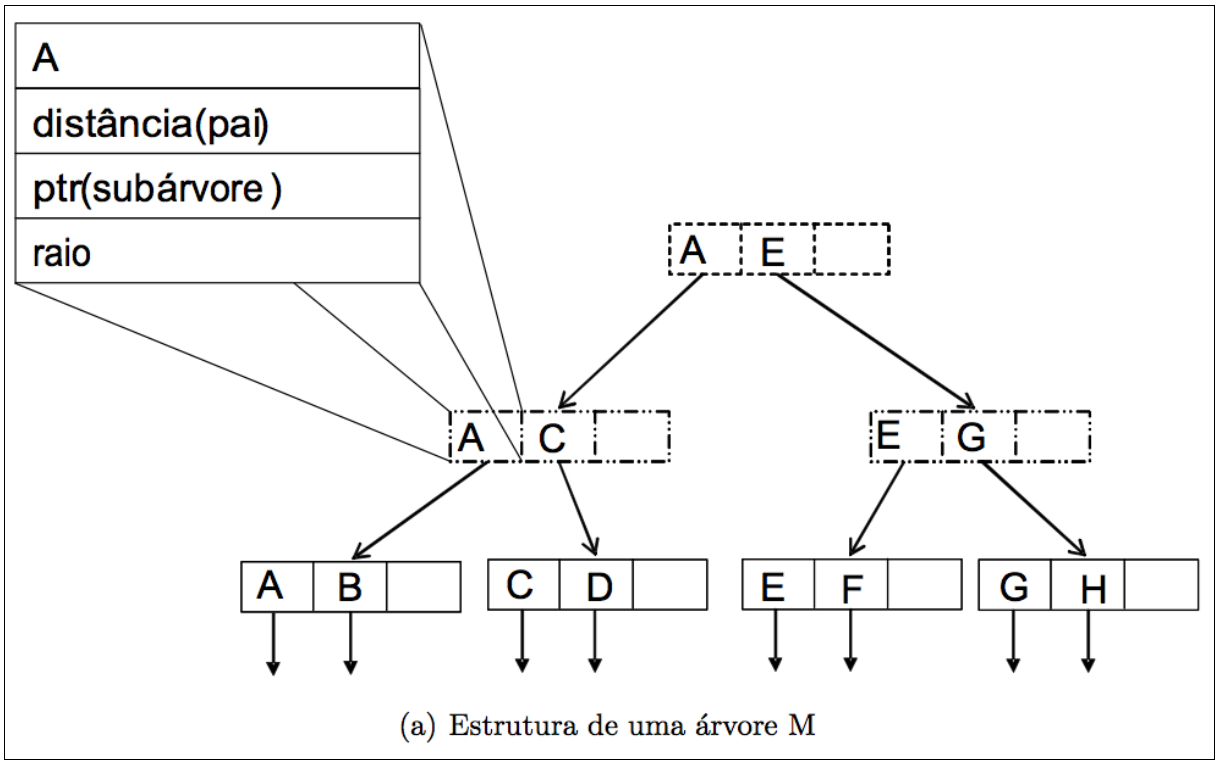
2.4.1.1 *M-tree*

Como já descrito acima, as árvores M (*M-tree*) introduziram características que as colocaram-na a frente de seus antecessores, como relata Nadvorny (2005). A primeira das importantes características é em relação ao **Uso de Memória Secundária**, onde a árvore possui nós com tamanhos fixos, permitindo assim o armazenamento como páginas de discos. A segunda característica presente nas árvores M é chamada de **Dinamismo,** que consiste do fato de operações de inserção ou remoção não serem tão custosas e não implicarem em uma reorganização drástica da árvore.

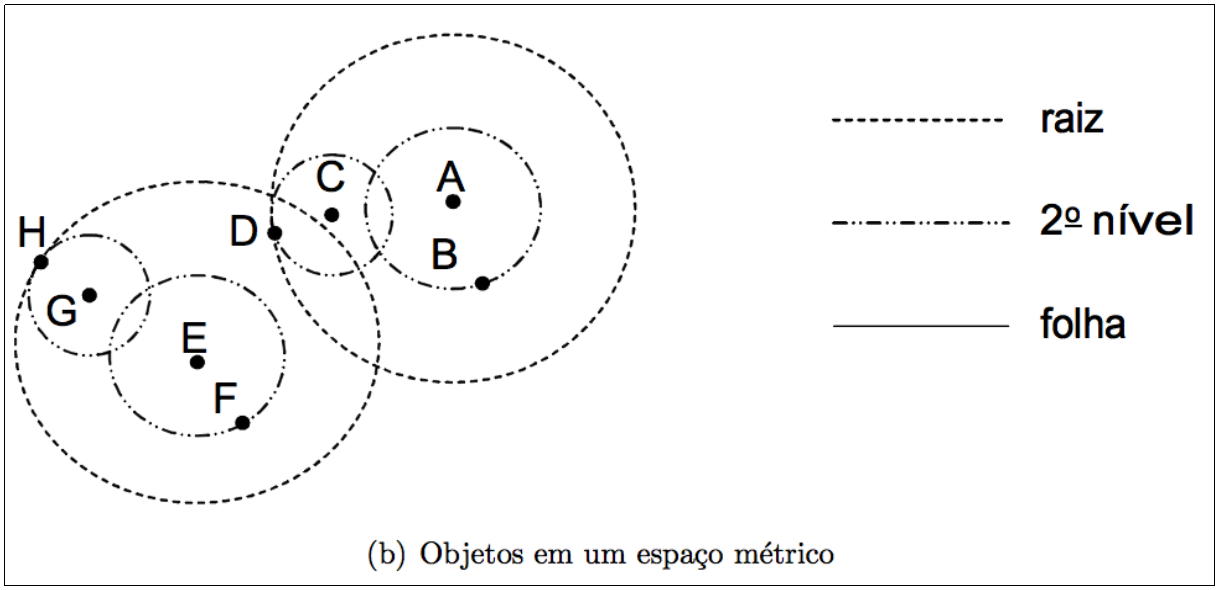
A terceira característica não menos importante diz respeito à **Escalabilidade**, em que o custo da busca não aumenta exponencialmente quando novos objetos são inseridos, pois uma árvore M está sempre balanceada. Isto ocorre devido à árvore ser construída de baixo para cima (*Bottom-Up*).

Os nós folha da árvore ou contém os objetos indexados ou apenas seus respectivos endereços (Ponteiros); os restantes dos nós possuem um endereço para uma sub-árvore, o raio de cobertura do nó atual e a distância métrica do nó atual até seu respectivo pai. As imagens

abaixo representam a estrutura de uma árvore M (a) e a disposição dos objetos em um espaço métrico.



(a) Estrutura de uma árvore M



(b) Objetos em um espaço métrico

Figura 2.5 (NARDVORNY, 2005 p 23)

2.4.1.2 *Slim-tree*

Slim-tree é um Método de Acesso Métrico derivado da *M-tree* e possui as mesmas características discutidas no tópico 2.4.1.1, como o fato de serem balanceadas, dinâmicas e desenvolvidas para indexar objetos pertencentes a um Domínio Métrico. Quando comparadas sob as mesmas condições, a *Slim-tree* sempre sobrepujou as *M-tree*, tanto em termos de número de acesso a disco quanto em termos de número de distância calculada para responder consultas por abrangência, conforme expõe Bueno Josiane (2001). Ela foi desenvolvida e exposta por Traina Jr et al no ano de 2000.

Assim como nas arvores B+, a *Slim-tree* possui todos os objetos armazenados nas folhas da árvore e todos os outros nós são apenas nós de caminhamento ou *index*.

2.4.1.2.1 Organização da *Slim-tree*

As *Slim-tree* possuem dois tipos de nós; os *IndexNodes,* que são os nós de caminhamento ou índice, e os *LeafNode* que são os nós folhas. Cada um dos tipos de nós citados acima armazenam características diferentes.

Os *IndexNodes* armazenam um vetor de *IndexEntity* de tamanho *C,* cujo valor de *C* representa a quantidade de *IndexEntity* que um nó pode armazenar. *Cada IndexEntity* contém um ponteiro ou referência para um objeto chamado *S*, a distância do objeto *S* para seu representativo ($dist(S,S_{rep})$), um ponteiro ou referência (*Ptr*) para a sub-árvore, um raio de abrangência daquele objeto *S* (R_i) e a quantidade de objetos armazenados na sub-árvore apontada por *Ptr.* A notação a seguir representa melhor um *IndexNode.*

- *IndexNode <vetor de* $<S_i, R_i, dist(S_i, REP(S_i)), Ptr(TS_i), Nentries(Ptr(TS_i)>>$, onde *i* pertence ao intervalo [0, *C*[; *C* é o numero máximo de objetos que um nó na árvore pode armazenar.

Os *LeafNodes* armazenam um vetor de *LeafEntity* de tamanho C*,* em que o valor de *C* representa a quantidade de *LeafEntity* que um nó pode armazenar. Cada *LeafEntity* contém um objeto (*S*), um identificador único para cada objeto (O_{id}) e a distância entre o objeto *S* e seu representativo S_{rep}. Uma outra representação de *LeafNode* é descrita abaixo.

- *LeafNode <vetor de* $<S_i , O_{id} , dist(S_i, REP(S_i)) >>$, onde *i* pertence ao intervalo [0, *C*[, e C é o numero máximo de objetos que um nó na arvore pode armazenar.

Vale lembrar que o tamanho de *C* é uma constante de configuração de um *Slim-tree* e seu valor, que deve ser configurada dependendo da quantidade de objetos que se quer armazenar.

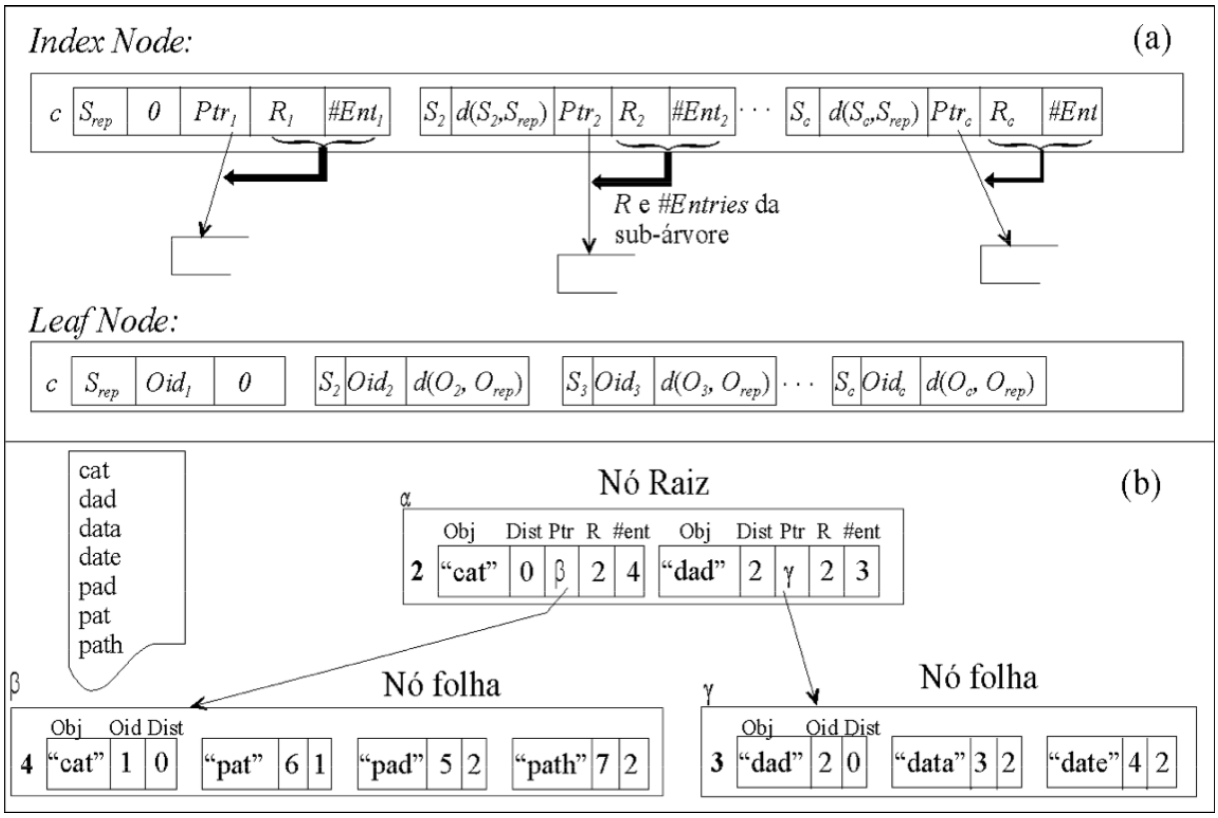


Figura 2.6, (BUENO JOSIANE. 2001, p 22)

2.4.2.2.2 Implementando uma *Slim-tree*

A *Slim-tree* possui o mesmo conjunto de métodos que uma árvore B+ (inserção, atualização, remoção e consulta). O passo a passo do método de inserção é descrito logo em seguida.

Segundo Bueno Josiane (2001), para inserir um novo objeto percorre-se a árvore a partir da raiz, procurando nós que se qualifiquem para receber o objeto. Se nenhum nó se qualificar, então seleciona-se o nó cujo centro está mais perto do novo objeto. Por outro lado, se mais de um nó se qualificar, um outro método chamado *ChooseSubtree* é executado. Este passo é aplicado recursivamente em todos os níveis da árvore. O método de *ChooseSubtree*

pode ser implementado de três formas diferentes:

- *Random*: seleciona aleatoriamente um nó no conjunto de nós que se qualificaram para inserir o novo objeto;
- *Mindist*: seleciona o nó que possui a menor distância entre o objeto a ser inserido e o objeto representativo do nó;
- *Minoccup*: seleciona o nó que possuir a menor quantidade de objetos armazenados.

É importante lembrar que tanto os métodos *mindist* e *ninoccup* podem chegar a um impasse sobre qual nó retorna. Neste caso o nó é escolhido aleatoriamente, e isto ocorre, por exemplo, quando a ocupação ou a menor distância entre dois ou mais nós forem iguais; Apesar de qualquer um dos três tipos de *ChooseSubtree* poder ser utilizado, o minoccup resulta em um número menor de acessos ao disco, melhorando o desempenho da *Slim-tree,* pois, "utilizando a opção minoccup do algoritmo ChooseSubtree, obteve-se árvores mais compactas (maior taxa de ocupação dos nós), o que redundou em um número menor de acessos a disco para responder consultas por similaridade"(BUENO JOSIANE, 2001).

Outra etapa do processo de inserção ocorre quando o nó escolhido para armazenar o objeto a ser inserido já estiver em sua capacidade máxima de armazenamento. Neste caso, o algoritmo de quebra de nós chamado de *Splitting* é executado. O método de *Splitting* aloca um novo nó no mesmo nível do nó a ser "quebrado", e os objetos do nó a ser quebrado, como o objeto a ser inserido, são distribuídos entre os nós. A Slim-tree possui três maneiras diferentes de efetuar a quebra de nós (*Splitting*):

- *Random*: seleciona aleatoriamente dois objetos representativos para comporem os novos nós, os objetos restantes são distribuídos com base na menor distância entre os objetos e os dois representativos. Deve-se respeitar a taxa de ocupação mínima que normalmente é a metade da quantidade máxima de armazenamento por nós.
- *MinMax*: percorre cada um dos pares de objetos considerando que todos são candidatos para se tornarem representativos. A cada iteração associa-se os demais objetos a um dos representativos. Serão escolhidos para serem representativos o par de objetos que minimizar o raio de cobertura da subárvore. Apesar da complexidade do algoritmo ser $O(C^3)$, onde C é a capacidade máxima de armazenamento, ele consegue

obter árvores que possibilitam consultas mais eficientes (CIACCIA 1998 apud BUENO JOSIANE 2001).

- *Minimal Panning Tree (MST)*: constrói-se a árvore de caminho mínimo apresentada por Kruskal em 1956, e um dos arcos mais longos da MST é removido, obtendo assim dois agrupamentos de objetos, que serão associados a cada nó. Segundo Bueno Josiane (2001), testes provam que esse algoritmo possibilita na construção de Slim-trees muito parecidas às que são construídas utilizando o algoritmo MinMax. Entretanto a complexidade computacional do algoritmo MST é de $O(C^2 \log C)$, efetuando $O(C^2)$ cálculos de distância. A figura 2.7 ilustra o algoritmo de quebra de nó utilizando o algoritmo *MST*.

Percebe-se que a *Slim-tree* cresce em altura quando a raiz da árvore está armazenando C objetos. Quando a raiz for quebrada, uma nova raiz será gerada com dois representativos (TRAINA JR, 1999).

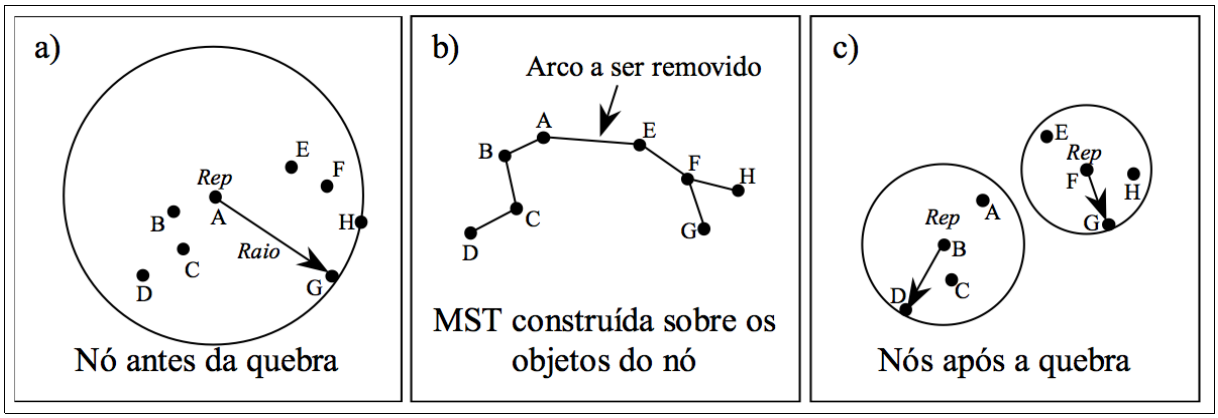


Figura 2.6: a) Nó antes da quebra; b) Cálculo do caminho mínimo e remoção do arco mais longo; c) Nós após a quebra (BUENO JOSEANE, 2001).

2.4.2.2.3 *Fat-Factor*

Segundo Bueno Josiane (2001), uns dos grandes problemas das estruturas de indexação métrica e espacial ocorre por causa do problema da sobreposição dos nós. A sobreposição acontece quando um determinado objeto pertence à interseção de duas ou mais áreas de abrangência entre nós, como mostra a figura 2.7, onde os objetos S_{13} estão sobreposto entres as áreas de abrangência de S_8 e S_6.

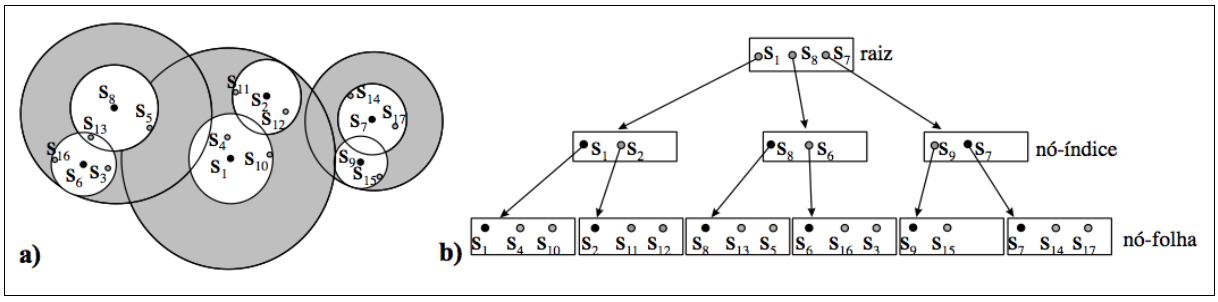


Figura 2.7 (BUENO et al, 2005)

O problema da sobreposição não ocorre em árvores binárias de busca ou em árvores-B, o que faz com que seja possível podar todos os nós que não possuam semelhança com o objeto passado na busca. O ideal seria que as estruturas métricas de indexação também possuíssem esta característica, porém, isto quase não acontece e quando acontece é para somente um conjunto reduzido de casos. A principal consequência da sobreposição dos nós

está no desempenho do algoritmo de consulta, no qual quanto maior a sobreposição, menor será o desempenho da consulta. A perda de desempenho é justificada pela maior quantidade de nós a serem consultados na *Slim-tree*, ou seja, "todos os nós que estão se sobrepondo à região de consulta" (BUENO JOSIANE, 2001, pág).

Procurando minimizar essa deficiência, a *Slim-tree* foi idealizada e codificada, com mecanismos de detecção de porcentagem de sobreposição existente em uma árvore. Esse mecanismo foi nomeado *Fat-Factor*. O *Fat-Factor* avalia o grau de sobreposição entre os nós da árvore *Slim-tree*. Para uma *Slim-tree T* com altura *H,* armazenado *N* elementos em *M* páginas de disco ou nós o *Fat-factor* é calculado por:

$$Fat-Factor\,(T)=(Ic-H*N/N)*(1/(M-H))$$

onde *Ic* representa a quantidade total de nós acessados necessários para responder consultas pontuais para cada um dos *N* objetos armazenados na *Slim-tree* conforme expõe Bueno (2009).

O valor retornado pelo *Fat-Factor* está sempre entre os intervalos [0, 1], onde quanto mais próximo de 1 maior o grau de sobreposição dos nós, e quando o grau for 0 significa que está árvore é ideal não existindo sobreposição entre seus nós.

2.4.2.2.4 *Slim-down*

Ainda segundo Bueno (2009) o *Slim-down* é uma técnica que possui a proposta de minimizar a sobreposição entre os nós de uma *Slim-tree*. Para que a técnica *Slim-down* seja ativada pode-se de tempos em tempos calcular-se o grau de sobreposição da árvore com o *Fat-Factor*. O *Slim-down* foi proposto inicialmente apenas para minimizar a sobreposição dos nós folhas. Quando ativado o *Slim-down* realiza a transição do objeto mais distante do representante do nó para um nó irmão (nós filhos do mesmo pai) que já possua um raio de abrangência que possa comportar o objeto a ser transferido. Com a transferência o raio de abrangência do nó que exportou o objeto pode ser reduzido sem aumentar o raio de abrangência de nenhum nó. Esse processo se repete até que não existam mais possíveis

transferências. A figura 2.8 exemplifica o processo do *Slim-down*.

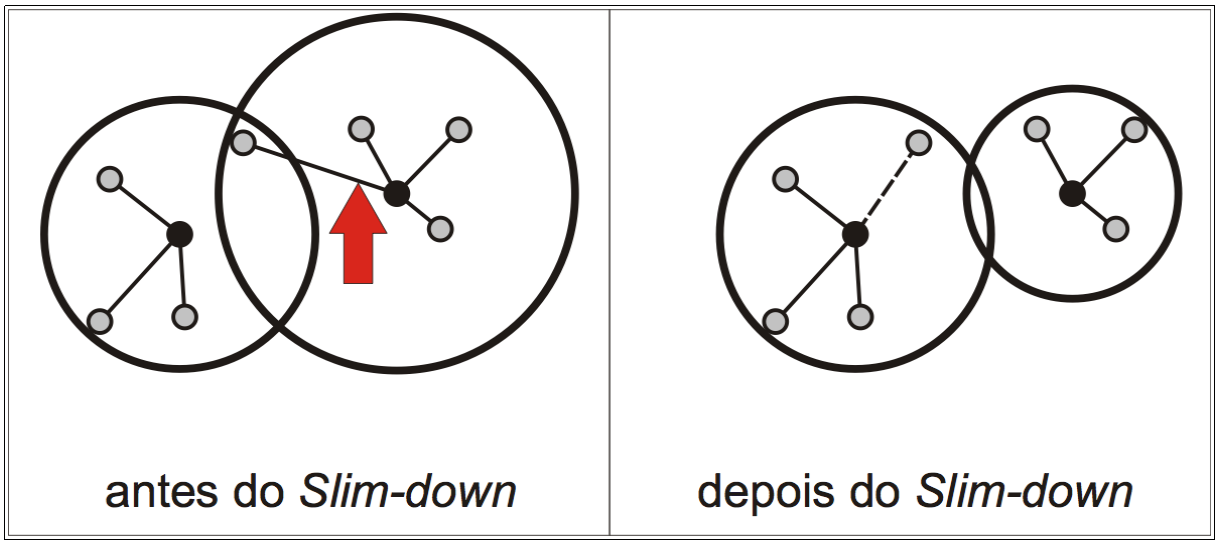


Figura 2.8 Otimizações com *Slim-down* (BUENO, 2009).

2.4.2.2.5 Executando Consultas por Similaridade

Para realizar consultas por similaridade em uma Slim-tree precisa-se percorrer a árvore do nó raiz até os nós folhas comparando os objetos armazenados nos nós com o objeto passado como parâmetro na consulta. Visando minimizar o numero de comparações a Slim-tree utiliza a propriedade da desigualdade triangular explicada no tópico 2.2 deste capítulo. A propriedade pode ser aplicada se a distancia entre um objeto representativo e o objeto pertencentes ao mesmo nó é menor do que a distancia entre objeto representativo e o objeto passado como parâmetro da consulta subtraído do raio de abrangência ou se a distancia entre um objeto representativo e o objeto pertencentes ao mesmo nó for maior do que a distancia entre objeto representativo e o objeto passado como parâmetro da consulta somado do raio de abrangência. No caso se alguma das duas possibilidades forem verdadeiras então o algoritmo de consulta pode podar toda aquela sub-árvore, fazendo com que o numero de comparações seja minimizado. As expressões abaixo exemplificam melhor as condições explicadas a cima:

- $d(S_{rep}, S_i) < d(S_{rep}, S_q) - r_q$
- $d(S_{rep}, S_i) > d(S_{rep}, S_q) + r_q$

A Figura 2.9

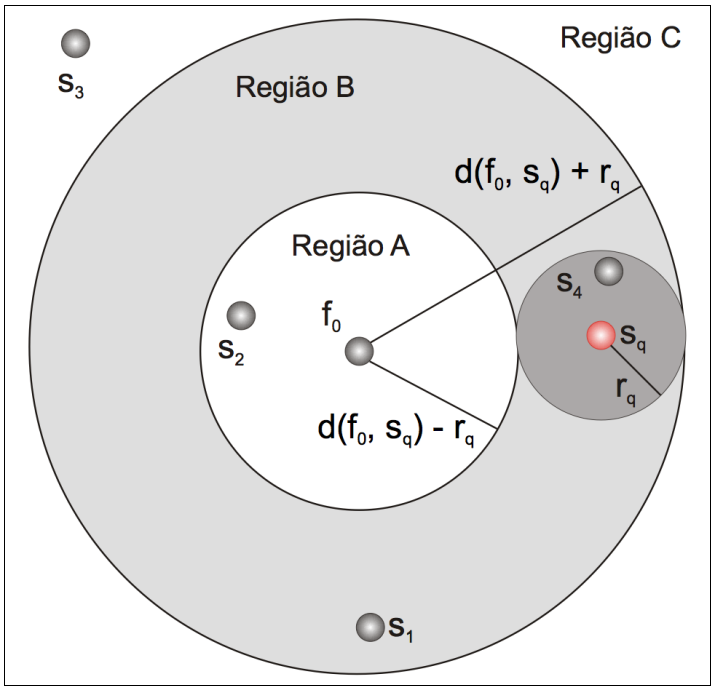


Figura 2.9 Propriedade da desigualdade triangular aplicada para corte de objetos (BUENO, 2009)

2.5 Considerações Finais

Neste capítulo, foram expostos o conceito e as definições de consultas por similaridade, mais especificamente o que seria um Domínio Métrico, os tipos de consultas por similaridade e os Métodos de Acesso Métrico. No próximo capítulo serão apresentados os Algoritmos Genéticos, bem como suas implicações na otimização de problemas.

3 ALGORITMOS GENÉTICOS

3.1 Considerações Iniciais

Os algoritmos genéticos (AG), do inglês *genetic algotithms* (*GA*), são uma técnica de busca extremamente eficiente no seu objetivo de varrer o espaço de soluções e encontrar soluções próximas da solução ótima. Seus mecanismos foram fortemente inspirados na teoria da evolução proposta por Darwin, em 1859. Os primeiros cientistas a introduzir e disseminar os Algoritmos Genéticos foram John Holland e seu aluno David Goldberg em 1975 e 1989 respectivamente, conforme expõem Lacerda e Carvalho (2002).

Alguns dos princípios dos algoritmos genéticos se baseiam na frase do livro **A Origem das Espécies** de Charles Darwin, que diz que quanto melhor um indivíduo se adaptar ao seu ambiente, maior será sua chance de sobreviver e gerar descendentes, ou seja, quanto melhor ou mais apto for um indivíduo, mais chances de sobreviver e de gerar descendentes ele vai ter.

Os algoritmos de otimização se caracterizam, por buscarem a melhor solução para um determinado problema, por isso os Algoritmos Genéticos se encaixam tão bem nessa classe de problemas. Conforme expõem Lacerda e Carvalho (2002), os algoritmos genéticos consistem em tentar várias soluções e utilizar a informação obtida neste processo, de forma a encontrar soluções cada vez melhores, por isso são tão eficientes em encontrar múltiplas soluções, provavelmente as melhores para resolver o problema.

Os ciclos de execução de um Algoritmo Genético possuem sete etapas bem definidas:

1. Geração da População Inicial;
2. População Atual;
3. Avaliação da População Atual;
4. Seleção de Indivíduos para compor a próxima População;

5. *Crossover* dos Indivíduos selecionados anteriormente ;
6. Mutação nos indivíduos gerados pelo *crossover*;
7. Critérios de Parada para saber se o ciclo se repete ou não.

As etapas serão explicadas separadamente, mais tarde. A figura 4.1 representa o ciclo de um Algoritmo Genético.

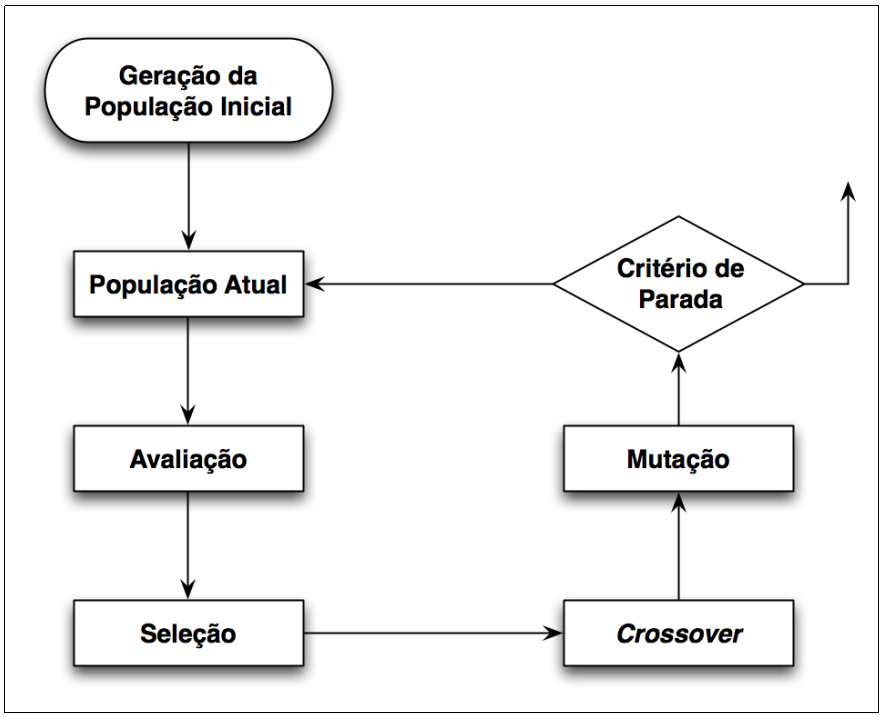


Figura 3.1 Ciclo de Execução dos Algoritmos Genéticos.

3.2 Terminologia

Os algoritmos genéticos possuem muitos termos originados da biologia, porque, segundo Lacerda e Carvalho (2002), eles são a metáfora dos processos de evolução e reprodução dos indivíduos mais adaptados ao ambiente. Durante a reprodução, ocorrem processos de *crossover* e mutação. Para entendermos melhor esse paradigma, precisamos conhecer alguns dos principais termos encontrados no âmbito dos Algoritmos Genéticos.

- **Cromossomo e Genoma:** Representam um ponto no espaço de busca; são a representação da solução codificada em uma estrutura de dados; geralmente não passam de um vetor ou uma árvore;
- **Gene:** Nada mais são do que características ou atributos do problema em questão, codificados em uma posição ou nó do cromossomo;
- **Indivíduo:** Um membro da população; São formados por um ou mais cromossomos e sua aptidão;
- **Genótipo:** Representa a informação contida dentro dos cromossomos;
- **Fenótipo:** Representa o objeto construído a partir da informação do genótipo;
- **Alelo:** São os diferentes valores que um gene pode assumir;

3.3 *Exploration X Exploitation*

Um método de otimização é considerado bom se realizar duas tarefas básicas. A primeira tarefa é a capacidade de explorar novas soluções no âmbito do espaço de soluções. A segunda tarefa é a capacidade de gerar novas soluções a partir da extração de informação de soluções já existentes e visitadas (LACERDA E CARVALHO 2002). Os principais métodos de otimização encontrados na literatura (Busca Aleatória, Subida de Encosta entre outros) só possuem a capacidade de realizar bem uma das tarefas descritas anteriormente, fazendo com que os Algoritmos Genéticos tenham uma vantagem sobre eles.

A capacidade de explorar novas soluções no espaço de busca é denominada *Exploration* e a capacidade de encontra novas soluções baseadas nas soluções já adqueridas é denominada de *Exploitation*. A pesar da tradução dos termos para o português serem iguais, "exploração", elas representam ideias diferentes.

Os operadores genéticos (*Crossover* e Mutação) que são descritos na seção 3.9 implementam a característica de *Exploration*, pois depois de executados soluções novas podem ser encontradas. Enquanto a etapa de seleção implementa a característica de *Exploitation* pois a busca é guiada para região onde se encontra as melhores soluções.

A pressão de seleção segundo Lacerda e Carvalho (2002) influencia diretamente a quantidade de *Exploration* e *Exploitation*. A pressão de seleção é obtida através da razão entre

avaliação do melhor indivíduo da população e a media das avaliações dos indivíduos pertencentes a população. Se a pressão for baixa demais o Algoritmo Genético tendera a ter um comportamento parecido com uma busca aleatória. Porem se a pressão for alta demais o Algoritmo Genético tendera a ter um comportamento semelhante aos métodos de subida de encosta.

3.4 Representação das Soluções

Quando foram idealizadas, as representações dos problemas eram codificadas utilizando dados binários (zeros e uns), mas existem casos em que uma representação real é mais cabível, por exemplo em problemas com parâmetros contínuos, matemáticos cuja precisão de casas decimais é muito grande. A representação binária, nestes casos, chega a ser inviável, "pois para cada ponto decimal acrescentado na precisão, é necessário adicionar 3,3 bits na cadeia" (LACERDA e CARVALHO, 2002, p. 107). A representação real, em certos casos, mostra-se bastante simples e robusta na resolução do problema. Na figura 4.2 e 4.3 temos um exemplo de como seria um conjunto de indivíduos em ambas as representações binária e real, respectivamente.

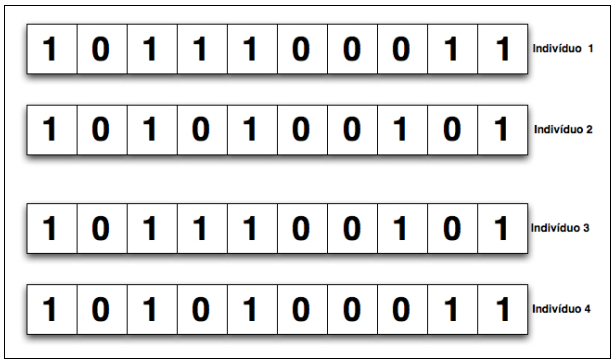


Figura 3.2 Exemplo de uma população de indivíduos utilizando a representação binária.

As duas representações mais difundidas são a binária e a real, mas na realidade qualquer outro tipo de estrutura de dados (lista ligada, pilha, fila ou árvores) podem ser utilizadas. A escolha de que tipo de representação escolher vai depender da natureza do

problema proposto para resolução.

Neste trabalho, foi feito uma análise sobre qual a melhor forma de representar as soluções para o problema dentro do cromossomo de cada indivíduo.

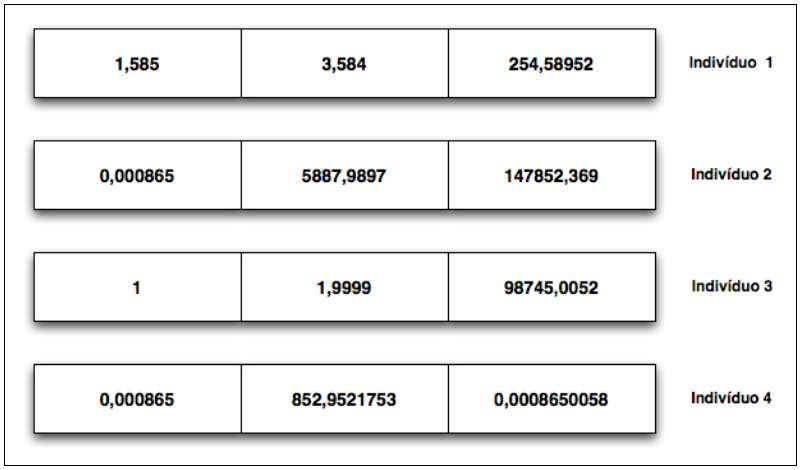


Figura 3.3 Exemplo de uma população de indivíduos utilizando a representação real.

3.4.1 Infactibilidade

Um dos grandes problemas das representações das soluções está no conjunto de todos os indivíduos que codificam uma solução inexistente. Os elementos pertencentes a este conjunto são chamados de infactíveis. A infactibilidade está presente na grande parte dos problemas do mundo real. Existem diversas formas de se tratar a infactibilidade de um elemento, todavia neste trabalho relataremos apenas as três mais utilizadas.

A primeira forma é simplesmente ignorando a infactibilidade, pois é muito provável que pelo menos um elemento da população de indivíduos seja uma solução utilizável (factível). A segunda maneira verifica se um determinado indivíduo é uma solução factível, se não for, esse elemento é descartado e um outro indivíduo é criado. Dependendo do problema, a segunda forma pode deixar nosso algoritmo quase que em *loop* infinito, pois dependendo do nosso espaço de solução, muitos indivíduos infactíveis podem ser gerados. A terceira forma é denominada de reparação, pois para cada indivíduo infactível é aplicado uma heurística, com a intenção de transformar a solução infactível em uma solução factível.

3.4.2 Princípio da Localidade

Outro grande problema na representação das soluções está ligado diretamente com o espaço de busca e as representações das soluções. Toda representação possui dois espaços de busca, o do genótipo e o do fenótipo. O problema existe quando uma pequena modificação do genótipo pode gerar uma mudança drástica no fenótipo. O problema ocorre, também, quando uma grande modificação do genótipo gera uma mudança suave no fenótipo. A figura 4.4 melhor exemplifica o problema e utiliza a representação binária de um numero natural. No indivíduo 1, alterando apenas um único bit é gerado um indivíduo (2) com o fenótipo muito menor comparado com o indivíduo 1. Por outro lado, alterando os 8 bits do indivíduo 3, é gerado um indivíduo (4) com o fenótipo quase inalterado.

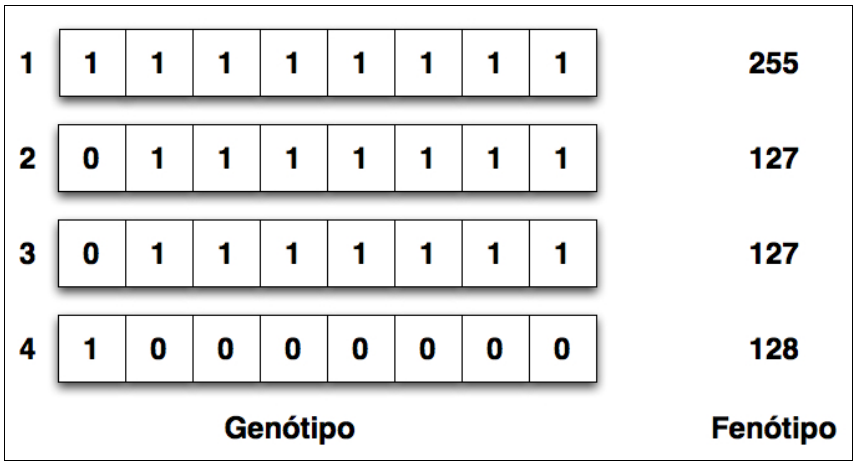


Figura 3.4 Implicação do principio da localidade.

3.5 População Inicial

A população inicial, como o próprio nome diz, é a primeira população de indivíduos com os quais os algoritmos genéticos trabalham. Elas podem ser geradas a partir de vários métodos. Nem sempre existe um método bom para se gerar uma nova população, pois a geração de indivíduos está diretamente relacionada com o espaço de solução do

problema tornando em muitos casos, a geração da população inicial uma dificuldade a ser resolvida especificamente para o problema em questão. O primeiro desses métodos é o aleatório, ou seja, a cada execução do algoritmo, uma nova e diferente população inicial é formada, e não se tem garantia de que essa população inicial seja igual à população inicial de outra execução. Outro método, é o de utilizar como população inicial soluções encontradas por outras técnicas de otimização, ou algumas soluções aceitáveis para o problema. Esse método é conhecido como "*seeding*", segundo Lacerda e Carvalho (2002).

3.6 Função Objetivo

Os problemas do mundo real são, em sua maioria, bastante complexos fazendo com que a função objetivo seja também bastante complexa e com alto custo dos recursos computacionais. É importante simplificar ao máximo a função objetivo, mas essa simplificação trará perdas à informação na solução final. Para reduzirmos o tempo computacional, é interessante utilizarmos uma simplificação da função objetivo nas primeiras gerações de indivíduos e deixarmos a função objetivo real para as últimas gerações. Uma outra técnica bastante difundida para a redução dos recursos computacionais é utilizar os algoritmos genéticos para localizar a região da função objetivo mais promissora a ser o máximo da função e depois utilizar um algoritmo de Subida de Encosta para localizar a possível melhor solução. Isso se dá porque os métodos de Subida de Encosta são extremamente eficientes na obtenção do ponto de máximo local.

3.7 Critérios de Parada

Com relação aos critérios de parada, observamos que estes são responsáveis por fazer com que o Algoritmo Genético não fique evoluindo infinitamente. Para cada tipo de problema pode ser utilizado um critério de parada diferente. Em alguns casos, podemos priorizar o tempo ou os recursos computacionais; em outros, a quantidade de gerações, ou

então, a chegada ao valor ótimo da função (somente se conhecermos esse valor). E, por último, o algoritmo poderia parar sua execução quando a convergência estiver acima de um certo ponto. A convergência ocorrerá quando a aptidão dos indivíduos de uma determinada população não melhorar por um dado número de gerações.

3.8 Seleção

Baseando-se no processo de seleção natural da biologia, o Algoritmo Genético seleciona os melhores indivíduos da população inicial, ou seja, os que possuem uma maior aptidão, para povoarem a próxima geração e para criarem novos indivíduos filhos (parecidos com seus pais). Os operadores de *crossover* e mutação são responsáveis pela criação dos novos indivíduos que irão povoar a próxima geração. Existem vários métodos de seleção, mas dois deles se destacam:

3.8.1 Roda da Roleta

A seleção por Roda da Roleta utiliza uma população intermediária conhecida como *mating pool*. A população intermediária serve para alocar os cromossomos pais selecionados, os quais são selecionados através da probabilidade proporcional à sua aptidão. Segundo Lacerda e Carvalho (2002) , "a probabilidade de seleção p_i , de um cromossomo s_i com aptidão f_i é dada por:

$$fi / \sum_{i=1}^{N} fi \quad "$$

A seleção por Roda da Roleta é realizada calculando uma coluna de aptidão acumulada. Depois, gera-se um número aleatório r no intervalo [0, Soma Total], cuja Soma Total é a soma de todas as aptidões. Por último, o indivíduo selecionado será o primeiro que

possuir a aptidão acumulada maior que *r*. Os passos são repetidos até toda a população intermediária ter sido preenchida.

O método de seleção Roda da Roleta foi o primeiro método desenvolvido e possui uma série de problemas, dos quais os dois problemas mais difundidos e estudados bem como as técnicas para contornar os problemas, serão expostos nas próximas seções.

3.8.1.1 Avaliação Negativa

Se existir um ou mais indivíduos da população com avaliação negativa, nunca será gerado um número aleatório, que represente a posição que o indivíduo ocupa na população e, portanto, ele nunca será selecionado para compor a próxima população.

Para resolver esse problema, basta somar uma constante maior que zero ao valor da função de avaliação. O valor da constante deve ser escolhido com base no maior valor negativo que a função de avaliação pode gerar.

3.8.1.2 Super Indivíduo

Um super indivíduo é aquele que possui uma avaliação muito superior à média do resto da população. Tomando como base a tabela 4.1, o indivíduo x5 será selecionado 99% das vezes, e pode ser que uma geração inteira de pais se resuma a este indivíduo.

Indivíduo	Avaliação
X1	20
X2	30
X3	40
X4	10
X5	9900

Tabela 4.1

Se uma população se resumir a um único indivíduo após uma seleção, acabando com a divergência da população, o algoritmo genético possuirá uma convergência genética prematura. Para resolver esse problema deverá ser utilizado alguma técnica de normalização. A mais utilizada é a normalização linear, que consiste em colocar os indivíduos em ordem decrescente de valor e criar novas funções de avaliação para cada um dos indivíduos e que o melhor de todos receba um valor fixo k e os outros recebam valores iguais ao do indivíduo imediatamente anterior na lista ordenada, menos um valor de decremento constante t.(Notas de sala de aula). Os parâmetros k e t precisam ser ajustados para cada problema.

Se o valor de t for muito pequeno, acontecerá uma aglomeração das avaliações, ou seja, todos os indivíduos passam a ter uma avaliação muito próxima. Por outro lado, se o valor de t for muito grande, ocorrerá uma desigualdade artificial entre os indivíduos que anteriormente tinham valores de avaliação extremamente próximos.

No caso do parâmetro k, o seu valor pouco importa, o mais importante é a relação k/t, a qual determina quão diferentes os indivíduos serão em termos proporcionais. Costuma-se fixar k=1.

3.8.2 Seleção Por Torneio

A seleção por torneio forma a população intermediária da seguinte maneira; 1) são escolhidos aleatoriamente e com probabilidade iguais, n cromossomos da população; 2) dados os cromossomos selecionados anteriormente, é escolhido o mais adaptado dentre eles, por isso a nomenclatura "torneio". O valor mais utilizado para n é de $n = 2$. Uma grande vantagem da seleção por torneio é o fato de "não precisar de escalonamento da aptidão e nem de ordenamento", conforme proposto por Lacerda e Carvalho (2002, p. 131), ou seja, a seleção por torneio é processada sem nenhuma modificação na população, o que reduz o consumo dos recursos computacionais.

Em uma outra abordagem, a seleção por torneio utiliza probabilidades diferenciadas. Dados n cromossomos para participarem do torneio, o primeiro cromossomo ganhará o torneio com probabilidade q; o segundo com probabilidade $q(1-q)$; o terceiro com $q(1-q)^2$, e assim por diante, até toda a população intermediária ter sido preenchida. Ainda

segundo a teoria proposta por Lacerda e Carvalho (2002, p 131-132), duas conclusões importantes podem ser tiradas:

- "Vale notar que se n = N, em que N é o tamanho da população, tal seleção é equivalente à seleção com ordenamento exponencial";
- "Aumentando o número n de cromossomos do torneio ou a probabilidade q de o primeiro cromossomo vencer, aumenta-se a pressão de seleção, isto é, cromossomos com aptidão acima da média terão mais chances de serem selecionados".

3.9 Operadores Genéticos

Os algoritmos genéticos possuem dois operadores bio-inspirados muito bem definidos e com algumas variações.

3.9.1 *Crossover*

O operador de mutação, juntamente com o operador de *crossover,* são as principais funções de busca dos Algoritmos Genéticos na exploração de regiões desconhecidas do espaço de busca. As operações de *crossover* ocorrem em um par de cromossomos obtidos a partir da população intermediária e geram dois novos cromossomos, nomeados de cromossomos filhos.

Os cromossomos filhos têm seus genes herdados dos cromossomos pais, sendo a cadeia de genes constituída por partes das cadeias de genes dos cromossomos pais. A cadeia de genes dos pais é cortada em uma posição aleatória, sendo que o primeiro cromossomo filho recebe a primeira e a segunda parte do primeiro e do segundo dos cromossomos pais, respectivamente. O segundo cromossomo filho, por outro lado, deverá receber a segunda parte do primeiro cromossomo pai e a primeira parte do segundo cromossomo pai. A figura abaixo melhor representa o comportamento do operador de *crossover*.

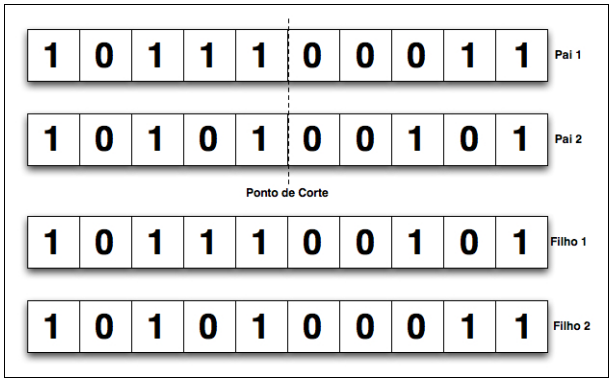


Figura 3.5 *Crossover* de Um Ponto

O *crossover* multiponto, também conhecido como *crossover* de *N* pontos, já foi descrito na seção anterior, sendo o *crossover* de 1 ponto apresentado. Os de 2 e 4 pontos são apresentados nas figuras logo abaixo. A principal diferença entre os operadores de *crossover* de 1, 2 e 4 pontos, está na quantidade de pontos de corte que aparecem na cadeia de genes no momento da permutação de características.

3.9.1.1 *Crossover* Uniforme

No *crossover* uniforme, para cada par de cromossomos pais é criada uma máscara de bits aleatórios. Essa máscara de bits não passa de uma cadeia de tamanho igual ao da cadeia de genes dos cromossomos e possui somente zeros e uns como valores armazenados, em cada posição da cadeia. Se o primeiro bit da máscara for 1, então o filho1 receberá o primeiro gene do pai1. Em caso contrário, receberá o primeiro gene do pai2, e assim por diante, até que toda a cadeia de gene do filho tenha sido preenchida. O segundo filho receberá o processo oposto ao do filho1, como descrito na figura abaixo.

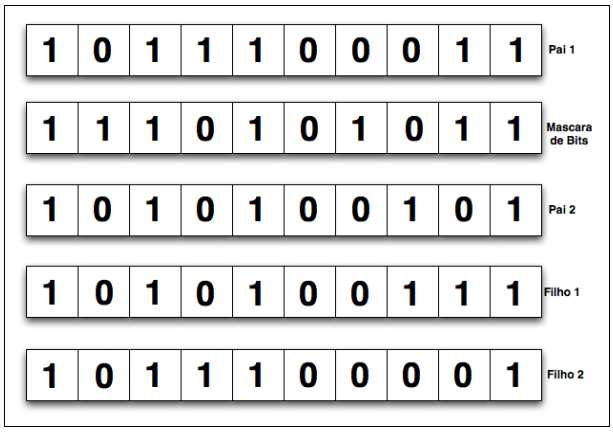


Figura 3.6 *Crossover* Uniforme

3.9.2 Mutação

Após aplicada a operação de *Crossover*, a operação de mutação é aplicada. Dada uma probabilidade normalmente entre 0.1% a 5% sobre cada gene dos cromossomos filhos gerados pelo operador de *crossover, a* operação de mutação inverte o valor dos genes, isso quer dizer que, se um gene possuir o valor 0 e ocorrer mutação, ele passará a armazenar o valor 1. A figura abaixo explica e exemplifica a operação de mutação.

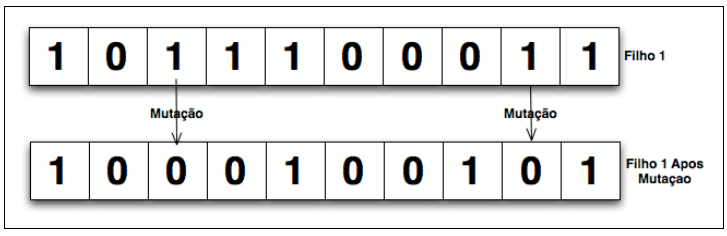


Figura 3.7 Mutação nos genes 3 e 9

"A mutação melhora a diversidade dos cromossomos na população, por outro lado, destrói a informação contida no cromossomo" (LACERDA e CARVALHO, 2002, p97). O que significa dizer que devemos ter cuidado ao aplicar o operador de mutação sobre os cromossomos, pois se forem muito altas o Algoritmo Genético se comportará como uma busca aleatória e se a taxa for muito baixa o operador de mutação não conseguirá gerar

diversidade nos cromossomos. Na grande maioria dos exemplos, as taxas de probabilidade variam entre 0.1% a 5%.

3.10 Elitismo

A cada interação do algoritmo genético, uma nova população de soluções candidatas à resolução do problema é formada. Nessa nova população, podemos perder informações valiosas, como a melhor solução já encontrada para o problema. Isso ocorre devido às operações de *crossover* ou mutação, pois ainda segundo Lacerda e Carvalho (2002 p 99) "é interessante transferir o melhor cromossomo de uma geração para a outra", finalmente, quando mantemos a melhor solução (cromossomo) da população anterior na nova população, dizemos que o algoritmo é elitista.

3.11 Considerações Finais

Neste capítulo, vimos a origem e o funcionamento dos algoritmos genéticos e o porque desta técnica evolutiva ter recebido tantas pesquisas, principalmente nas otimizações de algoritmos computacionais modernos. No capítulo 5, veremos como integrar os algoritmos genéticos para otimizarem o processo de consulta por similaridade aproximada.

4 SOLUÇÃO

4.1 Considerações Iniciais

Nos capítulos anteriores foi discutido o motivo pela adoção dos algoritmos Genéticos, na resolução do problema de otimização das consultas por similaridade. Neste capítulo, será apresentada uma maneira de interligar os conceitos de similaridade com a técnica de otimização global, evolutiva dos algoritmos genéticos. Os sub-tópicos apresentados logo em seguida apresentam a mesma ordem vista no capítulo 4, mostrando apenas como se dá a junção dos conceitos apresentados nos capítulos anteriores.

O algoritmo desenvolvido possui o ciclo de execução representando na Figura 4.1.

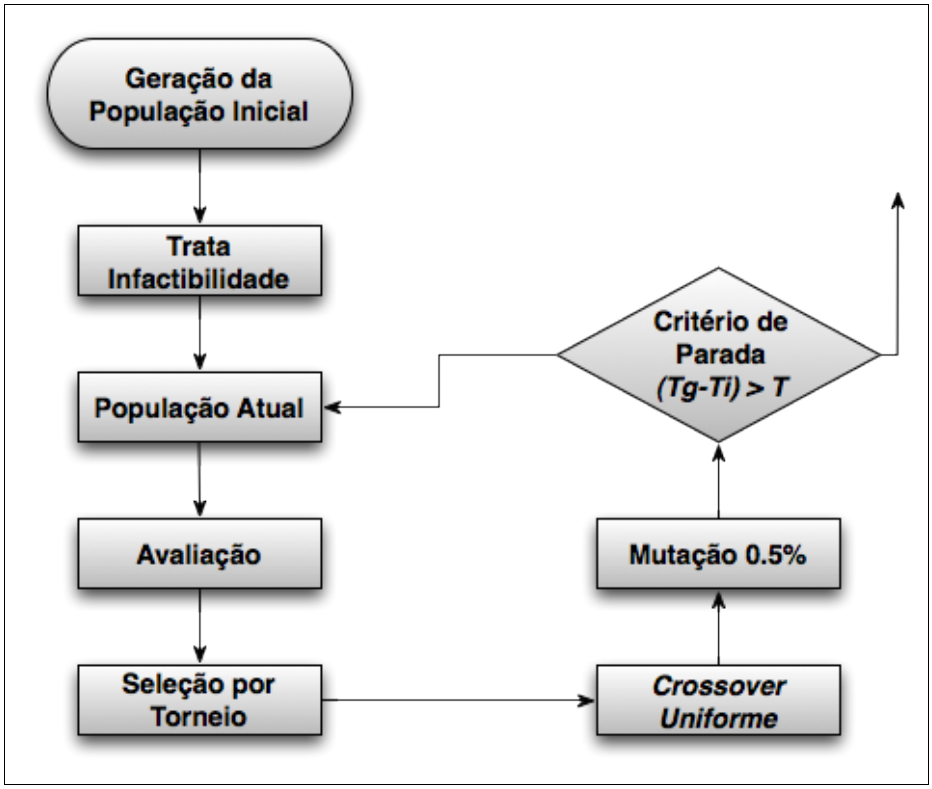


Figura 4.1 Ciclo de Execução do Algoritmo Desenvolvido. O critério de parada é definido pela condição: tempo da geração atual (*Tg*) menos tempo quando o algoritmo iniciou sua execução (*Ti*) for maior que o tempo padrão ou ao parâmetro passado no inicio da busca(*T*).

4.2 Representação das Soluções

Uma solução para o nosso problema é composta por um caminho de nós a serem percorridos em um método de acesso métrico (*Slim-tree*) a partir da raiz (*root*) da árvore até um nó folha (*Leaf Node*). Cada caminho a ser percorrido é uma característica do genótipo ou genes. Para uma árvore de altura *h,* são necessários *h-1* características, pois apenas os nós caminhos (*Index Node*) são utilizados. A figura 4.2 mostra como seria a árvore de caminhos em um método de acesso métrico:

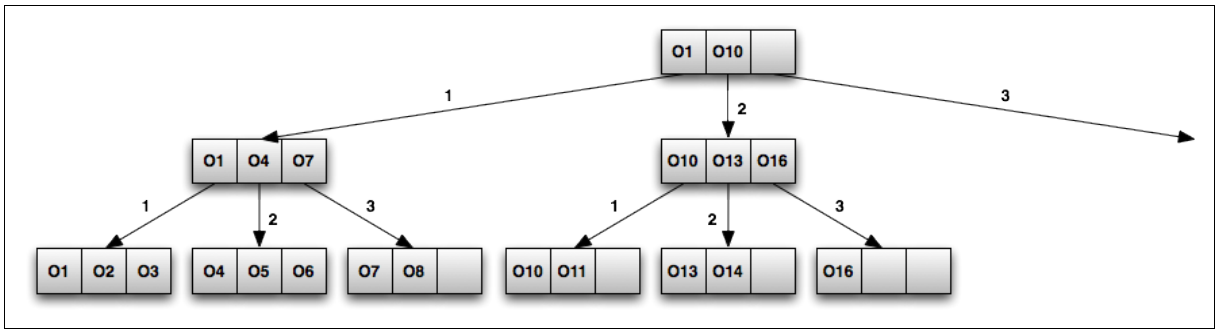


Figura 4.2 Árvore de Caminhos de uma *Slim-tree.*

Para se alcançar o objeto "O_{10}" devemos caminhar na árvore a partir da raiz passando pelos caminhos 2 e 1. Com essa representação, não precisamos efetuar um cálculo de distância para cada nível percorrido na árvore, o que aconteceria em uma consulta por similaridade tradicional. Porem precisamos dizer, de alguma forma, o quanto uma solução é boa ou não. Para inferir o quanto um caminho é bom ou não, foi criada uma função de avaliação que será descrita na seção 4.4 deste capítulo. Cada caminho poderia ser representado computacionalmente como um inteiro sem sinal, porém essa abordagem faz com que os operadores (*crossover* e mutação) dos algoritmos genéticos percam sua principal função de explorar o espaço de soluções.

Devido ao problema citado no parágrafo anterior, foi escolhida uma representação binária em que cada caminho (gene) é convertido para um número binário. É importante notar que a faixa de valores que um gene pode assumir pertence ao intervalo [0 ; *C* [, onde *C* é o

número máximo de objetos que um nó de caminho (*Index Node*) pode armazenar.

4.2.1 Infactibilidade

Um dos problemas que podem ocorrer se a árvore não estiver completamente na sua capacidade máxima de armazenamento é de uma solução tentar acessar um caminho que não possui nenhum objeto armazenado. Quando essa situação ocorre, dizemos que o caminho ou a solução são infactíveis fazendo com que seja, executado uma rotina de tratamento de infactibilidade.

A rotina executada simplesmente verifica qual caminho não está acessível e modifica o valor do caminho para a primeira posição anterior, válida. A rotina de tratamento, então, caminha na árvore perguntando para cada nó se o caminho é acessível; em caso positivo, então percorremos o caminho; caso negativo, verificamos qual o menor caminho mais próximo do caminho inválido, a rotina é executada recursivamente para cada caminho gerado. Tomando como exemplo a Figura 5.1, se uma solução possui os caminhos 3 e 2, mas o caminho 3 leva a uma área de armazenamento vazia (Nula ou *NULL*), neste caso, a rotina de tratamento trocaria o valor de 3 por 2 já que o menor caminho mais próximo de 3, e a solução assume agora os valores {2 ,2}, que levam ao nó que possui o objeto "O_{13}", como seu representativo.

Outras abordagens podem ser adotadas para se converter o problema da infactibilidade. Em Bueno et al (2005) é discutida e implementada uma abordagem de penalidade, onde uma solução que tiver algum caminho infactível será penalizada em sua avaliação fazendo com o método de seleção do Algoritmo Genético elimine essa solução com o passar das gerações.

4.3 População Inicial

A População inicial é formada por um conjunto de indivíduos e é gerada

aleatoriamente. Pelo fato de as características dos indivíduos serem geradas aleatoriamente, é muito provável a presença de caminhos infactíveis dentro da população inicial. Por essa razão depois de gerada a rotina de tratamento de infactibilidade é executada para cada indivíduo pertencente a população inicial.

4.4 Função Objetivo

Para avaliar a qualidade de cada indivíduo, foi definido uma função de distância.

4.5 Critérios de Parada

Para cada tipo de aplicação de consultas por similaridade podemos definir um conjunto de critérios de parada. Como explicado na na seção 3.7 do capítulo 3, os critério de parada são necessários para que o Algoritmo Genético não fique executando infinitamente.

Um critério de parada bastante utilizado pela comunidade é definido pelo tempo máximo que a aplicação poderá ser executada. Apesar do valor máximo do tempo de execução poder ser um parâmetro enviado a aplicação, essa abordagem não é recomendada pois para cada tipo de consulta por similaridade dentro de um domínio métrico um tempo minimo é necessário, para que o Algoritmo Genético possa varrer a região promissora do espaço de busca. No algoritmo desenvolvido o critério de parada utilizado é o de tempo máximo e é definido com base no tempo médio de execução do cálculo de distância entre dois objetos.

4.6 Seleção

Em Bueno et al (2005) é utilizada o método da roleta como método de seleção do

Algoritmo Genético. Devido a todos os micros problemas gerados em torno do método da roleta (capítulo 3, seção 3.8.1) foi escolhido o método do torneio como método de seleção, pois alem de não possuir os mesmos problemas encontrados no método da roleta ainda possui uma implementação mais simples

4.7 Operadores Genéticos

O operador genético de *crossover* escolhido foi o *crossover* uniforme, pois com ele podemos obter os mesmos resultados possíveis cujo os quais seriam obtidos pelo *crossover* de um ponto ou multi-ponto e ainda mas possibilidades que nenhum outro método implementa.

O outro operador genético de mutação possui uma taxa de mutação baixa de 0.5% fazendo com que o operador de mutação sirva apenas para realizar um ajuste fino nas soluções encontradas.

4.8 Elitismo

Aki

4.9 Algoritmo em Execução

Logo abaixo é mostrado a execução de um ciclo do algoritmo, exibindo cada indivíduo pertencente a população e antes e depois da execução de cada operador genético.

4.10 Considerações Finais

Neste capitulo foi explicado como mesclar os conceitos de similaridade com os conceitos dos algoritmos genéticos, bem como a forma de implementação do algoritmo. No próximo capitulo será apresentando alguns dos resultados obtidos.

5 RESULTADOS

6 CONCLUSÕES

REFERÊNCIAS

BÄCK, Thomas; FOGEL, David B; MICHALEWICZ, Zbigniew. ***Evolutionary Computation 1:*** b*asic algorithms and operators*. Institute of Physics Publishing. 2000. Acesso em http://www.google.com no dia 22/22/2000.

BUENO, Renato; TRAINA, Agma J. M; TRAINA JR, Caetano. **Algoritmos Genéticos para Consultas por Similaridade**. 2005. Acesso em http://www.google.com no dia 22/22/2000.

BUENO, Renato. **Tratamento do Tempo e Dinamicidade em Dados Representados em Espaços Métricos**. 2009. Acesso em http://www.google.com no dia 22/22/2000.

CHINO, Fabio Jun Takada. **Visualizando a Organização e o Comportamento de Estruturas Métricas:** aplicações em consultas por similaridade. 2004.

FERREIRA, Mônica Ribeiro Porto. **Suporte a Consultas por Similaridade Unárias em SQL**. 2008. Acesso em http://www.google.com no dia 22/22/2000.

GOLDBERG, David E. ***Genetic Algorithms in Search, Optimization, and Machine Learning***. *Cidade*, Addison-Wesley Company Inc. 1989.

JARDINI, Evandro; GONZAGA, Adilson; TRAINA, JR Caetano. **Metodologia para Indexação de Busca de Impressões Digitais, Através do Uso de Função de Distância Métrica**. 2006. Acesso em http://www.google.com no dia 22/22/2000.

Josiane M. BUENO. **Suporte à Recuperação de Imagens Médicas Baseada em Conteúdo através de Histogramas Métricos.** 2001. Acesso em http://www.google.com no dia 22/22/2000.

LACERDA, Estéfane; CARVALHO, André. **Introdução aos Algoritmos Genéticos**. Capítulo 3. Revista de Informática Teórica e Aplicada, v.9, p.7 - 39, 2002.

NADVORNY, César Feijó. **TM-*tree*:** um método de acesso para consultas por similaridade. 2005. Acesso em http://www.google.com no dia 22/22/2000.

PACHECO, Marco**. Algoritmos Genéticos:** princípios e aplicações. 1999 Acesso em **http://www.google.com** no dia 22/22/2000.

RUSSEL, Stuart; NORVIG, Peter. **Inteligência Artificial**. 2 ed. Rio de Janeiro: Editora Campus, 2004.

SILVA, Maria; BORGES, Eduardo; GALANTE, Renata. **XSimilarity:** Uma ferramenta para Consultas por Similaridade embutidas na Linguagem Xquery. 2008. Acesso em http://www.google.com no dia 22/22/2000.

TRAINA JR, Caetano; TRAINA, Agma; SEEGER, Bernhard; FALOUTSOS, Christos. ***Slim-trees: High Performace Metric Tree Minimizing Overlap Between Nodes***. 1999. Acesso em http://www.google.com no dia 22/22/2000.

VIERA, MARCOS; TRAINA JR, Caetano; CHINO, Fabio; TRAINA, Agma. ***DBM-Tree:*** *a dynamic metric access method sensitive to local density data*. 2004. Acesso em http://www.google.com no dia 22/22/2000.

YAMAROTO, Cláudio Haruo. **Uso de Lógica Nebulosa na Construção e na Utilização da Arvore Métrica *Slim-tree.*** *2002. Acesso em* http://www.google.com *no dia 22/22/2000.*

ZUBEN, Fernando J. Von. **Computação Evolutiva:** Uma Abordagem Pragmática. Acesso em http://www.google.com no dia 22/22/2000.

TODO List

Na seção 4.5 pode-se falar mais sobre a abordagem de visitar todos os caminhos possíveis