Universidade Estadual de Londrina

MATHEUS BARBIERO BASTOS

ÍNDICE BASEADO EM APRENDIZADO DE MÁQUINA PARA BUSCAS POR SIMILARIDADE

LONDRINA
2022

MATHEUS BARBIERO BASTOS

ÍNDICE BASEADO EM APRENDIZADO DE MÁQUINA PARA BUSCAS POR SIMILARIDADE

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Daniel dos Santos Kaster

LONDRINA

2022

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração Automática do Sistema de Bibliotecas da UEL

Bastos, Matheus Barbiero.
Índice Baseado em Aprendizado de Máquina para Buscas por Similaridade / Matheus Barbiero Bastos. - Londrina, 2022.
61 f. : il.

Orientador: Daniel dos Santos Kaster.
Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Graduação em Ciência da Computação, 2022.
Inclui bibliografia.

1. Buscas por similaridade. - TCC. 2. Índices basedos em grafos. - TCC. 3. Aprendizado de máquina. - TCC. 4. Buscas em espaços métricos. - TCC. I. dos Santos Kaster, Daniel. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Graduação em Ciência da Computação. III. Título.

CDU 519

MATHEUS BARBIERO BASTOS

ÍNDICE BASEADO EM APRENDIZADO DE MÁQUINA PARA BUSCAS POR SIMILARIDADE

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

Orientador: Prof. Dr. Daniel dos Santos Kaster
Universidade Estadual de Londrina

Prof. Dr. Wesley Attrot
Universidade Estadual de Londrina – UEL

Me. Larissa Capobianco Shimomura
Universidade Tecnológica de Eindhoven – TU/e

Londrina, 03 de junho de 2022.

AGRADECIMENTOS

Agradeço primeiramente ao professor Daniel por ser um orientador fantástico, que sempre esteve presente durante o desenvolvimento deste trabalho e dispôs de seu incrível conhecimento e experiência para me guiar na melhor direção. Agradeço também à minha família e à minha namorada Maria por me apoiarem imensamente durante todo o curso com tudo que precisei, sempre me envolvendo com muito carinho e afeto. Por fim, agradeço aos colegas e amigos que fiz durante esses anos, principalmente ao Melvi e ao Vinicius, pois aprendemos muito juntos e certamente levarei essas amizades para toda a vida.

BASTOS, M. B.. **Índice baseado em aprendizado de máquina para buscas por similaridade**. 2022. 61f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2022.

RESUMO

Dados complexos são cada vez mais armazenados em bancos de dados, demandando técnicas eficientes para fazer consultas sobre eles. Esses dados são geralmente representados em espaços métricos e as consultas são buscas por similaridade. Um exemplo desse tipo de consulta é a dos k vizinhos mais próximos, que retorna os k objetos mais similares em todo o conjunto de dados. Para acelerar a busca, índices são empregados, nesse caso bem diferentes dos tradicionais. Métodos baseados em grafos têm sido boas alternativas para diminuir o número de cálculos necessários, assim reduzindo o tempo de busca, mas ainda podem ser melhorados. Mais recentemente, *learned indexes* surgiram com o uso de aprendizado de máquina para se aproveitar da distribuição de dados e de outras características, obtendo resultados promissores. Este trabalho tem o objetivo de criar um índice eficiente para acelerar métodos baseados em grafos para buscas por similaridade. Este índice tem como princípio a predição dos vértices iniciais de busca usando aprendizado supervisionado. Com isso, o número de computações de distância necessárias para uma alta taxa de *recall* ser atingida diminuiu em vários *datasets* testados. Os resultados indicam uma melhora geral na qualidade das buscas quando comparado com a seleção aleatória de vértices iniciais. Também é evidente que essa é uma ideia promissora e que pode ser desenvolvida ainda mais.

Palavras-chave: Buscas por similaridade. Índices basedos em grafos. Aprendizado de máquina. Buscas em espaços métricos.

BASTOS, M. B.. **Machine-learning-based index for similarity searching**. 2022. 61p. Final Project (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2022.

ABSTRACT

With the increase of complex data that is used and stored, efficient techniques to query this type of data are demanded. Complex data are often represented in metric spaces and queried through similarity searches, such as the k-nearest neighbors query, which retrieves the k most similar objects in the entire dataset. To speed up the complex data retrieval, indexes for this specific type of data are used, since traditional indexing techniques don't support similarity searches. Graph-based methods showed promising results in decreasing the number of calculations needed, consequently decreasing the search time, but they can be improved. Recently, learned indexes have emerged with new concepts, like using machine learning to take advantage of the data distribution and other characteristics, showing promising results. This work aims to create an efficient learned index to speed up graph-based methods for similarity searching. Our method is based on using supervised learning to predict the seed vertex. This method was able to reduce the number of distance computations needed to obtain high recall rates on various datasets. The results show that the search quality was improved overall when compared to random vertex selection. Moreover, this is a promising idea that can be further explored.

Keywords: Similarity searching. Graph-based indexes. Machine learning. Metric space searching.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

RGB	*Red, Green, Blue*
kNNq	*k-Nearest Neighbors query*
Rq	*Range query*
NSW	*Navigable Small World*
k-NNG	*k-Nearest Neighbors Graph*
SAT	*Spatial Approximation Tree*
RNG	*Relative Neighborhood Graph*
GNNS	*Graph Nearest Neighbor Search*
CDF	*Cumulative Distribution Function*
RMI	*Recursive Model Index*
LMI	*Learned Metric Index*
RAM	*Random-access Memory*

SUMÁRIO

1 INTRODUÇÃO

Em bancos de dados, técnicas de indexação de dados vêm sendo pesquisadas há muitas décadas. Por exemplo, a B-Tree [3], uma das estruturas de dados usadas em índices mais conhecidas, foi proposta em 1970. Apesar da área já ter sido explorada de diversas formas, técnicas novas surgem frequentemente na literatura, ao passo que a taxa de armazenamento (e consumo) de dados no mundo cresce exponencialmente [4].

Estudos clássicos no assunto abordam principalmente dados estruturados (ou escalares), em que os objetos têm seus atributos tratados como dimensões independentes. Eles podem ser números ou pequenas cadeias de caracteres [5]. Seus registros podem ser ordenados, possibilitando consultas como busca exata, busca parcial e por intervalo. A criação de um índice em um conjunto de dados desse tipo geralmente envolve a ordenação de seus elementos, muitas vezes empregando algoritmos baseados em busca binária.

Por outro lado, mais recentemente aplicações que usam dados complexos viram sua utilização crescer, assim como a necessidade de armazenar esses dados. Eles possuem muito menos estrutura definida e sua especificação é menos precisa. Exemplos incluem imagens, sons e dados georreferenciados. Diferentemente dos estruturados, eles não podem ser ordenados e comparações exatas não fazem tanto sentido [6]. Por exemplo, a busca exata por uma imagem não seria útil na maioria dos casos (a não ser que se estivesse verificando a existência exata de uma dada imagem), pois até a variação em um único pixel, mesmo que imperceptível ao olho humano, causaria diferenças na representação dos dados. No caso desse tipo de dados, buscas por similaridade são mais adequadas, que retornam, por exemplo, qual são os k elementos mais similares a um objeto.

Assim como nos dados estruturados, a construção de índices acelera consultas em conjuntos de dados complexos. Porém, o desafio neste caso é um pouco diferente devido à maneira em que os dados são representados: são representados através **vetores de características**, que tipicamente são conjuntos de valores escalares [6]. Esses valores representam diferentes características do objeto, por exemplo, cor e forma no caso de uma imagem. Assim, a similaridade entre dois objetos pode ser medida através da aplicação de uma função de distância.

Para realizar a tarefa de busca por proximidade, várias estruturas de índice foram propostas na literatura, como o M-Index [7], a Slim-tree [8] e algoritmos baseados em grafos de proximidade [9, 10]. Nota-se que muitos desses trabalhos focam na busca *aproximada* dos vizinhos mais próximos, que é, no geral, mais eficiente que a exata [11]. A troca de exatidão por desempenho é um *trade-off* aceitável em muitos casos de uso.

Entre os métodos baseados em grafos, o ***HGraph*** [1] tem grande importância

para este trabalho. Ele é um método para construção de grafos de proximidade, como grafos *k-NN*. A construção é feita através de uma estratégia de divisão e conquista, que cria várias partições. Elas são conectadas através de regiões de sobreposição e *long-range edges*, arestas criadas posteriormente que aumentam a qualidade das consultas. Com o grafo construído, um algoritmo baseado em *aproximação espacial* [12] pode ser utilizado para realizar consultas sobre ele.

Um problema muito grande nos algoritmos de busca baseados em aproximação espacial é quando o vértice inicial está "muito longe" do resultado ideal. Dependendo do tamanho do grafo, essa condição causa uma necessidade de percorrer uma região muito grande do grafo [1]. Para contornar esse problema, é necessário determinar os vértices iniciais de busca de maneira mais adequada.

Mais recentemente, *learned indexes* surgiram como fortes alternativas ou complementos a estruturas tradicionais [13, 14, 15, 16]. Utilizando modelos de aprendizado de máquina, muitos deles conseguem se aproveitar de padrões existentes nos dados para acelerar consultas. O *Learned Metric Index* [13], por exemplo, pode se basear em índices como o M-Index [7] para construir sua própria estrutura constituída por modelos, eliminando cálculos de distância em passos intermediários da busca.

Inspirando-se nos *learned indexes* presentes na literatura, principalmente no LMI [13], este trabalho tem o objetivo de apresentar um novo *learned index* baseado em grafos construídos por particionamento, como é o caso do *HGraph*. A ideia principal é aproveitar a estrutura do grafo para treinar modelos que terão o papel de determinar os vértices iniciais das consultas por similaridade. Outro objetivo é validar que, com a predição, as buscas são iniciadas em regiões mais adequadas e necessitam de menos cálculos de distância para atingir o mesmo resultado.

Para tal, uma estrutura auxiliar ao *HGraph* foi implementada utilizando duas abordagens: (1) utilizando um único modelo e (2) utilizando uma hierarquia de modelos. Além disso, foram feitas comparações do novo método com o original, ao mesmo tempo que os parâmetros envolvidos tiveram seus impactos analisados.

No capítulo 2, a fundamentação teórica do trabalho é detalhada. No capítulo 3, é apresentado o problema e os trabalhos relacionados a este trabalho. No capítulo 4, é introduzido o método de predição com sua implementação e diferentes abordagens. Por fim, no capítulo 5, são apresentados e discutidos os resultados dos experimentos realizados com o *learned index*.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 Buscas por similaridade

Tradicionalmente na computação, buscas em conjuntos de dados são feitas em um domínio cujos valores são naturalmente ordenáveis. São valores como nomes, que podem ser organizados em ordem alfabética, e datas, que podem seguir a ordem cronológica. Nesses domínios, é sempre possível, dados dois elementos, responder qual deles é o anterior (de acordo com certo critério). Seguindo esse princípio de ordenação dos elementos, existem inúmeras técnicas que otimizam as consultas feitas sobre esses conjuntos de dados. Como exemplo, um dos algoritmos mais triviais que atua sobre um vetor ordenado é a busca binária, a qual é eficiente na tarefa de encontrar um elemento específico. Porém, nem todos os dados são ordenáveis.

Diferentemente dos números, não faz sentido tomar um conjunto de imagens e ordená-las (sem considerar metadados como nome e tamanho, apenas o aspecto visual delas). Também, não se faz buscas por exatidão: uma simples diferença em um *pixel* já seria capaz de tornar duas imagens diferentes [17]. Por outro lado, pode-se estabelecer critérios para determinar o quão semelhantes elas são. Dados como imagens, sons e sequências genéticas são denominados **dados complexos**. Neles, não é factível empregar algoritmos tradicionais de busca, mas sim **buscas por similaridade**. Ao invés de realizar consultas como "encontre o elemento Q" ou "encontre os elementos entre P e Q", são feitas buscas como "encontre o elemento mais similar a Q", por exemplo.

2.1.1 Conceitos básicos

Para realizar buscas por similaridade em um domínio, é preciso definir uma maneira de comparar dois elementos quaisquer pertencentes a ele. O primeiro passo é selecionar características de interesse que serão extraídas dos objetos. Elas são representadas em um **vetor de características** [18]. Algoritmos geradores desses vetores em imagens, por exemplo, são chamados de **descritores de imagens**. Um descritor de imagens simples pode tomar uma imagem RGB e colocar em sequência suas três camadas de cores, gerando uma saída unidimensional que pode ser considerada um vetor de características. Outro exemplo mais trabalhado seria gerar um histograma das cores presentes na imagem, caracterizando sua distribuição. Na maioria dos casos, essa segunda opção teria muito menos valores, o que é interessante para fins de comparação.

O próximo passo é definir uma **função de distância**, que indica de forma quantitativa o quão distantes são dois elementos, ou seja, o nível de dissimilaridade deles. Um resultado próximo de zero representa grande similaridade, enquanto que valores maiores

ocorrem em pares mais dissimilares [19]. Formalmente, uma função de distância em um domínio $\mathbb{S}$ é definida por: $d : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{R}^+$.

Na literatura, há várias funções de distância, sendo que a mais adequada depende do domínio de aplicação. Por serem genéricas e adequadas para comparar vetores de características, as mais empregadas são as da família Minkowski [20]. A distância Minkowski de ordem p está representada na Relação 2.1, onde n é o tamanho do vetor de características, X e Y são vetores da forma $X = \{x_1, x_2, ..., x_n\}$ e $Y = \{y_1, y_2, ..., y_n\}$, e $1 \leq p < \infty$.

$$L_p(X, Y) = \sqrt[p]{\sum_{k=1}^{n} |x_k - y_k|^p} \tag{2.1}$$

De todos os possíveis valores para p, alguns dos mais utilizados são: $p = \infty$, correspondente à função L_∞ ou L_0, conhecida como distância de Chebyshev; $p = 1$, correspondente à L_1, chamada de distância de Manhattan; e $p = 2$, correspondente à L_2, a distância Euclidiana. Essa última é a mais próxima do senso comum de distância, pois representa a distância entre dois pontos em uma linha reta. A Figura 1 ilustra a diferença entre essas funções no espaço, exibindo os pontos equidistantes a um elemento central u em cada uma delas.

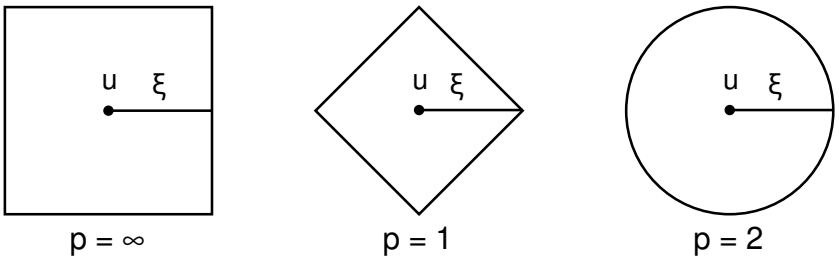


Figura 1 – Ilustração do formato de pontos equidistantes a um elemento central em um espaço bidimensional considerando as distâncias L_∞, L_1 e L_2.

2.1.1.1 Espaço métrico

Em alguns casos, os vetores de características criados podem ter comprimentos grandes, constituindo um espaço com alta dimensionalidade. Nesses espaços, ocorrem vários fenômenos chamados de "maldição da dimensionalidade" [21]. Entre os fenômenos, um dos principais é o fato de que conforme a dimensionalidade aumenta, os dados se tornam cada vez mais esparsos. Representações em espaços euclidianos, com objetivo de realizar otimizações na consulta baseadas em regras do espaço, também perdem cada vez mais o sentido. Por esse motivo, dados complexos muitas vezes são representados em **espaços métricos**, constituídos somente pelo conjunto dos dados (nesse caso, os vetores de características) e a função de distância. Diferentemente de espaços euclidianos, não há a noção de posição nos elementos.

Formalmente, um espaço métrico é um par $(\mathbb{S}, d)$, onde $\mathbb{S}$ é o domínio e d é a função de distância. Também, para quaisquer elementos $x, y, z \in \mathbb{S}$, as seguintes propriedades são mantidas [19]:

- $d(x, y) \geq 0$ (não-negatividade);
- $d(x, y) = 0 \Leftrightarrow x = y$ (identidade);
- $d(x, y) = d(y, x)$ (simetria);
- $d(x, y) \leq d(x, z) + d(y, z)$ (desigualdade triangular).

Essa última propriedade, a da desiguldade triangular, é fundamental para técnicas que visam otimizar o número de computações de distância feitas em uma consulta. Por exemplo, ela pode ser usada para determinar a impossibilidade de determinado elemento no conjunto de dados fazer parte da resposta, tornando desnecessários cálculos de distância a ele.

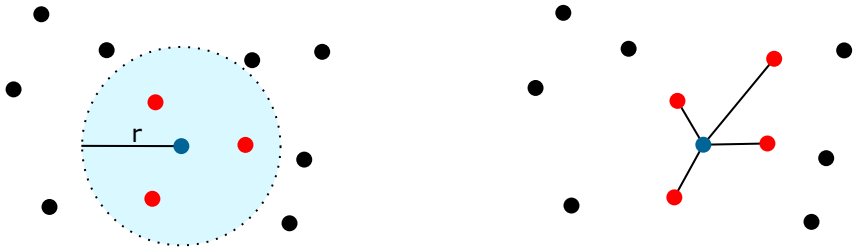
2.1.2 Tipos de consultas por similaridade

Em conjuntos de dados complexos, são feitas consultas por similaridade para buscar dados de interesse. Elas consistem em selecionar elementos do conjunto que satisfaçam determinado critério de similaridade. Existem dois tipos de consulta por similaridade fundamentais: as consultas aos *k-vizinhos mais próximos* ($kNNq$) e as consultas por abrangência ou *range queries* (Rq) [22].

Seja $\mathbb{S}$ um domínio de dados, $S \subseteq \mathbb{S}$ o conjunto dos dados, $q \in \mathbb{S}$ um elemento de consulta e d uma função de distância definida sobre $\mathbb{S}$:

- a consulta $kNNq(q, k)$ encontra os k elementos de menor distância a q que estão presentes no conjunto S. Formalmente, o resultado da consulta K é definido por $K = \{s \in S \mid \forall t \in S \setminus K, |K| = k, d(q, s) \leq d(q, t)\}$.
- a consulta $Rq(q, r)$, com $r \in \mathbb{R}^+$, encontra todos os elementos no conjunto S cuja distância para q é menor ou igual a r. Formalmente, o resultado da consulta é dado por $\{s \in S \mid d(q, s) \leq r\}$.

As duas consultas são ilustradas em um espaço euclidiano bidimensional na Figura 2, onde os elementos de consulta estão em azul e os elementos retornados nas consultas estão em vermelho. É interessante notar que a consulta Rq pode retornar nenhum elemento, e seu parâmetro r deve ser definido levando em consideração o domínio dos dados e a função de distância empregada. Enquanto isso, a $kNNq$ é mais independente do conjunto e garante que ao menos um elemento será retornado.



(a) Rq (b) kNNq

Figura 2 – Exemplo de Rq com raio r e $kNNq$ com $k = 4$.

2.1.3 Métodos de acesso aos dados

Uma maneira trivial para se realizar uma busca por similaridade, como a *kNNq* apresentada na subseção anterior, é comparar cada objeto do conjunto de dados com o elemento de busca, retornando os que satisfazem as condições desejadas. Essa estratégia resolve o problema, porém não de maneira eficiente: funções de distância geralmente são custosas [23], e executá-las para todos os elementos é inviável, principalmente em espaços com alta dimensionalidade e muitos dados. Com isso em mente, várias técnicas de busca surgiram para evitar ao máximo computações de distância. Pode-se englobá-las em algumas categorias [24], sendo as principais apresentadas a seguir.

- Baseadas em *hashing*: a função *hash* tem o papel de agrupar elementos similares, tendo a probabilidade de colisão muito maior nesses casos [25]. Um método conhecido é o *locality-sensitive hashing* [26].

- Baseadas em árvores: as estruturas dos índices são árvores, utilizando diferentes técnicas de particionamento hierárquico. Exemplos incluem a *M-Tree* [27] e a *Slim tree* [8].

- Baseadas em grafos: O espaço de similaridade é modelado como um grafo. Geralmente, os elementos do conjunto de dados são representados como vértices. Alguns exemplos são o NSW [9] e o k-NNG [10].

Na literatura, encontra-se vários métodos baseados em árvore e em grafos por serem genéricos e poderem ser empregados em vários domínios diferentes. Eles são capazes de particionar o conjunto de dados de maneira similar, mas há uma limitação inerente com as árvores [28]: a busca começa pela raiz, e quando um caminho errado é escolhido, não há como corrigir o erro em camadas mais inferiores. Portanto, é necessário realizar *backtracks*, forçando com que a árvore inteira ou partes dela sejam percorridas novamente. Vários *backtracks* geralmente ocorrem em uma única busca, percorrendo um caminho levemente diferente em cada um [29].

Esse problema tem relação com as estruturas hierárquicas globais presentes nas árvores [29], as quais impossibilitam a resposta de ser encontrada (sem *backtracks*) caso a subárvore incorreta esteja sendo percorrida, mesmo tendo elementos próximos. Os métodos baseados em grafos evitam essa característica, permitindo que buscas iniciadas em vértices mais próximos do elemento de consulta, no geral, exijam poucos cálculos de distância [30]. Esse fator é chave para este trabalho, e é devido a ele que o foco será dado em métodos baseados em grafos.

2.2 Grafos de proximidade

Um grafo G é definido por $G(V, E)$, onde V é um conjunto de vértices e E é um conjunto de arestas que conectam pares de vértices de V. Grafos de proximidade são grafos comumente utilizados para a realização de consultas por similaridade [30]. Neles, uma propriedade P, chamada de *critério de vizinhança*, é definida, e cada par de vértices $u, v \in V$ é conectado por uma aresta $e \in E$ se, e somente se, eles satisfizerem a propriedade P. As arestas podem ou não ter pesos, e caso tenham ele geralmente é a distância entre os dois elementos correspondentes aos vértices [23]. Esses grafos refletem a proximidade entre os elementos conectando objetos espacialmente próximos.

Uma das maneiras mais reconhecidas de se responder consultas de similaridade com um grafo de proximidade é através da *aproximação espacial*, definida por Navarro [31]. Sendo $N(u)$ os vizinhos de u, inicia-se o processo em um elemento qualquer de V, que é atribuído à variável a. Em cada passo, é escolhido um dos vizinhos $b \in N(a)$, o qual deve ter menos distância ao elemento de consulta do que a e todos os seus outros vizinhos, e atribuído b à variável a. O processo é repetido até que não seja possível escolher um vizinho de a com esse critério, o que significa que o nó atual é o mais próximo. A Figura 3 mostra um exemplo do caminho percorrido por uma aproximação espacial.

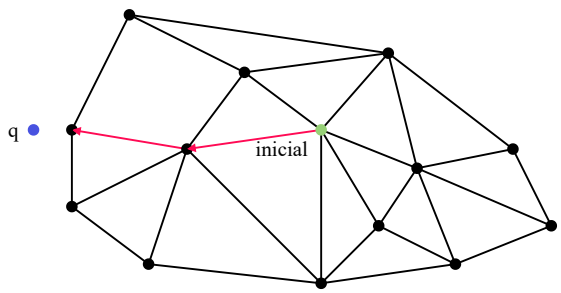


Figura 3 – Exemplo de aproximação espacial em um grafo de proximidade, iniciando-se em um vértice arbitrário e tendo q como objeto de consulta.

O grafo mais simples onde esse algoritmo funciona é o grafo completo, onde todos os vértices são adjacentes a todos os outros vértices. Porém, tal estrutura é extremamente

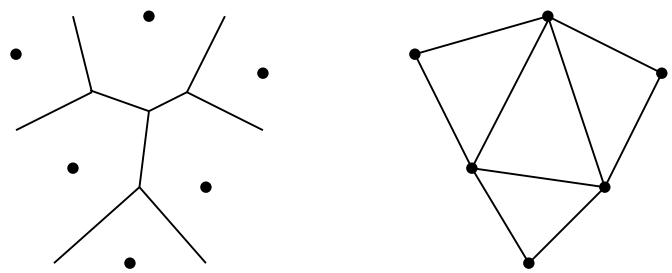
ineficiente, pois cada iteração requer $|V|$ cálculos de distância. O objetivo, portanto, é obter um grafo com o menor número de arestas possível, sem deixar de responder as consultas corretamente. Tal grafo[1] $G(V, \{(a, b), a \in V, b \in N(a)\})$, representando um espaço métrico $(\mathbb{S}, d)$, onde $V \in \mathbb{S}$, deve respeitar a seguinte propriedade [31]:

$$\forall a \in V, \forall q \in \mathbb{S}, \text{ se } \forall b \in N(a), d(q, a) \leq d(q, b), \text{ então } \forall c \in \mathbb{S}, d(q, a) \leq d(q, c) \qquad (2.2)$$

Isso significa que, para determinado elemento q, se não for possível se aproximar de q a partir de a escolhendo algum de seus vizinhos $b \in N(a)$, a é o elemento mais próximo de q em todo o conjunto V.

2.2.1 Exemplos de grafos de proximidade

Um exemplo de grafo que satisfaz a propriedade de aproximação espacial é o da triangulação de Delaunay. Nele, os vértices que são vizinhos no diagrama de Voronoi são conectados. O diagrama de Voronoi divide o espaço em subrregiões, onde cada uma é correspondente a um ponto p do conjunto de dados e qualquer ponto nela tem como elemento mais próximo o ponto p [31]. Uma visualização do mesmo conjunto de pontos para o diagrama de Voronoi e o grafo de Delaunay se encontra na Figura 4.



(a) Diagrama de Voronoi (b) Grafo de Delaunay

Figura 4 – Diagrama de Voronoi e grafo de Delaunay com os mesmos pontos.

Esse grafo, se generalizado para espaços arbitrários, seria o ideal para o método de aproximação espacial, contendo o número mínimo de arestas enquanto satisfaz a propriedade necessária. No entanto, ele é adequado apenas para espaços euclidianos, ao passo que Navarro mostra que o grafo de Delaunay pode ser o grafo completo em espaços métricos [31]. Ainda, em dados com alta dimensionalidade, o Delaunay e extensões propostas a ele sofrem com a *maldição da dimensionalidade*, rapidamente se tornando um grafo completo conforme o número de dimensões aumenta [1].

Como uma alternativa que engloba espaços métricos, Navarro propõe a *Spatial Approximation Tree* (SAT), que é um caso particular do grafo de Delaunay [31]. Para

[1] e qualquer outro onde se deseja realizar aproximação espacial

construí-la, primeiramente um elemento aleatório $a \in V$ é selecionado para se tornar a raiz da árvore. Então, os nós seguintes são inseridos de acordo com a propriedade da aproximação espacial. O restante dos elementos é ordenado de acordo com a distância para a, e então eles começam a ser inseridos como filhos de a, mas com uma regra: o elemento deve ter sua distância para a menor do que sua distância para qualquer outro elemento $b \in N(a)$ já inserido anteriormente. Esse processo é repetido recursivamente até que todos os elementos estejam na árvore.

Por ser uma árvore, a escolha da raiz é crítica na SAT, e uma raiz ruim pode levar à exploração de uma parte muito significativa da árvore, conforme já foi discutido anteriormente. Ocsa et al. [30] sugerem o uso de *Relative Neighborhood Graphs* [32] (RNG). Um RNG é construído conectando vértices u e v por uma aresta quando não existe um vértice w mais próximo a ambos u e v do que eles são entre eles. Por serem subgrafos dos grafos de Delaunay, a propriedade de aproximação espacial é parcialmente garantida neles. Ao mesmo tempo, podem ser construídos com apenas as distâncias entre os elementos do conjunto de dados disponíveis, sendo adequados para espaços métricos. Apesar disso, sua construção é muito custosa, com sua complexidade de tempo sendo $O(n^3)$ [32].

Outro grafo de proximidade muito usado é o *k-Nearest Neighbor Graph* (*k-NNG*) [33]. Nele, cada vértice é conectado com os k elementos mais próximos a ele no conjunto de dados, de acordo com determinada função de distância. A complexidade de tempo para construção desse grafo, utilizando um algoritmo trivial de força bruta, é $O(n^2)$. Isso não é aceitável para grandes bases de dados, motivando várias pesquisas focadas em algoritmos mais eficientes para construção de *k-NNGs* [34, 35].

2.3 O método de construção *HGraph*

O *HGraph* é um método que visa acelerar o tempo de construção de qualquer tipo de grafo de proximidade proposto na literatura, incluindo o *k-NNG*, ao mesmo tempo que melhora a qualidade dos resultados através da adição de *arestas de longa distância* [1]. Essas arestas não seguem o critério de proximidade estabelecido no grafo, e conectam os vértices selecionados no processo de particionamento do *HGraph*, consequentemente conectando as regiões do grafo gerado.

O método está no contexto de buscas *aproximadas* por similaridade, que são consultas que retornam resultados aproximados. Elas são motivadas pelo alto custo de se produzir respostas exatas para buscas por similaridade, tendo o objetivo de reduzir o tempo gasto com o mínimo de erro possível. Para aumentar a velocidade, os algoritmos têm a permissão de diminuir a qualidade da resposta, podendo retornar elementos que não fazem parte do resultado exato. Isso geralmente não é um problema em espaços métricos, considerando que muitas vezes a imprecisão é inerente aos dados [13].

A fundamentação do *HGraph* se dá em duas partes: (1) uma estratégia de divisão e conquista para construir qualquer tipo de grafo, dividindo o conjunto de dados recursivamente até que uma certa cardinalidade de um subconjunto seja atingida; (2) a adição de arestas de longa distância a vértices selecionados do grafo, que são os pivôs em cada partição recursiva. Isso é feito seguindo o critério de vizinhança do grafo cada vez que um conjunto é dividido [1]. Uma visão geral do *HGraph* pode ser vista na Figura 5.

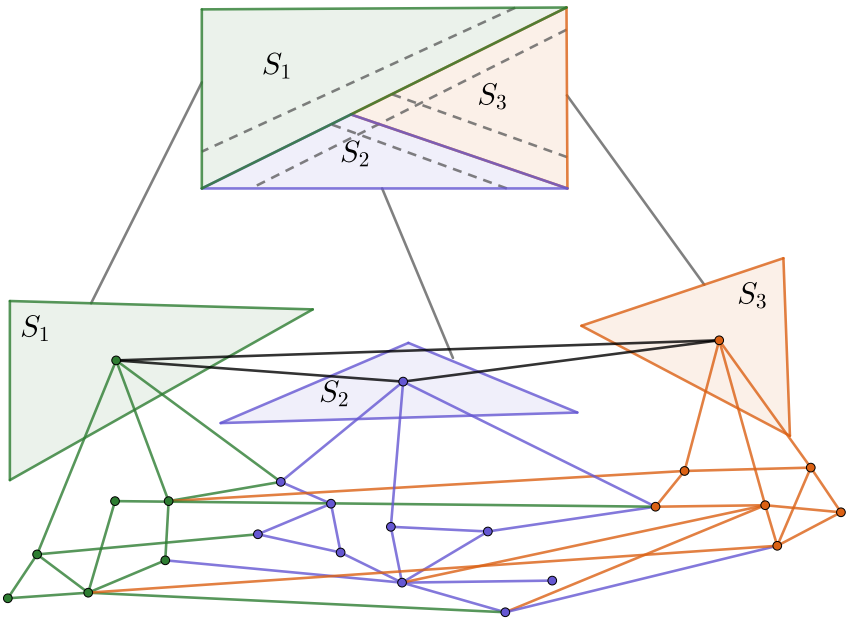


Figura 5 – Visão geral do *HGraph*, exibindo partições que se sobrepõem e que um grafo é construído em cada uma delas. Extraído do trabalho original [1].

Nesta seção, o método é apresentado de forma breve, dando ênfase nas partes mais importantes para este trabalho. Mais detalhes sobre o processo de construção e resultados obtidos podem ser encontrados no trabalho dos autores [1].

2.3.1 Método de construção

O *HGraph* possui alguns parâmetros de construção:

- o número de pivôs n_P;
- a estratégia de seleção de pivôs *ptype*;
- o número mínimo de elementos em cada grafo m;
- a taxa de sobreposição o;
- o tipo de grafo que será construído no nível folha, assim como seus parâmetros de construção, g_1;
- outro tipo de grafo g_2, que será utilizado para um refinamento feito nos pivôs.

Conforme já foi mencionado, o *HGraph* se baseia em uma estratégia de divisão e consquista. Seu objetivo é construir um grafo próximo ao g_1 de forma (potencialmente) paralelizada e com algumas características que acabam gerando benefícios para consultas feitas no grafo resultante, muitas vezes reduzindo o tempo de busca e aumentando a qualidade dos resultados.

A ideia base do método de construção é dividir o conjunto de dados recursivamente (através de estratégias que serão discutidas a seguir) até que as partições tenham m ou menos elementos. Em cada uma dessas partições resultantes, é construído um grafo g_1 com seus elementos. Os grafos construídos são conectados por meio de arestas de longa distância e objetos duplicados por estarem em regiões de sobreposição.

2.3.1.1 Particionamento

O particionamento do conjunto de dados começa selecionando n_P elementos que serão usados como pivôs de acordo com a estratégia de seleção *ptype*. Então, o conjunto é dividido em n_P subconjuntos usando a abordagem de partições por *hiperplanos generalizados* [36]. Primeiro, calcula-se a distância de cada elemento do conjunto para cada um dos pivôs selecionados. Então, atribui-se cada elemento ao subconjunto do pivô para qual a distância é a menor.

Os pivôs selecionados são conectados entre si, criando arestas denominadas *arestas de longa distância*. Apenas essas arestas não são suficientes para tornar o grafo final suficientemente conectado. Dois elementos de diferentes subconjuntos não ficarão conectados mesmo que sejam muito similares, causando um grande problema para algoritmos de busca baseados em aproximação espacial, que produzirão resultados com menor qualidade ou levarão mais tempo para executar. Por esse motivo, são criadas regiões de sobreposição entre os subconjuntos.

2.3.1.2 Região de *overlap*

Pensando no problema mencionado, elementos que estiverem em uma região de sobreposição entre dois subconjuntos são duplicados de modo a ficarem nas duas partições. Desse modo, arestas podem ser feitas entre os vértices das bordas sem afetar a independência das partições no processamento. Os autores do *HGraph* consideram duas maneiras principais para definir as regiões de sobreposição: bolas ou hiperplanos [1].

A abordagem utilizando bolas funciona tomando a região de intersecção entre as bolas com centro nos pivôs das partições vizinhas. Dessa maneira, o tamanho da região resultante é determinado pelos raios das bolas. Por outro lado, o método por hiperplanos cria um hiperplano imaginário que divide as partições vizinhas, definido pelos seus pivôs. Então, a região de sobreposição é dada pelos pontos cuja distância para o hiperplano é menor ou igual a um valor definido [1]. Uma ilustração da diferença na construção

utilizando as duas abordagens, antes da duplicação dos elementos, pode ser vista na Figura 6.

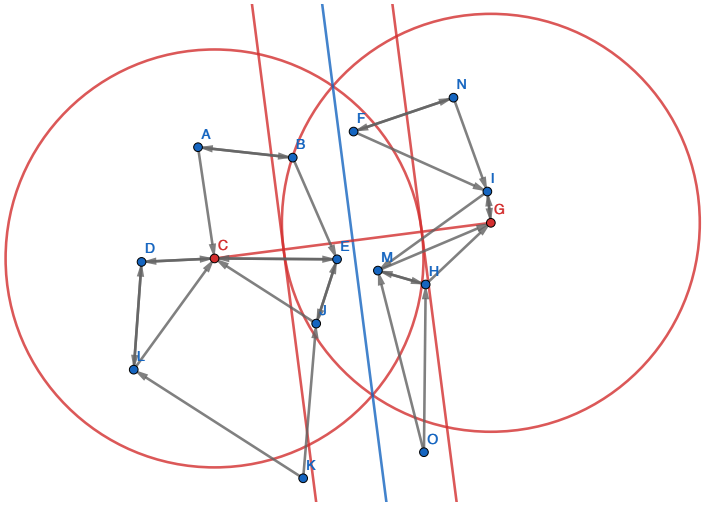


Figura 6 – Construção de um 2-NNG através do *HGraph* sem região de sobreposição. É possível perceber que, utilizando a abordagem das bolas, o elemento O ficaria de fora. Figura extraída do trabalho original [1].

O trabalho dá preferência à segunda abordagem (hiperplanos) [1]. Entre os motivos para tal, é que a área abrange mais do que apenas elementos com pouca distância ao pivô, como acontece na outra abordagem. Porém, por se tratar de um espaço métrico, o hiperplano não é real, então ele deve ser aproximado. Isso é feito considerando a diferença das distâncias para os pivôs. Também, para garantir que sempre exista no mínimo um elemento na região de sobreposição, os autores optaram por definir uma proporção do conjunto que sempre estará nessa região. Assim, com o parâmetro de taxa de *overlap* (o), o número de elementos na região de sobreposição entre dois subconjuntos S_1 e S_2, durante o processo de partição do conjunto S, é dado pela Relação 2.3:

$$|S_1 \cap S_2| = \lceil o * |S| \rceil \tag{2.3}$$

Com $n_o = \lceil o * |S| \rceil$, são selecionados os n_o elementos com menor diferença de distância para os dois pivôs para compor a região de sobreposição de um dado subconjunto. Por exemplo, se em dada etapa da recursão são formados dois subconjuntos S_1 e S_2, para definir a região de sobreposição de S_1, são selecionados os n_o elementos $s_i \in S_2$ com a menor diferença $d(s_i, p_1) - d(s_i, p_2)$. Esses elementos selecionados são duplicados e adicionados a S_1.

2.3.2 Método de busca

O grafo gerado é uma aproximação do grafo de tipo g_1. Portanto, qualquer algoritmo que realize buscas por aproximação espacial pode ser aplicado nas consultas

realizadas no grafo. Um exemplo é o *GNNS*, que foi utilizado em vários dos experimentos feitos com o *HGraph* [1]. Com o objetivo de responder a consulta *k-NN* de um elemento de consulta q, ele começa selecionando um vértice aleatório $u \in V$ do grafo e realiza o processo de aproximação espacial: em cada iteração, troca u por $v \in N(u)$, que é o vizinho de u mais similar ao elemento de consulta q. O número máximo de iterações é definido por T, e quando $T = |V|$, o vértice final será um que não possui vizinhos mais similares a q do que ele. Como é uma busca gulosa, o algoritmo sofre de problemas de ótimo local, ou seja, pode retornar respostas longe do esperado. Para contornar isso, o processo é repetido R vezes, cada vez com um vértice inicial diferente [10].

É possível perceber um claro *trade-off* no algoritmo. Com um valor baixo para R, a busca será realizada de maneira mais rápida, mas a chance de cair em um ótimo local que é longe do resultado esperado pode ser muito alta, diminuindo o *recall*[2]. Por outro lado, com um valor alto, a chance de encontrar a resposta exata é maior, pois o algoritmo visitará uma parcela maior do grafo. O problema é que o tempo de busca é prejudicado, tendo em vista que será necessário fazer muito mais computações de distância.

[2] O *recall*, ou revocação, é definido neste contexto pela proporção entre elementos efetivamente retornados e os que deveriam ter sido retornados em uma resposta exata. Ou seja, se 7 de 10 elementos corretos foram retornados na resposta, o *recall* é de 0,7.

3 PROBLEMA E TRABALHOS RELACIONADOS

Em uma busca por aproximação espacial, é importante que algum dos vértices iniciais não esteja longe da resposta esperada na consulta, pois quanto mais longe, maior o número de computações necessárias e maior a chance de cair em ótimos locais. Porém, ao escolher vértices aleatórios, conta-se com a sorte para que isso aconteça. Muitas vezes, é necessário definir um valor alto para R para se obter respostas mais precisas, o que não é ideal quando se deseja rapidez nas consultas. A Figura 7 ilustra um cenário onde, através de algum método como escolha aleatória, foi escolhido um vértice que está "longe" do objeto de consulta, necessitando que uma grande parte do grafo seja percorrida para se atingir a resposta.

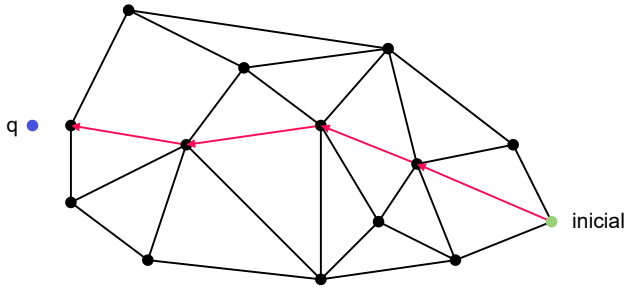


Figura 7 – Exemplo de aproximação espacial em um grafo de proximidade onde o vértice inicial escolhido está "longe" do objeto de consulta q.

Para melhorar esse cenário e evitar que vértices inadequados sejam escolhidos, técnicas de *aprendizagem de máquina* podem ser empregadas. O uso de aprendizagem de máquina para fazer consultas em conjuntos de dados tem sido o foco de vários trabalhos recentes. Nas próximas seções, serão apresentados os *learned indexes* e dois métodos, assim como a relação que eles carregam com este trabalho.

3.1 *Learned Indexes*

Como muitas estruturas de dados, os índices tradicionais são de propósito geral, ou seja, não assumem características fortes a respeito dos dados sendo armazenados [16]. Sendo assim, é comum que não sejam capazes de tirar vantagem da distribuição dos dados ou de aproveitar padrões existentes neles. Ao mesmo tempo, dados do mundo real muitas vezes não seguem padrões previamente conhecidos e que podem ser generalizados. Desenvolver soluções especializadas para cada caso de uso envolveria um esforço muito grande.

Trabalhos recentes têm tratado dessa falta de proveito através da criação de ***learned indexes*** [13, 14, 15, 16]. Eles fazem o uso de aprendizado de máquina, o que pode os tornar capazes de aprender modelos que refletem padrões intrínsecos aos dados em questão. Um dos precursores, o artigo de Tim Kraska et al. considera índices tradicionais como modelos, os quais podem ser aprimorados, ou até substituídos, por outros tipos de modelos, incluindo os que utilizam redes neurais [16]. Os autores também argumentam que aparentes obstáculos, como o pensamento tradicional de que modelos de *Machine Learning* são muito custosos, não são tão problemáticos como podem parecer.

3.2 *Learned indexes* em dados ordenáveis e o RMI

Seguindo a linha do artigo supracitado, índices para buscas por intervalo, como a B-Tree, já são modelos: dada uma chave, "predizem" a posição de um valor no conjunto indexado [16]. Por questão de eficiência, é comum que não sejam todas as chaves indexadas na B-Tree, mas sim, por exemplo, apenas a primeira chave de cada página. Assim, a posição retornada pelo índice não será exata: terá um erro mínimo e máximo, sendo o mínimo 0 e o máximo o tamanho da página. Vendo através desse ângulo, é possível considerar a B-Tree como uma árvore de regressão, ou seja, um modelo de *Machine Learning*.

Ainda mais, para buscas por intervalo, onde os dados devem estar ordenados de acordo com a chave de busca (se for desejado que elas sejam eficientes), um modelo que prediz a posição de uma chave no conjunto ordenado dos dados aproxima a **função de distribuição acumulada** [2] (CDF, sigla em inglês). Com uma CDF, é intuitivo inferir a posição aproximada de uma dada chave através da Relação 3.1:

$$p = F(Chave) * N \tag{3.1}$$

onde p é a posição estimada, $F(Chave)$ é o resultado da CDF que estima a probabilidade de existir uma chave menor ou igual à chave de busca $P(X \leq Chave)$ e N é o número total de chaves. Esse fato significa que, para que a indexação seja feita, basta aprender a distribuição dos dados para construir a CDF. É nesse ponto que são baseados alguns *learned indexes*, como o RMI [16], o PGM-index [37] e o RadixSpline [38].

O ***Recursive Model Index***, ou RMI [16], é uma estrutura motivada pela dificuldade de se atingir uma precisão satisfatória com apenas um modelo quando há muitos elementos. Por exemplo, reduzir o erro de predição para a ordem das centenas quando existem 100 milhões de elementos dessa maneira não é simples. Por outro lado, a redução de 100 milhões para 10 mil é muito mais fácil até com modelos simples, e de 10 mil para 100 é um problema similar. Então, a ideia do RMI é construir uma hierarquia de modelos

com vários estágios, onde em cada estágio é escolhido um modelo com base na chave, até que no estágio final a posição é predita. No primeiro estágio, há apenas um modelo.

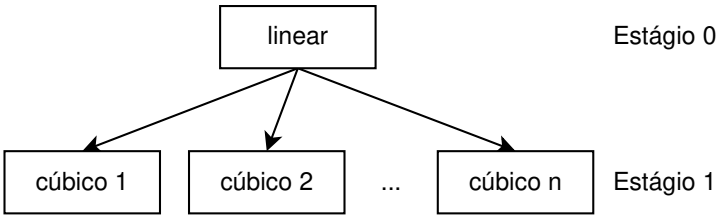


Figura 8 – Um RMI de dois estágios. Fonte: [2]

.

Uma vantagem importante é que os modelos podem ser de tipos diferentes. Por exemplo, um RMI de dois estágios pode usar um modelo linear no estágio 0, o qual seleciona um modelo cúbico do estágio 1 para efetivamente predizer a posição da chave. Esse exemplo é ilustrado na Figura 8. Os modelos no RMI são treinados de maneira *top-down*, ou seja, primeiramente é treinado o modelo no estágio 0 (o primeiro), com ele são treinados os modelos do estágio 1, os quais são posteriormente usados para treinar os modelos do estágio 2, e assim sucessivamente. Formalmente, com x sendo a chave, $y \in [0, N)$ a posição correspondente e M_ℓ o número de modelos no estágio ℓ, cada modelo k no estágio ℓ, denotado por $f_\ell^{(k)}$, é treinado visando minimizar os seguintes erros:

$$L_\ell = \sum_{(x,y)} (f_\ell^{(\lfloor M_\ell f_{\ell-1}(x)/N \rfloor)}(x) - y)^2 \qquad L_0 = \sum_{(x,y)} (f_0(x) - y)^2 \tag{3.2}$$

O RMI se mostrou competitivo com os índices tradicionais, reduzindo pela metade, em relação a uma B-Tree otimizada, o tempo de execução de consultas em vários conjuntos de dados. Mais detalhes podem ser encontrados no artigo [16].

Asim como este trabalho, o RMI é um índice que utiliza conceitos de aprendizado de máquina e hierarquia de modelos. Porém, o foco é em dados ordenáveis, não sendo adequado para dados complexos em espaços métricos. Além disso, ele não se apoia em nenhuma estrutura pré-construída, como um outro índice tradicional, tendo sua construção feita diretamente com os dados que estão sendo indexados.

3.3 LMI, um *learned index* para dados complexos

As estruturas apresentadas na Seção 3.2 são capazes de indexar dados ordenáveis, sendo habilitadas a substituir índices tradicionais como a B-Tree. Elas obtêm resultados satisfatórios, muitas vezes reduzindo o tempo de consulta e o tamanho do índice [2]. Porém, o intuito original delas não trata dados complexos, desestruturados, que se encaixam no modelo de um espaço métrico. Em contrapartida, um trabalho recente aplica o conceito de *learned indexes* a esse tipo de dados, introduzindo o ***Learned Metric Index***, ou

LMI [13]. Antol et al. acreditam que sua pesquisa é original, apesar de reconhecerem alguns experimentos anteriores utilizando aprendizado de máquina para realizar buscas por similaridade em espaços métricos. No mínimo, é a primeira do tipo a ter sucesso.

O LMI pode constituir um problema de aprendizado supervisionado, onde uma estrutura já existente é utilizada, considerando como *label* de um elemento a sua posição no índice original. Também pode ser um problema de aprendizado não supervisionado, onde o índice é construído por inteiro, determinando por si só as divisões dos dados. Os autores optaram pela primeira alternativa por fins de experimentação, apesar de considerarem o LMI não supervisionado como uma das futuras direções de pesquisa.

Nos experimentos, as estruturas-base utilizadas para o treinamento supervisionado foram a M-Tree [27] e o M-Index [7]. O objetivo não é simular a estrutura do índice original através de modelos, mas usá-la de ponto de partida a fim de criar uma potencialmente melhor. Como *label* de um objeto, é considerada sua posição na estrutura original. Por exemplo, com o M-Index de base, se determinado objeto estiver contido no *cluster* $C_{2,17,1,3}$, a *label* para o modelo do primeiro nível seria 17; para modelos do segundo, 1; e do terceiro, 3.

O processo de treinamento se constitui em substituir cada nó interno da árvore original do índice por um modelo. Se inicia no nó raiz, treinando um modelo com todo o conjunto dos dados. Seus descendentes são treinados em subconjuntos cada vez menores, conforme a profundidade na árvore. O treinamento é sequencial, ou seja, um nó é treinado somente após seu nó pai ser. Cada modelo tem um problema de classificação a ser resolvido, que é basicamente determinar as probabilidades de cada um dos filhos ter a(s) resposta(s) para alguma consulta. É importante notar que não são todos os nós preservados da estrutura original, bem como podem surgir novos nós. Mais detalhes podem ser encontrados no artigo.

Uma busca é iniciada no nó raiz e, em cada passo, um modelo é executado. A saída dele será uma distribuição de probabilidades, gerando um valor para cada filho do nó em questão. As probabilidades são inseridas em uma fila de prioridade, que é ordenada do nó mais provável ao menos provável. Então, o próximo nó escolhido será o primeiro da fila, e com ele o processo é repetido. Ao se escolher um nó-folha, que contém uma pequena partição do *dataset*, é feita uma busca linear que resultará na resposta para a consulta, ou em parte dela. Caso parte da resposta estiver faltando[1], é escolhido mais um nó na fila de prioridade, se repetindo o processo até que a resposta esteja enfim completa.

Vários tipos de modelos podem ser utilizados no LMI. Nos experimentos realizados, foram utilizados quatro, cada um com suas particularidades, desde o tempo de construção até os ajustes possíveis em hiperparâmetros. Entre eles, estão: a **Regressão Logística**,

[1] Por exemplo, se a consulta for para obter os 30-NN de um elemento de consulta q qualquer, e até o momento foram encontrados apenas 15.

que é mais simples e consumiu menos recursos computacionais em sua construção, além de obter bons resultados em consultas; as **Redes Neurais** de 2 e 3 camadas ocultas, que foram competitivas com a regressão logística, mas tiveram sua performance reduzida em alguns casos; e a ***Random Forest*** [39], que obteve alta precisão em consultas *1-NN* (que retorna o elemento mais próximo), mas não teve um resultado parecido em consultas mais genéricas *k-NN*, além de consumir muitos recursos computacionais.

O LMI carrega forte semelhança com este trabalho, principalmente pelo fato de ter sido uma das inspirações. Ele também lida com dados complexos em espaços métricos e utiliza hierarquias de modelos no processo, porém há algumas diferenças: é baseado em árvores e não em grafos; assim como nas suas estruturas base, ele trabalha com dados em disco ao invés de puramente em memória; ele serve como substituição e não como apoio ao índice original; e após os nós de nível folha serem determinados, a busca é feita sequencialmente neles ao invés de um algoritmo de aproximação espacial como o *GNNS* ser usado (por não se tratar de um grafo).

3.4 Aprendizagem de máquina na aproximação espacial

Para melhorar a situação discutida no início deste capítulo, é necessário utilizar técnicas para escolher os vértices iniciais de forma mais criteriosa. Se inspirando no RMI [16] e no LMI [13], no Capítulo 4 é apresentada uma nova técnica com esse objetivo, fazendo o uso de modelos de aprendizado de máquina para predizer vértices próximos ao objeto de consulta. Dessa maneira, é aumentada a chance de encontrar rapidamente uma resposta próxima à esperada.

4 PROPOSTA - *LEARNED INDEX* BASEADO NA PREDIÇÃO DE VÉRTICES INICIAIS

Conforme explicado no Capítulo 3, a escolha dos vértices inicias em uma busca por aproximação espacial é um fator importante, contribuindo para a precisão e o *recall* dos resultados obtidos, assim como o tempo de busca. Tendo isso em mente, o objetivo deste trabalho é desenvolver um método para **predizer vértices iniciais de busca em um grafo de aproximação construídos por particionamento**, visando tomar proveito da sua estrutura e do processo de construção realizado.

O *HGraph* é um método que constrói grafos por particionamento, sendo essa uma das características que podem ser aproveitadas. Durante a construção, vários subconjuntos vão sendo criados até que apenas existam subconjuntos com m ou menos elementos, sendo m um parâmetro normalmente definido para ser muito menor do que o tamanho original do conjunto de dados. O grafo final é constituído por vários agrupamentos, e a tendência é que os elementos em cada um não sejam muito dissimilares um do outro. Com isso, é possível imaginar que restringir os vértices iniciais para a partição onde se encontraria o resultado (ou partições "vizinhas" a ela) diminuiria o trajeto que o algoritmo de busca deve realizar para retornar a resposta correta.

Com base em trabalhos recentes na literatura, que tratam de *learned indexes*, uma abordagem para escolher vértices iniciais é através de modelos de aprendizagem de máquina. A ideia principal é que, dado um elemento de busca, o modelo prediz em quais das partições no grafo é mais provável de se encontrar o resultado. Com isso, a busca pode ser iniciada em algum elemento da partição escolhida, como no pivô. No caso de buscas com vários *restarts*[1], outros elementos aleatórios na partição são escolhidos.

Esse método tem inspiração no *Learned Metric Index* [13], que se baseia em estruturas já construídas como o *M-Index* e a *M-Tree* para treinar uma hierarquia de modelos. Dessa forma, constitui-se um problema de aprendizagem supervisionada, onde as *labels* são conhecidas (nesse caso, são as posições dos elementos na estrutura montada anteriormente).

Neste capítulo, será introduzido esse novo método, detalhando seu funcionamento. Alguns passos foram introduzidos na fase de construção do *HGraph*, detalhados nas seções 4.1 e 4.2, e um passo simples foi adicionado na fase de busca, discutido na seção 4.3. Por fim, a implementação é apresentada na seção 4.4.

[1] Como no algoritmo *GNNS*.

4.1 Pré-processamento

4.1.1 Atribuição de *labels*

Conforme foi discutido anteriormente, o *HGraph* forma uma árvore de recursão durante seu processo de divisão. Cada nó representa uma fatia do conjunto original, sendo que grafos como o *kNNG* são construídos nas folhas, as quais possuem cardinalidade igual ou inferior a m (um dos parâmetros fornecidos ao algoritmo). Esse processo foi modificado para salvar o caminho percorrido até cada nó, representado pelos índices em cada nível da árvore. Uma visualização de exemplo dos identificadores atribuídos durante a construção do *HGraph* é representada na Figura 9.

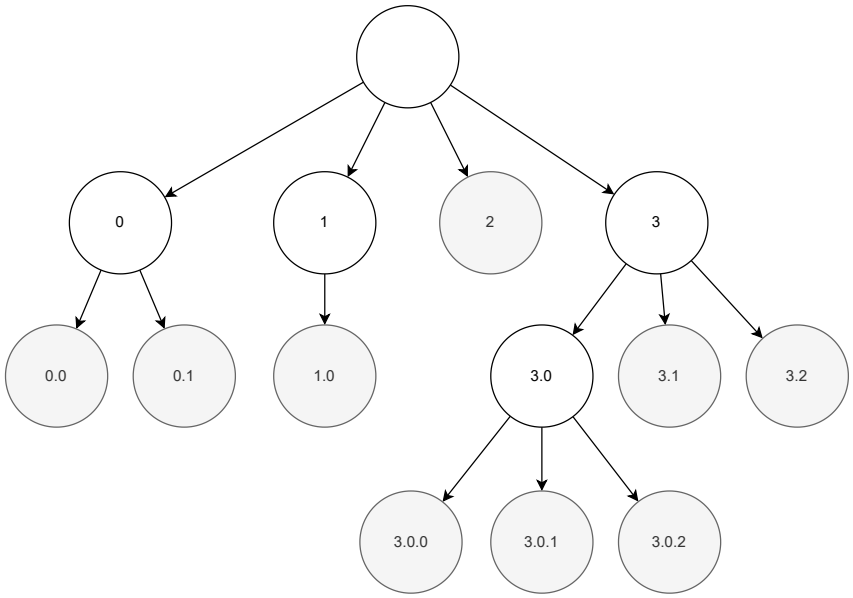


Figura 9 – *Labels* atribuídas a cada nó na árvore de recursão do *HGraph*.

Como pode ser visto na Figura 9, os identificadores são simples concatenações dos índices em níveis superiores, nesse caso divididos por um ponto. Assim, cada partição final no grafo (correspondente a um nó folha) ganha um identificador único e que pode ser interpretado posteriormente para realizar agrupamentos - o que será discutido na seção 4.2.2.

Ao término da construção do grafo com o *HGraph*, é montado o *dataset* de treinamento: cada vértice no grafo é um elemento, cuja *label* é o identificador que foi atribuído à sua partição. Com isso, move-se para a próxima etapa: o tratamento dos dados.

4.1.2 Tratamento dos dados

Os modelos empregados são de classificação, mais especificamente, classificação **multiclasse**; ou seja, eles devem classificar um objeto entre várias classes (nesse caso, entre as partições de nível folha) de acordo com suas características. Existem vários que cumprem esse papel, como por exemplo redes neurais em camadas e a *Random Forest* [39].

Para treinar modelos, seja qual for o tipo escolhido, os dados devem ser tratados antes. No caso do *dataset* gerado, um dos problemas é a duplicação de elementos: por conta da região de *overlap*, o mesmo elemento pode aparecer várias vezes com diferentes *labels*. Isso não faz sentido para um classificador, pois apenas uma resposta será dada para cada classificação. Portanto, os duplicados devem ser eliminados, levando em consideração algum critério como a proximidade para seu pivô.

Outros tipos de tratamento podem ser realizados, como a padronização das features através de *scalers*. Eles transformam a escala dos valores para intervalos padrão e geralmente tornam os dados mais amigáveis para os modelos. Isso ocorre pois evitam que características que apresentem valores muito superiores às outras tenham muita influência na classificação. Portanto, um *scaler* que padroniza as características para ficarem contidas no intervalo $[-1, 1]$ foi adotado no tratamento.

4.2 Treinamento do classificador

4.2.1 Abordagem de um único modelo

O classificador pode ser constituído por um único modelo, o que pode ser adequado para conjuntos de dados não muito complexos. Assim, todo o *dataset* gerado e tratado é utilizado para treinar esse modelo, o qual, dado um vetor de características, retornará a partição resposta diretamente. Um diagrama ilustrando o processo completo utilizando modelo único pode ser visto na Figura 10.

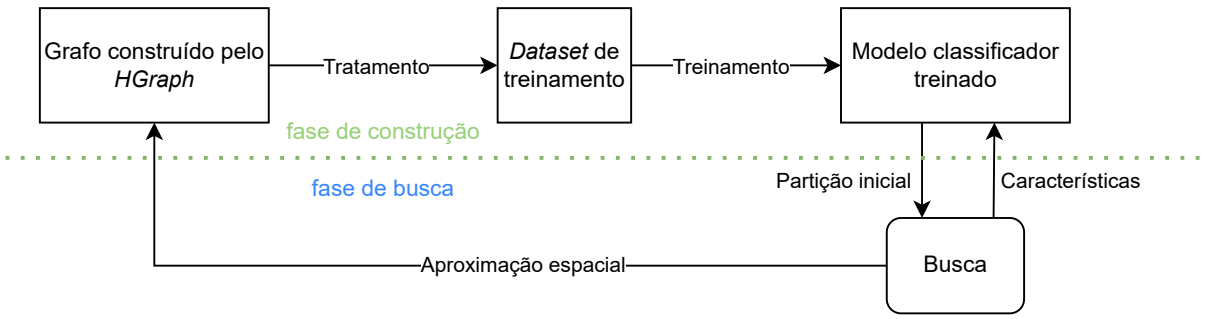


Figura 10 – Processo completo quando um modelo único é utilizado.

4.2.2 Abordagem com hierarquia de modelos

Com *datasets* maiores e/ou com dimensionalidade mais alta, o modelo único pode sofrer problemas de precisão e revocação, de forma similar ao enfrentado pelo RMI [16]. Portanto, é sugerido formar uma hierarquia de modelos, capaz de dividir o conjunto de dados em partes menores e potencialmente tornando o problema de classificação mais fácil para os modelos. A maneira com que isso é feito é através do agrupamento de partições, por meio de dois métodos propostos: um baseado em algoritmos de árvore

geradora mínima como o Kruskal e outro que une partições irmãs. Ambos tentam agrupar partições "semelhantes" para que o agrupamento faça sentido e os classificadores consigam identificar padrões.

4.2.2.1 Agrupamento baseado em algoritmos que encontram a árvore geradora mínima

Um dos métodos para agrupar partições é a utilização de uma versão modificada de algoritmos que encontram a árvore geradora mínima do grafo, como o algoritmo de Kruskal. Com os pivôs de cada partição como vértices, são feitos os passos do Kruskal, conectando os vértices de menor distância sem formar ciclos. A diferença é que dois vértices só são conectados se a árvore resultante pela conexão não contiver mais do que N elementos, número fornecido por um parâmetro. O resultado, ao invés de apenas uma árvore geradora como no Kruskal original, são várias árvores com N ou menos elementos, as quais são interpretadas como grupos. Na Figura 11, é possível ver parte do processo de um agrupamento utilizando esse algoritmo. As arestas que seriam criadas pelo algoritmo original, mas foram descartadas devido à modificação, estão indicadas na terceira imagem.

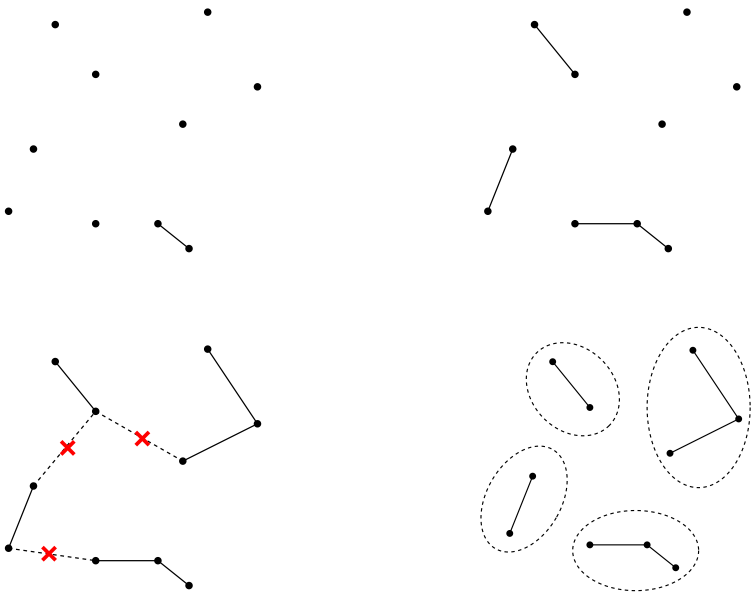


Figura 11 – Exemplo de aplicação do agrupamento através do algoritmo de Kruskal modificado com $N = 3$ em um espaço euclidiano 2D.

4.2.2.2 Agrupamento através da união de partições irmãs

Outra maneira proposta para realizar o agrupamento se aproveita de forma trivial das *labels* dos elementos. Com elas, é possível identificar partições que são irmãs: basta analisar se todos os índices, com exceção do último, são iguais. Assim, as partições irmãs são agrupadas com um limite de N partições por agrupamento, número fornecido por um

parâmetro. A motivação é por conta de partições com o mesmo nó pai terem a tendência de possuírem elementos similares, devido à forma que o *HGraph* realiza as divisões. O algoritmo 1 demonstra o processo de formação dos grupos, determinando as partições (representadas por suas respectivas *labels*) que compõem cada um.

Algoritmo 1: Agrupamento por partições irmãs

Entrada: N, *labels*
Saída : Lista de agrupamentos de *labels*

```
1  grupos ← {}
2  labels ← ordenar(labels)
3  grpatual ← {}
4  ultimalabel ← ""
5  para cada label em labels faça
6  |  se (|grpatual| = N) ∨ (ultimalabel.naoirmã(label)) ∧ (|grpatual| > 1)
   |   então
7  |  |  grupos ← grupos ∪ {grpatual}
8  |  |  grpatual ← {}
9  |  fim
10 |  ultimalabel ← label
11 |  grpatual ← grpatual ∪ {label}
12 fim
13 se grpatual ≠ ∅ então
14 |  grupos ← grupos ∪ {grpatual}
15 fim
16 retorna grupos
```

Um ponto chave do algoritmo está na linha 6. Nela, há a condição para o grupo que está sendo formado fechar e ser adicionado à lista de grupos. Como as *labels* estão ordenadas, as que correspondem a partições irmãs são adjacentes na lista, e é nesse fator que a condição se baseia. Se a *label* anterior não for irmã da atual, o grupo atual pode ser fechado. Porém, isso só é feito se ele contiver mais de uma partição, a fim de evitar que sejam formados grupos com apenas uma classe. Na Figura 12 é exibido um exemplo de aplicação do algoritmo.

Na prática, os dois métodos se mostraram semelhantes em termos de qualidade dos grupos gerados e como eles afetam a precisão dos modelos. Porém, o segundo se mostrou mais simples de ser implementado de maneira eficiente, sem exigir cálculos adicionais de distância, o que é essencial para conjuntos de dados muito grandes. Para o primeiro ser utilizado eficientemente, as distâncias entre os pivôs já calculadas durante o processo de construção deveriam ser reaproveitadas. Dessa forma, os experimentos apresentados no Capítulo 5 utilizam o método das partições irmãs.

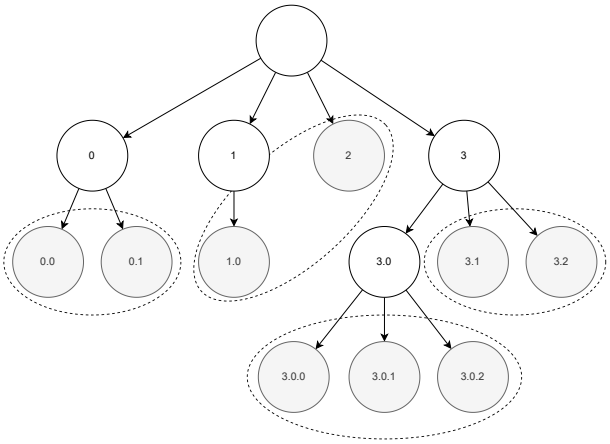


Figura 12 – Exemplo de aplicação do agrupamento por partições irmãs com $N = 3$.

4.2.2.3 Treinamento dos modelos

Tendo feito os agrupamentos por qualquer um dos métodos, cada um dos grupos, com todos os elementos pertencentes às suas partições, deve ser fornecido então como dados de treinamento para um modelo. Ou seja, se k grupos foram formados, k modelos serão treinados. Esses modelos são denominados os **modelos de nível folha**: são eles que efetivamente retornam o identificador de uma partição. No caso, cada um deles é limitado a retornar apenas resultados entre as partições que estavam contidas no grupo utilizado para seu treinamento. Assim como na abordagem utilizando modelo único, qualquer tipo de modelo para classificação multiclasse pode ser utilizado.

Vários modelos foram treinados, cada um com uma pequena parte do conjunto de dados original e, consequentemente, menos classes para predizer. Isso facilita o trabalho de predição deles, porém mais um modelo é necessário, denominado **modelo raiz**. O papel dele é determinar em qual dos grupos um objeto tem mais probabilidade de se situar - isto é, qual modelo utilizar.

Novamente, todo o conjunto de dados é usado no treinamento. Todavia, a diferença está no número de classes. As *labels* são trocadas para os índices dos grupos onde está cada elemento. Tendo que predizer apenas o grupo, e não a partição exata, esse número é dividido por aproximadamente N (tamanho de cada grupo). Dessa maneira, se o agrupamento foi capaz de dar preferência à união de partições semelhantes, o modelo terá mais facilidade para fazer predições corretas.

Na Figura 13 é mostrado um diagrama que ilustra todo o processo de treinamento dos modelos na abordagem hierárquica, desde o tratamento dos dados até o treinamento do modelo raiz. O *dataset* dos grupos é o que foi comentado por último, que é semelhante ao original com a diferença de suas *labels* corresponderem a grupos e não a partições.

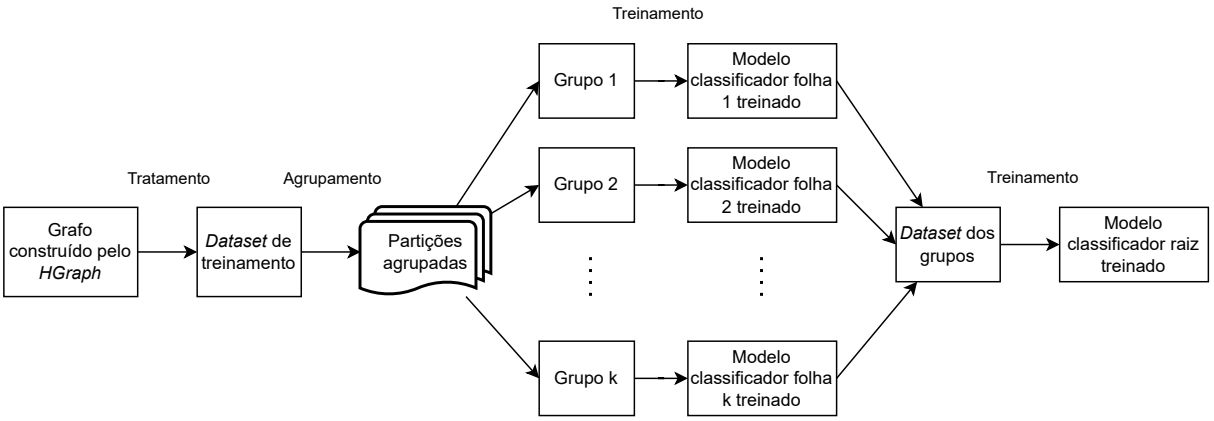


Figura 13 – Diagrama ilustrando o processo completo de treinamento dos modelos em hierarquia.

4.3 Fase de busca

A fim de tomar proveito do(s) modelo(s) treinado(s) durante a fase de construção, a fase de busca também foi modificada para que a partição onde ela se iniciará seja predita. Da partição predita, são escolhidos vértices quaisquer (a quantidade é determinada pelo número de *restarts*), porém o primeiro é sempre seu pivô, que possui, no geral, mais conexões do que outros vértices. Isso substitui a escolha do vértice inicial praticada anteriormente, que era aleatória. Porém, deve-se ressaltar que, se o número de *restarts* causar o esgotamento dos vértices da partição, vértices aleatórios passarão a ser selecionados.

Dependendo se foi utilizado um modelo único ou dois níveis, a forma com que a predição é feita muda. Com um modelo único, conforme discutido anteriormente, esse passo é realizado de forma direta: as características do objeto de consulta são entradas no modelo, o qual retorna o identificador de uma partição. Já com uma hierarquia, duas predições são necessárias: primeiramente o modelo raiz seleciona um modelo de nível folha, o qual é executado com o objeto de consulta e retorna, enfim, a partição.

Na Figura 14, é mostrada uma hierarquia de exemplo, assim como a entrada e saída de cada nível. No nível folha, as partições são representadas por círculos, contidos em retângulos referentes aos grupos onde foram inseridas.

4.4 Implementação

Para implementar as técnicas, como o *HGraph* já havia sido implementado na ***Non-Metric Space Library*** [40], foi utilizada a mesma base de código. A parte que atribui identificadores às partições foi integrada ao próprio código de construção em C++. Por outro lado, o tratamento dos dados, agrupamento (no caso de ser uma hierarquia), treinamento do(s) modelo(s) e predição foram implementados na linguagem *Python*. A comunicação entre os dois *runtimes* é feita através da gravação e leitura de arquivos na

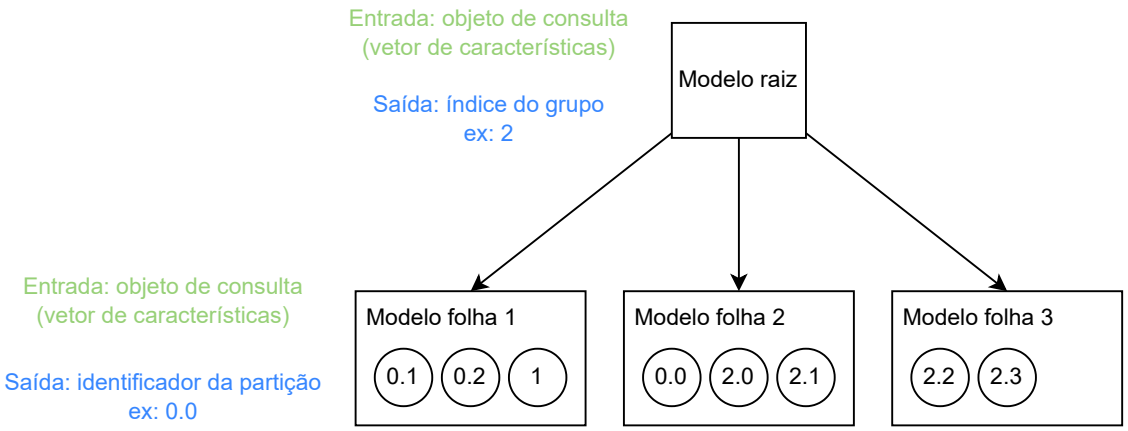


Figura 14 – Processo de predição em uma hierarquia de modelos.

fase de construção, e por meio de *pipes* na fase de busca.

O tratamento dos dados é feito com as bibliotecas *numpy*[2] e *pandas*[3], enquanto que a *scikit-learn* [41] fornece o modelo de classificação escolhido: a ***Random Forest*** [39]. Ela foi escolhida por geralmente não necessitar de muitos ajustes de hiperparâmetros (se comparada com redes neurais com múltiplas camadas, por exemplo) e ser capaz de classificar os objetos com alta precisão. Foi utilizada no caso de modelo único e também em todos os níveis da hierarquia de modelos.

É importante observar que o objetivo do trabalho é verificar se os passos adicionais resultam em menos cálculos de distância ao mesmo tempo que não prejudicam o *recall*. Portanto, a implementação separada em *Python* se tornou aceitável mesmo aumentando o tempo de construção e de busca (devido à troca de contexto e à forma como a comunicação é feita). Assim, os experimentos apresentados no Capítulo 5 não visam comparar essas medidas. Trabalhos futuros envolveriam uma implementação sem trocas de contexto, eliminando atrasos ocasionados por isso e tornando possível a comparação direta de tempos de execução.

[2] https://numpy.org/
[3] https://pandas.pydata.org/

5 EXPERIMENTOS REALIZADOS

Neste capítulo, serão apresentados e discutidos os resultados dos experimentos realizados com a implementação. Os *datasets* e o *setup* utilizado para os testes são mostrados na seção 5.1, enquanto que os gráficos e discussões aparecem na seção 5.2.

5.1 *Setup* de testes

Dataset	**Cardinalidade**	**Dimensões**	**Fonte**
USCities	25.375	2	Censo Americano[1]
Color Moments	68.040	9	*Corel Image Features* [42]
Co-occurrence Texture	68.040	16	*Corel Image Features* [42]
Color Histogram	68.040	32	*Corel Image Features* [42]
CoPhIR Colour Layout	1.000.000	12	CoPhIR [43]

Tabela 1 – *Datasets* utilizados nos experimentos.

Os *datasets* utilizados são listados na Tabela 1. O *USCities* é o mais simples em questão de cardinalidade e dimensionalidade, e é composto por coordenadas geográficas das cidades estadunidenses. Os conjuntos *Color Moments*, *Co-occurrence Texture* e *Color Histogram* são características extraídas de uma coleção de imagens. Mais informações podem ser encontradas no *UCI Machine Learning Repository*[2]. Já o *CoPhIR Colour Layout* é o maior utilizado, contendo informações de cores extraídas de um conjunto de um milhão de imagens.

Para conduzir os testes, foi empregada uma única *thread* de um Intel(R) Core(TM) i7-8700 e 32GB de *RAM* a 2666 MHz, executando o Ubuntu 20.04.2 LTS. Como foi discutido na seção 4.4, os tempos de construção e execução não serão medidos, então o *hardware* não impõe uma diferença notável, mas foi apresentado aqui por fins de clareza.

Os parâmetros utilizados na *RandomForestClassifier*, do *scikit-learn*, foram 100 árvores (*n_estimators*) e 30 de profundidade máxima (*max_depth*). No *HGraph*, foi escolhido o grafo *kNNG*, com $o = 0{,}1$ e $n_P = 10$ fixos e $NN = \{5, 10, 50\}$, $m = \{5000, 10000\}$ variando. Foram feitas consultas *k-NN* com $k = \{1, 10, 30\}$ e $r = \{1, 5, 10, 20, 40, 80, 120, 240\}$.

5.2 Resultados obtidos

Nesta seção, serão apresentados e discutidos os gráficos gerados nos experimentos. Em todas as comparações, o *HGraph* original é rotulado como **HGraph**; o método uti-

[1] https://www.census.gov/geographies/reference-files/time-series/geo/gazetteer-files.2000.html
[2] https://archive.ics.uci.edu/ml/datasets/corel+image+features

lizando a abordagem hierárquica, como **HGraph_H**; e o com a abordágem de modelo único, como **HGraph_S**.

5.2.1 Computações de distância vs. *Recall*

Para analisar os ganhos dos passos adicionais, foram feitas comparações entre os diferentes métodos relacionando o número de computações de distância feitas nas consultas e o *recall* atingido. A Figura 15 exibe um caso onde a abordagem hierárquica apresentou melhor desempenho, com menos cálculos sendo exigidos para atingir um *recall* de 0,9. A de modelo único vem em segundo lugar, ainda apresentando ganhos em relação ao *HGraph* original. Os parâmetros utilizados neste teste foram $NN = 10$, $m = 5000$ e $K = 1$.

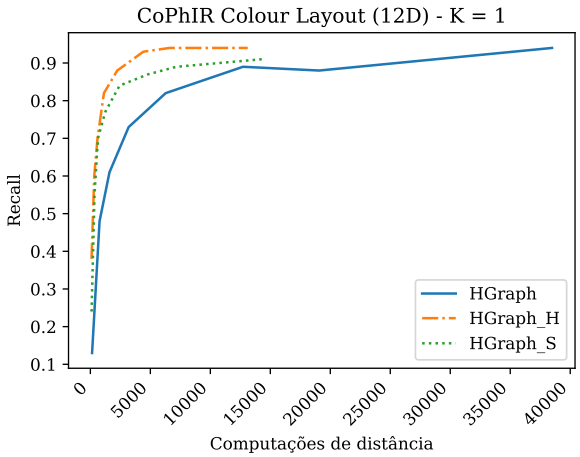


Figura 15 – Comparação de computações de distância vs. *recall* entre os métodos no *dataset CoPhIR Colour Layout* para uma consulta 1-NN.

Uma tendência ao utilizar a predição de vértices é o menor número máximo de computações de distância, mesmo com 240 *restarts*. Isso é explicado pelo foco dado à partição P escolhida: todos os vértices iniciais serão dela, pelo menos se $r \leq |P|$ (como comentado na seção 4.3). Esse fator faz com que menos regiões do grafo sejam exploradas do que com a escolha sendo completamente aleatória.

Na Figura 16, são exibidos resultados para os mesmos parâmetros de construção, mas fazendo consultas 10-NN e 30-NN. A diferença entre a abordagem hierárquica e a simples foi reduzida, principalmente nas consultas 30-NN, mas o ganho em comparação ao original permaneceu existindo. Um resultado melhor em consultas 1-NN sugere que a existência de múltiplos níveis está contribuindo para encontrar a partição correta, mas pelo mesmo motivo do foco ficar concentrado em uma região, a melhora acaba não sendo tão grande quando se deseja encontrar muitos elementos, que podem estar em regiões mais distantes.

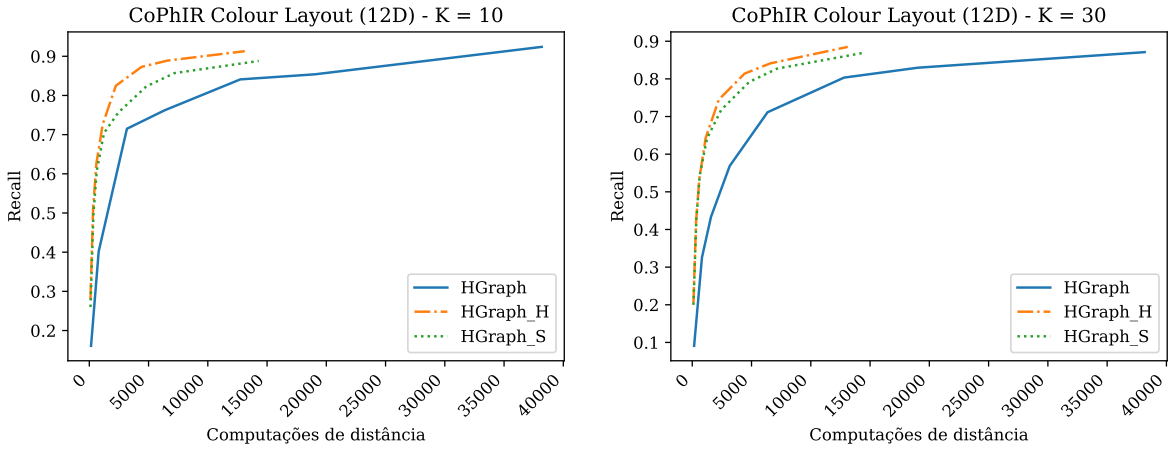


Figura 16 – Comparação de computações de distância vs. *recall* entre os métodos no *dataset CoPhIR Colour Layout* para as consultas 10-NN e 30-NN.

5.2.2 Impacto da complexidade do *dataset*

No *USCities*, o *dataset* de mais baixa cardinalidade e dimensionalidade, como esperado, a abordagem hierárquica sofreu perdas em relação à de modelo simples em alguns casos. Os passos adicionais envolvidos acabaram sendo prejudiciais para um conjunto desse tamanho. A Figura 17 indica esse cenário para consultas 1-NN e 30-NN, tendo como parâmetros de construção $NN = 10$ e $m = 5000$. No caso exibido, a abordagem de modelo simples atingiu uma maior taxa de *recall* do que a hierárquica, sem necesssitar de mais cálculos de distância.

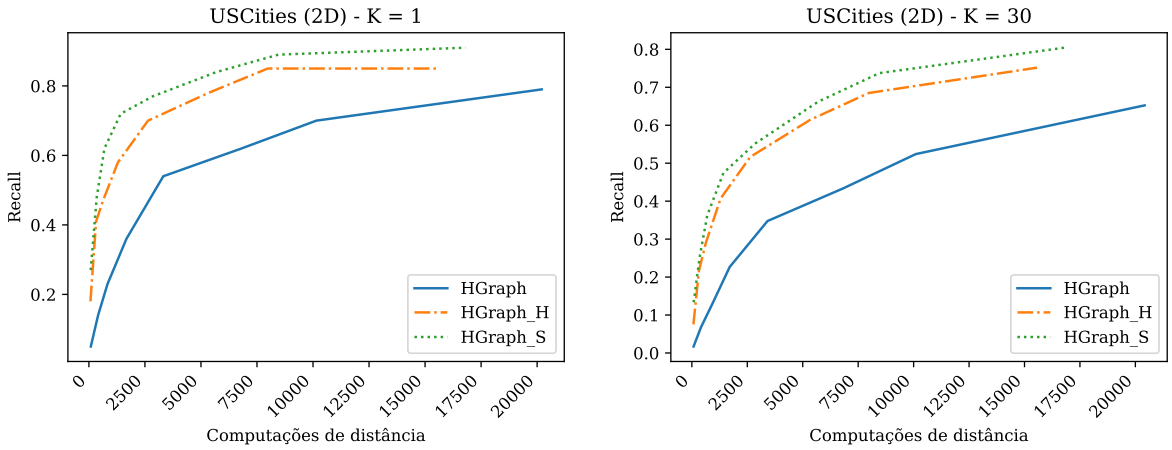


Figura 17 – Comparação de computações de distância vs. *recall* entre os métodos no *dataset USCities* para as consultas 1-NN e 30-NN.

5.2.3 Impacto dos parâmetros de construção do *HGraph*

5.2.3.1 Tamanho das partições

Para analisar o impacto do parâmetro m do *HGraph*, que determina o tamanho máximo de cada partição e, consequentemente, o número total de partições, foram fixados $NN = 5$ (por ser um valor baixo e causar menos influência no grafo) e $k = 10$ (a fim de testar consultas mais complexas que $k = 1$, mas não excessivamente). A Figura 18 exibe as computações de distância e *recall* em forma de *boxplot* para cada dataset. O *boxplot* foi utilizado pois há vários valores para o parâmetro r (listados na seção 5.1).

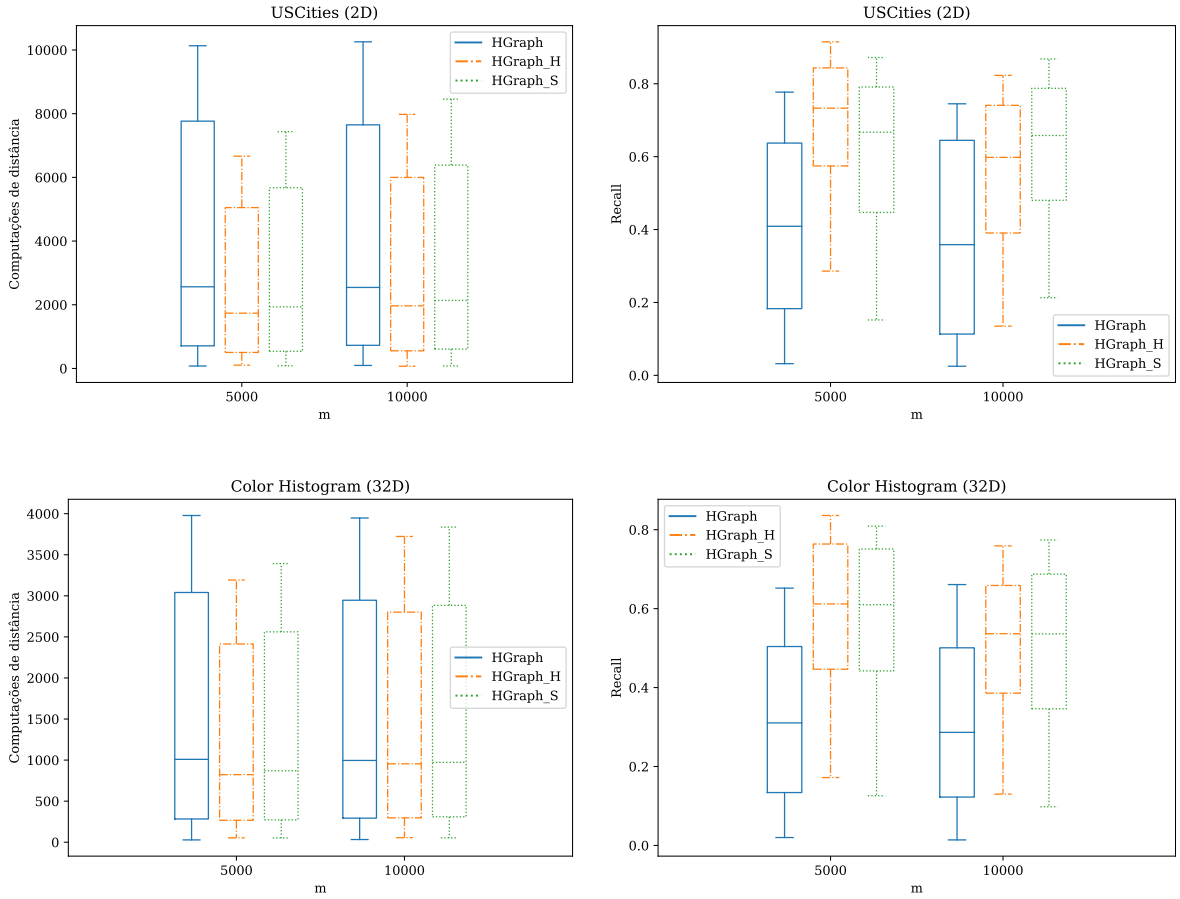


Figura 18 – Computações de distância e *recall* conforme o parâmetro m.

Comparando a abordagem hierárquica entre os valores de m, percebe-se que o *recall* tem média e máximas maiores com $m = 5000$, ao mesmo tempo que o número de computações de distância se mantém ou diminui. Enquanto isso, as outras abordagens não são muito afetadas: há ganhos menos expressivos (ou inexistentes) no *recall* e o número de computações de distância não apresenta muitas alterações com $m = 5000$. Isso indica que, como esperado, a abordagem hierárquica lida melhor com um maior número de partições para predizer, o que ocorre em *datasets* maiores.

5.2.3.2 Número de arestas (NN)

O parâmetro NN do *kNNG* determina o número de vizinhos mais próximos que serão conectados a cada vértice através de arestas no grafo. A fim de analisar o impacto desse parâmetro, foram fixados $m = 5000$ (para maximizar o número de partições) e $k = 10$ (pelo mesmo motivo dos testes com o parâmetro m). A Figura 19 mostra os cenários para cada *dataset* também em forma de *boxplots*. Evidentemente, quanto maior o NN, maior o *recall*. Em contrapartida, mais cálculos de distância são feitos.

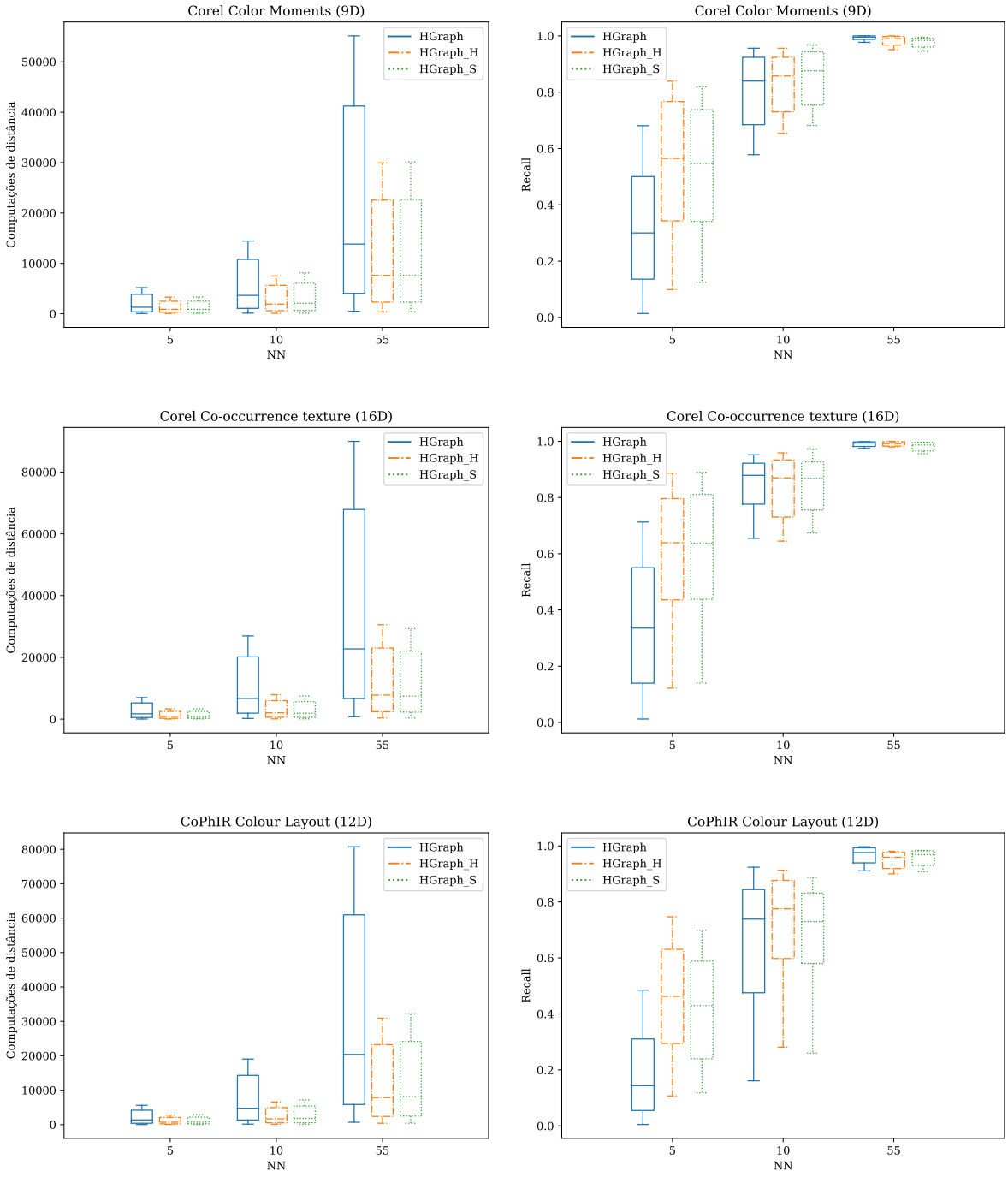


Figura 19 – Computações de distância e *recall* conforme o parâmetro NN.

Algo interessante de se observar é que, para $NN = 55$, os ganhos das novas abordagens em questão de *recall* são menos expressivos ou praticamente inexistentes, principalmente para *datasets* menores. Isso é explicado pela seleção do vértice inicial ter um impacto menor na aproximação espacial quando há tantas arestas em relação a K. Em compensação, esse valor pode causar mais cálculos de distância e aumentar o tempo de construção do grafo, como evidenciado no trabalho original [1].

A Figura 20 compara $N = 10$ com $N = 55$ no *Corel co-occurrence texture* utilizando os mesmos parâmetros. Nela, é possível perceber que a diferença entre as abordagens com predição e o *HGraph* original é muito menor quando $N = 55$. Ainda mais, a Figura 21 mostra que, no *CoPhIR Colour Layout*, a predição foi capaz de proporcionar um *recall* alto com menos computações de distância, mas impôs um limite que foi ultrapassado pelo *HGraph* original com mais cálculos quando $N = 55$.

(a) $N = 10$ (b) $N = 55$

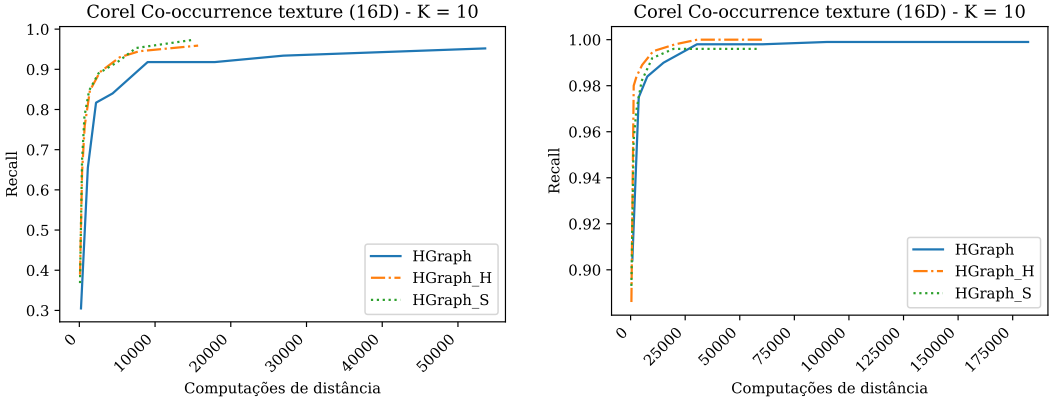


Figura 20 – Impacto do parâmetro NN no *dataset Corel Co-occurrence texture* com $m = 5000$ e $k = 10$.

(a) $N = 10$ (b) $N = 55$

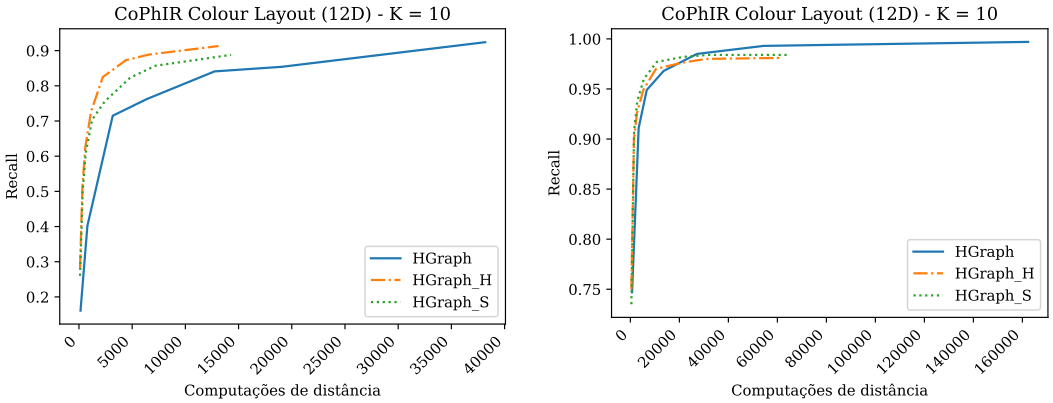


Figura 21 – Impacto do parâmetro NN no *dataset CoPhIR Colour Layout* com $m = 5000$ e $k = 10$.

5.2.4 Melhores configurações com *recall* maior ou igual a 0,9

A fim de compreender melhor os resultados, foi feita uma análise para cada *dataset* em cima das diferentes configurações que foram capazes de obter valores de *recall* $\geq 0{,}9$ nas consultas. As configurações foram ordenadas por cálculos de distância realizados e as 10 melhores de cada *dataset* foram selecionadas. As figuras 22, 23, 24, 25 e 26 mostram o resultado dessa análise para consultas $k = 10$.

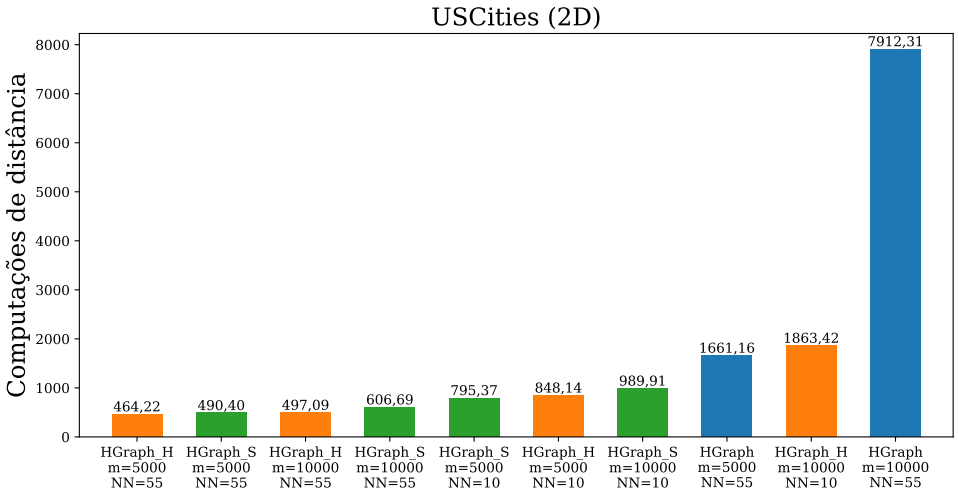


Figura 22 – Configurações com *recall* $\geq 0{,}9$ para consultas 10-NN no *USCities*.

A Figura 22 mostra que várias configurações do *HGraph* com predição de vértice inicial foram capazes de diminuir os cálculos de distância necessários para se obter um recall maior ou igual a 0,9 no *USCities*. A melhor configuração obtida, utilizando a abordagem hierárquica com $m = 5000$ e $NN = 55$, necessitou de 72% menos cálculos que a melhor configuração do *HGraph* original. Isso mostra que a predição também ajuda a melhorar os resultados de consultas em *datasets* de baixa cardinalidade.

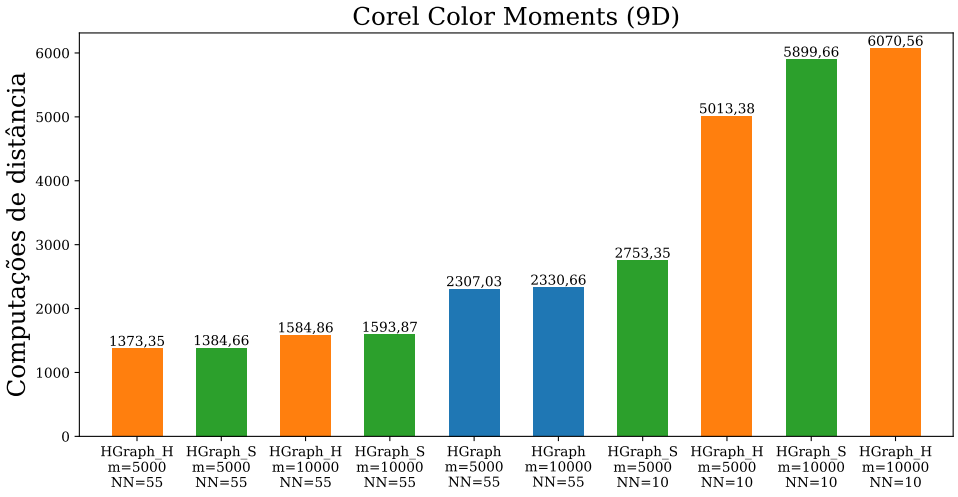


Figura 23 – Configurações com *recall* $\geq 0{,}9$ para consultas 10-NN no *Corel color moments*.

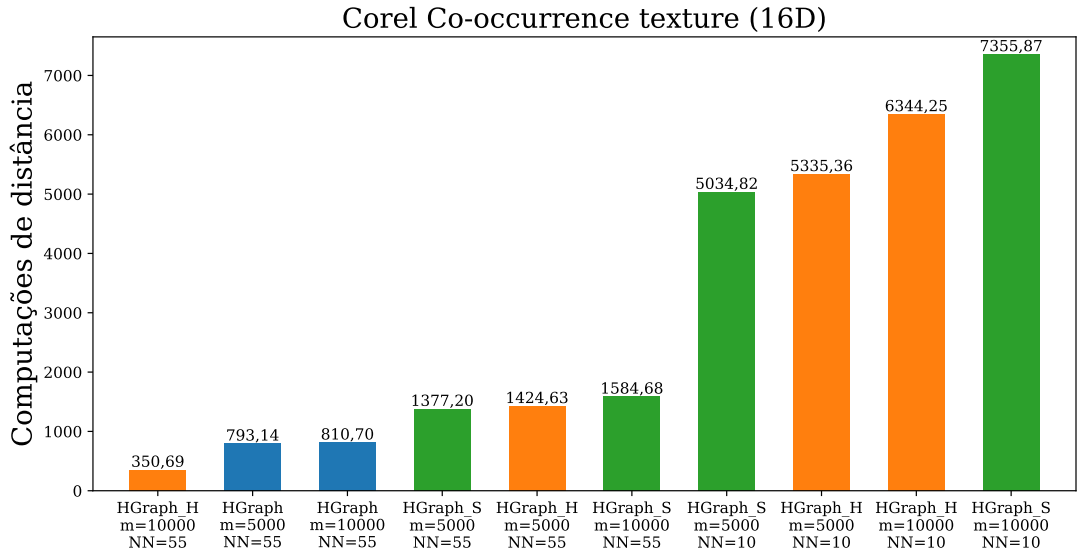


Figura 24 – Configurações com *recall* $\geq$ 0,9 para consultas 10-NN no *Corel co-occurrence texture*.

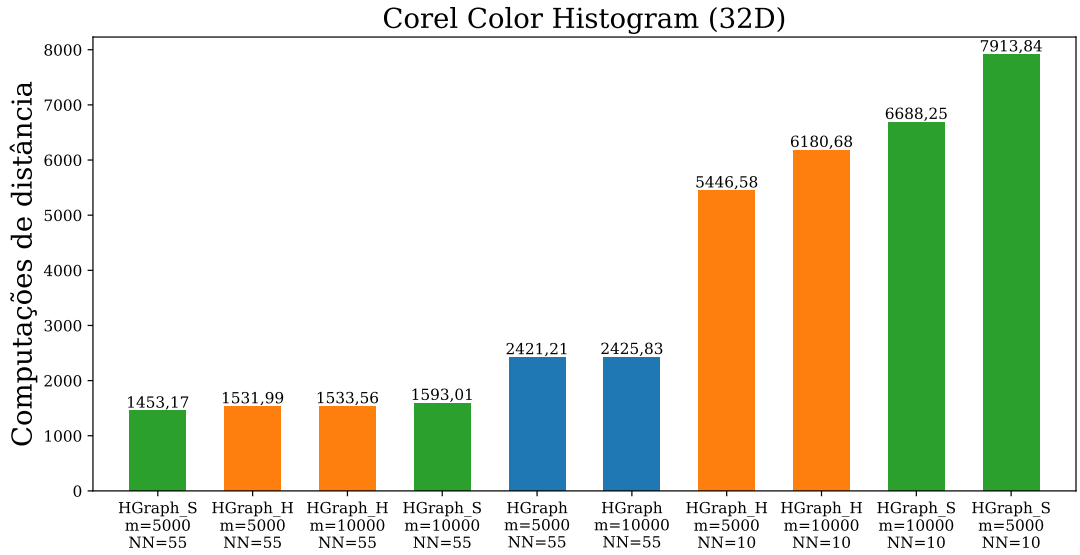


Figura 25 – Configurações com *recall* $\geq$ 0,9 para consultas 10-NN no *Corel color histogram*.

As figuras 23, 24 e 25 demonstram situações parecidas, com as melhores configurações necessitando, respectivamente, de 40%, 56% e 40% menos cálculos do que na melhor do *HGraph* original. É interessante notar que, no *Corel Co-occurrence texture*, a abordagem hierárquica com $m = 10000$ e $NN = 55$ se saiu muito melhor do que as outras que também usam predição.

Para o maior *dataset* testado, o *CoPhIR Colour Layout*, também houve claros ganhos. Na Figura 26, é possível perceber que, na melhor configuração, o *HGraph* original teve que realizar 138% mais cálculos de distância do que a abordagem hierárquica com

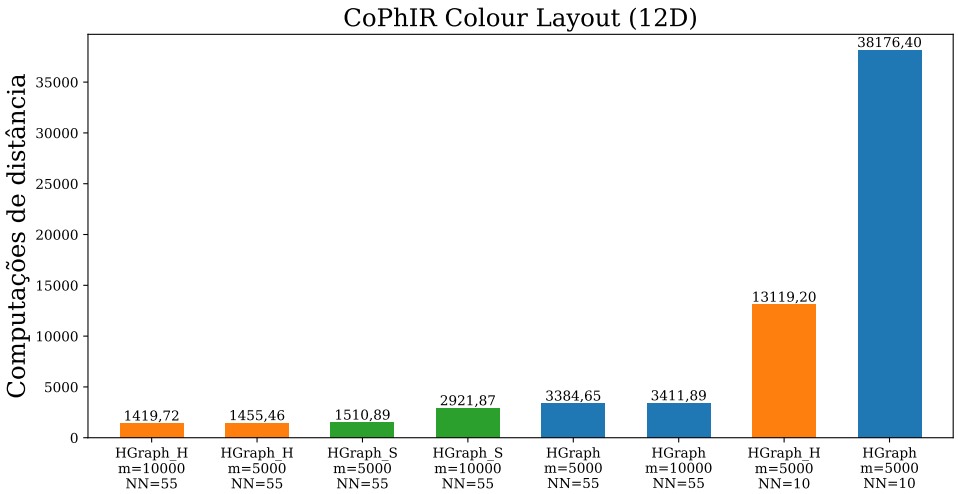


Figura 26 – Configurações com *recall* $\geq$ 0,9 para consultas 10-NN no *CoPhIR Colour Layout*. Neste caso, apenas 8 foram capazes de atingir esse valor.

$m = 10000$ e $NN = 55$. Apenas 8 configurações são exibidas no gráfico pois elas foram as únicas capazes de atingir um *recall* de 0,9 ou mais nos experimentos realizados.

6 CONCLUSÃO

Com o avanço da tecnologia, há cada vez mais dados complexos sendo produzidos, como imagens, sons e dados georreferenciados. Ao mesmo tempo, cresce a necessidade de armazenar e consultar esses dados de maneira eficiente ao passo que surgem aplicações mais e mais sofisticadas. As consultas são geralmente feitas através de buscas por similaridade.

Para acelerar essas buscas, índices apropriados devem ser utilizados. Por se tratar de dados complexos, índices tradicionais como a *B-Tree* geralmente não são adequados devido a muitos deles dependerem da ordenação dos dados, o que não faz sentido nesse contexto. Assim, várias estruturas que conseguem lidar com esse tipo de dados foram propostas na literatura, como o *Locality Sensitive Hashing* [26], a *M-tree* [27] e o grafo de proximidade *Navigable Small World* [9].

Recentemente, índices baseados em aprendizado de máquina, denominados *learned indexes*, receberam o foco de vários estudos [13, 14, 15, 16]. Os resultados apresentados em vários trabalhos indicam que as técnicas de aprendizado de máquina tem capacidade de contribuir com a qualidade e velocidade de muitos tipos de consulta. Alguns índices propostos como o *Learned Metric Index* [13] tratam especificamente de dados complexos.

Seguindo essa linha, a contribuição principal deste trabalho foi um novo *learned index* baseado em grafos construídos com particionamento, como o *HGraph* [1]. O índice se aproveita do particionamento realizado durante a construção do grafo para treinar um classificador (um modelo único ou uma hierarquia de modelos), o qual é capaz de prever uma posição inicial adequada para determinada busca se iniciar. Esse treinamento constitui um aprendizado supervisionado, trazendo o(s) modelo(s) como uma estrutura auxiliar ao grafo e ainda dependendo desse grafo para efetivamente realizar as consultas.

Experimentos foram feitos a fim de testar as diferenças entre a abordagem de modelo único e a com hierarquia de modelos, além de comparar o novo índice com o grafo original (sem predição). Foi concluído que: a abordagem hierárquica é mais adequada para *datasets* mais complexos e grafos construídos com mais partições; com grafos *k-NNG*, parâmetros NN muito altos podem diminuir as vantagens de se utilizar predição; e, no geral, ambas as abordagens foram capazes de reduzir o número de cálculos de distância necessário para se atingir um *recall* satisfatório.

6.1 Trabalhos futuros

Como trabalhos futuros, *datasets* maiores e/ou com mais dimensionalidade devem ser testados, a fim de verificar se os ganhos são intensificados nesses casos. Além disso,

uma implementação mais otimizada, sem trocas de contexto, deve ser feita para que seja possível comparar tempos de construção e consulta. Possivelmente, essa implementação seria realizada em C++ para acompanhar o *HGraph*. Isso abriria caminho para comparações mais adequadas com outros métodos da literatura, como o próprio LMI. Também, outros tipos de modelos de classificação, como redes neurais, devem ser explorados para compará-los com a *random forest*.

REFERÊNCIAS

[1] SHIMOMURA, L. C.; KASTER, D. S. Hgraph: a connected-partition approach to proximity graphs for similarity search. In: SPRINGER. *International Conference on Database and Expert Systems Applications*. [S.l.], 2019. p. 106–121.

[2] MARCUS, R. et al. Benchmarking learned indexes. *arXiv preprint arXiv:2006.12804*, 2020.

[3] BAYER, R.; MCCREIGHT, E. Organization and maintenance of large ordered indices. In: *Proc. 1970 ACM-SIGFIDENT Workshop Data Description and Access*. [S.l.: s.n.], 1970. p. 107–141.

[4] HOLST, A. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2024. *Statista. Dec*, v. 3, 2020.

[5] SANTOS, L. F. D. *Similaridade em big data*. Tese (Doutorado) — Universidade de São Paulo, 2017.

[6] HJALTASON, G. R.; SAMET, H. Index-driven similarity search in metric spaces (survey article). *ACM Transactions on Database Systems (TODS)*, ACM New York, NY, USA, v. 28, n. 4, p. 517–580, 2003.

[7] NOVAK, D.; BATKO, M.; ZEZULA, P. Metric index: An efficient and scalable solution for precise and approximate similarity search. *Information Systems*, Elsevier, v. 36, n. 4, p. 721–733, 2011.

[8] TRAINA, C. et al. Slim-trees: High performance metric trees minimizing overlap between nodes. In: SPRINGER. *International Conference on Extending Database Technology*. [S.l.], 2000. p. 51–65.

[9] MALKOV, Y. et al. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, Elsevier, v. 45, p. 61–68, 2014.

[10] HAJEBI, K. et al. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In: *Twenty-Second International Joint Conference on Artificial Intelligence*. [S.l.: s.n.], 2011.

[11] LI, W. et al. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, IEEE, v. 32, n. 8, p. 1475–1488, 2019.

[12] AOYAMA, K. et al. Fast similarity search in small-world networks. In: *Complex Networks*. [S.l.]: Springer, 2009. p. 185–196.

[13] ANTOL, M. et al. Learned metric index—proposition of learned indexing for unstructured data. *Information Systems*, Elsevier, v. 100, p. 101774, 2021.

[14] DING, J. et al. Alex: an updatable adaptive learned index. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. [S.l.: s.n.], 2020. p. 969–984.

[15] DING, J. et al. Tsunami: A learned multi-dimensional index for correlated data and skewed workloads. *arXiv preprint arXiv:2006.13282*, 2020.

[16] KRASKA, T. et al. The case for learned index structures. In: *Proceedings of the 2018 International Conference on Management of Data*. [S.l.: s.n.], 2018. p. 489–504.

[17] FALOUTSOS, C. *Searching multimedia databases by content*. [S.l.]: Springer Science & Business Media, 2012. v. 3.

[18] KASTER, D. d. S. *Tratamento de condições especiais para busca por similaridade em bancos de dados complexos*. Tese (Doutorado) — Universidade de São Paulo, 2012.

[19] ZEZULA, P. et al. *Similarity search: the metric space approach*. [S.l.]: Springer Science & Business Media, 2006. v. 32.

[20] DEZA, M. M.; DEZA, E. Encyclopedia of distances. In: *Encyclopedia of distances*. [S.l.]: Springer, 2009. p. 1–583.

[21] AGGARWAL, C. C.; HINNEBURG, A.; KEIM, D. A. On the surprising behavior of distance metrics in high dimensional space. In: SPRINGER. *International conference on database theory*. [S.l.], 2001. p. 420–434.

[22] CHÁVEZ, E. et al. Searching in metric spaces. *ACM computing surveys (CSUR)*, ACM New York, NY, USA, v. 33, n. 3, p. 273–321, 2001.

[23] PAREDES, R.; CHÁVEZ, E. Using the k-nearest neighbor graph for proximity searching in metric spaces. In: SPRINGER. *International Symposium on String Processing and Information Retrieval*. [S.l.], 2005. p. 127–138.

[24] WANG, M. et al. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *arXiv preprint arXiv:2101.12631*, 2021.

[25] DATAR, M. et al. Locality-sensitive hashing scheme based on p-stable distributions. In: *Proceedings of the twentieth annual symposium on Computational geometry*. [S.l.: s.n.], 2004. p. 253–262.

[26] INDYK, P.; MOTWANI, R. Approximate nearest neighbors: towards removing the curse of dimensionality. In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. [S.l.: s.n.], 1998. p. 604–613.

[27] CIACCIA, P.; PATELLA, M.; ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. In: CITESEER. *Vldb*. [S.l.], 1997. v. 97, p. 426–435.

[28] FLOREZ, O. U.; LIM, S. Hrg: A graph structure for fast similarity search in metric spaces. In: SPRINGER. *International Conference on Database and Expert Systems Applications*. [S.l.], 2008. p. 57–64.

[29] HARWOOD, B.; DRUMMOND, T. Fanng: Fast approximate nearest neighbour graphs. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. [S.l.: s.n.], 2016. p. 5713–5722.

[30] OCSA, A.; BEDREGAL, C.; CUADROS-VARGAS, E. A new approach for similarity queries using neighborhood graphs. In: *SBBD*. [S.l.: s.n.], 2007. p. 131–142.

[31] NAVARRO, G. Searching in metric spaces by spatial approximation. *The VLDB Journal*, Springer, v. 11, n. 1, p. 28–46, 2002.

[32] JAROMCZYK, J. W.; TOUSSAINT, G. T. Relative neighborhood graphs and their relatives. *Proceedings of the IEEE*, IEEE, v. 80, n. 9, p. 1502–1517, 1992.

[33] ARYA, S. et al. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, ACM New York, NY, USA, v. 45, n. 6, p. 891–923, 1998.

[34] DONG, W.; MOSES, C.; LI, K. Efficient k-nearest neighbor graph construction for generic similarity measures. In: *Proceedings of the 20th international conference on World wide web*. [S.l.: s.n.], 2011. p. 577–586.

[35] PAREDES, R. et al. Practical construction of k-nearest neighbor graphs in metric spaces. In: SPRINGER. *International Workshop on Experimental and Efficient Algorithms*. [S.l.], 2006. p. 85–97.

[36] UHLMANN, J. K. Satisfying general proximity/similarity queries with metric trees. *Information processing letters*, Elsevier, v. 40, n. 4, p. 175–179, 1991.

[37] FERRAGINA, P.; VINCIGUERRA, G. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proceedings of the VLDB Endowment*, VLDB Endowment, v. 13, n. 8, p. 1162–1175, 2020.

[38] KIPF, A. et al. Radixspline: a single-pass learned index. In: *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. [S.l.: s.n.], 2020. p. 1–5.

[39] CUTLER, A.; CUTLER, D. R.; STEVENS, J. R. Random forests. In: *Ensemble machine learning*. [S.l.]: Springer, 2012. p. 157–175.

[40] BOYTSOV, L.; NAIDAN, B. Engineering efficient and effective non-metric space library. In: SPRINGER. *International Conference on Similarity Search and Applications*. [S.l.], 2013. p. 280–293.

[41] PEDREGOSA, F. et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, JMLR. org, v. 12, p. 2825–2830, 2011.

[42] ORTEGA, M. et al. Supporting ranked boolean similarity queries in mars. *IEEE Transactions on Knowledge and Data Engineering*, IEEE, v. 10, n. 6, p. 905–925, 1998.

[43] BOLETTIERI, P. et al. Cophir: a test collection for content-based image retrieval. *arXiv preprint arXiv:0905.4627*, 2009.