Indexação de Fingerprints

João Felipe Contreras de Moraes

February 14, 2025

Contents

1 Introdução à Indexação **3**
- 1.1 O que é indexação? . . . 3
  - 1.1.1 Busca por Semelhança . . . 3
- 1.2 Implementações de índices . . . 5
  - 1.2.1 B Tree . . . 5
  - 1.2.2 B+ Tree . . . 7
- 1.3 Indexação Multidimensional . . . 7
  - 1.3.1 KD-Tree . . . 7
  - 1.3.2 R-Tree . . . 8
  - 1.3.3 M-Tree . . . 8
  - 1.3.4 SA Tree . . . 8
- 1.4 The Curse of Dimensionality . . . 8
  - 1.4.1 Conceito de Dimensão . . . 9
  - 1.4.2 Impacto na Busca por Similaridade . . . 9

2 Indexação para Fingerprints **13**
- 2.1 Introdução . . . 13
- 2.2 Minutia Cylinder-Code . . . 13
  - 2.2.1 Representação . . . 14
  - 2.2.2 Similaridade . . . 14
  - 2.2.3 Indexação . . . 15

3 Vector Databases **19**
- 3.1 Introdução . . . 19
  - 3.1.1 Aplicações . . . 19
- 3.2 Divisão dos métodos . . . 20
- 3.3 Métodos exatos . . . 20
  - 3.3.1 Linear Scan . . . 20
  - 3.3.2 Particionamento de Espaço . . . 20
- 3.4 Métodos baseados em clusterização . . . 21
  - 3.4.1 Inverted File Index (IVF) . . . 21

3.5 Métodos baseados em árvores 23
3.5.1 sa-tree 23
3.5.2 Approximate Nearest Neighbors Oh Yeah (ANNOY) . 23
3.6 Métodos baseados em grafos 23
3.6.1 Navigable Small World 23
3.6.2 Hierarchical Navigable Small World (HNSW) 27
3.7 Métodos baseados em hashing 32
3.7.1 LSH 32
3.8 Quantização 32
3.8.1 Scalar Quantization 32
3.8.2 Product Quantization 32
3.9 Opções comerciais 32

Chapter 1

Introdução à Indexação

1.1 O que é indexação?

Um índice, no contexto de *dados estruturados*, é uma estrutura de dados que melhora a velocidade das operações de query de dados em uma tabela, ao custo de escritas adicionais e espaço de armazenamento para manter a estrutura do índice. Índices permitem localizar dados rapidamente sem precisar buscar sequencialmente em cada linha de uma tabela.

A maioria dos softwares de banco de dados inclui tecnologia de indexação que permite buscas em tempo sub-linear para melhorar o desempenho, já que a busca linear é ineficiente para grandes bancos de dados.[1]

1.1.1 Busca por Semelhança

No caso de *dados não estruturados*, como bancos de dados multimídia e dados gerados por extração de características, a busca realizada não tem as mesmas características da busca exata. Ao invés disso, deseja-se encontrar os elementos mais semelhantes de um elemento de consulta.

O tema de busca por semelhança[2] se desenvolveu em diversas áreas de forma independente. A grande quantidade de dados não estruturados com representações complexas originou uma gama de métodos distintos. Existe até uma conferência específica para o tema, SISAP.

Nesse tipo de cenário, o conceito de espaço métrico serve como modelo para o problema. Um espaço métrico é um conjunto X com uma função de distância $d : X \times X \to \mathbb{R}$ que satisfaz as seguintes propriedades:

1. $d(x, y) \geq 0$ (não-negatividade)
2. $d(x, y) = 0 \iff x = y$ (identidade dos indiscerníveis)

3. $d(x, y) = d(y, x)$ (simetria)

4. $d(x, z) \leq d(x, y) + d(y, z)$ (desigualdade triangular)

   A simple corollary of triangle inequality is that, if any two objects within the space are far apart, then no third object can be close to both

A última restrição é usada em algumas técnicas espaciais para dividir o espaço. No entanto, existem métodos com outros princípios que podem relaxar a 4a condição, dando origem a uma *semimétrica*.

Example 1.1.1. O espaço euclidiano $\mathbb{R}^n$ com a norma L^p, tal que

$$d(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$$

e para o caso $p = \infty$,

$$d(x, y) = \max_{1 \leq i \leq n} |x_i - y_i|$$

Diante disso, cada elemento do banco de dados é um elemento de X e quanto menor a distância entre dois objetos, $d(x, y)$, maior a similaridade entre eles.

Note também que a busca por similaridade toma lugar em um espaço multidimensional (section 1.3) e são de dois tipos

Range Query recupera todos os elementos até uma distância r da query q, i.e.,

$$\{x \in X : d(x, q) \leq r\}$$

k-Nearest Neighbor Search (k-NNS) recupera os k elementos mais próximos de q em X, i.e., retornar $Y \subset X$ tal que $|Y| = k$ e $\forall y \in Y, \forall x \in X \setminus Y, d(y, q) \leq d(x, q)$.

Solução Força Bruta k-NNS

A maneira mais simples de fazer isso é realizando a comparação de todos os elementos do conjunto e retornar os k mais próximos, com complexidade $O(n)$.

No entanto, considerando o custo associado ao cálculo da distância, a escalabilidade linear é uma limitação para grandes conjuntos de dados. Talvez com uso de GPU, paralelismo, seja aplicável em determinados cenários.

Algorithm 1 Busca por Força Bruta

```
1: procedure BruteForceSearch(X, q, k)
2:     Y ← ∅
3:     for x ∈ X do
4:         if |Y| < k then
5:             Y ← Y ∪ {x}
6:         else
7:             y ← arg min_{y∈Y} d(y, q)
8:             if d(x, q) < d(y, q) then
9:                 Y ← Y \ {y} ∪ {x}
10:            end if
11:        end if
12:    end for
13:    return Y
14: end procedure
```

1.2 Implementações de índices

1.2.1 B Tree

Uma *B-tree* é uma estrutura de dados em árvore auto-balanceada que mantém dados ordenados e permite buscas, acessos sequenciais, inserções e deleções em tempo logarítmico. A B-tree generaliza a árvore binária de busca, permitindo nós com mais de dois filhos.

É amplamente utilizada em sistemas de arquivos e bancos de dados. É uma estrutura que se beneficia da leitura e escrita em bloco, levando vantagem em um aspecto historicamente relevante, uma vez que o número de operações de I/O (em discos magnéticos) era igualmente relevante para o desempenho quanto o número de operações de comparação.

Foi inventada por Rudolf Bayer e Edward M. McCreight em 1972 [3] (o B não foi explicado por eles).

> *What Rudy (Bayer) likes to say is, the more you think about what the B in B-Tree means, the better you understand B-Trees!*

Os principais algoritmos associados a B-trees são: busca (algorithm 2) e inserção (algorithm 3) (existem variações para a operação de deleção).

São necessárias duas funções auxiliares para a inserção: splitChild, que divide um nó cheio em dois, e insertNonFull, que insere uma chave em um nó não cheio.

Bulk loading?

Algorithm 2 Algoritmo de busca na B Tree, assumindo que a chave k é o valor a ser buscado e x é o nó onde a busca começa.

```
1: procedure BTREESEARCH(x, k)
2:     i ← 0
3:     while i < x.n and k > x.key[i] do
4:         i ← i + 1
5:     end while
6:     if i < x.n and k = x.key[i] then
7:         return x
8:     end if
9:     if x.leaf then
10:        return None
11:    end if
12:    return BTREESEARCH(x.child[i], k)
13: end procedure
```

Algorithm 3 Algoritmo de inserção na B Tree, assumindo que a chave k é o valor a ser inserido.

```
1: procedure BTREEINSERT(T, k)
2:     r ← T.root
3:     if r.n = 2(T.d) − 1 then
4:         s ← new Node
5:         T.root ← s
6:         s.child[1] ← r
7:         SPLITCHILD(s, 1)
8:         INSERTNONFULL(s, k)
9:     else
10:        INSERTNONFULL(r, k)
11:    end if
12: end procedure
```

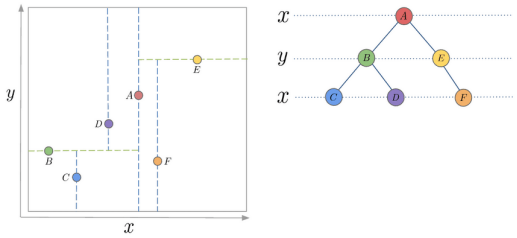


Figure 1.1: Exemplo de uma KD-Tree.

[3]–[5]

1.2.2 B+ Tree

Uma B+ tree pode ser vista como uma B-tree onde cada nó contém apenas chaves (não pares chave-valor), com um nível adicional de folhas ligadas na parte inferior.

O principal valor de uma B+ tree está no armazenamento de dados para recuperação eficiente em um contexto de armazenamento orientado a blocos, como sistemas de arquivos. Diferente das árvores binárias de busca, as B+ trees têm um fanout muito alto (número de ponteiros para nós filhos em um nó, tipicamente na ordem de 100 ou mais), o que reduz o número de operações de I/O necessárias para encontrar um elemento na árvore.

Aplicações: iDistance

[6]

1.3 Indexação Multidimensional

Exemplo de como estava sendo realizada a indexação multidimensional em multimidia(imagens)

Efficient and Effective Querying by Image Content

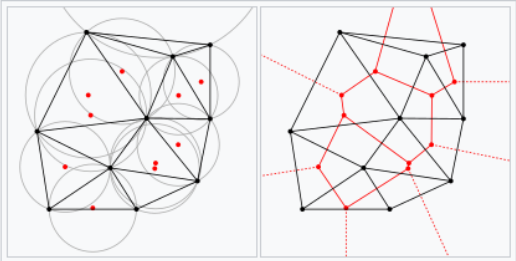
Survey [7], [8]

1.3.1 KD-Tree

fig. 1.1

1975 [9], [10]

$O(\log n)$ em dimensões 2 e 3.



The Delaunay triangulation with all the circumcircles and their centers (in red).

Connecting the centers of the circumcircles produces the Voronoi diagram (in red).

1.3.2 R-Tree

Usada para multidimensional

1.3.3 M-Tree

Usada para espaços métricos[11]

1.3.4 SA Tree

Space Approximation Tree**searching:navarro2002**

Delaunay[12] e a relação com a busca gulosa no grafo dual do diagrama de voronoi

1.4 The Curse of Dimensionality

O termo "*curse of dimensionality*" é atribuído a Richard Bellman, que se referiu isso no contexto de programação dinâmica e o impacto do número de estados na performance do algoritmo. É o fenômeno relacionado ao aumento da dificuldade/complexidade de uma tarefa com o aumento da dimensionalidade dos dados. Isso significa que a tarefa pode ser realizada de forma satisfatória em dimensões baixas, mas tem performance muito pior ou se torna inviável em dimensões altas. Existem outras definições, como uma associada ao "*fenômeno de Hughes*" [13], o qual diz que, com um número fixo de amostras de treino, o poder preditivo de um modelo aumenta inicialmente com o aumento de dimensões (parâmetros), mas depois de um certo ponto, começar a deteriorar. Mas em geral o termo é utilizado para descrever dificuldades encontradas ao entrar em um contexto com alta dimensionalidade.

1.4.1 Conceito de Dimensão

Dimensão intrínseca(i.e., the real number of dimensions in which the points can be embedded while keeping the distances among them)[8]

1.4.2 Impacto na Busca por Similaridade

O conceito de dimensão está associado, em problemas de busca por similaridade, a dificuldade da busca.

Pensando na distribuição de pontos, é possível fazer uma pequena análise que demonstre a esparsidade dos dados em altas dimensões. Para isso, calcularemos a razão de volumes em uma esfera inscrita em um cubo. A princípio, a razão entre os volumes é de

$$\left(\frac{4}{3}\pi r^3\right)\frac{1}{8r^3} \approx 0.52$$

conforme a dimensão d aumenta, a razão

$$\left(\frac{2r^d\pi^{d/2}}{d\Gamma(d/2)}\right)\frac{1}{(2r)^d} \to 0$$

indicando que o volume da hiperesfera se torna insignificante em relação ao volume do hipercubo. [14]

Em particular, no contexto de espaços métricos, as medidas de distância entre pares de pontos colapsam, fazendo com que a distância entre todos os pontos seja aproximadamente a mesma. Dessa maneira, descartar elementos com uma distância maior que a média, em um cenário com alta dimensionalidade, descarta poucos elementos. fig. 1.2

Na kd-tree, a complexidade é $O(dN^{1-1/k})$[15].

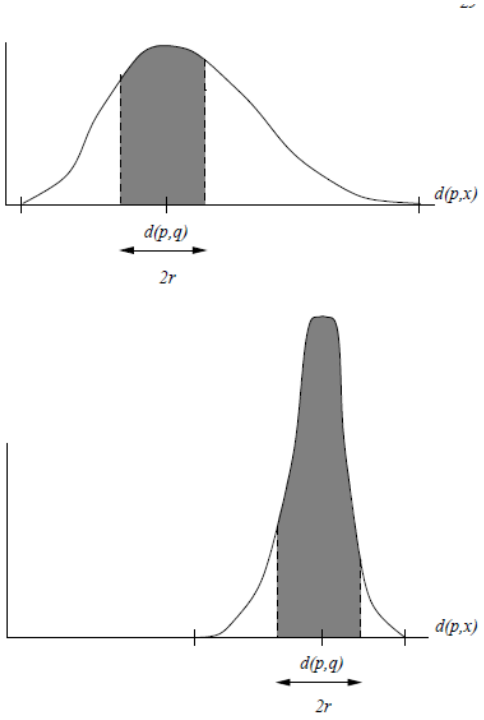


Figure 1.2: Ilustração da "*curse of dimensionality*" na distribuição de distâncias. **searching:navarro2002**

Bibliography

[1] Wikipedia, *Database index*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Database_index.

[2] Wikipedia, *Similarity search*, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Similarity_search.

[3] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, 1970, pp. 107–141.

[4] Wikipedia, *B tree*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/B-tree.

[5] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[6] Wikipedia, *B+ tree*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/B%2B_tree.

[7] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Computing Surveys (CSUR)*, vol. 30, no. 2, pp. 170–231, 1998.

[8] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín, "Searching in metric spaces," *ACM computing surveys (CSUR)*, vol. 33, no. 3, pp. 273–321, 2001.

[9] Wikipedia, *K-d tree*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/K-d_tree.

[10] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[11] P. Ciaccia, M. Patella, P. Zezula, *et al.*, "M-tree: An efficient access method for similarity search in metric spaces," in *Vldb*, Citeseer, vol. 97, 1997, pp. 426–435.

[12] Wikipedia, *Delaunay triangulation*, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Delaunay_triangulation.

[13] G. Hughes, "On the mean accuracy of statistical pattern recognizers," *IEEE transactions on information theory*, vol. 14, no. 1, pp. 55–63, 1968.

[14] Wikipedia, *Curse of dimensionality*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Curse_of_dimensionality.

[15] D.-T. Lee and C.-K. Wong, "Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees," *Acta Informatica*, vol. 9, no. 1, pp. 23–29, 1977.

Chapter 2

Indexação para Fingerprints

2.1 Introdução

- Matching local baseado em minúcias
  - Abordagens antigas [1], [2]
  - Associa cada minúcia as suas vizinhas em estruturas invariantes a rotação e distâncias[3], [4]
  - Baseada em cilindros, veja section 2.2
  - Outros métodos que incluem mais características: local orientation field, local frequency, ridge shapes

Estruturas locais de uma minúcia central podem ser baseadas em:

- *Vizinhos mais próximos*, que consideram as k minúcias mais próximas [3].

  A vantagem dessa representação é com relação ao tamanho fixo, facilitando no procedimento de comparação.

- *Raio fixo*, que considera todas as minúcias dentro de um raio fixo, usada em [4].

  A vantagem dessa representação é a tolerância com relação a ruído (minúcias extras ou faltantes).

2.2 Minutia Cylinder-Code

Baseada em [5], [6]

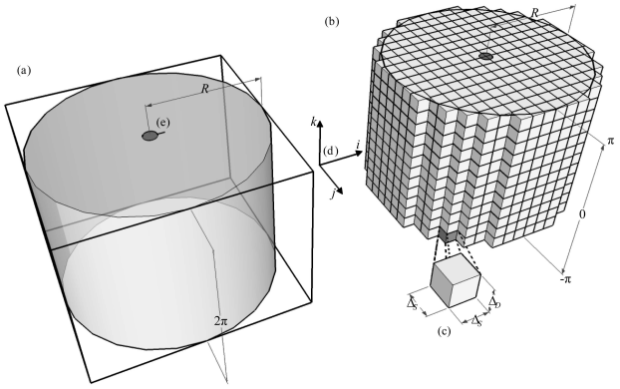


Figure 2.1: Representação de um MCC.

2.2.1 Representação

Uma representação tridimensional de minúcias baseada em distâncias entre minúcias e ângulos relativos. A representação recebe o nome de Minutia Cylinder-Code (MCC) e tem como características principais: invariância de rotação, tamanho fixo e orientada a codificação binária.

O esquema de representação é um cilindro segmentado, como na fig. 2.1. O cilindro é um recorte de um cubo dividido em células, onde cada uma possui um valor indexado por $C_m[i, j, k]$.

O cálculo dos valores de $C_m[i, j, k]$ é complicado, mas essencialmente envolve as seguintes ideias:

- Verifica se está em uma região válida: dentro do cilindro e dentro do *convex hull* da fingerprint.
- Calcula a contribuição de cada minúcia vizinha usando uma Gaussiana, fig. 2.2.
- Calcula a contribuição de cada minúcia usando a diferença entre a orientação.

2.2.2 Similaridade

A similaridade entre dois cilindros é obtida a partir do seguinte procedimento:

1. Lineariza o cilindro em um vetor, similar a operação de `reshape`. Por exemplo, o cilindro de uma minúcia a, $C_a[i, j, k]$, é linearizado em $\mathbf{c}_a$.
2. Seleciona todas as entradas comparáveis desses vetores (células que são *válidas* em ambos) que dão origem aos vetores $\tilde{\mathbf{v}}_a$ e $\tilde{\mathbf{v}}_b$.

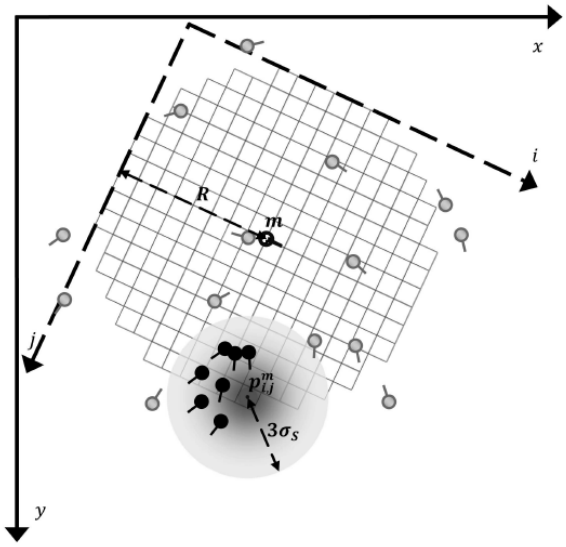


Figure 2.2: Contribuição da distância de uma minúcia vizinha. Cores mais escuras refletem uma maior contribuição.

3. Na implementação binária, é realizado um XOR bit a bit entre os vetores.

$$\gamma(\tilde{\mathbf{v}}_a, \tilde{\mathbf{v}}_b) = 1 - \frac{\|\tilde{\mathbf{v}}_a \oplus \tilde{\mathbf{v}}_b\|}{\|\tilde{\mathbf{v}}_a\| + \|\tilde{\mathbf{v}}_b\|} \tag{2.1}$$

2.2.3 Indexação

Para a indexação de um MCC é empregado um esquema de *locality-sensitive hashing* (LSH), [7], [8].

O procedimento consiste em:

1. Seja $\mathbf{v}_m$ o vetor obtido da linearização de um cilindro.

2. É feita a projeção de $\mathbf{v}_m$ em um subespaço $\mathbf{h}_m$. A projeção é obtida simplesmente ao escolher um subconjunto dos índices H do vetor original.

$$\mathbf{h}_m = \mathbf{v}_m[H] \tag{2.2}$$

3. O conjunto H define uma função que mapeia um vetor binário $\mathbf{v}_m$ em um número natural obtido ao interpretar o vetor binário $\mathbf{h}_m$ como um número natural.

$$h_H : \{0,1\}^n \to \mathbb{N} \tag{2.3}$$

4. São definidos ℓ conjuntos $H_1, H_2, \ldots, H_\ell$, cada um com uma função h_{H_i}.

5. O índice por sua vez é um conjunto de *hash tables*, $\mathbb{H}_1, \mathbb{H}_2, \ldots, \mathbb{H}_\ell$, onde cada hash table tem os seus buckets definidos pela função h_{H_i}.

6. A indexação segue fazendo a consulta do vetor desejado em cada hash table, retornando os candidatos.

7. Por fim, os candidatos são ranqueados usando a *distância de Hamming* entre os vetores.

O procedimento enumerado acima é ilustrado na fig. 2.3.

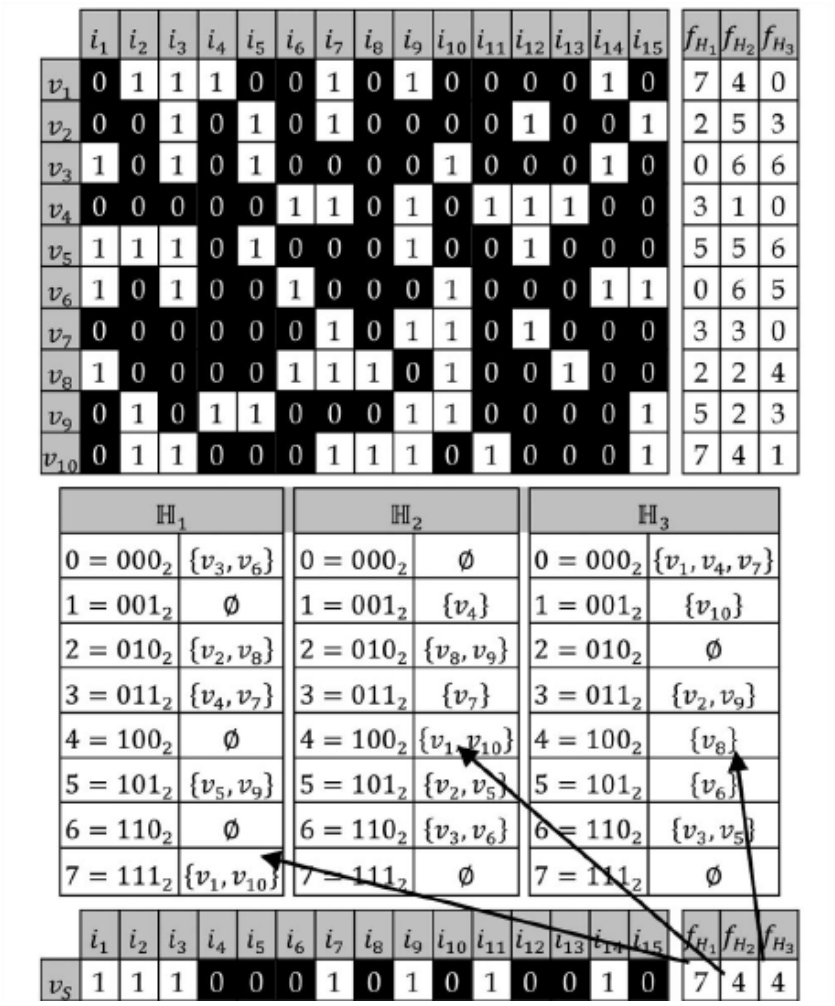


Figure 2.3: Ilustração do procedimento de indexação usando LSH de um MCC.

Bibliography

[1] A. K. Hrechak and J. A. McHugh, "Automated fingerprint recognition using structural matching," *Pattern recognition*, vol. 23, no. 8, pp. 893–904, 1990.

[2] A. J. Willis and L. Myers, "A cost-effective fingerprint recognition system for use with low-quality prints and damaged fingertips," *Pattern recognition*, vol. 34, no. 2, pp. 255–270, 2001.

[3] X. Jiang and W.-Y. Yau, "Fingerprint minutiae matching based on the local and global structures," in *Proceedings 15th international conference on pattern recognition. ICPR-2000*, IEEE, vol. 2, 2000, pp. 1038–1041.

[4] N. K. Ratha, R. M. Bolle, V. D. Pandit, and V. Vaish, "Robust fingerprint authentication using local structural similarity," in *Proceedings Fifth IEEE Workshop on Applications of Computer Vision*, IEEE, 2000, pp. 29–34.

[5] R. Cappelli, M. Ferrara, and D. Maltoni, "Minutia cylinder-code: A new representation and matching technique for fingerprint recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 32, no. 12, pp. 2128–2141, 2010.

[6] R. Cappelli, M. Ferrara, and D. Maltoni, "Fingerprint indexing based on minutia cylinder-code," *IEEE transactions on pattern analysis and machine intelligence*, vol. 33, no. 5, pp. 1051–1057, 2010.

[7] A. Gionis, P. Indyk, R. Motwani, *et al.*, "Similarity search in high dimensions via hashing," in *Vldb*, vol. 99, 1999, pp. 518–529.

[8] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 604–613.

Chapter 3

Vector Databases

3.1 Introdução

Os chamados *Vector Databases* (VDB) surgem em um cenário no qual *dados não estruturados* são dominantes em praticamente todas as esferas e as previsões apontam para um crescimento contínuo e acelerado. Exemplos de dados dessa forma são aqueles que não podem ser armazenados de forma padronizada em tabelas, como imagens, vídeos, áudios e textos.

Para contrapor a definição de dados não estruturados, entende-se por *dados estruturados* aqueles que podem ser colocados em forma de tabelas, como aqueles armazenados em bancos de dados relacionais. Define-se também os *dados semi-estruturados*, que são aqueles que possuem uma estrutura menos rígida, como arquivos XML e JSON.

A mudança de paradigmas apresentadas pelos dados não estruturados diz respeito a como deve ser feita a interpretação desses dados. Diferentes imagens de cachorros são objetivamente diferentes analisando o seu conteúdo bruto (valor dos pixels), mas todas as imagens apresentam uma *similaridade semântica*. O desafio passa a ser então como representar, armazenar e realizar busca nesses dados, de forma que eles reflitam esse aspecto.

A abordagem que segue é a de representar os dados por meio de métodos de *aprendizado profundo*, gerando a partir de um pedaço de dado não estruturado um vetor multidimensional, chamado de *embedding*. Observa-se na literatura que uma rede neural bem treinada é capaz de capturar a semântica desses dados e, portanto, a similaridade entre eles é refletida por uma medida de distância entre os vetores.

3.1.1 Aplicações

...

3.2 Divisão dos métodos

- Busca exata
  - Linear Scan
  - Space Partitioning (KD-Tree, Ball-Tree, Inverted File Index)
- Busca aproximada
  - Cluster-based
  - Graph-based
  - Tree-based
  - Hash-based
- Quantização
  - Scalar Quantization
  - Product Quantization

3.3 Métodos exatos

3.3.1 Linear Scan

O algoritmo de busca linear, *flat indexing*, calcula a distância entre a query Q e todos os elementos do conjunto de dados $\mathcal{D}$, retornando o elemento mais próximo. A complexidade desse algoritmo é $O(n)$, onde n é o número de elementos em $\mathcal{D}$, tornando-o muitas vezes impraticável em cenários reais. O algoritmo é apresentado no algorithm 4.

3.3.2 Particionamento de Espaço

Métodos exatos de busca podem ser vistos em section 1.3.

O que acontece nesses caso é a maldição da dimensionalidade. Sendo assim, métodos de busca aproximados devem ser utilizados, trocando um pouco de precisão por eficiência.

Algorithm 4 Algoritmo de busca linear de uma query Q em um conjunto de dados $\mathcal{D}$.

```
1: procedure LinearScan(Q, D)
2:     best_dist ← ∞
3:     best_match ← None
4:     for d ∈ D do
5:         dist ← dist(Q, d)
6:         if dist < best_dist then
7:             best_dist ← dist
8:             best_match ← d
9:         end if
10:    end for
11:    return best_match
12: end procedure
```

3.4 Métodos baseados em clusterização

3.4.1 Inverted File Index (IVF)

A estratégia de IVF é baseada em dividir o espaço em partições, representadas a partir do centroide de todos os vetores que pertencem a essa partição.

Os centroides de cada partição, ou *clusters*, são determinados pelo algoritmo de clusterização *k-means*. O funcionamento desse algoritmo consiste em: inicialmente selecionar aleatoriamente k vetores do conjunto de dados e atribuir a cada um deles um cluster; em seguida adicionar a cada cluster os vetores mais próximos a ele; atualizar o novo centroide de cada cluster; repetir o processo até a convergência [1], [2].

A busca, dessa forma, pode ser limitada somente aos vetores que pertencem a uma partição ou conjunto de partições, reduzindo o espaço de busca, como ilustra algorithm 5.

Algorithm 5 Algoritmo de busca usando o IVF de uma query Q em um índice $\mathcal{I}$.

procedure IVFSEARCH($\mathcal{I}, query, top\_k, nprobe$)
    $distances \leftarrow []$
    for all $c \in \mathcal{I}.centroids$ **do**
        $d \leftarrow \|query - c\|$
        $distances \leftarrow distances \cup \{(c, d)\}$
    end for
    $candidateCentroids \leftarrow$ **selecionar os** $nprobe$ **menores** d
    $candidateVectors \leftarrow \{\}$
    for all $ctr \in candidateCentroids$ **do**
        $candidateVectors \leftarrow candidateVectors \cup \mathcal{I}[ctr]$
    end for
    $scored \leftarrow []$
    for all $v \in candidateVectors$ **do**
        $dist \leftarrow \|query - v\|$
        $scored \leftarrow scored \cup \{(v, dist)\}$
    end for
    ordenar $scored$ **por** $dist$ **ascendente**
    return primeiros top_k **de** $scored$
end procedure

3.5 Métodos baseados em árvores

3.5.1 sa-tree

3.5.2 Approximate Nearest Neighbors Oh Yeah (ANNOY)

3.6 Métodos baseados em grafos

3.6.1 Navigable Small World

A princípio[3], [4] o autor, Yury Malkov, estava interessado no problema de vizinhos mais próximos em uma configuração distribuída.

Quais as vantagens apontadas para essa escolha?

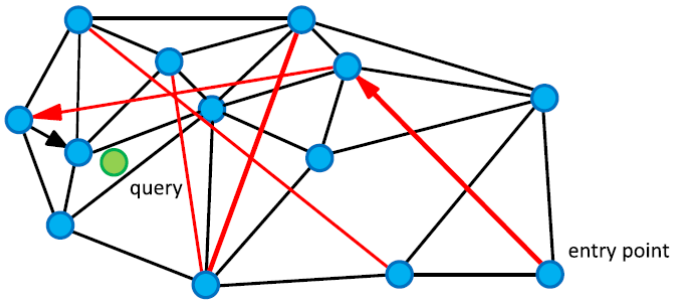
- Existem algoritmos para construir essas redes,
- Não tem elementos raízes,
- Operações de adição e busca usam somente informações locais e podem ser iniciadas de qualquer elemento.

> *This gives an opportunity for building decentralized similarity search oriented storage systems where physical data location doesn't depend on the content because every data object can be placed on an arbitrary physical machine and can be connected with others by links like in p2p systems.*

Uma das técnicas de busca em grafos com vértices contendo elementos de um espaço métrico é a busca gulosa. Nesse caso, para que a busca seja exata, i.e., encontra sempre os verdadeiros vizinhos mais próximos, é necessário que o grafo contenha o grafo de Delaunay como subgrafo (dual a tesselação de Voronoi). As limitações da busca exata estão associadas a necessidade de conhecimento prévio da estrutura interna do espaço**searching:navarro2002** e a maldição da dimensionalidade [5]. Caso a condição de busca exata seja relaxa, é necessário conter somente uma aproximação do grado de Delaunay.

Para que a busca gulosa tenha escalabilidade logarítmica, a rede small world deve ter a propriedade de ser *navegável*.[6]

As arestas E da rede são divididas em dois grupos: arestas de baixo alcance, que aproximam o grafo de Delaunay e responsáveis pelo resultado do roteamento guloso, e as arestas de longo alcance, que servem para fornecer a escalabilidade logarítmica da busca gulosa devido a propriedade de navigabilidade.



Busca

O algoritmo algorithm 6 encontra o objeto no conjunto de dados mais próximo ou um falso mais próximo. Caso todo o elemento na estrutura tivesse em sua lista de vizinhos todos os vizinhos no sentido de Voronoi, isso seria a condição para o grafo de Delaunay e da busca exata.

Algorithm 6 Busca gulosa em busca de um mínimo local em um grafo Small World.

```
1:  procedure GreedySearch(q, ep)
2:      curr ← ep
3:      dmin ← d(q, curr)
4:      next ← Null
5:      for all n ∈ curr.neighbors do
6:          if d(q, n) < dmin then
7:              dmin ← d(q, n)
8:              next ← n
9:          end if
10:     end for
11:     if next = Null then
12:         return curr                          ▷ Reached local minima
13:     else
14:         return GreedySearch(q, next)
15:     end if
16: end procedure
```

Uma estratégia para diminuir o erro, fig. 3.1 é realizar um conjunto de m buscas independentes, começando de pontos distintos, e retornar o melhor resultado. Caso a probabilidade de encontrar o mínimo global seja p, então a probabilidade de encontrar o mínimo global em pelo menos uma das buscas é $1-(1-p)^m$, indicando que a probabilidade de falha decai exponencialmente com o número de buscas.

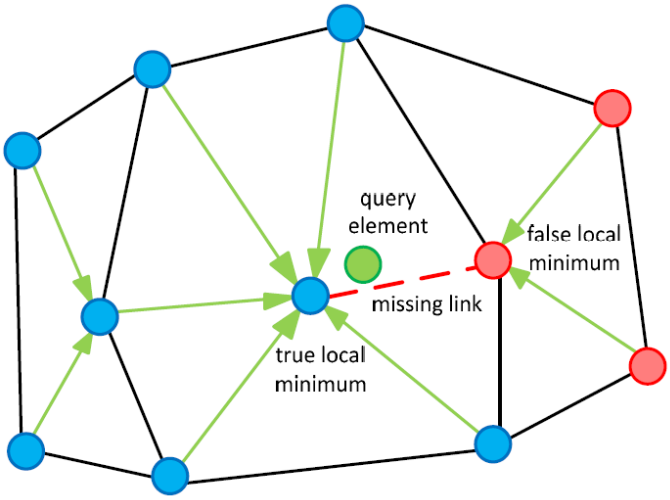


Figure 3.1: Caso ilustrativo de um falso positivo na busca gulosa.

Algorithm 7 MultiSearch

```
1: procedure MultiSearch(q, m)
2:     candidatos ← []
3:     for i ← 1 to m do
4:         ep ← getRandomEntryPoint
5:         result ← GreedySearch(q, ep)
6:         candidatos ← candidatos ∪ {result}
7:     end for
8:     ordenar candidatos por d(q, ·) ascendente
9:     return candidatos
10: end procedure
```

Em [4], o algoritmo é refinado. Nova condição de parada: as iterações ocorrem somente em não visitados, para quando o próximo elemento não altera os k mais próximos.

Construção

Sabendo que o objetivo é construir somente uma aproximação do grafo de Delaunay, existe grande liberdade na etapa de construção. Existe um método que busca minimizar o volume das células de Voronoi por Monte Carlo. O método proposto é construir o grafo de forma iterativa, uma adição por vez.

Usando a ideia de MultiSearch, podemos fazer o seguinte procedimento, ilustrado em algorithm 8, obter um conjunto de mínimos locais a partir de m buscas, ordenar os resultados pela distância com o objeto inserido e conectar os k mais próximos.

Algorithm 8 Adição de um novo objeto ao grafo Small World.

```
1: procedure kNN-Add(q, m, k)
2:     localMins ← MultiSearch(q, m)
3:     vizinhos ← []
4:     for all lm ∈ localMins do
5:         vizinhos ← vizinhos ∪ {lm}
6:     end for
7:     ordenar vizinhos por d(q, ·) ascendente
8:     for i ← 1 to k do
9:         Connect(q, vizinhos[i])
10:    end for
11: end procedure
```

A ideia por trás disso é que a intersecção entre os vizinhos de Voronoi e os k vizinhos mais próximos deve ser grande. Além disso, com dados chegando de forma aleatória, a característica de navigabilidade é obtida sem passos adicionais (os primeiros elementos tendem a gerar os links de longo alcance).

Resultados

- Verifica propriedade de navigabilidade do Small World: medir o tamanho médio do caminho percorrido durante a busca. De fato, é logaritmico com relação ao tamanho do dataset.
- Número de buscas do MultiSearch: Para manter uma taxa de acerto de $p = .95$, mostrou-se que $m > A \log n$. Isso mostra que o número de

buscas, durante a fase de construção, para obter um valor alto de p cresce de forma logarítmica.

- Número de conexões (k): Aproximadamente $3d$, que mantém baixa taxa de erro para uma baixa taxa de elementos visitados durante a busca.
- Fração de elementos visitados: Diminui com aumento do dataset. A relação entre número de elementos e porcentagem de visitados é, em escala log-log, uma reta com inclinação de -45 graus, porcentagem = $-kn^{-1}$.
- Complexidade da busca (tamanho dataset): É $\log^2 n$, sendo um "log" do tamanho do caminho e outro do número de "buscas".
- Complexidade da busca (dimensão): É $d^{1.7}$.

No geral, complexidade de busca

$$O(d^{1.7} \log^2 n \log(1/p_{\text{fail}}))$$

de construção é

$$O(d^{1.7} n \log^2 n)$$

3.6.2 Hierarchical Navigable Small World (HNSW)

Em grafos de proximidade, sofrem de problemas de escalabilidade por lei de potência e possível desconexão global em dados clusterizados. Soluções envolvendo uma etapa de busca grosseira (coarse search) usando outras estruturas para definir o ponto de entrada foram sugeridas.

Nos trabalhos anteriores, um grafo de proximidade com a característica de navegabilidade (grafo NSW) foi proposto. Essas estruturas apresentam escalabilidade logarítmica ou polilogarítmica no número de saltos durante a busca com relação ao tamanho do grafo. O método é baseado na construção incremental da estrutura, sempre ligando um novo elemento ao conjunto de k elementos mais próximos. Para encontrar os k mais próximos, são feitas múltiplas buscas a partir de pontos de entrada diferentes. Conexões antigas, servem como conexões de longo alcance que permite a escalabilidade logarítmica.

No entanto, a escalabilidade polilogarítmica do número de cálculos de distância na busca era uma limitação. Essa escalabilidade é devido a: $\langle \#\text{ hops} \rangle \times \langle \text{node degree} \rangle$. O primeiro termo é logaritmico como discutido e o segundo

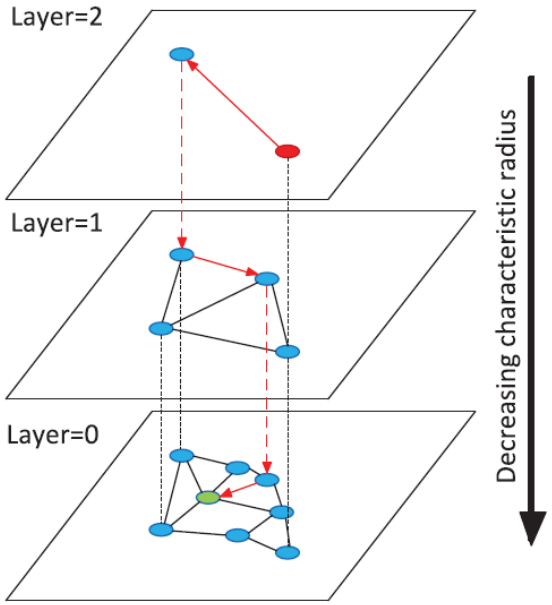


Figure 3.2: Hierarchical Navigable Small World.

também, devido ao fato do número de hubs crescer logaritmicamente com o tamanho do grafo.

Para melhorar o método,

- começar em um nó com alto grau, para evitar ficar preso em um mínimo local. Os nós iniciais são ideais para isso e aumentam a chance de um roteamento bem sucedido.
- dividir o grafo segundo a distância das conexões. Dessa maneira, fixando o número de conexões avaliadas para um número constante, permitindo o tempo geral logaritmico.

A essência desse novo método[7] é começar na camada mais superior, que apresentam os maiores links, realizar uma busca gulosa até um mínimo local e continuar a busca a partir desse mesmo elemento na camada inferior, repetindo o processo, fig. 3.2.

Para construir essa estrutura por camadas é sortear o nível máximo que um elemento vai ocupar usando uma distribuição exponencial. A criação de links também usa uma heurística para diversificar as conexões.

Construção

algorithm 9.

$O(n \log n)$

Algorithm 9 Inserção de um elemento.

```
1: procedure INSERT(q, M, M_max, efConstruction, m_L)
2:     NN ← []
3:     ep ← hsnw.ep
4:     L ← ep.level
5:     l ← ⌊−m_L ln U(0, 1)⌋
6:     for l_c ← L to l + 1 do        ▷ Camadas nas quais não está presente...
7:         NN ← SEARCHLAYER(q, ep, ef = 1, l_c)
8:         ep ← nearest element in NN to q
9:     end for
10:    for l_c ← min(l, L) to 0 do
11:        NN ← SEARCHLAYER(q, ep, efConstruction, l_c)
12:        vizinhos ←SELECTNEIGHBORS(q, NN, M, l_c)
13:        CONNECT(q, vizinhos) no nível l_c
14:        for all v ∈ vizinhos do
15:            if v.neighbors(l_c) > M_max then
16:                novos ← SELECTNEIGHBORS(v, v.neighbors(l_c), M_max, l_c)
17:                v.neighbors(l_c) ← novos
18:            end if
19:        end for
20:        ep ← NN                    ▷ ep pode ser uma lista de elementos
21:    end for
22:    if L < l then
23:        hsnw.ep ← q
24:    end if
25: end procedure
```

Busca

algorithm 10.

A complexidade de uma única busca pode ser obtida considerando que construímos um grafo de Delaunay exato e, então, o elemento selecionado em um dado nível é o mais próximo. Ao selecionar um elemento, a probabilidade dele estar no próximo nível é $p = \exp(-m_L)$. No entanto, durante a busca nesse nível, se são realizados s passos, então esses s elementos não podem estar no nível acima, pois caso contrário eles seriam retornados.

Agora considerando que para alcançar o melhor elemento são necessários s passos, então a probabilidade de não alcançar o elemento é limitada por

$p^s = \exp(-m_L s)$. O número esperado ???
$O(\log n)$

Algorithm 10 Busca dos ef elementos mais próximos de uma query q em uma camada.

```
1: procedure SearchLayer(q, ep, ef, l_c)
2:     visitados ← {ep}
3:     candidatos ← {ep}
4:     NN ← []
5:     while candidatos ≠ ∅ do
6:         proximo ← pop candidatos
7:         distante ← NN.last
8:         if ||q − proximo|| > ||q − distante|| then
9:             break
10:        end if
11:        for v ∈ proximo.neighbors(l_c) do
12:            if v ∉ visitados then
13:                visitados ← visitados ∪ {v}
14:                distante ← NN.last                ▷ Atualiza o mais distante
15:                if ||q − v|| < ||q − distante|| or NN.size < ef then
16:                    candidatos ← candidatos ∪ {v}
17:                    NN ← NN ∪ {v}
18:                    if NN.size > ef then
19:                        remove o mais distante de NN
20:                    end if
21:                end if
22:            end if
23:        end for
24:    end while
25:    return NN
26: end procedure
```

Escolhendo os vizinhos

A maneira mais simples é escolher os vizinhos mais próximos. A outra é adotando uma heurística fig. 3.3 que cria conexões em direções diversas, levando em conta a distância entre candidatos. A heurística analisa os candidatos, começando do mais próximo, e escolhe para fazer a conexão somente aqueles que são mais próximos do objeto inserido q do que dos outros candidatos inseridos.

Algorithm 11 Seleção de vizinhos via heurística.

1: **procedure** SELECTNEIGHBORS($q, candidatos, M, l_c$)
2: todo...
3: **end procedure**

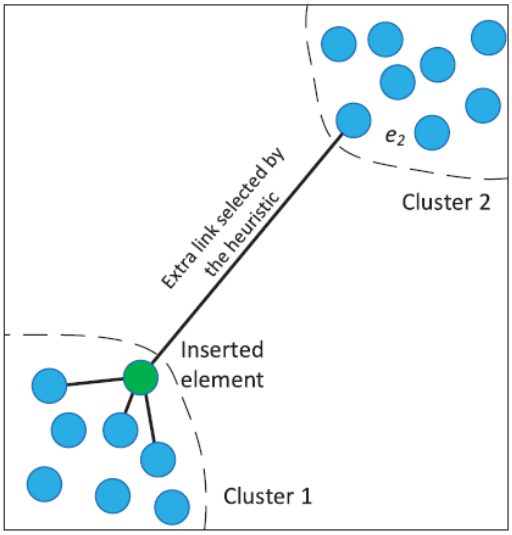


Fig. 2. Illustration of the heuristic used to select the graph neighbors for two isolated clusters. A new element is inserted on the boundary of Cluster 1. All of the closest neighbors of the element belong to the Cluster 1, thus missing the edges of Delaunay graph between the clusters. The heuristic, however, selects element e_0 from Cluster 2, thus, maintaining the global connectivity in case the inserted element is the closest to e_0 compared to any other element from Cluster 1.

Figure 3.3: Heurística de seleção de vizinhos.

Parâmetros

Os parâmetros que dão as características principais dos métodos quando comparados aos anteriores são m_L e $M_{\max}$. Se $m_L = 0$ e $M_{\max} = M$, é um grafo kNN com escalabilidade dada por lei de potência. Se $m_L = 0$ e $M_{\max} = \infty$ é um grafo NSW com escalabilidade polilogarítmica. Ao escolher $m_L > 0$, passa a ser possível controlar a hierarquia do grafo, com a escalabilidade logarítmica.

3.7 Métodos baseados em hashing

3.7.1 LSH

A popular approach for similarity search is locality sensitive hashing (LSH). It hashes input items so that similar items map to the same "buckets" in memory with high probability (the number of buckets being much smaller than the universe of possible input items).

3.8 Quantização

A quantização busca diminuir o tamanho do banco de dados, representando os vetores por suas representações quantizadas. Note que isso é diferente do método de redução de dimensionalidade, como (PCA, t-SNE, UMAP).

3.8.1 Scalar Quantization

Transforma vetores de floats em vetores de inteiros. Para cada dimensão, o método busca todo o alcance e divide essa faixa uniformemente em bins. Se queremos armazenar o inteiro em um `uint8`, então temos $2^8 = 256$ bins.

3.8.2 Product Quantization

Dado um vetor que possui d bits. Cada vetor é dividido em m subvetores, cada um com d/m bits. Em seguida, para todos os subvetores, é feita uma clusterização por k-means. Substituímos todos os subvetores por seus respectivos centroides.

3.9 Opções comerciais

Bibliography

[1] Wikipedia, *k-means clustering*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/K-means_clustering.

[2] Wikipedia, *Lloyd's algorithm*, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Lloyd%27s_algorithm.

[3] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Scalable distributed algorithm for approximate nearest neighbor search problem in high dimensional general metric spaces," 2012.

[4] Y. Malkov, A. Ponomarenko, A. Logvinov, and V. Krylov, "Approximate nearest neighbor algorithm based on navigable small world graphs," *Information Systems*, vol. 45, pp. 61–68, 2014.

[5] F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure," *ACM Computing Surveys (CSUR)*, vol. 23, no. 3, pp. 345–405, 1991.

[6] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," in *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, 2000, pp. 163–170.

[7] Y. A. Malkov, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE transactions on pattern analysis and machine intelligence*, vol. 42, no. 4, pp. 824–836, 2018.