UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DOUGLAS ROLINS DE SANTANA

Junções por Similaridade Aproximadas em Espaços Vetoriais Densos

Goiânia
2023

UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO (TECA) PARA DISPONIBILIZAR VERSÕES ELETRÔNICAS DE TESES

E DISSERTAÇÕES NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei 9.610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou download, a título de divulgação da produção científica brasileira, a partir desta data.

O conteúdo das Teses e Dissertações disponibilizado na BDTD/UFG é de responsabilidade exclusiva do autor. Ao encaminhar o produto final, o autor(a) e o(a) orientador(a) firmam o compromisso de que o trabalho não contém nenhuma violação de quaisquer direitos autorais ou outro direito de terceiros.

1. Identificação do material bibliográfico

[x] Dissertação [ ] Tese [ ] Outro*:_____________

*No caso de mestrado/doutorado profissional, indique o formato do Trabalho de Conclusão de Curso, permitido no documento de área, correspondente ao programa de pós-graduação, orientado pela legislação vigente da CAPES.

Exemplos: Estudo de caso ou Revisão sistemática ou outros formatos.

2. Nome completo do autor

Douglas Rolins de Santana

3. Título do trabalho

Junções por Similaridade Aproximadas em Espaços Vetoriais Densos

4. Informações de acesso ao documento (este campo deve ser preenchido pelo orientador)

Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. Após esse período, a possível disponibilização ocorrerá apenas mediante:
a) consulta ao(à) autor(a) e ao(à) orientador(a);
b) novo Termo de Ciência e de Autorização (TECA) assinado e inserido no arquivo da tese ou dissertação.
O documento não será disponibilizado durante o período de embargo.
Casos de embargo:
- Solicitação de registro de patente;
- Submissão de artigo em revista científica;
- Publicação como capítulo de livro;
- Publicação da dissertação/tese em livro.

Obs. Este termo deverá ser assinado no SEI pelo orientador e pelo autor.

Documento assinado eletronicamente por **Leonardo Andrade Ribeiro**, **Professor do Magistério Superior**, em 04/09/2023, às 16:45, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Douglas Rolins De Santana**, **Usuário Externo**, em 05/09/2023, às 10:46, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **4015650** e o código CRC **DB8467C8**.

Referência: Processo nº 23070.041859/2023-45 SEI nº 4015650

DOUGLAS ROLINS DE SANTANA

Junções por Similaridade Aproximadas em Espaços Vetoriais Densos

Dissertação apresentada ao Programa de Pós–Graduação em Ciência da Computação do Instituto de Informática da Universidade Federal de Goiás, como requisito para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Linha: Metodologias e Técnicas de Computação

Orientador: Prof. Dr. Leonardo Andrade Ribeiro

Goiânia
2023

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

Santana, Douglas Rolins de
Junções por Similaridade Aproximadas em Espaços Vetoriais Densos [manuscrito] / Douglas Rolins de Santana. - 2023.
CI, 101 f.

Orientador: Prof. Dr. Leonardo Andrade Ribeiro.
Dissertação (Mestrado) - Universidade Federal de Goiás, Instituto de Informática (INF), Programa de Pós-Graduação em Ciência da Computação, Goiânia, 2023.
Bibliografia. Apêndice.
Inclui siglas, gráfico, tabelas, algoritmos, lista de figuras, lista de tabelas.

1. junção por similaridade. 2. word embeddings. 3. vetores densos. 4. HNSW. I. Ribeiro, Leonardo Andrade, orient. II. Título.

CDU 004

UNIVERSIDADE FEDERAL DE GOIÁS

INSTITUTO DE INFORMÁTICA

ATA DE DEFESA DE DISSERTAÇÃO

Ata nº **12** da sessão de Defesa de Dissertação de **Douglas Rolins de Santana**, que confere o título de Mestre em Ciência da Computação, na área de concentração em Ciência da Computação.

Aos vinte e quatro dias do mês de agosto de dois mil e vinte e três, a partir das dezenove horas, na sala 150 do INF, realizou-se a sessão pública de Defesa de Dissertação intitulada **"Junções por Similaridade Aproximadas em Espaços Vetoriais Densos"**. Os trabalhos foram instalados pelo Orientador, Professor Doutor Leonardo Andrade Ribeiro (INF/UFG) com a participação dos demais membros da Banca Examinadora: Professor Doutor Marcos Vinicius Naves Bedo (IC/UFF), membro titular externo; Professor Doutor Wellington Santos Martins (INF/UFG), membro titular interno. A realização da banca ocorreu por meio de videoconferência. Durante a arguição os membros da banca não fizeram sugestão de alteração do título do trabalho. A Banca Examinadora reuniu-se em sessão secreta a fim de concluir o julgamento da Dissertação, tendo sido o candidato **aprovado** pelos seus membros. Proclamados os resultados pelo Professor Doutor Leonardo Andrade Ribeiro, Presidente da Banca Examinadora, foram encerrados os trabalhos e, para constar, lavrou-se a presente ata que é assinada pelos Membros da Banca Examinadora, aos vinte e quatro dias do mês de agosto de dois mil e vinte e três.

TÍTULO SUGERIDO PELA BANCA

Documento assinado eletronicamente por **Leonardo Andrade Ribeiro**, **Professor do Magistério Superior**, em 24/08/2023, às 21:24, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Marcos Vinicius Naves Bedo**, **Usuário Externo**, em 24/08/2023, às 21:24, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Wellington Santos Martins**, **Professor do Magistério Superior**, em 24/08/2023, às 21:25, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Douglas Rolins De Santana**, **Usuário Externo**, em 24/08/2023, às 21:25, conforme horário oficial de Brasília, com fundamento no § 3º do art. 4º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **3927224** e o código CRC **6DFCBA6B**.

Referência: Processo nº 23070.041859/2023-45 SEI nº 3927224

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador(a).

Douglas Rolins de Santana

Bacharel em Sistemas de Informação pela Universidade Estadual de Goiás e Especialista em Desenvolvimento de Sistemas para WEB pela Anhanguera Educacional. Professor efetivo de Tecnologia da Informação no Instituto Federal de Educação, Ciência e Tecnologia de Goiás (IFG). Atuou como Diretor de Tecnologia da Informação no IFG de 2013 a 2022. Experiência nas áreas de governança e gestão de TI, desenvolvimento de sistemas, banco de dados, redes e infraestrutura de TI. Atualmente com dedicação em pesquisas na linha de metodologia e técnicas de computação.
Lattes: https://lattes.cnpq.br/6843698978977791
Linkedin: https://www.linkedin.com/in/douglasrolins

Dedico este trabalho a todos aqueles que me apoiaram ao longo desta jornada acadêmica. Em especial, dedico-o à minha esposa, Mayara Vieira Silva, que esteve ao meu lado durante todo o percurso, compartilhando das alegrias, desafios e realizações. Seu amor, compreensão e incentivo foram a força motriz por trás deste trabalho e essenciais para que eu pudesse superar os obstáculos e alcançar meus objetivos. A você, minha amada esposa, dedico este trabalho com todo o meu carinho e gratidão.

Agradecimentos

Expresso aqui meus sinceros agradecimentos a todas as pessoas que contribuíram para a realização deste trabalho.

Em especial, meu profundo agradecimento ao meu estimado orientador, Leonardo Andrade Ribeiro, pela sua orientação, dedicação e expertise ao longo deste percurso. Sua orientação foi fundamental para o desenvolvimento deste trabalho, sendo uma fonte constante de inspiração e aprendizado. Suas sugestões, críticas construtivas e encorajamento foram de valor inestimável, e sou imensamente grato por todo o seu apoio.

Agradeço ao Instituto Federal de Goiás (IFG) pelo incentivo e pela liberação para a realização do mestrado. A oportunidade de aprofundar meus estudos e desenvolver esta pesquisa só foi possível graças ao apoio e suporte oferecidos pela instituição. Sou grato pelo ambiente propício ao aprendizado e crescimento acadêmico proporcionado pelo Instituto.

Também expresso meu agradecimento à Universidade Federal de Goiás por todo o suporte e recursos disponibilizados ao longo desta jornada. Agradeço aos professores, funcionários e colegas de curso pela troca de conhecimentos, discussões enriquecedoras e amizades construídas.

Quero mencionar, de forma especial, os servidores da Diretoria de Tecnologia da Informação do IFG, com quem tive a oportunidade de conviver e aprender durante muitos anos antes de ingressar no mestrado. A experiência e conhecimentos adquiridos nesse período foram de grande valia para minha formação acadêmica e para a realização deste trabalho. Agradeço por todo o apoio, colaboração e amizade compartilhada.

Não posso deixar de mencionar meus familiares e amigos, que estiveram sempre ao meu lado, oferecendo seu apoio incondicional, compreensão e incentivo. Seu amor e encorajamento foram fundamentais para que eu persistisse e enfrentasse os desafios que surgiram ao longo dessa trajetória.

Meus sinceros agradecimentos a todos que fizeram parte desta jornada, pois sem a contribuição de cada um, este trabalho não seria possível.

A verdadeira descoberta consiste em ver o que todos viram e pensar no que ninguém pensou.

Szent-Györgyi,
Albert.

Resumo

Santana, Douglas. **Junções por Similaridade Aproximadas em Espaços Vetoriais Densos**. Goiânia, 2023. 101p. Dissertação de Mestrado. Programa de Pós-Graduação em Ciência da Computação, Instituto de Informática, Universidade Federal de Goiás.

Junção por similaridade é uma operação que retorna pares de objetos cuja similaridade é maior ou igual a um limite especificado, sendo essencial para tarefas como limpeza, mineração e integração de dados. Uma abordagem comum é utilizar representações vetoriais dos dados, como o método TF-IDF, e medir a similaridade entre vetores usando a função cosseno. No entanto, calcular a similaridade para todos os pares de vetores pode ser computacionalmente proibitivo em grandes conjuntos de dados. Algoritmos tradicionais exploram a esparsidade dos vetores e aplicam filtros para reduzir o espaço de comparação. Recentemente, avanços no processamento de linguagem natural resultaram em vetores semanticamente mais ricos, melhorando a qualidade dos resultados. No entanto, esses vetores têm características distintas dos gerados por métodos tradicionais, sendo densos e de alta dimensionalidade. Experimentos preliminares demonstraram que o L2AP, melhor algoritmo conhecido para junção por similaridade, não é eficiente em espaços vetoriais densos. Devido às características intrínsecas de tais vetores, soluções aproximadas baseadas em índices especializados são predominantes para lidar com grandes conjuntos de dados. Nesse contexto, foram investigadas formas de realizar junções por similaridade usando o Hierarchical Navigable Small World (HNSW), um índice baseado em grafos de última geração projetado para buscas aproximadas de k-vizinho mais próximo (kNN). Foi explorado o espaço de projeto de possíveis soluções, variando de alternativas do topo do HNSW à uma integração mais profunda do processamento de junção por similaridade nessa estrutura. Os experimentos realizados demonstraram acelerações de até 2,48 e 3,47 ordens de magnitude em relação ao método exato e à abordagem *baseline*, respectivamente, mantendo taxas de recuperação próximas a 100%.

Palavras–chave

<junção por similaridade, word embeddings, vetores densos, HNSW>

Abstract

Santana, Douglas. **Approximate Similarity Joins over Dense Vector Embeddings.** Goiânia, 2023. 101p. MSc. Dissertation. Programa de Pós-Graduação em Ciência da Computação, Instituto de Informática, Universidade Federal de Goiás.

Similarity Join is an operation that returns pairs of objects whose similarity is greater than or equal to a specified threshold, and is essential for tasks such as cleaning, mining, and data integration. A common approach is to use data vector representations, such as the TF-IDF method, and measure the similarity between vectors using the cosine function. However, computing the similarity for all pairs of vectors can be computationally prohibitive on large data sets. Traditional algorithms exploit the sparsity of vectors and apply filters to reduce the comparison space. Recently, advances in natural language processing have produced in semantically richer vectors, improving the results quality. However, these vectors have different characteristics from those generated by traditional methods, being dense and of high dimensionality. Preliminary experiments demonstrated that L2AP, the best known algorithm for similarity join, is not efficient for dense vector spaces. Due to the intrinsic characteristics of such vectors, approximate solutions based on specialized indices are predominant for dealing with large datasets. In this context, we investigate how to perform similarity joins using the Hierarchical Navigable Small World (HNSW), a state-of-the-art graph-based index designed for approximate k-nearest neighbor (kNN) queries. We explored the design space of possible solutions, ranging from top-end alternatives to HNSW to deeper integration of similarity join processing into this framework. The experiments carried out demonstrated accelerations of up to 2.48 and 3.47 orders of magnitude in relation to the exact method and the baseline approach, respectively, maintaining recovery rates close to 100%.

Keywords

<similarity join, word embeddings, dense vectors, HNSW>

Sumário

Lista de Figuras 13

Lista de Tabelas 14

Lista de Algoritmos 15

1 Introdução 16
1.1 Justificativa 18
1.2 Questões de Pesquisa 19
1.3 Objetivos 19
1.3.1 Objetivo Geral 19
1.3.2 Objetivos Específicos 19
1.4 Organização do Documento 20

2 Referencial Teórico 21
2.1 Definições das Operações de Similaridade 21
2.2 Representação de Dados em Vetores 22
2.2.1 Tokenização 23
2.2.2 Representação com TF-IDF 25
2.2.3 Representação com *Sentence-Transformers* 26
2.3 Operações de Similaridade 29
2.3.1 Funções de Similaridade 29
2.3.2 Operações de Busca 30
2.3.3 Junção por Similaridade 32
2.4 Algoritmo L2AP 34
2.4.1 Filtro de Prefixo 36
2.4.2 Filtro por Tamanho 37
2.4.3 Filtro Residual 37
2.4.4 Poda de Candidatos 38
2.5 Estrutura de Grafos HNSW 38

3 Trabalhos Relacionados 40
3.1 Trabalhos de Junção por Similaridade 40
3.2 Junção por Similaridade em Vetores 41

4 L2AP em Espaços Vetoriais Densos 44
4.1 Experimentos 44
4.2 Resultados 45
4.3 Algoritmo Básico com Filtro da Norma L2 46

5 Junções por Similaridade com Algoritmos ANN 48
5.1 Algoritmos ANN 48
5.2 Experimentos na Junção por Similaridade 49

6 Junções por Similaridade com Grafos HNSW 51
6.1 Junções no Topo do HNSW 51
6.2 Junções por Similaridade Integradas ao HNSW 53

7 Experimentos e Resultados 58
7.1 Método Exato 58
7.2 Abordagem *Baseline* 59
7.3 Configuração dos Experimentos 60
7.4 Conjuntos de Dados 61
7.5 Resultados e Discussão 62
7.5.1 Tempo de Execução 63
7.5.2 Taxa de Recuperação 66
7.5.3 Comparação com Algoritmo L2AP 68
7.5.4 Consumo de Memória 69
7.5.5 Sumário dos Resultados 70

8 Conclusões 71
8.1 Contribuições 72
8.2 Trabalhos Futuros 72
8.3 Publicações Geradas 73

Referências Bibliográficas 74

A Algoritmo L2AP 85
A.1 AllPairs 87
A.1.1 Indexação - AllPairs 87
A.1.2 Geração de Candidatos - AllPairs 88
A.1.3 Verificação - AllPairs 89
A.2 L2AP 90
A.2.1 Indexação - L2AP 90
A.2.2 Geração de Candidatos - L2AP 91
A.2.3 Verificação - L2AP 93

B Algoritmo HNSW 95
B.1 Construção do Grafo 95
B.2 Busca no Grafo 99
B.3 Análise da Complexidade e Consumo de Memória 101

Lista de Figuras

2.1 Exemplo de vetor denso e esparso. 23
2.2 Exemplo de relações vetoriais no word2vec [67]. 27
(a) Relação de gênero 27
(b) Singular/Plural 27
2.3 Exemplo de índice invertido. 34
2.4 Indexação com aplicação do pscore. 36
2.5 Filtragem residual. 37
2.6 Estrutura multicamada do HNSW. 39

4.1 Execução do L2AP em vetores densos x esparsos. 45

7.1 Tempo de execução dos algoritmos. 64

B.1 Ilustração da heurística utilizada para selecionar os vizinhos do grafo HNSW para dois clusters isolados. 99

Lista de Tabelas

2.1 Exemplo de tokenização. 24

4.1 Descrição do dataset DBLP. 45
4.2 Tempo de execução do algoritmo L2AP e NestedL2Join. 47

5.1 Resultados dos experimentos de junção por similaridade com algoritmos ANN. 50

7.1 Descrição dos datasets semissintéticos. 62
7.2 Taxa de recuperação dos algoritmos. 66
7.3 Tempos de execução da junção por similaridade com L2AP em representação esparsa versus HSJ em representação densa. 69
7.4 Consumo de memória do índice HNSW. 69

A.1 Notações utilizadas nos algoritmos. 86
A.2 Equações para o limite *dpscore*. 93

Lista de Algoritmos

2.1	Base-L2AP	35
4.1	NestedL2Join	46
6.1	HSJ-Ext	53
6.2	HSJ-Ths	54
6.3	RANGE-SEARCH	54
6.4	RANGE-SEARCH-LAYER	55
6.5	HSJ-Ths^{Inc}	57
7.1	Método-Exato	59
7.2	Baseline	60
A.1	Index-AllPairs	88
A.2	CandGen-AllPairs	89
A.3	Verify-AllPairs	90
A.4	Index-L2AP	91
A.5	CandGen-L2AP	92
A.6	Verify-L2AP	94
B.1	INSERT	96
B.2	SELECT-NEIGHBORS-SIMPLE	98
B.3	SELECT-NEIGHBORS-HEURISTIC	98
B.4	K-NN-SEARCH	100
B.5	SEARCH-LAYER	100

CAPÍTULO 1

Introdução

O contínuo desenvolvimento de sistemas para apoiar processos de negócios e proporcionar soluções tecnológicas tem impulsionado empresas a coletar e armazenar volumes cada vez maiores de dados. Para lidar com essa quantidade de dados, são utilizadas diversas fontes para construir repositórios de armazenamento altamente escaláveis, conhecidos como *Data Lakes* [74]. Esses repositórios são capazes de manter grandes quantidades de dados brutos em seu formato nativo.

Para facilitar uma melhor tomada de decisão, esses repositórios são explorados por meio de tarefas analíticas. Portanto, a qualidade dos dados obtidos nesses repositórios é um aspecto crucial no processo de análise, uma vez que dados sujos (dados duplicados ou incorretos) podem comprometer os resultados [25]. Pesquisas em ciência de dados e aprendizado de máquina mostram que profissionais que lidam com dados gastam a maior parte do tempo em atividades relacionadas à compreensão e análise dos dados [54], e que dados sujos são uma barreira comum enfrentada por esses profissionais [53]. Além disso, o processo de limpeza e preparação dos dados pode consumir mais tempo do que o próprio processo de análise. Portanto, investir em soluções que otimizem essa etapa do processo é crucial para minimizar o tempo necessário para obter resultados.

Além da presença de dados sujos, um desafio adicional é lidar com dados que representam a mesma entidade do mundo real. A correspondência entre entidades refere-se ao problema de identificar quais instâncias de dados se referem à mesma entidade do mundo real [71, 94, 63]. Esse problema tem sido extensivamente estudado e abordado por várias técnicas, no entanto, ainda é um desafio significativo e há espaço para melhorias [8].

Para lidar com os desafios mencionados anteriormente, muitas aplicações fazem uso da operação de junção por similaridade, que busca pares de objetos cuja similaridade exceda um limite pré-definido [9]. Essa operação emprega funções de similaridade, as quais calculam o grau de semelhança entre duas entidades ou objetos [87].

Essas funções de similaridade fornecem medidas numéricas que representam o quão similares são dois objetos. A escolha da função apropriada depende da tarefa a ser realizada e também deve estar alinhada com o tipo de representação dos dados utilizada.

Por exemplo, para dados representados em conjuntos, pode-se utilizar o coeficiente de Jaccard ou o coeficiente de Dice. Para espaços vetoriais, a similaridade de cosseno ou a distância euclidiana podem ser empregadas. Já para sequências, a distância de edição é comumente utilizada [57].

Antes da aplicação de uma função de similaridade, os dados precisam ser representados em um formato que permita a realização de operações. O espaço vetorial tem sido amplamente utilizado para representação de dados [90], especialmente em dados semiestruturados e em aplicações mais recentes no processamento de linguagem natural (PNL) [43]. A depender do método de representação utilizado, podem ser gerados vetores de baixa ou alta dimensão, densos ou esparsos, o que pode ter impacto nos custos computacionais da operação e na qualidade dos resultados obtidos.

Dentre os métodos tradicionais para representação de dados no espaço vetorial destaca-se o TF-IDF. Este método utiliza estatísticas para atribuir valores aos componentes textuais. No entanto, métodos mais recentes, como o *word2vec* [68], *glove* [80] e *fasttext* [69], empregam redes neurais para definir o contexto das palavras e aplicar uma melhor representação.

Mais recentemente, o *framework Sentence-Transformers* [85] tem se destacado. Ele se baseia em modelos de PNL de última geração, utilizando a arquitetura *Transformer* [101], como o BERT [30], para obter a representações semanticamente significativas de sentenças textuais por meio de vetores densos de tamanho fixo.

Há uma tendência crescente no processamento de vetores densos impulsionada por modelos modernos de aprendizado de máquina. Estes modelos geram representações vetoriais que retêm informações semanticamente relevantes de dados textuais bem como de outros dados não estruturados complexos, como imagem e áudio. Este paradigma de processamento de dados baseado em representações vetoriais densas fornece a espinha dorsal de uma ampla gama de aplicações, incluindo recomendação, pesquisa de vídeo, recuperação de imagem-texto e resposta a perguntas, entre muitas outras [102].

No entanto, observa-se uma ausência de trabalhos que abordem especificamente a operação de junção por similaridade nesses espaços vetoriais. Conforme mencionado anteriormente, essa operação desempenha um papel crucial no processamento de dados e na realização de tarefas relacionadas à recuperação e análise de informações. Diante desse contexto, o presente trabalho se dedica a investigar e desenvolver técnicas para otimizar a junção por similaridade em dados representados por vetores densos gerados por modelos de aprendizado de máquina.

Os trabalhos de junção por similaridade têm se concentrado em reduzir a complexidade da operação ao reduzir o espaço de comparação. Vários algoritmos de junção por similaridade, como AllPairs [9] e L2AP [2], exploram a esparsidade da representação vetorial para derivar filtros e reduzir o número de computações da similaridade.

Estes espaços vetoriais gerados por modelos de aprendizado de máquina possuem características que tornam desafiadora a junção por similaridade em grandes conjuntos de dados. Primeiro, esses vetores são de alta dimensionalidade, o que torna muitos métodos de indexação ineficazes devido à conhecida maldição da dimensionalidade [21]. Em segundo, os modelos de aprendizado de máquina geram vetores densos, ou seja, todas as dimensões contêm valores diferentes de zero, em contraste com métodos tradicionais de *tokenização* baseados em termos que geram vetores esparsos onde a maioria das dimensões é zero.

Uma abordagem comum para lidar com o processamento de vetores densos é utilizar soluções aproximadas, onde algoritmos sacrificam a precisão em troca de um melhor desempenho. Neste contexto, ressalta-se soluções de indexação e busca do *k*-vizinho mais próximo (*k*NN), que retorna os k vetores mais semelhantes a um vetor de consulta. Dentre as soluções mais populares, destaca-se o *Hierarchical Navigable Small World* (HNSW) [65], considerado como estado da arte por estudos de referência [6]. Resumindo, HNSW é uma organização hierárquica em memória de aproximações de gráficos de Delaunay com parâmetros ajustáveis para controlar contrapartidas de desempenho e recuperação. Porém, essa estrutura de indexação foi projetada para busca *k*NN e não oferece suporte nativo a consultas baseadas em limite de similaridade necessário na operação de junção por similaridade.

Neste contexto, o presente trabalho investiga a operação de junção por similaridade em espaços vetoriais densos, avaliando a aplicação do algoritmo L2AP nesses vetores e explorando algoritmos de buscas *k*NN, com ênfase no HNSW, para realizar essa operação.

1.1 Justificativa

A utilização de técnicas de junção por similaridade em espaços vetoriais enfrenta desafios relevantes, como a alta dimensionalidade e a enorme quantidade de dados, o que pode tornar computacionalmente proibitiva a realização dessa operação. Embora existam métodos desenvolvidos para lidar com esses desafios, como a redução do espaço de comparação por meio de filtros derivados de vetores esparsos e diferentes estruturas de indexação, ainda há uma carência de pesquisas voltadas especificamente para otimizar a junção por similaridade em espaços vetoriais densos gerados por modelos de aprendizado de máquina.

A necessidade de lidar de forma eficiente com a crescente geração e processamento de dados em espaços vetoriais densos, bem como a busca por soluções que permitam explorar todo o potencial das representações semânticas geradas por modelos de aprendizado de máquina, são os principais motivadores para a realização desta pesquisa.

Além disso, a operação de junção por similaridade desempenha um papel crucial em diversas áreas, como na análise, limpeza e integração de dados, identificação de duplicatas e recomendação de conteúdo, onde a capacidade de identificar objetos semelhantes ou relacionados é essencial para a obtenção de resultados precisos e relevantes.

Portanto, este trabalho busca investigar métodos que possam aprimorar a operação de junção por similaridade em espaços vetoriais densos, buscando reduzir o tempo de processamento, aumentar a escalabilidade para conjuntos de dados volumosos e aproveitar ao máximo as representações semânticas geradas por modelos de aprendizado de máquina. Espera-se que os resultados obtidos contribuam tanto para o avanço científico nessa área quanto para o desenvolvimento de aplicações práticas que dependem da operação de junção por similaridade, permitindo uma melhor exploração e compreensão dos dados em espaços vetoriais densos.

1.2 Questões de Pesquisa

Para guiar o processo de pesquisa, foram definidas as seguintes questões:

QP1. Como as características dos espaços vetoriais densos afetam o desempenho de algoritmos existentes de junção por similaridade?

QP2. É possível alcançar desempenho comparável aos métodos tradicionais de representação na junção por similaridade em espaços vetoriais densos gerados por modelos de aprendizado de máquina?

QP3. É viável otimizar uma técnica de busca do vizinho mais próximo (kNN) para a operação de junção por similaridade em espaços vetoriais densos?

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo geral deste trabalho é desenvolver e avaliar técnica de junção por similaridade em dados representados por vetores densos gerados por modelos de aprendizado de máquina.

1.3.2 Objetivos Específicos

Como objetivos específicos para esta pesquisa têm-se:

- Avaliar a aplicação, em espaços vetoriais densos, de algoritmo de junção por similaridade com função cosseno do estado da arte;

- Adaptar e avaliar a aplicação de algoritmos de busca kNN, otimizados para espaços vetoriais densos, na junção por similaridade;
- Desenvolver técnica de junção por similaridade otimizada para espaços vetoriais densos;
- Realizar experimentos para avaliar a técnica proposta, bem como analisar os resultados obtidos e discutir possíveis melhorias e limitações.

1.4 Organização do Documento

O restante do documento está organizado da seguinte forma: O Capítulo 2 aborda o referencial teórico do trabalho, apresentando os principais conceitos e técnicas relacionadas às operações de similaridade em espaços vetoriais. Em seguida, o Capítulo 3 apresenta trabalhos relacionados, com uma revisão bibliográfica sobre trabalhos anteriores relacionados ao tema desta pesquisa.

No Capítulo 4, é tratada a aplicação do algoritmo de junção por similaridade L2AP em espaços vetoriais densos. O Capítulo 5 aborda a utilização de algoritmos de busca do vizinho mais próximo para a junção por similaridade em vetores densos.

O Capítulo 6 é dedicado a apresentar as principais contribuições deste trabalho, com foco nas técnicas de junções por similaridade com grafos HNSW. Em seguida, o Capítulo 7 apresenta os experimentos realizados, incluindo as configurações utilizadas, os resultados obtidos e uma análise detalhada dos mesmos.

Por fim, o Capítulo 8 traz as conclusões desta pesquisa, contendo uma discussão sobre os resultados obtidos, as contribuições alcançadas e possíveis trabalhos futuros. Além disso, são apresentadas as publicações geradas durante o desenvolvimento deste trabalho de pesquisa.

CAPÍTULO 2

Referencial Teórico

O presente capítulo apresenta os principais conceitos e definições fundamentais para o entendimento e desenvolvimento desta pesquisa. Inicialmente, são apresentadas as definições formais das operações de similaridade contidas neste trabalho (Seção 2.1). Em seguida, são abordados temas como a representação de dados textuais em vetores (Seção 2.2), as operações de similaridade utilizadas para medir a proximidade entre vetores (Seção 2.3), além dos algoritmos L2AP e HNSW (Seções 2.4 e 2.5, respectivamente).

2.1 Definições das Operações de Similaridade

Assume-se um conjunto de vetores $\mathcal{V}$ de dimensionalidade fixa n. Dado dois vetores x e y, seja $sim(x,y)$ uma função de similaridade comutativa que retorna um valor em $[0,1]$. Definem-se formalmente as operações de similaridade relevantes para o contexto deste trabalho da seguinte forma:

Definição 2.1 (Busca kNN) *Dado um vetor de consulta x e um inteiro k, uma busca kNN sobre $\mathcal{V}$ retorna o conjunto de respostas $\mathcal{A}_{knn} = \{(\{y_1, ..., y_k\} \subseteq \mathcal{V} : \forall y \in \mathcal{A}_{kNN}$ e $\forall y' \in \mathcal{V} - \mathcal{A}_{kNN}, sim(x,y) \geq sim\left(x,y'\right)\}$.*

Definição 2.2 (Busca por Similaridade) *Dado um vetor de consulta x e um limite de similaridade τ, uma busca por similaridade em $\mathcal{V}$ retorna o conjunto de respostas $\mathcal{A}_{ss} = \{y \in \mathcal{V} : sim(x,y) \geq \tau\}$.*

Definição 2.3 (Junção por Similaridade) *Dado um limite de similaridade $\tau \in [0,1]$, uma junção por similaridade sobre $\mathcal{V}$ retorna o conjunto de respostas $\mathcal{A}_{sj} = \{(x,y) \in \mathcal{V} \times \mathcal{V} : sim(x,y) \geq \tau\}$.*

Obviamente, pode-se calcular a junção por similaridade realizando uma busca por similaridade para cada $x \in \mathcal{V}$ em uma junção de loop aninhado. O Lema a seguir expressa a condição de contenção e equivalência entre buscas kNN e por similaridade e, por sua vez, estabelece uma conexão entre buscas kNN e junção por similaridade.

Lema 2.4 *Dado um vetor de consulta x, considere dois métodos de busca, kNN e SS, sobre $\mathcal{V}$: a primeira é uma busca kNN e retorna o conjunto de respostas $\mathcal{A}_{knn}$ para um determinado valor k, e o último é uma busca por similaridade e retorna o conjunto de respostas $\mathcal{A}_{ss}$ para um determinado valor limite τ. Seja y_k o vetor de menor similaridade em $\mathcal{A}_{knn}$, ou seja, $y_k \in \mathcal{A}_{knn}$ e $\forall y \in \mathcal{A}_{knn}, sim(x, y_k) \leq sim(x, y)$. Além disso, seja y_{k+1} o vetor com maior similaridade que não está em $\mathcal{A}_{knn}$, ou seja, $y_{k+1} \notin \mathcal{A}_{knn}$ e $\forall y' \notin \mathcal{A}_{knn}, sim(x, y_{k+1}) \geq sim\left(x, y'\right)$. Se $sim(x, y_{k+1}) < \tau$, então SS está contido em kNN como $\mathcal{A}_{ss} \subseteq \mathcal{A}_{knn}$. Se $sim(x, y_{k+1}) < \tau$ e $sim(x, y_k) \geq \tau$, então SS e kNN são equivalentes, ou seja, $\mathcal{A}_{ss} \subseteq \mathcal{A}_{knn}$ e $\mathcal{A}_{knn} \subseteq \mathcal{A}_{ss}$.*

Prova por contradição: Seja $sim(x, y_{k+1}) < \tau$ e existe um vetor $y \in \mathcal{A}_{ss} \setminus \mathcal{A}_{knn}$, ou seja, $\mathcal{A}_{ss} \nsubseteq \mathcal{A}_{knn}$. Se $y \in \mathcal{A}_{ss}$, então $sim(x, y) \geq \tau$; mas se $y \notin \mathcal{A}_{knn}$, então $sim(x, y_{k+1}) \geq sim(x, y) \geq \tau$, o que contradiz $sim(x, y_{k+1}) < \tau$. Agora, seja $sim(x, y_k) \geq \tau$ e existe um vetor $y \in \mathcal{A}_{knn} \setminus \mathcal{A}_{ss}$, ou seja, $\mathcal{A}_{knn} \nsubseteq \mathcal{A}_{ss}$. Se $y \in \mathcal{A}_{knn}$, então $sim(x, y_k) \leq sim(x, y)$; mas se $y \notin \mathcal{A}_{ss}$, então $sim(x, y_k) \leq sim(x, y) < \tau$, o que contradiz $sim(x, y_k) \geq \tau$.

O Lema 2.4 não aborda a situação de empate nas extremidades do conjunto $\mathcal{A}_{knn}$, ou seja, quando $sim(x, y_k) = sim(x, y_{k+1})$. No entanto, é importante observar que essa situação não afetará a corretude dos algoritmos de junção por similaridade apresentados.

Neste trabalho, o foco está na similaridade do cosseno e assume-se que todos os vetores de entrada foram normalizados para comprimento unitário, ou seja, $||x|| = 1, \forall x \in \mathcal{V}$. Assim, a similaridade do cosseno entre dois vetores x e y corresponde ao seu produto escalar: $sim(x, y) \equiv \langle x, y \rangle \equiv \sum_{i=1}^{n} x_i \cdot y_i$, onde x_i é o valor da i-ésima dimensão de x.

2.2 Representação de Dados em Vetores

Para possibilitar o processamento de textos por sistemas computacionais, é necessário representá-los em uma estrutura que permita a realização de operações, como a verificação da similaridade entre eles. Um modelo amplamente utilizado nesse contexto é o modelo do espaço vetorial [90], especialmente no processamento de linguagem natural. Nesse modelo, documentos, textos ou palavras, dependendo do nível de granularidade definido, são representados por vetores. Dessa forma, o cosseno entre dois vetores fornece uma medida de similaridade entre eles, que pode ser usada para classificar os resultados. Essa abordagem permite a aplicação de técnicas de recuperação de informações, classificação de textos e outras tarefas relacionadas ao processamento de linguagem natural.

Ao longo do tempo, várias técnicas foram desenvolvidas para representar dados textuais em vetores, com o objetivo de medir a similaridade entre textos [44]. Dependendo

da técnica utilizada, podem ser gerados vetores esparsos ou densos, com alta ou baixa dimensionalidade. Em um vetor esparso, a maioria das dimensões contém o valor zero, enquanto em um vetor denso todas as dimensões possuem valores diferentes de zero. A escolha entre vetores esparsos e densos também está relacionada à dimensionalidade intrínseca dos dados, ou seja, à quantidade de dimensões necessárias para capturar de forma eficaz as características dos textos e suas relações. A Figura 2.1 ilustra a diferença entre esses dois tipos de vetores.

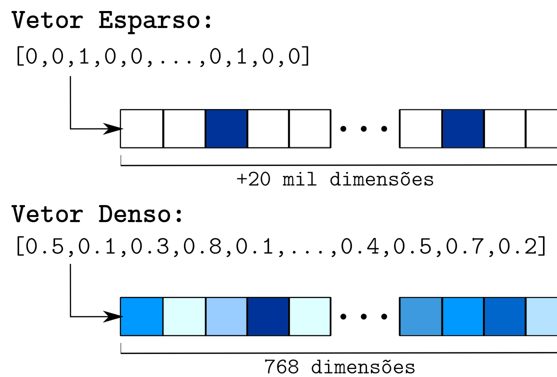


Figura 2.1: Exemplo de vetor denso e esparso.

Esta pesquisa concentra-se em duas abordagens para representar dados textuais em vetores:

- Representação com método tradicional TF-IDF: Nessa abordagem, o processo de *tokenização* (Seção 2.2.1) é aplicado ao texto para dividir em unidades significativas, como palavras ou caracteres. Em seguida, são gerados vetores esparsos utilizando a medida TF-IDF (Seção 2.2.2), que pondera a importância dos termos nos documentos com base na frequência e na raridade nos dados textuais;
- Representação com modelos de redes neurais pré-treinadas: Nessa abordagem, busca-se capturar um maior significado semântico dos dados textuais. Neste trabalho, foi utilizado o *framework Sentence-Transformers* para gerar vetores densos que representam o conteúdo dos textos (Seção 2.2.3).

2.2.1 Tokenização

Uma tarefa inicial para representar dados textuais em vetores é a *tokenização*, onde o texto é dividido em unidades básicas e atômicas chamadas *tokens* [47]. Esses *tokens* podem representar palavras individuais ou subdivisões do texto, como *q-grams*. A *tokenização* permite a transformação do texto em uma sequência estruturada de elementos que podem ser processados de forma mais eficiente e utilizados na construção de representações vetoriais.

Na *tokenização* por palavras, cada palavra do conjunto de dados é considerada como um token individual. Essa abordagem é comumente utilizada e é considerada a forma mais simples de realizar a *tokenização*.

Na *tokenização* por q-grams, os *tokens* são gerados deslizando uma janela de tamanho q nos caracteres do texto [100]. Para garantir que cada *token* tenha tamanho q, são adicionados caracteres especiais no início e no final do texto, com tamanho $q-1$. Esses caracteres extras permitem que a janela de deslizamento percorra todo o texto e gere os *tokens* corretamente. A técnica por q-grams pode ser aplicada nos casos em que a estrutura interna das palavras é relevante para a aplicação, em vez de apenas palavras individuais.

Por exemplo, considere o texto "Exemplo de token", ao realizar a *tokenização* por palavra, 2-gram e 3-gram, são gerados os *tokens* conforme Tabela 2.1. Este exemplo é apenas para demonstrar a diferença nos tipos de *tokenização* e não considera outros tratamentos do texto, como remoção de *stop-words*.

Tokenização	**Entrada**	***Tokens***
Palavra (words)	Exemplo de token	{"Exemplo","de","token"}
q-gram, q = 2	#Exemplo de token#	{"#E","Ex","xe","em","mp","pl", "lo","o "," d","de","e "," t","to", "ok","ke","en","n#"}
q-gram, q = 3	##Exemplo de token##	{"##E","#Ex","Exe","xem","emp", "mpl","plo","lo ","o d"," de","de ", "e t"," to","tok","oke","ken","en#", "n##"}

Tabela 2.1: Exemplo de tokenização.

A escolha do tipo de *tokenização* depende da tarefa em questão. Por exemplo, para a identificação de dados sujos, a *tokenização* por palavras pode não ser adequada, pois palavras com erros de digitação seriam tratadas como *tokens* diferentes, dificultando sua identificação. Nesse caso, a *tokenização* por q-grams pode ser mais adequada para capturar a similaridade entre os dados.

A partir da *tokenização*, é possível atribuir valores e pesos aos *tokens* para obter um vetor numérico que representa o texto. Abordagens mais tradicionais, como TF-IDF (Seção 2.2.2), utilizam medidas estatísticas para determinar esses valores. Por outro lado, técnicas mais recentes de processamento de linguagem natural se baseiam em aprendizado de máquina, utilizando redes neurais para aprender a representação de texto com maior significado semântico. Essas técnicas podem se beneficiar do uso de modelos pré-treinados em grandes *corporas*[1], como Word2Vec [68], GloVe [80], FastText

[1]Conjunto de textos escritos ou falados em um ou mais idiomas.

[69], BERT [30] e GPT-3 [16]. Além disso, o *framework Sentence-Transformers* (Seção 2.2.3) oferece modelos pré-treinados especializados em codificar e extrair informações semânticas de sentenças, permitindo a obtenção de vetores de alta qualidade e relevância semântica para representar o texto.

2.2.2 Representação com TF-IDF

Em um conjunto de dados textual com N sentenças, o TF-IDF (*Term Frequency-Inverse Document Frequency*) é uma medida estatística que fornece um valor representativo para cada *token* em uma sentença. Essa medida calcula a importância de cada *token* com base na frequência com que ele ocorre em uma determinada sentença (TF - *Term Frequency*) e na frequência inversa desse *token* no conjunto de dados como um todo (IDF - *Inverse Document Frequency*). O TF-IDF é o produto dessas duas medidas, e seu valor resultante é usado para avaliar a relevância de um *token* em relação a uma sentença específica e ao conjunto de dados em geral.

A medida TF representa a frequência de um *token* em uma sentença específica. Para normalizar essa frequência e evitar que *tokens* que aparecem muitas vezes dominem a pontuação, realiza-se a divisão da frequência pelo total de *tokens* presentes na sentença (Definição 2-1). Assim, os *tokens* mais frequentes na sentença terão uma pontuação mais alta em relação aos menos frequentes.

$$TF_{(t)} = \frac{tc}{st}, \tag{2-1}$$

onde tc representa a quantidade de vezes que o *token* t aparece na sentença e st a quantidade total de *tokens* desta sentença.

Ao considerar que características raras são mais importantes para a avaliação da semelhança das sentenças [89], o IDF obtém o valor de cada *token* de forma inversa à sua frequência em todo o conjunto de dados. É calculada a partir do número total de sentenças, dividindo-o pelo número de sentenças que contêm o *token* e aplicando o logaritmo (Definição 2-2). Assim, quanto mais próximo de 0 significa que mais comum é o token, caso contrário, mais raro será o *token*.

$$IDF_{(t)} = \log\left(\frac{N}{sc(t)}\right), \tag{2-2}$$

onde N representa o total de sentenças do conjunto de dados e $sc(t)$ representa a quantidade de sentenças em que o *token* t aparece no conjunto de dados.

Alguns *tokens* serão mais comuns que outros e, portanto, os valores de TF serão muito maiores. Neste caso, multiplicar pelo inverso da frequência no documento (IDF) obterá um valor que melhor representa o *token* na sentença e assim se obtém o TF-IDF (Definição 2-3).

$$TF\text{-}IDF_{(t)} = TF_{(t)} \cdot IDF_{(t)} \tag{2-3}$$

Desta forma, é levado em consideração tanto a importância do *token* na sentença quanto a raridade do *token* em todo o conjunto de dados. Assim, tokens mais frequentes terão pontuação baixa, enquanto *tokens* mais raros, que aparecem com menos frequência no conjunto de dados, terão pontuação mais alta. É importante mencionar que, além dessa formulação do TF-IDF baseada em frequência e raridade, existem outras formulações possíveis [77].

Com a técnica TF-IDF, cada sentença é representada por um vetor, onde cada dimensão desse vetor corresponde a um *token* único no conjunto de todos os *tokens* da base de dados. Portanto, no vetor TF-IDF, apenas as dimensões dos *tokens* presentes na sentença terão valores diferentes de zero. Como o universo de *tokens* geralmente é alto e a quantidade de *tokens* por sentença é baixa, os vetores resultantes tendem a ser esparsos e de alta dimensionalidade.

2.2.3 Representação com *Sentence-Transformers*

A representação TF-IDF calcula o valor de cada *token* independentemente, levando em consideração sua frequência no objeto e no conjunto de dados, sem considerar diretamente o contexto e o relacionamento semântico entre os *tokens*.

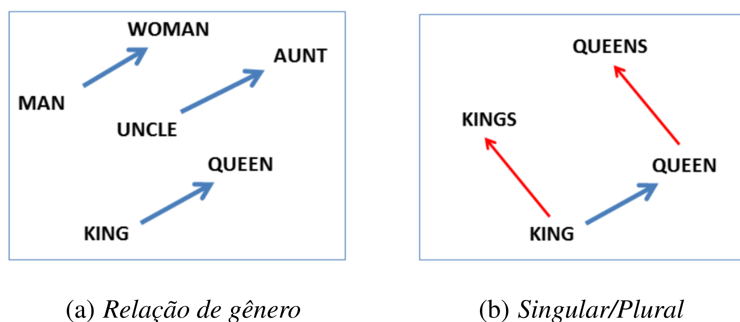
No entanto, surgiram outras abordagens para representar o texto, com destaque para o trabalho de Mikolov *et al.* [68], que trouxe avanços significativos na captura dos relacionamentos semânticos. Essas abordagens utilizam a técnica de *word embeddings* para gerar vetores densos e de dimensionalidade fixa.

Os *word embeddings* são obtidos por meio de técnicas de representação não supervisionada, em que palavras com significado semelhante são mapeadas para vetores próximos uns aos outros. Cada dimensão do vetor denso gerado representa uma característica importante do texto, capturando propriedades sintáticas e semânticas relevantes [98]. Esses modelos são frequentemente construídos usando redes neurais, que aprendem a prever o contexto das palavras em um texto. Além de dados textuais, a geração de *embeddings* também é amplamente utilizada na representação de outros tipos de dados, como imagens [95].

A abordagem de *word embeddings* tem se mostrado eficaz para capturar as relações semânticas entre as palavras e tem sido amplamente utilizada em diversas

tarefas de processamento de linguagem natural, como busca de informações, tradução automática, sumarização de texto, entre outras.

Além das questões sintáticas, como demonstrado por Mikolov *et al.*, os *word embeddings* também permitem capturar relações semânticas entre as palavras por meio do deslocamento dos vetores. Por exemplo, eles mostraram que a operação vetorial *"vetor('King') - vetor('Man') + vetor('Woman')"* resulta em um vetor que está mais próximo da representação vetorial da palavra *"Queen"* (Figura 2.2(a)). Essa propriedade de deslocamento vetorial permite realizar operações semânticas interessantes, como obter o singular ou plural de uma palavra (Figura 2.2(b)).



(a) *Relação de gênero* (b) *Singular/Plural*

Figura 2.2: Exemplo de relações vetoriais no word2vec [67].

O trabalho de Mikolov *et al.* impulsionou diversas pesquisas na área de processamento de linguagem natural a fim de obter melhores representações do texto. Diversas técnicas surgiram, como GloVe [80], FastText [69] e Universal Sentence Encoder [18], cada uma com suas abordagens para capturar informações semânticas das palavras.

No entanto, uma das abordagens de maior impacto na área foi a introdução da arquitetura *Transformer* [101]. Essa arquitetura utiliza um mecanismo de auto-atenção para calcular as representações de uma sequência, dispensando o uso de recorrência e convoluções, que eram amplamente utilizados em técnicas anteriores de aprendizado de máquina. O mecanismo de auto-atenção permite o relacionamento de diferentes posições da sequência para calcular sua representação e demonstrou desempenho e capacidade de generalização superior às técnicas anteriores.

A partir do *Transformer*, surgiram modelos que representam o estado da arte em várias aplicações de processamento de linguagem natural, como BERT [30], T5 [83] e GPT-3 [16]. Esses modelos conseguem generalizar as representações de texto de maneira superior aos modelos anteriores, que eram construídos para casos de uso específicos [15]. O BERT (*Bidirectional Encoder Representations from Transformers*), em particular, é um modelo pré-treinado em grandes quantidades de dados e disponibilizado publicamente, o que impulsionou seu uso em diversas tarefas, como perguntas e respostas e classificação.

A utilização de modelos baseados em BERT ou outros modelos do tipo *Transformer* para obter representações vetoriais de objetos textuais e avaliar a similaridade entre esses vetores pode parecer uma escolha óbvia. No entanto, esses modelos não foram originalmente projetados para esse propósito específico, tornando essa abordagem inviável em termos de custo computacional. Isso ocorre porque os modelos *Transformer* operam em nível de palavra (*token*) e não capturam diretamente o contexto de sentenças inteiras. Portanto, para comparar a similaridade entre duas sentenças, seria necessário alimentar ambas no modelo, o que demandaria muito tempo para os cálculos de inferência, especialmente quando se busca o par mais semelhante em todo o conjunto de dados [85].

Uma alternativa para o modelo BERT é representar uma sentença calculando a média das representações dos *tokens* que a compõem. No entanto, essa abordagem geralmente resulta em representações de sentenças de baixa qualidade e até inferiores em comparação com modelos anteriores como o GloVe [85].

Para superar essas limitações e obter representações vetoriais de sentenças mais precisas usando redes neurais com menor custo computacional, foi desenvolvido o SBERT [85]. O SBERT permite a geração de representações vetoriais de sentenças sem a necessidade de realizar cálculos de inferência completos. A partir dessas representações, é possível avaliar a similaridade semântica entre pares de sentenças utilizando a medida de similaridade de cosseno.

Além do SBERT, foram desenvolvidos outros modelos que superaram seu desempenho na tarefa de representação vetorial de sentenças. Esses modelos estão disponíveis publicamente e podem ser acessados através da organização Hugging Face[2], que hospeda diversos modelos pré-treinados para processamento de linguagem natural.

O Hugging Face disponibiliza esses modelos através de um *framework* chamado *Sentence-Transformers*, desenvolvido pelos mesmos autores do SBERT, Reimers e Gurevych. Esse *framework* permite uma troca simplificada de modelos, permitindo gerar representações de sentenças utilizando os modelos mais recentes e avançados disponíveis.

Segundo Reimers e Gurevych, a complexidade para encontrar um par de sentenças mais semelhantes em um conjunto de 10 mil sentenças, reduz de 65 horas com o BERT para 5 segundos com SBERT na fase de representação dos dados, e mais 0.01 segundos para o cálculo do cosseno. Uma base de 10 mil sentenças é relativamente pequena, sendo que para bases bem maiores, como 1 milhão de sentenças ou mais, técnicas para otimizar a fase de similaridade de cosseno podem ser úteis. Com base nisso, esta pesquisa investiga a junção por similaridade (Seção 2.3.3) em vetores gerados com uso do *Sentence-Transformers*.

[2]https://huggingface.co/sentence-transformers

2.3 Operações de Similaridade

As operações de similaridade são empregadas para calcular a proximidade entre objetos em conjuntos de dados. Essas operações permitem avaliar o quão semelhantes dois objetos estão em termos de suas características e são aplicadas em várias áreas, como recomendação de conteúdo, recuperação de informações, busca em bancos de dados, clusterização, classificação de documentos, dentre muitos outros [75, 27, 36, 32].

Uma abordagem quantitativa para medir a similaridade entre objetos é por meio de funções de similaridade (Seção 2.3.1). Essas funções fornecem uma medida numérica que representa a proximidade entre os objetos. Nesse contexto, as operações de similaridade podem ser classificadas em **busca *k*NN** (*k-nearest neighbors*) ou **busca por similaridade**, podendo os resultados serem **exatos** ou **aproximados** (Seção 2.3.2). Além das operações de busca, destaca-se a operação denominada **junção por similaridade**, que combina pares de objetos de um ou mais conjuntos de dados com base em sua similaridade (Seção 2.3.3).

2.3.1 Funções de Similaridade

Para determinar a similaridade entre objetos em um conjunto de dados, é necessário aplicar uma medida numérica que possibilite a comparação entre eles. Nesse contexto, as funções de similaridade desempenham o papel de fornecer essa medida, permitindo avaliar o grau de similaridade entre dois objetos [87].

Resumidamente, considerando dois objetos x e y em um conjunto de dados, uma função de similaridade $sim(x, y)$ retorna um valor no intervalo $[0, 1]$ (no caso de valores normalizados) que indica o grau de semelhança entre x e y. Quanto mais próximo de 1 for o valor retornado, maior será a semelhança entre os objetos.

Em relação a conjuntos de dados textuais é possível classificar as funções de similaridade em dois tipos: baseadas em *tokens* e baseadas em caracteres [44].

As funções de similaridade baseadas em *tokens* (consulte a Seção 2.2.1) envolvem a transformação do texto em *tokens*, utilizando conjuntos ou vetores para representar os dados. No caso de vetores, cada dimensão representa a presença, frequência ou uma medida como TF-IDF atribuída ao *token* (consulte a Seção 2.2.2). Para vetores com representação semântica, cada dimensão do vetor representa uma característica significativa do texto (consulte a Seção 2.2.3). Nessas funções, a similaridade entre dois objetos, representados por vetores, por exemplo, é calculada com base na distância entre eles. Alguns exemplos de funções tradicionais baseadas em *tokens* são jaccard, dice e cosseno [87].

Por outro lado, nas funções baseadas em caracteres, a similaridade é calculada com base na quantidade de operações necessárias para transformar o texto de consulta

no texto de entrada. Essa abordagem também é conhecida como similaridade baseada em edição. Uma das funções clássicas nesse tipo é a função de Levenshtein [59].

A escolha da função de similaridade deve estar em consonância com a forma de representação dos dados. Por exemplo, as similaridades de Jaccard ou Dice são usadas para representação baseadas conjuntos ou vetores binários, a similaridade de cosseno é apropriada para vetores, e as similaridades baseadas em edição são utilizadas para sequências [57].

Dado que o objetivo deste trabalho é investigar a junção por similaridade em dados com representação semântica utilizando a estrutura de vetores, foi escolhida a função cosseno para calcular a similaridade. Essa função tem sido amplamente utilizada em diversos domínios devido à sua alta qualidade de resultados, precisão e desempenho comprovados [97, 2, 39, 20, 55].

Para padronizar o conjunto de dados e facilitar a comparação entre os objetos, é possível atribuir pesos ponderados aos *tokens* e normalizar o vetor para ter uma unidade de comprimento. Essa operação pode ser realizada calculando a magnitude inicial do vetor (Definição 2-4) e, em seguida, ajustando o valor de cada *token* dividindo-o pela magnitude inicial do vetor (Definição 2-5). Isso garante que todos os vetores tenham a mesma escala e facilita a comparação de similaridade entre eles.

$$||x|| = \sqrt{x_1^2 + x_2^2 + x_3^2 + \ldots + x_d^2} \tag{2-4}$$

$$x_i = \frac{x_i}{||x||} \tag{2-5}$$

Considerando os vetores $x = [x_1, x_2, \ldots, x_d]$ e $y = [y_1, y_2, \ldots, y_d]$, que foram previamente normalizados para terem uma unidade de comprimento, ou seja, $||x|| = 1$ e $||y|| = 1$, a similaridade do cosseno entre x e y pode ser calculada como o produto escalar entre eles (Definição 2-6).

$$sim(x,y) \equiv \frac{\sum_{i=1}^{d} x_i \cdot y_i}{||x|| \cdot ||y||} \equiv \langle x, y \rangle \equiv \sum_{i=1}^{d} x_i \cdot y_i \tag{2-6}$$

2.3.2 Operações de Busca

A operação de busca é o processo de encontrar os objetos mais semelhantes em um conjunto de dados em relação a um objeto de consulta específico. Duas abordagens comuns para esta tarefa são:

- **Busca** k**NN**, onde os objetos são representados em um espaço vetorial comum e os k vetores mais próximos são retornados com base em uma função de similaridade como cosseno (Definição 2.1);

- **Busca por similaridade**, onde todos os objetos com similaridade igual ou acima de um limite pré-estabelecido (*threshold*) ao objeto de consulta são retornados (Definição 2.2).

A busca, seja kNN ou por similaridade, utilizando o método por força bruta, consiste em comparar o vetor de consulta com cada vetor candidato no conjunto de dados. No entanto, esse método pode ser computacionalmente custoso ou até mesmo inviável, especialmente em grandes conjuntos de dados e vetores de alta dimensionalidade.

Para melhorar a eficiência da busca em conjuntos de dados com alta dimensionalidade, foram desenvolvidas estruturas de dados especializadas e técnicas de quantização e redução de dimensionalidade [33, 79]. Essas abordagens visam otimizar o processo de busca e reduzir o custo computacional necessário para encontrar os objetos mais similares.

Existem duas categorias principais de soluções para a busca: **métodos exatos** e **aproximados**. As soluções exatas retornam os k-vizinhos mais próximos verdadeiros para o vetor de consulta no caso da busca kNN ou todos os objetos acima do *threshold* na busca por similaridade. Já nos métodos aproximados, a precisão dos resultados é sacrificada em troca de eficiência computacional. Os algoritmos de busca kNN aproximados também são conhecidos como algoritmos ANN (*Approximate Nearest Neighbor*) [6].

Embora tenham sido propostas diversas técnicas para a busca eficiente e exata [34, 62, 58, 99], o desempenho dessas abordagens pode ser insatisfatório para muitas aplicações [61]. Isso levou a um crescente interesse da comunidade de pesquisa no desenvolvimento de soluções aproximadas, especialmente em buscas kNN aproximadas [6]. Essas soluções podem ser classificadas em abordagens baseadas em árvores [73, 114], baseadas em *hash* [48, 49] e baseadas em grafos [42, 65, 41, 92, 108, 108], cada uma com características e propriedades específicas que visam melhorar a eficiência da busca. Destaca-se que as abordagens baseadas em árvores ou grafos também são utilizadas por métodos exatos [3, 119].

As abordagens baseadas em árvores utilizam estruturas de dados hierárquicas, como árvores *KD-tree*, *R-tree*, *ball tree* e *M-Tree*, para organizar os dados de forma eficiente e acelerar a busca por objetos semelhantes. Essas estruturas dividem o espaço multidimensional em regiões menores, permitindo a redução do espaço de busca necessário. Dessa forma, a busca é otimizada ao evitar a comparação com objetos distantes no espaço.

Já as abordagens baseadas em *hash* aproveitam funções de *hash* para mapear os vetores em chaves de busca. Essas funções transformam os vetores em valores *hash* que são usados para indexar e agrupar objetos semelhantes. Isso possibilita uma busca rápida e eficiente, pois os objetos semelhantes são agrupados em *buckets* com base em seus valores de *hash*.

Por fim, as abordagens baseadas em grafos representam os dados como grafos, onde cada nó representa um vetor e as arestas representam a similaridade entre os

vetores. Essa representação permite a aplicação de algoritmos de travessia de grafos para encontrar objetos semelhantes. Ao percorrer os nós do grafo, é possível identificar vizinhos próximos e calcular a similaridade entre eles.

Ressalta-se ainda as técnicas propostas para lidar com a enorme quantidade de dados encontrados em muitas aplicações. Entre essas técnicas, destacam-se a quantização [79, 52, 46] e a redução da dimensionalidade [1, 115]. A quantização envolve a representação dos vetores originais por vetores binários compactos, utilizando métodos como o produto escalar quantizado (QEP) e o produto escalar binário (BSP). Essa abordagem permite reduzir o tamanho dos vetores e, consequentemente, os custos computacionais da busca. Já a redução da dimensionalidade consiste em projetar os vetores em um espaço de menor dimensionalidade, utilizando técnicas como a análise de componentes principais (PCA) e a decomposição em valores singulares (SVD). Essas técnicas preservam, na medida do possível, a estrutura de similaridade entre os vetores, permitindo uma busca mais eficiente.

É importante destacar que essas técnicas de quantização ou redução da dimensionalidade geralmente são combinadas com abordagens baseadas em *hash*, árvores ou grafos, a fim de aprimorar ainda mais o desempenho da busca [51, 120].

Dentre as abordagens mencionadas, destaca-se a estrutura de grafos HNSW para busca aproximada kNN. Essa estrutura tem sido amplamente utilizada em plataformas de busca semântica [102, 51] e é considerada uma das abordagens mais avançadas [6, 92, 92]. Sua eficiência e desempenho foram comprovados experimentalmente, conforme apresentado na Seção 5.1.

Diante disso, este trabalho de pesquisa se concentrou no estudo aprofundado da estrutura de grafos HNSW, detalhada na Seção 2.5, para a realização da operação de junção por similaridade em vetores densos.

2.3.3 Junção por Similaridade

A junção por similaridade é uma operação que consiste na identificação de pares de objetos semelhantes em uma base de dados. Para tanto, são utilizadas funções de similaridade (Seção 2.3.1) para determinar a semelhança entre os objetos. Essa operação é formalizada na Definição 2.3. Uma abordagem básica é percorrer cada objeto da base de dados e realizar uma busca por similaridade para encontrar seus correspondentes semelhantes. Sob esta perspectiva, a operação de busca pode ser considerada como uma variação especializada do processo de junção.

O termo *"junção por similaridade"* possui uma ampla utilização na área de Bancos de Dados, designando a operação descrita. De acordo com Cohen [26], a junção por similaridade une objetos de uma base de dados em pares cuja similaridade não é

inferior a um limite especificado. Esse limite de similaridade também é referido pelo termo *"threshold"* ao longo do texto. Além disso, existem outras variações terminológicas para esta operação, tais como *"busca de similaridade para todos os pares"* [9] e *"Fuzzy Join"* [22]. Esta última terminologia é empregada em contextos de correspondência de *strings*, visando identificar todos os pares semelhantes de acordo com uma função de similaridade específica.

No contexto deste trabalho, o foco está na auto-junção, que se refere à aplicação da junção por similaridade em uma única base de dados. É importante ressaltar que as técnicas desenvolvidas e exploradas para a auto-junção podem ser adaptadas e estendidas para a aplicação da junção por similaridade em diferentes bases de dados, ampliando assim o escopo e a utilidade dessas abordagens.

A junção por similaridade têm sido extensivamente estudada na área de Banco de Dados devido à sua aplicabilidade em diversas tarefas da área. Ela é amplamente utilizada em correspondência de dados textuais, detecção de duplicatas, processamento de texto e são consideradas operações fundamentais em sistemas gerenciadores de bancos de dados [19, 104, 50]. Além disso, esta operação desempenha um papel importante em tarefas como integração e limpeza de dados [25, 22, 70, 113, 117], em processos de correspondência de entidades [64, 94, 78], bem como em aplicações de recomendação de conteúdo, recuperação de informações e análise de similaridade em geral.

Uma abordagem básica para a operação de junção por similaridade envolve percorrer todos os objetos no conjunto de dados e realizar uma operação de busca para cada objeto, procurando por seus objetos similares. No entanto, essa abordagem pode ser ineficiente, pois possui uma complexidade quadrática, o que torna inviável para conjuntos de dados grandes.

Para lidar com esse desafio, foram propostos diversos algoritmos que visam melhorar a eficiência da operação. Esses algoritmos se baseiam principalmente em técnicas de filtragem, que têm como objetivo reduzir o espaço de comparação, focando apenas nos objetos mais promissores em termos de similaridade.

Os algoritmos de junção por similaridade podem ser classificados de acordo com o tipo de representação dos dados para os quais a técnica foi otimizada. Entre as principais categorias, destacam-se a baseada em caracteres [116, 111, 103, 104, 107], em conjuntos [97, 112, 87, 66, 110, 24, 117, 86] e em vetores [9, 57, 2].

A junção por similaridade baseada em caracteres está relacionada à comparação de sequências de caracteres. O objetivo é encontrar pares de *strings* que sejam semelhantes de acordo com alguma medida de similaridade, como distância de edição ou distância Levenshtein.

A junção por similaridade em conjuntos diz respeito à comparação de conjuntos de elementos, representados por *tokens*. O objetivo é encontrar pares de conjuntos que

possuam elementos em comum acima de um limite de similaridade pré-estabelecido, calculado por medidas como dice, jaccard ou cosseno.

A junção por similaridade em vetores, por sua vez, é empregada quando os dados são representados como vetores numéricos. Nesse caso, a similaridade é medida por meio de funções para o espaço vetorial, como cosseno ou distância euclidiana. Essa abordagem encontra aplicações relevantes em áreas como recuperação de informação, mineração de dados e aprendizado de máquina, onde a semelhança entre vetores é um fator crítico.

É importante ressaltar que essas categorias não são mutuamente exclusivas, e muitas vezes os algoritmos podem ser adaptados para lidar com diferentes tipos de representação de dados.

No escopo deste trabalho de pesquisa, a ênfase está na junção por similaridade em dados representados em espaços vetoriais densos, utilizando a função cosseno como medida de similaridade. Nesse contexto, será discutido de maneira mais aprofundada o algoritmo L2AP [2], reconhecido como estado da arte para junção por similaridade com função cosseno.

O algoritmo L2AP se destaca pela introdução de novas estratégias de poda, que o colocam à frente de métodos de referência, como AllPairs [9], MMJoin [57] e BayesLSH [91], resultando em melhorias significativas de desempenho. Devido à sua relevância, esta pesquisa investigou sua aplicação em espaços vetoriais densos.

A próxima seção apresentará uma descrição do funcionamento do algoritmo L2AP, discutindo suas principais etapas e estratégias utilizadas.

2.4 Algoritmo L2AP

O algoritmo L2AP [2], proposto por Anastasiu e Karypis, é fundamentado em abordagem baseada em índices invertidos construídos dinamicamente (Figura 2.4).

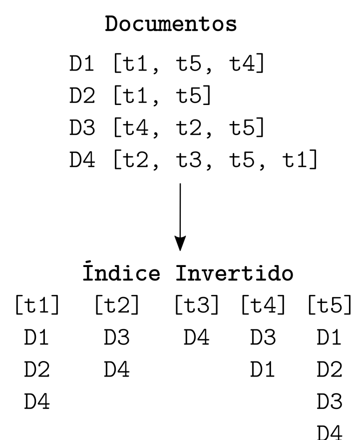


Figura 2.3: Exemplo de índice invertido.

A estrutura do algoritmo é composta por três fases principais: a fase de indexação, na qual é construído o índice invertido; a fase de geração de candidatos, que produz os pares de possíveis candidatos; e a fase de verificação, responsável por calcular a similaridade dos pares candidatos. Esta estrutura básica é demonstrada no Algoritmo 2.1.

Algoritmo 2.1: Base-L2AP

Entrada: Conjunto de vetores D e *threshold* τ

Saída: Conjunto de resultados $\mathcal{A}_{sj}$

```
1  A_sj ← ∅
2  I ← ∅ // Inicializa Índice Invertido
3  para cada x ∈ D faça
4      C ← GeraCandidatos(x, I, τ) // Produz pares de possíveis candidatos de x
5      A_ss ← VerificaCandidatos(x, C, I, τ) // Calcula a similaridade dos
           candidatos e retorna pares acima do τ
6      A_sj ← A_sj ∪ A_ss // Inclui pares com similaridade acima do τ no
           resultado
7      para cada x_j ∈ x ∧ x_j > 0 faça // Indexa tokens do vetor
8          I_j ← I_j ∪ {(x, x_j)}
9          x_j ← 0
10 retorna A_sj
```

Em cada uma das fases do algoritmo, são aplicadas técnicas de filtragem para melhorar a eficiência geral, reduzindo o tamanho do índice invertido, eliminando pares candidatos falsos-positivos e interrompendo o processamento da similaridade de pares que não atingirão o *threshold*. Entre as técnicas empregadas, destacam-se a filtragem de prefixo (Seção 2.4.1), filtragem por tamanho (Seção 2.4.2), filtragem residual (Seção 2.4.3) e poda de candidatos (Seção 2.4.4).

O L2AP incorpora técnicas de filtragem do algoritmo AllPairs [9], além de empregar suas próprias estratégias. Ele se beneficia de uma ordenação pré-definida dos vetores, que devem estar dispostos em ordem decrescente com base no *token* de maior valor em cada vetor, e os *tokens* de cada vetor em ordem decrescente de frequência.

Uma das estratégias de filtragem consiste em reduzir o tamanho do índice invertido ao indexar apenas os *tokens* do vetor a partir do ponto em que é provável que se atinja o *threshold* com os demais vetores. Essa estratégia divide o vetor em duas partes distintas: o prefixo (x'), que corresponde à porção não indexada do vetor x, e o sufixo (x''), que corresponde à parte já indexada.

Os principais filtros empregados pelo L2AP são baseados nos limites gerados pela desigualdade de Cauchy-Schwarz. Ela estabelece que a similaridade entre dois veto-

res é menor ou igual ao produto das normas dos vetores, ou seja $sim(x, y) \leq ||x|| \cdot ||y||$.

No caso do L2AP, todos os vetores devem ser normalizados previamente para um comprimento unitário, o que implica que norma do vetor candidato y é menor ou igual a 1, ou seja $||y|| \leq 1$. Portanto, ao considerar o prefixo x' do vetor x, temos que a similaridade entre x' e y é limitada pela norma de x', como segue: $sim(x', y) \leq ||x'|| \cdot ||y|| \leq ||x'||$. Consequentemente, podemos concluir que $sim(x', y) \leq ||x'||$.

Esses limites mais restritos permitem ao algoritmo L2AP gerar um índice invertido menor, além de filtrar e descartar um número maior de pares candidatos falsos-positivos.

O pseudo-código do algoritmo L2AP, juntamente com os detalhes das otimizações implementadas, pode ser encontrado no Apêndice A. As seções seguintes fornecem uma visão geral das técnicas de filtragem empregadas.

2.4.1 Filtro de Prefixo

A técnica proposta por Chaudhuri *et al.* [19], conhecida como técnica de filtragem de prefixo, requer uma ordenação prévia dos *tokens* para ser aplicada. Essa técnica é utilizada na fase de indexação e tem como objetivo reduzir o tamanho do índice invertido, indexando apenas alguns *tokens* que garantem a presença de pelo menos um *token* em comum entre o vetor e seus possíveis candidatos.

É importante destacar que o termo *token* utilizado no contexto do espaço vetorial, se refere à identificação da dimensão do vetor à qual o *token* está associado.

Para determinar quais *tokens* serão incluídos no índice invertido, é estabelecido um limite chamado de *pscore*, que representa a maior similaridade possível ao combinar os primeiros elementos diferente de zero do vetor x com qualquer outro vetor no conjunto de dados. Assim que o *pscore* é alcançado, significa que o vetor será capaz de atingir o *threshold* com outro vetor, e, portanto, os *tokens* subsequentes são incluídos no índice invertido (Figura 2.4.1). A parte não indexada do vetor é denotada por x', que corresponde ao prefixo residual do vetor x. Por outro lado, a parte indexada é representada por x'' e é conhecida como sufixo do vetor x.

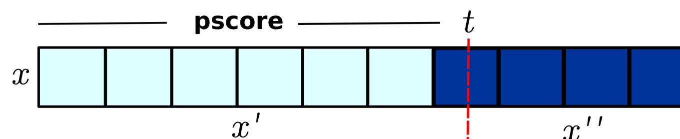


Figura 2.4: Indexação com aplicação do pscore.

2.4.2 Filtro por Tamanho

Tendo como premissa que vetores de tamanhos muito diferentes não são semelhantes, é possível eliminar um vetor se o seu tamanho for menor que um determinado limite. Essa técnica de filtragem é amplamente utilizada pelos algoritmos, juntamente com o filtro de prefixo [66]. No caso específico de vetores ordenados em ordem decrescente com base no valor do *token* como no L2AP, o tamanho mínimo necessário para que um vetor atinja o limite de similaridade com outro vetor pode ser calculado como a divisão do *threshold* pelo maior valor do *token* presente no vetor de consulta (Definição 2-7).

$$minsize = \frac{\tau}{rw_x}, \tag{2-7}$$

onde τ representa o *threshold* e rw_x o maior valor dentre os os *tokens* no vetor de consulta.

2.4.3 Filtro Residual

Na fase de geração de candidatos, é aplicado um filtro conhecido como *remscore* para eliminar os candidatos presentes no índice invertido cuja similaridade máxima com o vetor de busca atual seja menor que o *threshold* estabelecido. Esse filtro foi introduzido por Bayardo *et al.* [9].

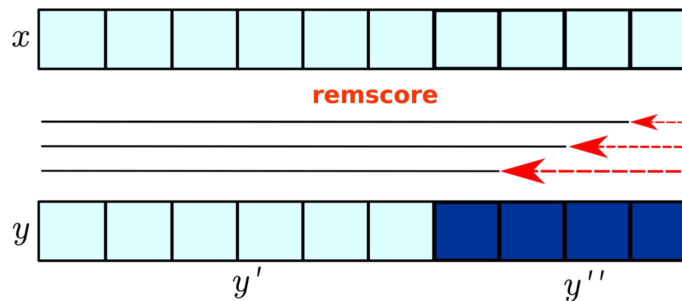


Figura 2.5: Filtragem residual.

Inicialmente, o valor máximo de similaridade entre o vetor x e qualquer outro vetor candidato é atribuído ao *remscore*. À medida que cada candidato é processado, é verificado se o valor do *remscore* está abaixo do *threshold*. Caso isso ocorra, o candidato pode ser descartado, uma vez que não alcançará o *threshold* determinado. É importante ressaltar que, para aplicar esse filtro, a iteração no vetor de busca deve ser realizada em ordem inversa em relação à ordem utilizada durante a indexação. Essa estratégia de filtragem é ilustrada na Figura 2.5.

2.4.4 Poda de Candidatos

A poda de candidatos é uma otimização realizada durante a fase de verificação, embora também possa ser aplicada na fase de geração de candidatos. Essa técnica utiliza vários limites para interromper o cálculo do produto escalar entre pares de vetores que, de acordo com esses limites, não seriam capazes de atingir o *threshold* de similaridade desejado.

Esses limites estimam a maior similaridade possível entre o vetor x e o vetor y. A cada iteração do cálculo do produto escalar, caso seja verificado que não será possível alcançar o *threshold*, a computação do produto entre esses vetores é interrompida.

2.5 Estrutura de Grafos HNSW

O Hierarchical Navigable Small Worlds (HNSW) é uma estrutura de indexação que se baseia em uma organização de camadas de grafos de proximidade [65]. Em cada camada, é construída uma aproximação do grafo de Delaunay [23], preservando apenas os links para os vizinhos mais próximos de cada nó.

O HNSW utiliza uma heurística para adicionar links de longo alcance e garantir as propriedades de navegação de um mundo pequeno [56], possibilitando a obtenção de complexidade logarítmica nas operações de busca. Isso permite a rápida navegação pelo grafo e a redução do espaço de busca para regiões mais promissoras.

A Figura 2.6 ilustra a estrutura multicamadas do HNSW, que se assemelha a uma estrutura *skip list*[3], onde os grafos de proximidade substituem as listas encadeadas. Cada vetor é representando por um nó que pode estar presente em mais de uma camada. A camada máxima de um nó é determinada por meio de uma escolha aleatória com distribuição de probabilidade de decaimento exponencial, o que resulta em uma escala logarítmica do número de camadas na estrutura.

A navegação pelo grafo inicia-se a partir do nó raiz na camada superior e segue em direção aos nós vizinhos que possuem a maior similaridade com o nó de consulta. Devido à maior distância entre os nós nas camadas superiores, a navegação ocorre de maneira mais rápida nessas camadas. À medida que a busca progride em direção à camada base, os nós mais similares à consulta são determinados nessa camada.

O HNSW foi projetado para buscas aproximadas kNN, o que implica que o conjunto de respostas $\mathcal{A}_{knn}$ pode conter elementos que não pertencem aos k-vizinhos mais próximos do vetor de consulta. Os parâmetros efConstruction (`efC`) e efSearch (`ef`) determinam o número de vizinhos considerados durante a construção e busca do grafo

[3]Estrutura de dados probabilística com base em listas ligadas e com eficiência na ordem de $O(log(n))$ para a maioria das operações.

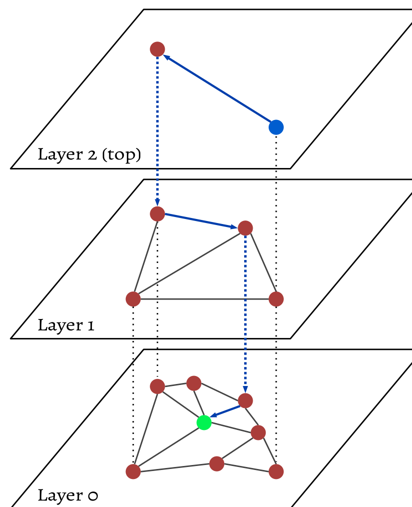


Figura 2.6: Estrutura multicamada do HNSW.

de proximidade, respectivamente, controlando assim a contrapartida entre desempenho e recuperação. Para cada elemento inserido, o número de conexões estabelecidas com seus vizinhos mais próximos é determinado pelo parâmetro `M`.

Para interromper uma busca, além dos parâmetros mencionados, também é utilizado um mínimo local, que representa um vetor com menor similaridade dentro dos `ef` vetores desejáveis, mas que possui maior similaridade ao objeto de consulta em relação aos demais vetores encontrados durante a travessia no grafo. No entanto, destaca-se que essa abordagem pode resultar em um "falso mínimo local", onde um vetor que, em um determinado momento durante o processo de busca, é considerado como mínimo local e a busca é interrompida, mas outros vetores ainda mais similares poderiam existir e que foram negligenciados. Essa situação pode ocorrer devido à natureza aproximada da construção do índice HNSW e também pode ser influenciado pela escolha dos parâmetros do algoritmo e pela distribuição dos dados no espaço vetorial.

O pseudo-código do algoritmo HNSW, juntamente com detalhes de seu funcionamento, pode ser encontrado no Apêndice B.

HNSW é considerado o estado da arte para buscas aproximadas *k*NN [6], superando soluções concorrentes no contexto de avaliação de junção por similaridade, conforme demonstrado no Capítulo 5, e é incorporado por sistemas gerenciadores de bancos de dados vetoriais modernos, como o Milvus [102].

Com base nas características do HNSW e nos resultados de ponta para busca *k*NN, este trabalho de pesquisa se concentrou no desenvolvimento de técnicas de junção por similaridade com utilização dessa estrutura. Foram propostos algoritmos específicos para realizar a junção por similaridade aproximada, e detalhes sobre esses algoritmos são apresentados no Capítulo 6.

CAPÍTULO 3

Trabalhos Relacionados

Este capítulo apresenta estudos relacionados ao tema desta pesquisa, com foco nos trabalhos que propõem métodos e algoritmos utilizados na junção por similaridade e busca em espaços vetoriais densos. Esses estudos têm como objetivo explorar técnicas avançadas de junção por similaridade, levando em consideração aspectos como escalabilidade, desempenho, precisão e otimização da busca.

Em geral, os estudos que se dedicam ao desenvolvimento de técnicas de similaridade entre objetos podem ser categorizados com base em diversos aspectos: (1) o tipo de operação, que pode envolver busca, retornando objetos similares a um elemento de consulta, ou junção, retornando pares de elementos similares; (2) o critério de geração do resultado, que pode ser os k objetos mais similares ou todos os objetos acima de um determinado *threshold*; e (3) o tipo de resultado, que pode ser exato ou aproximado. Além disso, esses estudos podem ser classificados de acordo com o tipo de representação dos dados para os quais a técnica é otimizada, como *strings*, conjuntos ou vetores, bem como a função de similaridade utilizada.

É importante ressaltar que este trabalho de pesquisa se concentrou na operação de junção com resultados aproximados acima de um *threshold*, em dados representados em vetores densos, utilizando a função cosseno.

3.1 Trabalhos de Junção por Similaridade

O trabalho realizado por Papadakis et al. [78] apresenta uma revisão abrangente dos algoritmos de junção por similaridade, com um enfoque especial nas técnicas de filtragem. De acordo com a pesquisa realizada, as técnicas de filtragem baseadas em prefixo têm sido amplamente utilizadas e têm demonstrado eficiência na aceleração da junção por similaridade [9, 14, 19, 81, 87, 105, 109, 113]. Essa abordagem utiliza uma estratégia onde partes iniciais dos vetores são consideradas para filtrar os candidatos. Além disso, as técnicas de filtragem baseadas em partição também foram exploradas, porém em menor frequência [5, 29, 60].

A revisão realizada revela que os algoritmos aproximados [24, 91, 118] são mais adequados para lidar com limites de baixa similaridade, enquanto os algoritmos exatos tendem a apresentar um desempenho superior em limites mais altos. Essa constatação ressalta a importância de selecionar a técnica de filtragem adequada com base no limite de similaridade desejado.

Além disso, os estudos incluídos na revisão abordam técnicas mais avançadas, como a correspondência de *strings* baseada em *substrings* ou abreviações [96, 106], cálculo de similaridade entre conjuntos considerando *matching* aproximado de *tokens* [28, 107] e a combinação de vários predicados de similaridade [86, 17]. Essas abordagens mais complexas têm demonstrado resultados promissores no desafio da junção por similaridade, ampliando as possibilidades e a precisão dos métodos utilizados.

No entanto, é importante considerar que o desempenho das técnicas de filtragem pode variar dependendo da implementação específica e das características do conjunto de dados em análise. Estudos que realizaram avaliações experimentais das técnicas desenvolvidas relatam resultados variados [40, 50, 66], o que destaca a importância de adaptar as técnicas de filtragem às particularidades do problema em questão.

Na área de bancos de dados, inúmeros estudos foram realizados com o objetivo de aprimorar o desempenho da junção por similaridade, principalmente em relação a resultados exatos [87, 117]. Esses pesquisadores dedicaram considerável atenção ao desenvolvimento de técnicas específicas para dados representados como conjuntos.

Adicionalmente, alguns algoritmos de junção por similaridade têm explorado técnicas de paralelismo, processamento distribuído e utilização de GPUs para melhorar o desempenho e a escalabilidade [4, 88, 31, 82, 12, 93]. Essas abordagens visam aproveitar os recursos computacionais disponíveis para acelerar o processo de junção por similaridade em grandes volumes de dados.

3.2 Junção por Similaridade em Vetores

Na junção por similaridade em vetores, ressalta-se um paradigma amplamente adotado para otimizar a operação, que consiste na utilização de técnicas de filtragem, geração de candidatos e computação da similaridade baseadas em índice invertido. Essa abordagem impulsionou a escalabilidade dos métodos, permitindo o processamento eficiente de conjuntos de dados cada vez maiores. O AllPairs [9] é um algoritmo de destaque nessa área, sendo amplamente referenciado e servindo de base para o desenvolvimento de outros trabalhos subsequentes.

O algoritmo L2AP, proposto por Anastasiu e Karypis [2], segue a mesma estrutura básica do algoritmo AllPairs e representa o estado da arte para a junção por similaridade exata em vetores com função cosseno. Uma contribuição significativa do algoritmo

L2AP é a introdução de técnicas avançadas de filtragem baseadas na desigualdade de Cauchy-Schwarz. Essas técnicas de filtragem demonstraram um desempenho superior, mesmo quando comparadas à técnica aproximada BayesLSH [91], que utiliza um conjunto de funções *hash* probabilísticas para transformar os objetos de dados em assinaturas *hash*. No entanto, o trabalho de Anastasiu explora principalmente as características de vetores esparsos para derivar os filtros. Neste contexto, o algoritmo L2AP é utilizado neste trabalho para avaliar o seu desempenho em espaços vetoriais densos que representam dados textuais, expandindo assim seu escopo de aplicação.

No contexto das operações de similaridade em vetores que representam imagens e outros dados complexos, destacam-se trabalhos relacionados à recuperação da informação com base em conteúdo [11] e busca de similaridade em bancos de dados multimídia [38]. Essas pesquisas enfocam o uso de métodos de acesso métrico e medidas de similaridade avançadas para lidar com grandes volumes de dados complexos [76], abrangendo áreas como reconhecimento de padrões [84] e processamento de imagens [10].

Para melhorar a escalabilidade e o desempenho na junção por similaridade em vetores, visando resultados exatos e em cenários com grandes volumes de dados, têm sido exploradas técnicas de paralelismo e o uso de GPUs [45]. No entanto, a quantidade e a dimensionalidade dos dados continuam sendo fatores críticos que impactam diretamente o tempo de execução necessário para obter resultados exatos, o que demanda técnicas de otimização baseadas em soluções aproximadas, além do aproveitamento de hardwares modernos.

Assim, a aplicação de técnicas como redução de dimensionalidade ou quantização tem sido investigadas para obter resultados aproximados [79]. Porém, embora a aplicação de técnicas de quantização e redução de dimensionalidade ofereçam ganhos significativos em desempenho, resulta em queda na taxa de recuperação, levando à perda de resultados relevantes. Portanto, é fundamental adotar estratégias adicionais de otimização da junção por similaridade que proporcionem um equilibro adequado entre o tempo de execução e a taxa de recuperação.

Nesse contexto, observa-se que há uma lacuna em pesquisas dedicadas à otimização específica da junção por similaridade em espaços vetoriais densos. No entanto, com o aumento do uso de representações de dados por vetores densos, especialmente na comunidade de ciência de dados e aprendizado de máquina, têm impulsionado a investigação de técnicas que visam otimizar a busca por similaridade nesses espaços. Um aspecto relevante dessas pesquisas é a criação de estruturas de indexação e algoritmos para realizar a busca aproximada dos k-vizinhos mais próximos [6]. Essas estruturas e algoritmos são incorporados em bibliotecas específicas de busca por similaridade [51] e em sistemas gerenciadores de bancos de dados vetoriais modernos [102]. Dentre as bibliotecas e algoritmos desenvolvidos para esse propósito, destacam-se:

FAISS [51] - uma biblioteca que oferece algoritmos eficientes para busca por similaridade e agrupamento de vetores densos. Permite a comparação de vetores usando distância euclidiana ou produtos internos e pode identificar vetores similares a partir de um vetor de consulta. O FAISS é conhecido por sua capacidade de lidar com conjuntos de vetores em grandes tamanhos, inclusive quando não cabem na memória RAM. Ele também possui implementações otimizadas para GPU para oferecer maior desempenho;

Annoy (*Approximate Nearest Neighbors Oh Yeah*) [13] - uma biblioteca em C++ com *bindings* para Python que permite buscar pontos no espaço próximos a um determinado ponto de consulta. Se destaca pela capacidade de compartilhamento dos índices, possibilitando o uso entre processos e distribuição em ambientes de produção. Oferece suporte a diversas métricas e implementa uma floresta de projeção aleatória [114]. Utilizado pelo serviço *Spotify* para recomendações musicais em um espaço de alta dimensionalidade;

HNSW (*Hierarchical Navigable Small World*) [65] - uma estrutura que constrói um grafo hierárquico com várias camadas, onde os elementos são conectados com base na proximidade. Utiliza uma abordagem de busca pelo vizinho mais próximo, percorrendo o grafo em camadas e aproximando-se gradualmente de um mínimo local. O algoritmo combina o fenômeno do 'mundo pequeno' [56] com links de longo alcance e utiliza pontos de partida estratégicos para melhorar a eficiência da busca; e

ScaNN (*Scalable Nearest Neighbors*) [46] - uma biblioteca para busca de similaridade vetorial em grande escala que utiliza técnicas de compressão de vetores para permitir cálculos de distância aproximada de forma rápida. Em seu artigo [46], os pesquisadores propõem uma abordagem de quantização anisotrópica de vetores, que penaliza mais o erro de quantização paralelo ao vetor original. Isso resulta em maior precisão para estimar produtos internos elevados, que são cruciais em busca de vizinhos mais próximos.

No entanto, é importante ressaltar que os métodos mencionados anteriormente foram projetados para buscas kNN e não foram otimizados especificamente para a operação de junção por similaridade. Portanto, esta pesquisa procura investigar como aproveitar estas estruturas existentes e aplicá-las de maneira eficiente na junção por similaridade.

Dentre as soluções mencionadas, destaca-se o HNSW por apresentar melhorias significativas no tempo de busca e uma mínima redução na taxa de recuperação [35, 6]. Ele tem sido amplamente utilizado em várias aplicações, desde buscas em tempo real até a descoberta de dados em *Data Lakes* [37]. Levando em consideração essas características, este trabalho de pesquisa se aprofundou no estudo dessa estrutura, explorando sua aplicação na operação de junção por similaridade.

CAPÍTULO 4

L2AP em Espaços Vetoriais Densos

Este capítulo apresenta uma avaliação do algoritmo L2AP aplicado em espaços vetoriais densos, gerados por um método de *embedding*, e esparsos, gerados pelo método TF-IDF. O objetivo foi responder a questão (1) de pesquisa: "como as características dos espaços vetoriais densos afetam o desempenho de algoritmos existentes de junção por similaridade?". Neste sentido, foram realizados experimentos (Seção 4.1), analisado os resultados (Seção 4.2) e comparado com método por força bruta (Seção 4.3).

4.1 Experimentos

Para realização dos experimentos utilizou-se o conjunto de dados *DBLP*[1] descrito na Tabela 4.1. As representações vetoriais dos dados, normalizadas para uma unidade de comprimento, foram realizadas de duas formas: (1) medida TF-IDF com *tokenização q-gram(3)* e (2) representação com uso do *framework Sentence-Transformers* e modelo pré-treinado multi-qa-MiniLM-L6-cos-v1[2] que gera vetores densos com 384 dimensões. Este modelo foi escolhido por apresentar maior rapidez na representação e melhores resultados na busca semântica de acordo com o site SBERT[3].

Uma característica comum nos vetores gerados pelos métodos de aprendizado de máquina são dimensões no intervalo [-1,1]. Portanto foi necessário o ajuste dos valores para o intervalo [0,1] como esperado pelo L2AP. Esta operação foi realizada através da fórmula $novo_valor = \frac{(valor_anterior + 1)}{2}$. O tempo necessário para gerar a representação vetorial pela medida TF-IDF e pelo *Sentence-Transformers* foi de 16 segundos e 59 segundos respectivamente. As execuções do algoritmo foram realizadas variando o *threshold* de 0.7 a 0.9 com incrementos de 0.1 e os tempos obtidos são a média de cinco execuções.

[1] http://dblp.uni-trier.de
[2] https://huggingface.co/sentence-transformers/multi-qa-MiniLM-L6-cos-v1
[3] https://www.sbert.net/docs/pretrained_models.html

Datasets	Atributos	Tam. Min	Tam. Max	Tam. Méd	Registros	Duplicatas	Total
DBLP	Title, Author	8	958	125	20000	5	100000

Tabela 4.1: Descrição do dataset DBLP.

4.2 Resultados

A Figura 4.1 apresenta: (a) os tempos de execução do algoritmo L2AP e (b) a quantidade de pares candidatos, filtrados e verificados nos vetores com representação por medida TF-IDF e com representação densa através do *Sentence-Transformers* (indicado nos gráficos pela sigla ST).

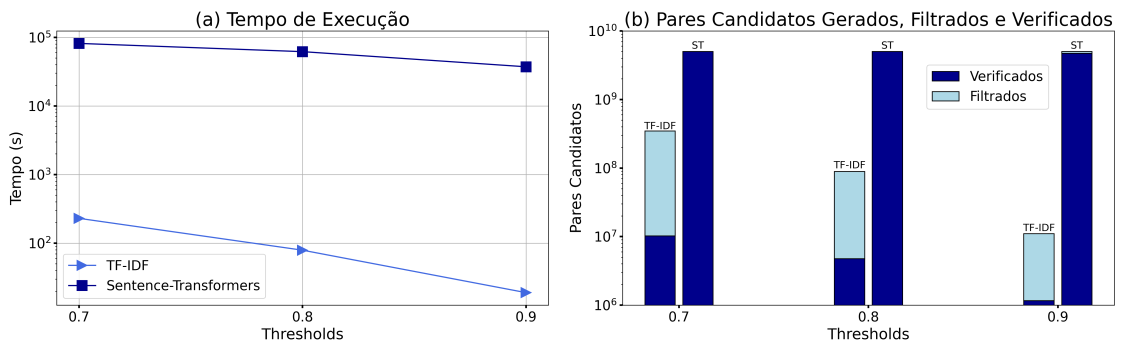


Figura 4.1: Execução do L2AP em vetores densos x esparsos.

Com *threshold* em 0.9, observa-se uma diferença significativa no tempo de execução entre a representação TF-IDF e a representação densa usando o algoritmo L2AP. Enquanto na representação TF-IDF o algoritmo L2AP fornece os resultados em 19 segundos, na representação densa o tempo de execução é de 37.228 segundos, o que corresponde a uma diferença de 1.950 vezes maior. Essa diferença de tempo é consistente para outros valores de *threshold* analisados. O algoritmo L2AP demonstra eficiência ao lidar com vetores gerados usando a medida TF-IDF. Embora esses vetores possuam um grande número de dimensões, os mesmos são esparsos, com a minoria de dimensões tendo valores diferentes de zero. As técnicas de filtragem e poda de candidatos empregadas pelo algoritmo se beneficiam dessas características para eliminar um grande número de candidatos e interromper a computação de falsos positivos, resultando em tempos de execução muito baixos, da ordem de poucos segundos.

No entanto, em espaços vetoriais densos e de tamanho fixo, o algoritmo L2AP não demonstrou eficiência. Nesses espaços, cada *token* ou representação do dado é mapeado para uma dimensão, e todos os vetores possuem os mesmos *tokens*. Como resultado, a fase de geração de candidatos do L2AP não consegue filtrar efetivamente os candidatos, o que sobrecarrega a fase de verificação, exigindo o cálculo da similaridade para um grande número de falsos positivos.

O principal filtro do L2AP, baseado na norma L2, não conseguiu filtrar uma quantidade significativa de candidatos. Além disso, os demais filtros do algoritmo, que se baseiam no tamanho do vetor, seu prefixo e sufixo, também não filtraram pares de candidatos. Consequentemente, o algoritmo acaba calculando a similaridade para praticamente todos os pares de vetores no espaço de comparação, o que aumenta substancialmente o tempo de execução.

Além disso, a estrutura de indexação invertida utilizada pelo L2AP, ao invés de reduzir o tempo de processamento, torna-se mais lenta nos vetores densos. Isso ocorre devido à necessidade de indexar todas as dimensões em todos os vetores, resultando em custos computacionais adicionais. Combinado com os filtros ineficazes, o algoritmo acaba sendo comparável ou até mais lento do que um método de força bruta, que realiza todas as comparações no espaço vetorial denso.

4.3 Algoritmo Básico com Filtro da Norma L2

Com base na avaliação do algoritmo L2AP, que demonstrou ser ineficiente em espaços vetoriais densos, foi desenvolvido um algoritmo básico de junção por similaridade denominado "NestedL2Join".

Algoritmo 4.1: NestedL2Join

Entrada: Conjunto de vetores D e *threshold* τ

Saída: Conjunto de resultados $\mathcal{A}_{sj}$

1. $\mathcal{A}_{sj} \leftarrow \varnothing$
2. $C \leftarrow \varnothing$ `// Conjunto de vetores candidatos`
3. **para cada** $x \in D$ **faça**
4.   **para cada** $y \in C, |C| > 0$ **faça**
5. $s \leftarrow 0$
6.   **para cada** $p = d \ldots 1, y_p \in y$ **faça** `// Em ordem inversa`
7. $s \leftarrow s + x_p \cdot y_p$
8.   **se** $s + ||x_p|| \cdot ||y_p|| < \tau$ **então** `// Poda com base na Norma L2`
9.   **break** `// Parar de acumular e passar para o próximo candidato`
10.   **se** $s \geq \tau$ **então**
11. $\mathcal{A}_{sj} \leftarrow \mathcal{A}_{sj} \cup \{\langle x, y \rangle\}$ `// Adicionar par similar`
12. $C \leftarrow C \cup \{x\}$ `// Adiciona vetor corrente nos candidatos`
13. **retorna** $\mathcal{A}_{sj}$

O NestedL2Join, descrito no Algoritmo 4.1, realiza a comparação de todos os pares de vetores, e obtém resultados exatos. O algoritmo utiliza um laço aninhado para computar a similaridade entre os pares de vetores (Linhas 3-4). No momento da computação da similaridade entre o par de vetores, é aplicado o filtro pela norma L2 do L2AP (Linhas 8-9). Esse filtro demonstrou, ainda que de forma mínima, resultados de poda na avaliação do L2AP nos vetores densos conforme apresentado na Seção anterior. É importante ressaltar que, para aplicar esse filtro, é necessário calcular e armazenar previamente a magnitude em cada posição do vetor de consulta e do vetor candidato. Se a similaridade entre o par de vetores for acima do *threshold*, o par é incluído no resultado (Linhas 10-11). Além disso, uma lista incremental de candidatos é empregada para evitar repetições de pares de vetores (Linha 12).

Thresholds	**L2AP (TF-IDF)**	**L2AP (ST)**	**NestedL2Join (ST)**
0.7	230s	81560s	64814s
0.8	79s	61858s	18031s
0.9	19s	37228s	7920s

Tabela 4.2: Tempo de execução do algoritmo L2AP e NestedL2Join.

Na Tabela 4.2, é possível consultar o tempo de execução dos algoritmos NestedL2Join e L2AP em vetores densos, bem como do L2AP em vetores com TF-IDF. Em vetores densos, o algoritmo NestedL2Join apresentou *speedups* de 4,7x, 3,4x e 1,2x nos *threshold* de 0.9, 0.8 e 0.7, respectivamente. Embora o método NestedL2Join tenha reduzido o tempo de execução em comparação com o algoritmo L2AP em vetores densos, ele ainda é ineficiente e não atinge um desempenho comparável ao L2AP em espaços vetoriais com TF-IDF. Enquanto o algoritmo L2AP executou em apenas 19 segundos no TF-IDF, o método NestedL2Join levou 7920 segundos nos vetores densos, ambos com um *threshold* de 0.9.

A análise realizada confirmou que a estrutura de indexação e os filtros utilizados pelo algoritmo L2AP não reduzem o custo computacional da operação de junção por similaridade em vetores densos. Pelo contrário, esses elementos adicionam complexidade computacional e cálculos adicionais, resultando em um maior tempo de execução.

Os resultados obtidos evidenciam as dificuldades do algoritmo L2AP na filtragem efetiva de candidatos em espaços vetoriais densos. A baixa eficiência dos filtros desse algoritmo indica a necessidade de buscar novas abordagens ou técnicas de filtragem para melhorar o desempenho nesses espaços.

Essas considerações destacam a importância de explorar alternativas e desenvolver métodos mais adequados para a operação de junção por similaridade em vetores densos.

CAPÍTULO 5

Junções por Similaridade com Algoritmos ANN

Neste capítulo, é abordado a utilização de algoritmos de busca do vizinho mais próximo aproximado para a junção por similaridade em vetores densos. Inicialmente, fornecemos uma visão geral de quatro algoritmos avançados de busca eficiente de vizinhos mais próximos em espaços multidimensionais. Em seguida, propomos uma abordagem baseada no Lema 2.4 para aplicar esses algoritmos na operação de junção por similaridade. Para avaliar o desempenho dessas abordagens, foi realizado experimentos comparativos entre diferentes estruturas de indexação e discutido os resultados obtidos.

5.1 Algoritmos ANN

Conforme abordado no Capítulo 3, nos estudos relacionados a vetores densos, há um foco significativo no desenvolvimento de estruturas de indexação e algoritmos para buscas aproximadas kNN, também conhecidos como algoritmos ANN. Dentre as principais estruturas e algoritmos desenvolvidos, destacam-se o FAISS [51], que utiliza, dentre outras, técnicas de índice invertido, quantização e métodos baseados em *hash*; o Annoy [13], que emprega uma estrutura baseada em floresta de projeção aleatória; o HNSW [65], que utiliza uma estrutura de grafo hierárquico e técnicas de busca em largura para encontrar os vizinhos mais próximos; e o ScaNN [46], que emprega uma estratégia de compactação através de uma nova abordagem de quantização dos vetores para aumentar a precisão. Demais características sobre o FAISS, Annoy e ScaNN são apresentadas na Seção 3.2 e o HNSW com mais profundidade na Seção 2.5.

A busca exata em grandes volumes de dados e com vetores densos de alta dimensionalidade é computacionalmente custosa ou até mesmo inviável. No entanto, a busca aproximada oferece uma solução viável e eficiente, alcançando altas taxas de recuperação, frequentemente superiores a 99%, em termos de distância, por meio de algoritmos como os mencionados anteriormente. Além desses, o trabalho de referência ANN-Benchmarks [6] aborda uma ampla gama de algoritmos e plataformas que podem ser comparados. Esses algoritmos são capazes de lidar com grandes volumes de dados

e têm sido amplamente aplicados em áreas como processamento de linguagem natural, visão computacional e ciência de dados [102].

5.2 Experimentos na Junção por Similaridade

Algoritmos ANN desempenham um papel crucial em aplicações de tempo real e são especialmente projetados para buscas *k*NN. No contexto da junção por similaridade, é possível adaptar a aplicação desses algoritmos para obter uma junção por similaridade aproximada, conforme mencionado no Lema 2.4. Uma abordagem simplificada consiste em realizar chamadas repetidas de buscas *k*NN, aumentando gradualmente o valor de k a cada iteração, até que o objeto menos semelhante retornado esteja abaixo do *threshold* estabelecido. Embora algumas estruturas de indexação fornecidas pelo FAISS, como o índice de arquivo invertido (IVF) e o índice plano, permitam buscas baseadas em *threshold*, as estruturas HNSW, Annoy e ScaNN não possuem esse recurso nativamente.

Para avaliar e comparar a aplicação do FAISS, Annoy, HNSW e ScaNN na junção por similaridade, foram realizados experimentos utilizando a base de dados DBLP indicada na Tabela 4.1. Essa base de dados foi representada por vetores densos normalizados com 384 dimensões gerados pelo modelo pré-treinado multi-qa-MiniLM-L6-cos-v1 do *Sentence-Transformers*.

Para cada estrutura de indexação, foi implementado um algoritmo de duas etapas: (1) indexa todos os vetores e (2) percorre cada vetor da base, consultando o índice em busca de vetores similares acima do *threshold*, utilizando a função cosseno. Utilizou-se a abordagem de busca *k*NN para *threshold*, conforme mencionado anteriormente ou, no caso do FAISS com o índice IVF, realizou-se uma busca do tipo *range-search*. Para esses experimentos, foram utilizadas as bibliotecas oficiais do FAISS[1], Annoy[2], HNSW[3] e ScaNN[4], com as *bindings* em Python. Utilizou-se a versão brute_force disponibilizada pela biblioteca nmslib[5] para obter resultados exatos. No FAISS, foram empregados os índices aproximados, sendo do tipo LSH, que aplica funções de *hash* sensíveis à localidade para maximizar as colisões e agrupar os vetores, e o tipo IVFFlat, que reduz o escopo de busca por meio de arquivo invertido e clusterização.

Os resultados experimentais são apresentados na Tabela 5.1, onde cada estrutura de indexação foi avaliada em termos de taxa de recuperação (*recall*), em relação ao método exato, e tempo de execução para diferentes *thresholds*.

[1] `https://github.com/facebookresearch/faiss`
[2] `https://github.com/spotify/annoy`
[3] `https://github.com/nmslib/nmslib`
[4] `https://github.com/google-research/google-research/tree/master/scann`
[5] `https://github.com/nmslib/nmslib`

Threshold	nmslib.brute_force		faiss.IndexLSH		faiss.IVFFlat	
	Recall	Tempo (s)	Recall	Tempo (s)	Recall	Tempo (s)
0.6	100%	1863	95.18%	1190	92.99%	153
0.7	100%	1797	99.98%	1127	95.29%	153
0.8	100%	1791	100.00%	1127	97.81%	152
0.9	100%	1764	100.00%	1126	99.49%	152
Threshold	**Annoy**		**nmslib.HNSW**		**ScaNN**	
	Recall	Tempo (s)	Recall	Tempo (s)	Recall	Tempo (s)
0.6	94.57%	327	97.43%	47	79.49%	88
0.7	97.33%	235	98.23%	39	97.29%	80
0.8	99.06%	186	99.12%	34	99.03%	71
0.9	99.26%	143	99.42%	31	99.81%	63

Tabela 5.1: Resultados dos experimentos de junção por similaridade com algoritmos ANN.

Os resultados revelam que o HNSW alcançou um equilíbrio superior entre tempo de execução e taxa de recuperação em comparação com as outras estruturas. Esses resultados corroboram os achados do projeto ANN-Benchmarks [6], que demonstra o desempenho destacado do HNSW em buscas aproximadas kNN em relação aos métodos concorrentes.

Esses resultados enfatizam o HNSW como uma escolha viável para a junção por similaridade aproximada em vetores densos. Ao atingir altas taxas de recuperação e tempos de execução competitivos, evidencia-se um potencial promissor na exploração de novas abordagens de uso do HNSW, visando aprimorar ainda mais a eficiência da junção por similaridade em vetores densos. Essas novas abordagens de uso do HNSW são detalhadas no próximo capítulo.

CAPÍTULO 6

Junções por Similaridade com Grafos HNSW

Neste capítulo, são apresentados os algoritmos de junção por similaridade baseados no HNSW. Como mencionado anteriormente, o HNSW retorna resultados aproximados, o que significa que as soluções propostas também são aproximadas. Além disso, o HNSW é uma estrutura de indexação para buscas kNN e não oferece suporte a consultas baseadas em *threshold*. Portanto, foram desenvolvidas estratégias para utilizar essa estrutura na junção por similaridade. Primeiro, é proposta uma abordagem para realizar junções por similaridade no topo do HNSW, ou seja, sem modificações de sua estrutura e com base no tratamento de forma criteriosa do parâmetro k (Seção 6.1). Em seguida, são apresentados os algoritmos que adaptam a estrutura interna do HNSW para realizar buscas baseadas em *threshold* (Seção 6.2).

Os algoritmos propostos receberam a denominação de *HNSW-Based Similarity Join*, sendo adotada a sigla "HSJ" para identificá-los. Três variações desses algoritmos são apresentadas: o HSJ-Ext, que não requer modificações na estrutura do HNSW, e os algoritmos HSJ-Ths e HSJ-Ths^{Inc}, que utilizam um novo método de busca baseado no *threshold*. Além disso, dentro do contexto dos algoritmos HSJ-Ths e HSJ-Ths^{Inc}, são propostos dois novos algoritmos que realizam a busca por similaridade dentro do índice HNSW, denominados RANGE-SEARCH e RANGE-SEARCH-LAYER.

6.1 Junções no Topo do HNSW

Uma abordagem básica para realizar a junção por similaridade usando o HNSW é primeiro indexar o conjunto de vetores $\mathcal{V}$ e, em seguida, realizar uma busca de k-vizinhos mais próximos (kNN) para cada vetor, a fim de encontrar suas contrapartes semelhantes, formando assim o conjunto de respostas $\mathcal{A}_{ss}$. Para isso, é necessário determinar um valor adequado para o parâmetro k que estabeleça uma relação de equivalência ou, pelo menos, de contenção com uma busca de similaridade baseada em *threshold* (conforme demonstrado no Lema 2.4). No entanto, o valor de k adequado não é conhecido antecipadamente.

Uma estratégia intuitiva é realizar múltiplas buscas kNN com valores crescentes de k até que todas os vetores similares sejam recuperados. As escolhas do valor inicial

de k e do incremento ao longo das buscas desempenham um papel crucial no desempenho geral do algoritmo. Se os valores escolhidos forem muito pequenos, será necessário realizar um maior número de buscas kNN para recuperar $\mathcal{A}_{ss}$. Por outro lado, se forem escolhidos valores muito grandes, muitos vetores dissimilares serão incluídos nas respostas, resultando em uma sobrecarga desnecessária.

Com base nas observações mencionadas anteriormente, foram desenvolvidas heurísticas para a seleção do valor inicial de k e o seu incremento em cada iteração. Para determinar o valor inicial de k, utilizou-se o parâmetro `ef` do HNSW, acrescido de um fator determinado pelo *threshold* conforme Fórmula 6-1. A proposta é utilizar um valor inicial de k maior para *thresholds* menores. Essa abordagem visa capturar um maior número de vizinhos próximos e aumentar a probabilidade de recuperar todas as contrapartes similares.

$$k_{inicial} = ef + (1 - \tau) \cdot ef \tag{6-1}$$

Para o incremento de k, considerou-se o elemento menos similar em $\mathcal{A}_{knn}$, denominado y_k. O incremento de k é calculado com base em $sim(x, y_k)$, que representa a similaridade entre y_k com o vetor de consulta x, seguindo uma diminuição monotônica a cada iteração conforme Fórmula 6-2. Essa heurística garante que o incremento de k diminua à medida que a similaridade de y_k se aproxima do *threshold*. Dessa forma, prioriza-se a inclusão de vetores mais similares e evita-se a adição excessiva de vetores dissimilares.

$$k_{novo} = k + \frac{(1 - \tau) \cdot k}{(1 - sim(x, y_k))} \tag{6-2}$$

O algoritmo 6.1 descreve o HSJ-Ext, a variante externa do HSJ que executa sobre o HNSW sem modificar sua estrutura interna. Primeiro, todos os vetores são indexados no grafo HNSW (Linha 4). Então, $\mathcal{V}$ é escaneado novamente (Linha 6), e para cada vetor x, o $\mathcal{A}_{knn} \supseteq \mathcal{A}_{ss}$ correspondente é computado pela execução de buscas kNN com x e aumentando k progressivamente (linhas 7–13); o valor inicial de k e seu incremento são calculados de acordo com as heurísticas descritas anteriormente (Linha 5 e 11, respectivamente). Os vetores em $\mathcal{A}_{knn}$ com similaridade maior que τ são unidos com o vetor x e inseridos em $\mathcal{A}_{sj}$ (linhas 14-16).

Ressalta-se o potencial do HNSW de utilizar paralelismo *multicore* para melhorar a eficiência e o desempenho. Ao empregar técnicas de paralelismo, é possível distribuir a carga computacional em vários núcleos de processamento, acelerando o processo de construção do grafo e permitindo que as operações de busca sejam executadas de forma mais rápida e eficiente.

O HSJ-Ext se beneficia totalmente do procedimento de busca kNN altamente eficiente do HNSW. Porém, mesmo com a heurística proposta para determinação do valor

de k, pode ser necessário um grande número de buscas para *thresholds* baixos. Cada busca requer uma travessia completa na hierarquia do grafo de proximidade da camada superior para a inferior. Para evitar essas buscas repetidas, propomos a integração do processamento de junção por similaridade no HNSW, conforme descrito a seguir.

Algoritmo 6.1: HSJ-Ext

Entrada: Conjunto de vetores $\mathcal{V}$, threshold τ, parâmetros HNSW M, efC e ef

Saída : Conjunto de resultados $\mathcal{A}_{sj}$

```
1  A_sj ← ∅
2  A_knn ← ∅ // em ordem decrescente de similaridade
3  hnsw ← buildHNSW(V.dimension, M, efC, ef)
4  hnsw.insertAll(V) // indexa todos os vetores com o algoritmo
     INSERT do HNSW
5  k_inicial = ef + ((1 − τ) · ef)
6  para cada x ∈ V faça
7      k ← k_inicial
8      A_knn ← K-NN-SEARCH(hnsw, x, k)
9      y_k ← vetor com a menor similaridade em A_knn
10     enquanto sim(x, y_k) ≥ τ faça
11         k ← k + ((1 − τ) · k)/(1 − sim(x, y_k))
12         A_knn ← K-NN-SEARCH(hnsw, x, k)
13         y_k ← vetor com a menor similaridade em A_knn
14     para cada y ∈ A_knn faça
15         se sim(x, y) ≥ τ então
16             A_sj.add(x, y)
17 retorna A_sj
```

6.2 Junções por Similaridade Integradas ao HNSW

Através da adoção de uma abordagem mais profunda no projeto e integração para o processamento de junção por similaridade no HNSW, tornou-se possível realizar a travessia do grafo enquanto elementos com similaridade acima do *threshold* são recuperados.

O algoritmo HSJ-Ths, apresentado no Algoritmo 6.2, descreve essa abordagem. Nele, realiza-se buscas de similaridade diretamente em uma estrutura HNSW. Assim como no algoritmo anterior, todos os vetores são indexados (Linha 5) e, em seguida, percorre-se $\mathcal{V}$ novamente, emitindo uma busca de similaridade para cada vetor (Linha 6).

Em contraste com HSJ-Ext, $\mathcal{A}_{ss}$ agora é calculado dentro do HNSW pelo novo método RANGE-SEARCH (Linha 7). Essa modificação permite que o HNSW percorra o grafo uma única vez, encontrando elementos similares acima do limite especificado. Com a utilização do método RANGE-SEARCH, podemos recuperar diretamente o conjunto $\mathcal{A}_{ss}$ de elementos similares, eliminando a necessidade de pós-processamento adicional.

Algoritmo 6.2: HSJ-Ths

Entrada: Conjunto de vetores $\mathcal{V}$, threshold τ, parâmetros HNSW M, efC e ef

Saída : Conjunto de resultados $\mathcal{A}_{sj}$

1 $\mathcal{A}_{sj} \leftarrow \emptyset$

2 $\mathcal{A}_{ss} \leftarrow \emptyset$

3 $ef = ef + ((1 - \tau) \cdot ef)$

4 $hnsw \leftarrow buildHNSW(\mathcal{V}.dimension, M, efC, ef)$

5 $hnsw.addAll(\mathcal{V})$ `// indexa todos os vetores com o algoritmo INSERT do HNSW`

6 **para cada** $x \in \mathcal{V}$ **faça**

7 $\quad \mathcal{A}_{ss} \leftarrow$ RANGE-SEARCH$(hnsw, x, ef, \tau)$

8 $\quad$ **para cada** $y \in \mathcal{A}_{ss}$ **faça**

9 $\quad\quad \mathcal{A}_{sj}.add(x, y)$

10 **retorna** $\mathcal{A}_{sj}$

Algoritmo 6.3: RANGE-SEARCH

Entrada: índice $hnsw$, vetor de busca x, tamanho da lista dinâmica de candidatos ef e *threshold* τ

Saída : Conjunto de resultados $\mathcal{A}_{ss}$

1 $\mathcal{W} \leftarrow \emptyset$ `// Lista dinâmica de vizinhos mais próximos encontrados`

2 $\mathcal{A}_{ss} \leftarrow \emptyset$

3 $ep \leftarrow$ obter nó de entrada no índice $hnsw$

4 $L \leftarrow$Nível do ep

5 **para** $lc \leftarrow L \ldots 1$ **faça**

6 $\quad \mathcal{W} \leftarrow$ SEARCH-LAYER$(x, ep, ef = 1, lc)$ `// algoritmo SEARCH-LAYER do HNSW`

7 $\quad ep \leftarrow$ obter o elemento mais próximo em $\mathcal{W}$ de x

8 $\mathcal{A}_{ss} \leftarrow$ RANGE-SEARCH-LAYER$(x, ep, ef, lc = 0, \tau)$

9 **retorna** $\mathcal{A}_{ss}$

O método RANGE-SEARCH, detalhado no Algoritmo 6.3, é uma extensão

do método K-NN-SEARCH do artigo original do HNSW. Ele segue uma abordagem semelhante na busca de objetos nas camadas superiores (Linhas 5-7), avançando até atingir o ponto de entrada na camada inferior. Quando alcança a camada inferior (Linha 8), aciona o novo método RANGE-SEARCH-LAYER (Algoritmo 6.4), que é baseado no método SEARCH-LAYER original do HNSW, para buscar diretamente os vetores com similaridade acima do *threshold*.

Algoritmo 6.4: RANGE-SEARCH-LAYER

Entrada: vetor de busca x, nó de entrada ep, tamanho máximo da lista dinâmica de vizinhos encontrados ef, número da camada lc, *threshold* τ

Saída : Conjunto de respostas $\mathcal{A}_{ss}$

1 $v \leftarrow ep$ `// conjunto de elementos visitados`
2 $C \leftarrow ep$ `// conjunto de candidatos`
3 $\mathcal{W} \leftarrow ep$ `// lista dinâmica de vizinhos mais próximos encontrados`
4 $\mathcal{A}_{ss} \leftarrow \emptyset$
5 **enquanto** $|C| > 0$ **faça**
6 $\quad c \leftarrow$ extrair o elemento mais similar em C para x
7 $\quad f \leftarrow$ obter o elemento menos similar em W para x
8 $\quad$**se** $sim(c,x) < sim(f,x)$ ***e*** $sim(f,x) < \tau$ **então**
9 $\quad\quad$break `// todos os elementos em` $\mathcal{W}$ `foram avaliados e o threshold atingido`
10 $\quad$**para cada** $e \in vizinhos(c)$ *na camada* lc **faça**
11 $\quad\quad$**se** $e \notin v$ **então**
12 $\quad\quad\quad v \leftarrow v \cup e$
13 $\quad\quad\quad f \leftarrow$ obter o elemento menos similar em $\mathcal{W}$ para x
14 $\quad\quad\quad$**se** $sim(e,x) \geq sim(f,x)$ ***ou*** $|\mathcal{W}| < ef$ ***ou*** $sim(e,x) \geq \tau$ **então**
15 $\quad\quad\quad\quad C \leftarrow C \cup e$
16 $\quad\quad\quad\quad \mathcal{W} \leftarrow \mathcal{W} \cup e$
17 $\quad\quad\quad\quad$**se** $|\mathcal{W}| > ef$ ***e*** $sim(f,x) < \tau$ **então**
18 $\quad\quad\quad\quad\quad$remova o elemento menos similar em $\mathcal{W}$ para x
19 **para cada** $y \in \mathcal{W}$ **faça**
20 $\quad$**se** $sim(x,y) \geq \tau$ **então**
21 $\quad\quad \mathcal{A}_{ss}.add(y)$
22 **retorna** $\mathcal{A}_{ss}$

O método RANGE-SEARCH-LAYER (Algoritmo 6.4) incorpora modificações

ao método de busca SEARCH-LAYER do HNSW, para utilizar o *threshold* como critério de parada da travessia no grafo. Além do parâmetro adicional do *threshold* τ, os demais parâmetros de entrada permanecem os mesmos do algoritmo original. Esses parâmetros incluem o vetor de busca x, o ponto de entrada no grafo ep, o valor ef que limita o tamanho máximo da lista dinâmica de vizinhos encontrados $\mathcal{W}$, e o número da camada que será realizada a busca l_c. As demais modificações no algoritmo são as seguintes:

- Utilização do mínimo local como condição de parada apenas se a similaridade do elemento for menor que o *threshold* (Linha 8). Isso evita que a busca seja interrompida prematuramente caso o mínimo local seja encontrado, mas sua similaridade atenda ao limite desejado;
- Além das condições existentes, é verificado se a similaridade do elemento atual e a consulta é maior que o *threshold* para incluí-lo como candidato (Linha 14). Essa verificação assegura que, mesmo que o tamanho máximo da lista dinâmica $\mathcal{W}$ seja atingido ou a similaridade do elemento atual seja o mínimo local, o elemento será incluído no resultado se sua similaridade com a consulta for superior ao *threshold*;
- A remoção do vizinho na lista dinâmica só ocorre se a sua similaridade for menor que o *threshold* (Linha 17). Esta condição garante que os candidatos com similaridade acima do *threshold* permaneçam na lista, mesmo se o tamanho da lista atingir seu limite;
- Durante a travessia no grafo, a fim de evitar que o mínimo local seja alcançado prematuramente, são adicionados à lista dinâmica de vizinhos mais próximos pelo menos a quantidade estabelecida pelo parâmetro `ef`. Como resultado, é possível que existam elementos que não atingiram o *threhsold*. Portanto, é necessário realizar uma verificação final e incluir no resultado apenas os elementos que possuem similaridade acima do *threshold* (Linhas 19–21).

Apesar da busca por similaridade ser orientada pelo *threshold*, é importante ressaltar que o parâmetro `ef` não pode ser negligenciado, pois ele estabelece um número mínimo de candidatos a serem considerados. Isso evita a possibilidade de se alcançar um mínimo local falso e assegura uma busca mais abrangente. Portanto, o parâmetro `ef` desempenha um papel fundamental na qualidade dos resultados obtidos.

Por fim, propomos o HSJ-Ths^{Inc} (Algoritmo 6.5), um versão incremental do HSJ-Ths. Ao contrário da abordagem anterior, em que todos os vetores são indexados previamente antes da busca, o HSJ-Ths^{Inc} realiza uma busca incremental no momento em que cada vetor é adicionado ao índice (Linha 7 e 10, respectivamente). Essa abordagem traz vantagens significativas, como a redução do escopo de busca, ficando limitado aos dados já indexados, evitando o processamento de pares de vetores semelhantes duplicados. Essa característica torna essa abordagem uma solução promissora para aplicações que exigem junções aproximadas por similaridade em ambientes dinâmicos, nos quais grandes volumes de dados estão sendo constantemente atualizados.

Algoritmo 6.5: HSJ-Ths^{Inc}

Entrada: Conjunto de vetores $\mathcal{V}$, threshold τ, parâmetros HNSW M, efC e ef

Saída : Conjunto de resultados $\mathcal{A}_{sj}$

```
1  A_sj ← ∅
2  A_ss ← ∅
3  ef = ef + ((1 − τ) · ef)
4  hnsw ← buildHNSW(V.dimension, M, efC, ef)
5  para x ∈ V faça
6      se hnsw ≠ ∅ então
7          A_ss ← RANGE-SEARCH(hnsw, x, ef, τ)
8          para cada y ∈ A_ss faça
9              A_sj.add(x, y)
10     INSERT(hnsw, x) // Algoritmo INSERT do HNSW
11 retorna A_sj
```

Como mencionado anteriormente, o HNSW possui potencial para utilizar de paralelismo *multicore*. No entanto, ao empregar essa técnica na busca do grafo usando o método incremental, é importante considerar as implicações na taxa de recuperação. Por um lado, o paralelismo *multicore* permite distribuir a carga computacional entre vários núcleos de processamento, acelerando o processo de busca e potencialmente aumentando a eficiência do algoritmo. Isso pode resultar em tempos de resposta mais rápidos e maior capacidade de processamento de volumes maiores de dados. Por outro lado, a concorrência entre os núcleos de processamento pode gerar problemas de sincronização e acesso simultâneo a recursos compartilhados, como a lista de candidatos. Isso pode afetar a consistência dos resultados e comprometer a precisão da busca por similaridade.

Portanto, ao aplicar o paralelismo *multicore* na busca do grafo usando o método incremental, é necessário realizar uma análise cuidadosa dos benefícios e limitações dessa abordagem, levando em consideração as características dos dados, os recursos de hardware disponíveis e os requisitos específicos da aplicação.

No próximo capítulo, será abordada a avaliação experimental realizada para verificar o desempenho dos algoritmos propostos, considerando o tempo de execução e a taxa de recuperação em diferentes bases de dados do mundo real. Além disso, será analisada a questão do paralelismo na versão incremental.

CAPÍTULO 7

Experimentos e Resultados

O presente capítulo apresenta um estudo experimental para avaliar o desempenho e a eficácia dos algoritmos propostos para a junção por similaridade aproximada em vetores densos. Os experimentos foram conduzidos com os seguintes objetivos: 1) avaliar o desempenho das técnicas em termos de tempo de execução e taxa de recuperação; 2) comparar as técnicas propostas com método que obtém resultados exatos e com abordagem *baseline*; 2) comparar o desempenho das versões HSJ-Ths e HSJ-Ths^{Inc} com a versão HSJ-Ext; e 4) testar a escalabilidade das propostas.

Inicialmente, são detalhados o método que obtém resultados exatos e a abordagem *baseline* (Seções 7.1 e 7.2). Em seguida, são apresentados os detalhes da configuração do ambiente experimental (Seção 7.3), abrangendo informações sobre a configuração e parâmetros para garantir a consistência e a reprodutividade dos resultados. Posteriormente, são fornecidos detalhes sobre os conjuntos de dados utilizados (Seção 7.4). Por fim, são apresentados os resultados obtidos juntamente com uma análise dos mesmos (Seção 7.5).

7.1 Método Exato

O método para obter resultados exatos segue a mesma estrutura do algoritmo HSJ-Ext, baseado no Lema 2.4, porém em vez de realizar buscas kNN aproximadas em um grafo HNSW, realiza buscas kNN exatas na coleção de todos os vetores.

O método é descrito pelo Algoritmo 7.1. Para implementar as buscas kNN exatas, foi utilizada a biblioteca NMSLIB[1] com o método *brute_force*. Além disso, foram aplicadas as mesmas heurísticas de definição do valor inicial de k e seu incremento a cada iteração conforme descrito na versão HSJ-Ext (Seção 6.1).

A aplicação deste método permite a verificação dos resultados exatos da junção por similaridade para comparação com os resultados obtidos pelos métodos aproximados baseados no HNSW.

[1] https://github.com/nmslib/nmslib

Algoritmo 7.1: Método-Exato

```
   Entrada: Conjunto de vetores 𝒱, threshold τ, k
   Saída   : Conjunto de resultados A_sj
 1 A_sj ← ∅
 2 A_knn ← ∅ // em ordem decrescente de similaridade
 3 index ← buildIndex(𝒱.dimension,)
 4 index.insertAll(𝒱) // Cria índice com todos os vetores
 5 k_inicial = k + ((1 − τ) · k)
 6 para cada x ∈ 𝒱 faça
 7     k ← k_inicial
 8     A_knn ← BRUTE-FORCE-SEARCH(index, x, k)
 9     y_k ← vetor com a menor similaridade em A_knn
10     enquanto sim(x, y_k) ≥ τ faça
11         k ← k + ((1 − τ) · k)/(1 − sim(x, y_k))
12         A_knn ← BRUTE-FORCE-SEARCH(index, x, k)
13         y_k ← vetor com a menor similaridade em A_knn
14     para cada y ∈ A_knn faça
15         se sim(x, y) ≥ τ então
16             A_sj.add(x, y)
17 retorna A_sj
```

7.2 Abordagem *Baseline*

A abordagem *baseline*, que é uma abordagem *naive*, permite estabelecer um ponto de referência para a comparação do desempenho e eficácia das técnicas propostas baseadas no HNSW. Desta forma, o Algoritmo 7.2 apresenta esta abordagem, que segue a mesma estrutura do Algoritmo HSJ-Ext, onde executa sobre o HNSW sem modificar sua estrutura interna baseado no Lema 2.4, porém, sem a aplicação da heurística de definição do k inicial e seu incremento. Nesta abordagem, o valor do k inicial é definido pelo parâmetro `ef` e realizado o incremento de 1 a cada iteração.

Algoritmo 7.2: Baseline

Entrada: Conjunto de vetores $\mathcal{V}$, threshold τ, parâmetros HNSW M, efC e ef
Saída : Conjunto de resultados $\mathcal{A}_{sj}$

```
1  A_sj ← ∅
2  A_knn ← ∅ // em ordem decrescente de similaridade
3  hnsw ← buildHNSW(V.dimension, M, efC, ef)
4  hnsw.insertAll(V) // indexa todos os vetores com o algoritmo
    INSERT do HNSW
5  k_inicial = ef
6  para cada x ∈ V faça
7      k ← k_inicial
8      A_knn ← K-NN-SEARCH(hnsw, x, k)
9      y_k ← vetor com a menor similaridade em A_knn
10     enquanto sim(x, y_k) ≥ τ faça
11         k ← k + 1
12         A_knn ← K-NN-SEARCH(hnsw, x, k)
13         y_k ← vetor com a menor similaridade em A_knn
14     para cada y ∈ A_knn faça
15         se sim(x, y) ≥ τ então
16             A_sj.add(x, y)
17 retorna A_sj
```

7.3 Configuração dos Experimentos

Os experimentos foram conduzidos no servidor do Instituto de Informática da Universidade Federal de Goiás denominado *Sirius*. Este servidor possui as seguintes especificações: Processador Intel(R) Xeon(R) E5-2620 v2 @ 2.10GHz (12 CPU), 32Gb de memória Ram, 550Gb de HD, sistema operacional Debian 10 Buster e Java na versão 17.0.2. Para gerar a representação vetorial dos dados a partir de modelos de redes neurais, foi utilizado a ferramenta *Google Colaboratory*[2].

Os algoritmos HSJ foram implementados na linguagem Java com uso da biblioteca HNSW disponível no GitHub[3] e com a exploração do paralelismo *multicore* do

[2] https://colab.research.google.com/
[3] https://github.com/jelmerk/hnswlib

processador. Foram utilizados os valores 64, 32 e 32 como parâmetros `M`, `efC` e `ef`, respectivamente, em todas as execuções.

Foram realizadas múltiplas execuções dos algoritmos, variando o *threshold* de 0.5 a 0.9 com incremento de 0.1, e os tempos registrados correspondem à média de cinco execuções. Foi definido um limite máximo de 100 horas para execução dos algoritmos. Caso esse limite seja atingido, a execução é finalizada.

7.4 Conjuntos de Dados

Para a realização dos experimentos, foram utilizadas as quatro bases de dados listadas abaixo, escolhidas para abranger diferentes domínios e fornecer uma variedade de características para os experimentos, permitindo uma análise abrangente do desempenho dos algoritmos propostos.

- DBLP[4], um repositório bibliográfico de ciência da computação, contendo informações sobre artigos, autores e conferências;
- IMDB[5], uma base de dados sobre filmes, incluindo informações como títulos, elenco, diretores e avaliações;
- Spotify[6], uma base de dados contendo informações sobre músicas, como artistas, álbuns e características musicais;
- GLOVE, uma base de dados amplamente utilizada em *benchmarks*, que contém vetores de palavras.

Os dados das bases DBLP, IMDB e Spotify foram gerados através da extração de amostras aleatórias e com inclusão de duplicatas nos registros para simular dados sujos. A geração destes dados sujos foi realizada através da substituição, exclusão ou inserção de até cinco caracteres usando uma ferramenta previamente desenvolvida e específica para esta finalidade. Duas versões diferentes foram geradas para as bases DBLP e IMDB a fim de avaliar diferentes quantidades de duplicatas e dimensões dos vetores. As características das versões destes conjuntos de dados podem ser encontradas na Tabela 7.1.

Os vetores densos normalizados para as bases DBLP, IMDB e Spotify foram gerados através do *framework Sentence-Transformers* [85], utilizando os modelos pré-treinados all-MiniLM-L12-v2[7] e all-mpnet-base-v2[8], que geram vetores com 384 e 768 dimensões, respectivamente. Destaca-se que os vetores gerados por esses modelos não

[4] `http://dblp.uni-trier.de`
[5] `https://www.imdb.com/interfaces`
[6] `https://research.atspotify.com/datasets/`
[7] `https://huggingface.co/sentence-transformers/all-MiniLM-L12-v2`
[8] `https://huggingface.co/sentence-transformers/all-mpnet-base-v2`

passaram por nenhum tratamento ou ajuste de escala, sendo que a normalização foi realizada pelo próprio *Sentence-Transformers*. Estes modelos foram selecionados devido apresentarem melhor equilíbrio entre a eficiência na representação e melhores resultados na busca semântica de acordo com o *Massive Text Embedding Benchmark* [72]. Para obter a representação vetorial de cada registro, os atributos de cada registro foram concatenados em uma única sentença, separados pelo *token* [SEP].

Para o conjunto de dados GLOVE, utilizou-se a versão com vetores de 100 dimensões e 1.183.514 registros disponibilizada pelo projeto ANN-Benchmarks[9]. Esses vetores são amplamente utilizados como um *benchmark* na comunidade científica e fornecem uma representação densa de palavras com base em modelos de linguagem pré-treinados.

Datasets	Atributos	Tam. Mínimo	Tam. Máximo	Tam. Médio	Registros	Duplicatas	Total	Dimensões
DBLP-1	Title, Author	6	1081	125	20000	5	100000	384
DBLP-2	Title, Author	6	1081	125	50000	2	100000	768
IMDB-1	Title, Actors, Description	31	477	161	50000	2	100000	384
IMDB-2	Title, Actors, Description	31	477	161	20000	5	100000	768
Spotify	Artist_name, Track_name	5	204	33	50000	2	100000	384

Tabela 7.1: Descrição dos datasets semissintéticos.

7.5 Resultados e Discussão

Nesta seção, são apresentados os resultados dos experimentos com objetivo de avaliar o tempo de execução da operação de junção por similaridade em vetores densos, bem como comparar a taxa de recuperação dos algoritmos com o método exato.

A taxa de recuperação (*recall*) se refere ao percentual de resultados retornados pelo método aproximado em comparação ao método exato e é calculada pela Fórmula 7-1. Vale ressaltar que não há possibilidade de ocorrência de falsos-positivos e, portanto, cada item retornado é de fato um verdadeiro-positivo.

[9]https://github.com/erikbern/ann-benchmarks

$$recall = \frac{|A_{sj}|^{aprox}}{|A_{sj}|^{exat}} \cdot 100, \quad (7\text{-}1)$$

onde $|A_{sj}|^{aprox}$ é a quantidade de pares retornados pelo método aproximado e $|A_{sj}|^{exat}$ é a quantidade de pares retornados pelo método exato.

Os algoritmos foram representados nos gráficos e tabelas da seguinte forma: `ME` para o método exato, `BS` para a abordagem *baseline*, `EXT` para o algoritmo HSJ-Ext, `THS` para o algoritmo HSJ-Ths e `INC` para o algoritmo HSJ-Ths^{Inc}.

7.5.1 Tempo de Execução

Os tempos de execução dos algoritmos nas diferentes bases de dados são apresentados nos gráficos da Figura 7.1. Ressalta-se que as execuções do método exato na base GLOVE com *threshold* 0.5, da abordagem *baseline* na base IMDB-2 com *threshold* 0.5 e da abordagem *baseline* na base GLOVE com os *thresholds* 0.6 e 0.5 foram interrompidas devido atingirem o limite de tempo máximo pré-determinado.

Em todas as bases de dados, é observada uma tendência semelhante em relação ao desempenho dos algoritmos. O método exato apresenta tempos de execução significativamente mais altos em comparação com os algoritmos baseados em HNSW. Por exemplo, na base DBLP-2 e *threshold* 0,9, o método exato leva 2861 segundos para executar, enquanto o HSJ-Ths^{Inc}, que possui o melhor desempenho entre os algoritmos HNSW, leva apenas 77 segundos. A maior diferença em relação ao método exato ocorre na base GLOVE no *threshold* 0.6, onde HSJ-Ths^{Inc} é aproximadamente 2,48 ordens de magnitude (302x) mais rápido que o método exato. Essa diferença é esperada, uma vez que o método exato compara todos os vetores entre si, resultando em um alto custo computacional.

Os algoritmos HSJ-Ext, HSJ-Ths e HSJ-Ths^{Inc}, por sua vez, mostram tempos de execução consideravelmente menores em todas as bases de dados. Esses algoritmos se beneficiam do uso da estruturas de grafos para otimizar a busca de elementos similares.

Observa-se que a abordagem *baseline* obteve tempos similares ao algoritmo HSJ-Ext em *thresholds* altos. No entanto, em *thresholds* baixos, o tempo de execução do *baseline* aumenta consideravelmente, ultrapassando inclusive o método exato, como pode ser visto na base IMDB-1 nos *thresholds* 0.6 e 0.5. Isso ocorre porque, em *thresholds* mais baixos, o número de pares similares tende a ser maior. Devido ao incremento do valor de k no *baseline* ser de 1 em 1, resulta em um grande número de buscas repetidas, realizando a computação da similaridade entre pares repetidas vezes, inclusive em quantidade maior do que a realizada pelo método exato. A maior diferença de tempo de execução ocorre na base IMDB-1 e *threshold* 0.5, onde HSJ-Ths^{Inc} é aproximadamente 3,47 ordens de magnitude (2980x) mais rápido que a abordagem *baseline*.

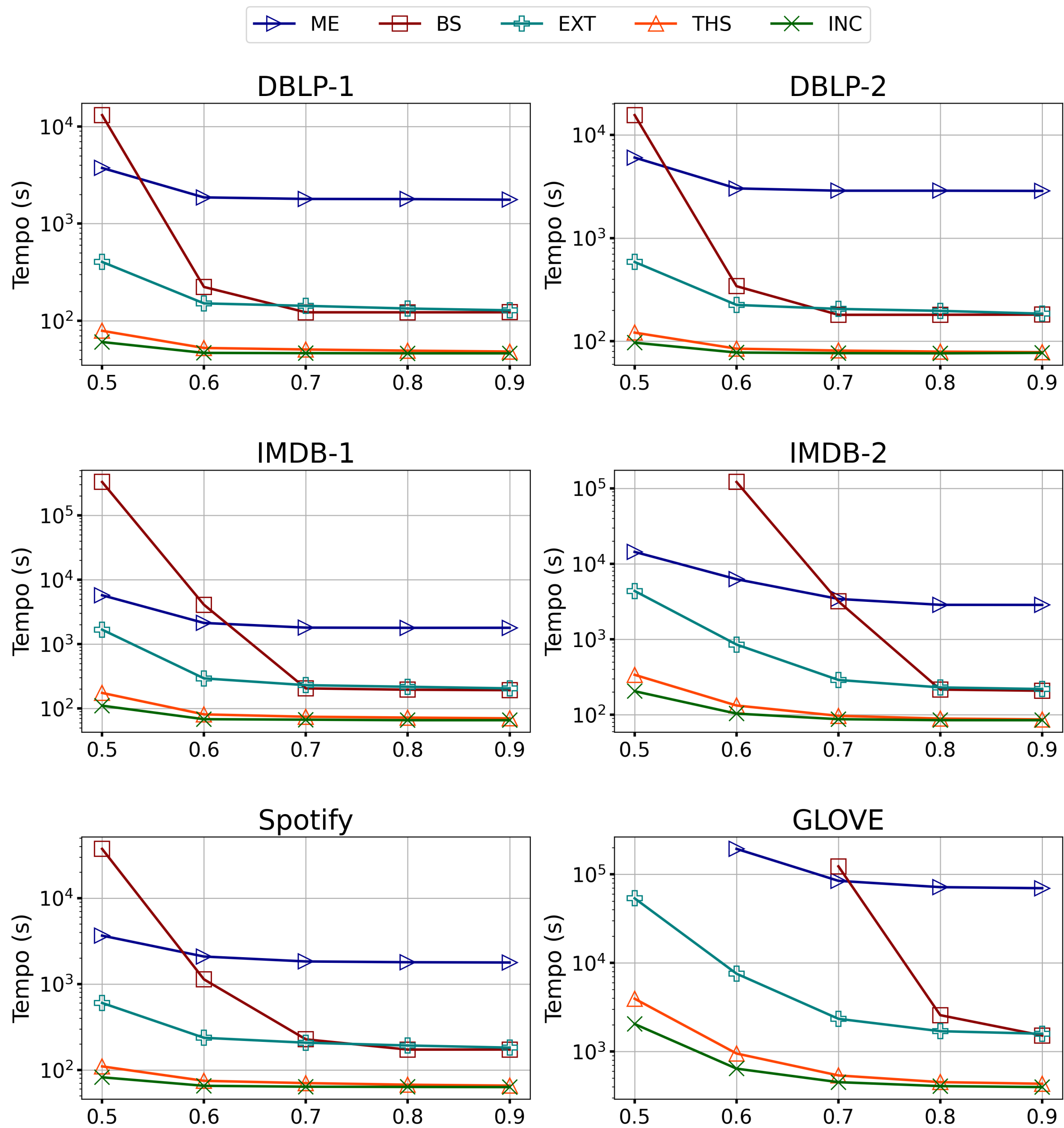


Figura 7.1: Tempo de execução dos algoritmos.

Ao comparar os resultados entre as bases de dados, podemos notar diferenças nos tempos de execução dos algoritmos. A base GLOVE, com mais de 1 milhão de vetores, apresenta os maiores tempos de execução em todas as abordagens. Por outro lado, as bases de dados DBLP, IMDB e Spotify, com 100 mil vetores, mostram tempos de execução menores em comparação com a base GLOVE. Isso é evidente devido os diferentes tamanhos das bases.

Verifica-se que o tempo de execução aumenta quando a dimensionalidade dos vetores de entrada é ampliada de 384 para 768 nas bases DBLP e IMDB. Isso confirma a influência da dimensionalidade dos vetores de entrada no desempenho dos algoritmos,

corroborando a conhecida maldição da dimensionalidade [21].

Ao analisar os resultados em relação aos *thresholds*, observa-se que, em geral, à medida que o *threshold* diminui, os tempos de execução tendem a aumentar para todos os algoritmos. Essa tendência é evidente nas bases de dados IMDB-2, Spotify e GLOVE onde os algoritmos HSJ apresentam tempos de execução crescentes à medida que o *threshold* diminui. Isso ocorre porque à medida que o *threshold* é reduzido, mais vetores similares são retornados durante a travessia do grafo, demandando maior processamento e, consequentemente, aumentando o tempo de execução.

Ao comparar as abordagens otimizadas HSJ-Ths e HSJ-Ths^{Inc} com a versão HSJ-Ext, fica claro que as modificações na estrutura de busca no grafo HNSW resultaram em melhorias significativas de desempenho. Em todas as bases de dados e *thresholds* analisados, as abordagens HSJ-Ths e HSJ-Ths^{Inc} apresentaram tempos de execução menores em relação ao HSJ-Ext. Isso ocorre porque essas abordagens evitam buscas repetidas e desnecessárias que acontecem no HSJ-Ext, retornando apenas os vetores acima do *threshold*, devido à modificação realizada na busca do grafo HNSW.

As diferenças de tempo de execução entre as abordagens otimizadas e o HSJ-Ext variaram conforme a base de dados e o *threshold*. Por exemplo, na base de dados IMDB-1, a abordagem HSJ-Ths^{Inc} foi aproximadamente 15 vezes mais rápida que o HSJ-Ext para o *threshold* de 0.5. Da mesma forma, na base de dados Glove, a abordagem HSJ-Ths^{Inc} superou o HSJ-Ext em aproximadamente 26 vezes para o mesmo *threshold*. Esses resultados demonstram claramente o impacto positivo das otimizações na estrutura de busca no grafo, evidenciando o potencial de melhorar o desempenho da junção por similaridade.

Verifica-se ainda que a abordagem incremental HSJ-Ths^{Inc} obteve tempos de execução ainda menores que a não incremental HSJ-Ths, indicando um desempenho superior. A diferença no desempenho da versão incremental foi maior com *thresholds* mais baixos, como 0.6 e 0.5. Ao analisar os resultados nos *thresholds* mais altos, observa-se que as abordagens apresentam tempos de execução próximos. Isso sugere que, nesses *thresholds* altos, as otimizações adicionais introduzidas na abordagem incremental, podem não ter um impacto significativo em relação ao HSJ-Ths. Uma possível explicação é que, nos *thresholds* mais altos, o número de vetores que atendem ao critério de similaridade é menor, resultando em um conjunto reduzido de candidatos para o HSJ-Ths processar. Portanto, as melhorias adicionais da abordagem HSJ-Ths^{Inc} podem não ser tão evidentes nesse contexto.

Os experimentos realizados com diferentes bases de dados confirmam que os algoritmos baseados em HNSW apresentam tempos de execução significativamente menores em comparação com o método exato e a abordagem *baseline*. Além disso, as abordagens otimizadas na busca do grafo conseguem reduzir ainda mais os tempos em relação

à abordagem sem modificações na estrutura. Os resultados indicam que esses algoritmos são uma opção viável e eficiente para aplicações de junção por similaridade.

7.5.2 Taxa de Recuperação

As taxas de recuperação dos algoritmos nas diferentes bases de dados e *thresholds* são apresentadas na Tabela 7.2. Ao analisar os resultados, observa-se que a maioria das execuções apresentou taxas de recuperação acima de 99%. Isso indica que os métodos aproximados baseados no HNSW foram capazes de recuperar a grande maioria dos vetores em relação ao método exato.

Ressalta-se que as execuções do método *baseline* na base IMDB-2 e *threshold* 0.5, assim como na base GLOVE e *thresholds* 0.6 e 0.5, foram interrompidas por alcançarem o limite de tempo pré-determinado, e, portanto, as taxas de recuperação não estão disponíveis. Além disso, os valores de taxa de recuperação para os métodos HSJ-Ext, HSJ-Ths e HSJ-Ths^{Inc} na base GLOVE com *threshold* 0.5 não puderam ser obtidos devido a execução do método exato para esse *threshold* também ter atingido o limite de tempo, impossibilitando o cálculo do número total de pares similares.

Os algoritmos HSJ-Ths, HSJ-Ths^{Inc} e a abordagem *baseline* obtiveram taxas de recuperação ligeiramente inferiores na maioria das bases e *thresholds*, em comparação com o método HSJ-Ext. Essa redução pode ser atribuída à natureza aproximada desses

Threshold	**ME**	**DBLP-1**				**DBLP-2**			
		BS	**EXT**	**THS**	**INC**	**BS**	**EXT**	**THS**	**INC**
0.9	100%	99.98%	99.98%	99.97%	99.95%	99.99%	99.99%	99.97%	99.94%
0.8	100%	99.94%	99.94%	99.86%	99.92%	99.95%	99.95%	99.93%	99.87%
0.7	100%	99.77%	99.81%	99.75%	99.87%	99.90%	99.92%	99.91%	99.61%
0.6	100%	99.71%	99.75%	99.57%	99.78%	99.87%	99.90%	99.89%	99.41%
0.5	100%	99.13%	99.70%	99.49%	99.12%	98.91%	99.37%	99.23%	98.80%
Threshold	**ME**	**IMDB-1**				**IMDB-2**			
		BS	**EXT**	**THS**	**INC**	**BS**	**EXT**	**THS**	**INC**
0.9	100%	99.95%	99.96%	99.66%	99.58%	99.96%	99.97%	99.97%	99.65%
0.8	100%	99.87%	99.88%	99.30%	98.98%	99.78%	99.81%	99.81%	99.57%
0.7	100%	99.04%	99.49%	99.13%	98.02%	99.51%	99.74%	99.70%	99.49%
0.6	100%	98.19%	99.25%	98.93%	97.83%	99.42%	99.68%	99.55%	99.21%
0.5	100%	98.18%	98.91%	98.63%	97.27%	-	99.52%	99.23%	98.79%
Threshold	**ME**	**Spotify**				**GLOVE**			
		BS	**EXT**	**THS**	**INC**	**BS**	**EXT**	**THS**	**INC**
0.9	100%	99.97%	99.97%	99.91%	99.93%	99.70%	99.75%	99.74%	99.75%
0.8	100%	99.89%	99.90%	99.87%	99.88%	99.53%	99.69%	99.67%	99.59%
0.7	100%	99.75%	99.87%	99.80%	99.61%	98.72%	98.82%	99.03%	98.89%
0.6	100%	99.63%	99.86%	99.75%	99.17%	-	98.01%	98.33%	98.17%
0.5	100%	97.76%	98.61%	98.24%	97.50%	-	-	-	-

Tabela 7.2: Taxa de recuperação dos algoritmos.

métodos e também à tendência do HSJ-Ext de recuperar mais resultados acima do *threshold*. Isso ocorre devido a heurística do incremento do valor de k a cada busca, permitindo uma maior exploração do grafo na busca final e, consequentemente, aumentando o número de resultados, embora seja necessário um pós-processamento para remover vetores com similaridade abaixo do *threshold*.

Percebe-se que na abordagem *baseline*, que não emprega a heurística de incremento do k, mas sim incrementa de 1 em 1, a taxa de recuperação foi ligeiramente inferior à do HSJ-Ext. O baixo incremento na exploração do grafo a cada busca aumenta a probabilidade de encontrar um falso mínimo local mais rapidamente.

Em relação ao método de busca direta por vetores acima do *threshold* no grafo, realizado pelo HSJ-Ths e HSJ-Ths^{Inc}, pode ocorrer a interrupção da busca ao encontrar um falso mínimo local, deixando de fora vetores com similaridade acima do *threshold* que ainda não haviam sido considerados como candidatos, e assim diminuindo a taxa de recuperação.

Além disso, ao analisar os *thresholds* mais altos, é interessante observar que as abordagens HSJ-Ths e HSJ-Ths^{Inc} apresentaram taxas de recuperação próximas umas das outras. Isso sugere que a abordagem incremental do HSJ-Ths^{Inc} não afetou significativamente a capacidade de recuperação em relação ao HSJ-Ths.

No que diz respeito ao paralelismo *multicore*, é importante considerar suas implicações nas abordagens não incrementais (*baseline*, HSJ-Ext e HSJ-Ths) e incremental (HSJ-Ths^{Inc}). Nas abordagens não incrementais, todos os vetores são incluídos previamente no índice, o que significa que, durante a busca, não há problemas de sincronização entre os processos. Cada processo pode realizar a busca no mesmo índice completo, aproveitando o paralelismo *multicore* para acelerar a recuperação. Assim, a capacidade de recuperação não é prejudicada e os resultados obtidos são consistentes.

Por outro lado, na versão incremental, em que cada vetor é indexado após a realização da busca por seus similares, pode ocorrer desafios relacionados à sincronização entre os processos. É possível que uma busca não retorne todos os resultados desejados, pois o vetor que está sendo indexado pode não ter sido considerado por um processo paralelo. Isso ocorre devido à natureza assíncrona da abordagem incremental, em que diferentes processos realizam etapas de busca e indexação simultaneamente.

No entanto, ao analisar os resultados dos experimentos, verifica-se que, mesmo com o paralelismo na versão incremental, a taxa de recuperação manteve próxima às versões não incrementais. Este resultado pode ser atribuído a distribuição dos vetores de entrada, os quais apresentam uma menor quantidade de pares similares em sequência, evitando possíveis problemas de sincronização entre os processos paralelos. Porém, é importante ressaltar que essa conclusão se baseia nos experimentos realizados e nas características das bases de dados utilizadas. Para uma avaliação mais abrangente do

paralelismo na versão incremental, é prudente realizar mais experimentos com outras bases de dados, abrangendo diferentes densidades e distribuição de vetores.

A menor taxa de recuperação ocorreu na base de dados IMDB-1, no algoritmo HSJ-Ths^{Inc} e no *threshold* 0.5, com uma taxa de recuperação de 97.27%. Essa diminuição pode ser atribuída a dois fatores principais. Primeiramente, como mencionado anteriormente, a distribuição dos vetores nesta base pode ter impactado negativamente o sincronismo no processamento paralelo, levando a resultados incompletos em algumas buscas. Além disso, uma densidade maior entre os vetores na base IMDB-1 pode contribuir para o surgimento de falsos mínimos locais durante a busca. Esses falsos mínimos locais podem levar à interrupção prematura da busca no grafo, impedindo a descoberta de todos os vetores com similaridade acima do *threhsold*.

Os resultados obtidos nos experimentos demonstram que os algoritmos aproximados propostos foram altamente eficazes na recuperação dos vetores, apresentando taxas de recuperação geralmente acima de 99%. Tanto o algoritmo HSJ-Ths quanto o HSJ-Ths^{Inc} mostraram melhorias significativas no desempenho em relação ao HSJ-Ext, ao mesmo tempo em que mantiveram taxas de recuperação elevadas. Esses resultados evidenciam a eficiência e eficácia das abordagens propostas na operação de junção por similaridade em espaços vetoriais densos, mesmo considerando a natureza aproximada dos métodos baseados no HNSW.

7.5.3 Comparação com Algoritmo L2AP

Para abordar a Questão 2 de pesquisa, foi realizada uma comparação entre os tempos de execução do algoritmo L2AP aplicado aos dados representados por TF-IDF e dos algoritmos HSJ empregados nos dados com representação densa. Essa comparação foi realizada utilizando a base de dados DBLP, conforme descrito na Tabela 4.1. Na representação densa, foram utilizados vetores de 384 dimensões.

Os resultados dessa comparação estão apresentados na Tabela 7.3. Esses resultados demonstram que é possível obter um desempenho comparável aos métodos tradicionais de representação ao utilizar a junção por similaridade em espaços vetoriais densos gerados por modelos de aprendizado de máquina.

Ao utilizar o *threshold* de 0.9, o algoritmo L2AP com a representação TF-IDF obteve o menor tempo de execução, com apenas 19 segundos. No entanto, os algoritmos HSJ-Ext, HSJ-Ths e HSJ-Ths^{Inc}, que utilizam a representação densa gerada pelo *Sentence-Transformers*, apresentaram tempos de execução de 128s, 48s e 46s, respectivamente. Isso indica que a abordagem de representação densa pode oferecer uma alternativa viável e eficiente em termos de tempo para a junção por similaridade.

Thresholds	Representação com TF-IDF	Representação densa com *Sentence-Transformers*		
	L2AP	EXT	THS	INC
0.9	19s	128s	48s	46s
0.8	79s	134s	49s	46s
0.7	230s	143s	51s	46s
0.6	507s	161s	53s	47s
0.5	904s	404s	79s	60s

Tabela 7.3: Tempos de execução da junção por similaridade com L2AP em representação esparsa versus HSJ em representação densa.

Ao reduzir o *threshold* para 0.8, observa-se um aumento no tempo de execução para todos os algoritmos. O L2AP agora leva 79 segundos, enquanto os algoritmos HSJ-Ext, HSJ-Ths e HSJ-Ths^{Inc} apresentam tempos de execução de 134s, 49s e 46s, respectivamente. Essa diferença sugere que os algoritmos especializados para a representação densa podem ser mais robustos em termos de desempenho à medida que o *threshold* é reduzido. Esta mesma observação é verificada com o *threshold* em 0.7, corroborando a capacidade dos algoritmos otimizados para representação densa de oferecer desempenho comparável aos algoritmos baseados nos métodos tradicionais de representação, mesmo em cenários com *thresholds* mais baixos. Além disso, foi observada uma robustez dos algoritmos HSJ em relação à variação do *threshold*. O algoritmo HSJ-Ths^{Inc} apresentou tempos de execução praticamente iguais nos *thresholds* de 0.9 a 0.6, demonstrando a capacidade do algoritmo de lidar de forma consistente com diferentes valores de *threshold*.

7.5.4 Consumo de Memória

Além do eficiente desempenho que o índice HNSW demonstrou nos experimentos de junção por similaridade, é importante considerar o levantamento do consumo de memória durante o processo de construção do índice. O consumo de memória no HNSW é principalmente influenciado pelo armazenamento das conexões do grafo, sendo que o gasto médio de memória por elemento é definido pela Fórmula B-1, onde o parâmetro M desempenha um papel central. No contexto dos experimentos conduzidos neste estudo, o parâmetro M foi fixado em 64 para todas as execuções.

Qantidade de Vetores	Dimensões	Consumo de Memória
100000	384	375 Mb
100000	768	675 Mb
1183514	100	1815 Mb

Tabela 7.4: Consumo de memória do índice HNSW.

A tabela 7.4 apresenta o consumo de memória associados à construção do índice HNSW, considerando os diversos conjuntos de dados empregados nos experimentos. Os dados revelam que, além das conexões do grafo, o gasto de memória varia de acordo com a quantidade de vetores e a dimensionalidade do espaço vetorial. Para uma coleção de 100.000 vetores com 384 dimensões, o tamanho do índice se estabelece em 375 Mb. Já para a mesma quantidade de vetores, porém com 768 dimensões, o tamanho do índice expande-se para 675 Mb. Enquanto que para um conjunto de 1.183.514 vetores em um espaço de 100 dimensões, o tamanho do índice eleva-se a 1.815 Mb.

Ao apresentar os dados referentes ao consumo de memória durante a construção do índice HNSW, torna-se possível obter uma compreensão dos requisitos de recursos associados à operação dentro dessa estrutura. Essa exposição viabiliza uma análise da escalabilidade inerente à abordagem proposta, contemplando distintos conjuntos de dados, dimensionalidades e volumes de vetores. Adicionalmente, tem-se como intuito proporcionar a profissionais e pesquisadores uma percepção dos recursos de memória necessários, na etapa de construção do índice HNSW, para implementar a operação de junção por similaridade de maneira consistente e efetiva em contextos do mundo real, especialmente em cenários que envolvam grandes volumes de dados.

7.5.5 Sumário dos Resultados

Os resultados dos experimentos demonstram que as técnicas propostas superaram significativamente o método exato e à abordagem *baseline*, alcançando uma melhoria de aproximadamente 2,48 e 3,47 ordens de magnitude, respectivamente, e apresentando percentuais de recuperação próximos a 100%, para os diferentes *thresholds* e bases de dados avaliadas. Além disso, observou-se que as versões HSJ-Ths e HSJ-Ths^{Inc}, que utilizam o novo método de busca no grafo HNSW, sendo a última de forma incremental durante a indexação, alcançaram *speedups* de até 13x e 26x, respectivamente, em relação à versão HSJ-Ext que não modifica os algoritmos HNSW.

Essas análises evidenciam a relevância das modificações introduzidas nas abordagens HSJ-Ths e HSJ-Ths^{Inc} para melhorar o tempo de execução na junção por similaridade. Os resultados indicam que é viável otimizar a técnica de busca do vizinho mais próximo para a operação de junção por similaridade em espaços vetoriais densos. Dessa forma, essas abordagens respondem positivamente à Questão 3 de pesquisa.

CAPÍTULO 8

Conclusões

A junção por similaridade desempenha um papel crucial em várias aplicações, mas calcular a similaridade exata entre pares de objetos pode ser computacionalmente inviável para grandes conjuntos de dados. Essa limitação é ainda mais significativa em vetores densos, onde as técnicas de filtragem empregadas pelos algoritmos de junção por similaridade existentes como o L2AP são inadequadas. Neste trabalho, foi abordado essa limitação e investigado a junção por similaridade aproximada usando a estrutura de grafos HNSW.

Com base no trabalho de pesquisa realizado, pode-se fornecer respostas às três questões levantadas inicialmente:

QP1. As características dos espaços vetoriais densos impactam significativamente o desempenho dos algoritmos existentes de junção por similaridade. Através de experimentos, observa-se que os algoritmos tradicionais baseados em filtragem de vetores esparsos como L2AP não são adequados para lidar com vetores densos, resultando em baixo desempenho e limitações no processamento de grandes conjuntos de dados.

QP2. Com a proposta de junção por similaridade aproximada usando a estrutura do grafo HNSW, foi demonstrado que é possível obter desempenho, comparável aos métodos tradicionais de representação, em espaços vetoriais densos gerados por modelos de aprendizado de máquina.

QP3. O trabalho de pesquisa mostrou que é possível otimizar a técnica de busca do vizinho mais próximo (kNN) para a junção por similaridade em espaços vetoriais densos. O algoritmo proposto de busca no grafo HNSW baseado no *threshold* como critério de parada, conseguiu recuperar eficientemente pares de objetos similares em grandes conjuntos de dados. Por meio de experimentos em vários conjuntos de dados do mundo real, foi demonstrado a redução significativa no tempo de execução em comparação à versão sem modificação na estrutura, até 26x mais rápido, enquanto se manteve taxas de recuperação próximas a 100%.

8.1 Contribuições

As principais contribuições deste trabalho de pesquisa são:

- Nova técnica: A principal contribuição deste trabalho é a proposta de uma técnica de junção por similaridade otimizada para espaços vetoriais densos, fundamentada na utilização da estrutura de grafos HNSW. Além disso, também é apresentada uma abordagem de busca no grafo HNSW baseada em *threshold*.
- Experimentos e avaliações empíricas: Para validar a eficiência e eficácia das técnicas propostas, foram realizados experimentos e avaliações empíricas utilizando vários conjuntos de dados do mundo real. Os resultados obtidos demonstram a viabilidade da técnica proposta em relação ao tempo de execução e taxa de recuperação comparados a método exato e à abordagem *baseline*.

8.2 Trabalhos Futuros

Como trabalhos futuros, destacam-se algumas iniciativas em andamento que complementam e expandem os resultados desta pesquisa. Uma delas é a disponibilização do código fonte dos algoritmos implementados para a comunidade, permitindo que outros pesquisadores possam reproduzir e estender os experimentos realizados. Além disso, está em desenvolvimento uma aplicação prática de correspondência de entidades que utiliza os algoritmos HSJ como parte integrante de um *pipeline* de processamento. Essa aplicação visa solucionar desafios específicos de correspondência de entidades em um contexto real.

Destaca-se que a técnica proposta neste trabalho pode ser aplicada em diversas áreas, como por exemplo, sistemas de recomendação de conteúdo, busca de documentos em sistemas de informação, reconhecimento de padrões, limpeza e integração em bancos de dados, correspondência entre entidades, bem como servir de operação auxiliar em *pipelines* de aprendizado de máquina e análise de dados.

É interessante testar a aplicação dos algoritmos propostos em outros tipos de dados, como imagens e áudios, bem como explorar vetores multimodais que combinam diferentes modalidades de informação. Essa análise exploratória permitirá avaliar a capacidade dos algoritmos em lidar com dados de naturezas diversas, ampliando suas possibilidades de aplicação.

Adicionalmente, há espaço para explorar otimizações na abordagem proposta. Uma possibilidade é utilizar o limite de similaridade durante a construção do grafo HNSW para derivar filtros que possam podar candidatos. Essa estratégia pode contribuir para aprimorar o desempenho do método e reduzir ainda mais o tempo de processamento.

Outra iniciativa futura promissora consiste na exploração de paralelismo via GPUs para aprimorar o desempenho dos algoritmos HSJ. A utilização de GPUs permite

distribuir as tarefas de processamento entre vários núcleos, acelerando operações intensivas e possibilitando que a operação seja mais eficiente em grandes volumes de dados. Essa abordagem requer adaptações e otimizações dos algoritmos, considerando a eficiência na transferência de dados e explorando técnicas específicas de otimização para GPUs.

Também é relevante conduzir uma avaliação estratificada das bases de dados, considerando diferentes níveis de complexidade e dimensionalidade dentro da mesma base de dados. Esse enfoque permitirá uma investigação mais profunda sobre como os métodos propostos se comportam em cenários variados.

Além disso, é relevante investigar outras estruturas de indexação e algoritmos de busca aproximada, além das abordadas nesta pesquisa, para comparações com o método proposto na operação de junção por similaridade em vetores densos. Também é interessante verificar a combinação do HNSW com outras técnicas, como quantização. Essas comparações permitirão uma avaliação mais abrangente das vantagens e limitações, fornecendo *insights* valiosos sobre a eficiência e a eficácia da abordagem proposta em relação a outras técnicas.

Por fim, é importante considerar a análise do consumo de memória das técnicas propostas durante a fase de junção, uma vez que no presente estudo somente foi avaliada essa métrica na construção do índice HNSW. Também é pertinente analisar em mais detalhes o impacto dos parâmetros do HNSW na eficiência e eficácia da técnica proposta.

Ao explorar essas iniciativas, será possível aprimorar e fortalecer as contribuições desta pesquisa, fornecendo uma base sólida para avanços futuros na área de junção por similaridade em espaços vetoriais densos.

8.3 Publicações Geradas

Durante o desenvolvimento deste trabalho de pesquisa, foram realizadas as seguintes publicações:

- Santana, D.; Ribeiro, L. A. **Junções por Similaridade em Espaços Vetoriais Semânticos.** Simpósio Brasileiro de Banco de Dados (SBBD), 37, 2022, Búzios. Anais. Porto Alegre: Sociedade Brasileira de Computação, 2022 . p. 147-153.
- Lima, P.; Santana, D.; Santos Martins, W.; Ribeiro, L. A. **Evaluation of Deep Learning Techniques for Entity Matching.** In Proceedings of the 25th International Conference on Enterprise Information Systems - Volume 1: ICEIS, 2023.
- (*artigo aceito*) Santana, D.; Ribeiro, L. A. **Approximate Similarity Joins over Dense Vector Embeddings.** Simpósio Brasileiro de Banco de Dados (SBBD), 25 a 29 de setembro de 2023, Belo Horizonte.

Referências Bibliográficas

[1] ABDI, H.; WILLIAMS, L. J. **Principal component analysis**. *WIREs Computational Statistics*, 2(4):433–459, 2010.

[2] ANASTASIU, D. C.; KARYPIS, G. **L2AP: Fast Cosine Similarity Search with Prefix L-2 Norm Bounds**. In: *Proceedings of the ICDE Conference*, p. 784–795, 2014.

[3] ANASTASIU, D. C.; KARYPIS, G. **L2knng: Fast exact k-nearest neighbor graph construction with l2-norm pruning**. In: *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, CIKM '15, p. 791–800, New York, NY, USA, 2015. Association for Computing Machinery.

[4] ANASTASIU, D. C.; KARYPIS, G. **Pl2ap: Fast parallel cosine similarity search**. In: *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '15, New York, NY, USA, 2015. Association for Computing Machinery.

[5] ARASU, A.; GANTI, V.; KAUSHIK, R. **Efficient exact set-similarity joins**. In: *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, p. 918–929. VLDB Endowment, 2006.

[6] AUMÜLLER, M.; BERNHARDSSON, E.; FAITHFULL, A. J. **ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms**. *Information Systems*, 87, 2020.

[7] AWEKAR, A.; SAMATOVA, N. F. **Fast matching for all pairs similarity search**. In: *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, volume 1, p. 295–300, 2009.

[8] BARLAUG, N.; GULLA, J. A. **Neural networks for entity matching: A survey**. *ACM Trans. Knowl. Discov. Data*, 15(3), apr 2021.

[9] BAYARDO, R. J.; MA, Y.; SRIKANT, R. **Scaling up All Pairs Similarity Search**. In: *Proceedings of the WWW Conference*, p. 131–140, 2007.

[10] BEDO, M.; RAMOS, J. S.; TRAINA, A. J. M.; TRAINA, C.; NOGUEIRA-BARBOSA, M. H.; AZEVEDO-MARQUES, P. M. **Wia-spine: A cbir environment with embedded radiomic features to assess fragility fractures**. In: *2022 IEEE 35th International Symposium on Computer-Based Medical Systems (CBMS)*, p. 72–77, 2022.

[11] BEDO, M. V. N.; TRAINA, A. J. M.; TRAINA JR., C. **Seamless integration of distance functions and feature vectors for similarity-queries processing**. *Journal of Information and Data Management*, 5(3):308, Oct. 2014.

[12] BELLAS, C.; GOUNARIS, A. **An empirical evaluation of exact set similarity join techniques using gpus**. *Information Systems*, 89:101485, 2020.

[13] BERNHARDSSON, E. **Spotify/annoy: Approximate nearest neighbors in c++/python optimized for memory usage and loading/saving to disk**, 2015.

[14] BOUROS, P.; GE, S.; MAMOULIS, N. **Spatio-textual similarity joins**. *Proc. VLDB Endow.*, 6(1):1–12, nov 2012.

[15] BRIGGS, J. **Natural language processing (nlp) for semantic search**, 2022.

[16] BROWN, T. B.; MANN, B.; RYDER, N.; SUBBIAH, M.; KAPLAN, J.; DHARIWAL, P.; NEELAKANTAN, A.; SHYAM, P.; SASTRY, G.; ASKELL, A.; AGARWAL, S.; HERBERT-VOSS, A.; KRUEGER, G.; HENIGHAN, T.; CHILD, R.; RAMESH, A.; ZIEGLER, D. M.; WU, J.; WINTER, C.; HESSE, C.; CHEN, M.; SIGLER, E.; LITWIN, M.; GRAY, S.; CHESS, B.; CLARK, J.; BERNER, C.; MCCANDLISH, S.; RADFORD, A.; SUTSKEVER, I.; AMODEI, D. **Language models are few-shot learners**, 2020.

[17] C., P. S. G.; ARDALAN, A.; DOAN, A.; AKELLA, A. **Smurf: Self-service string matching using random forests**. *Proc. VLDB Endow.*, 12(3):278–291, nov 2018.

[18] CER, D.; YANG, Y.; KONG, S.-Y.; HUA, N.; LIMTIACO, N.; JOHN, R. S.; CONSTANT, N.; GUAJARDO-CESPEDES, M.; YUAN, S.; TAR, C.; SUNG, Y.-H.; STROPE, B.; KURZWEIL, R. **Universal sentence encoder**, 2018.

[19] CHAUDHURI, S.; GANTI, V.; KAUSHIK, R. **A Primitive Operator for Similarity Joins in Data Cleaning**. In: *Proceedings of the ICDE Conference*, p. 5, 2006.

[20] CHEN, G. **Scalable spectral clustering with cosine similarity**. In: *2018 24th International Conference on Pattern Recognition (ICPR)*, p. 314–319, 2018.

[21] CHEN, L. **Curse of Dimensionality**, p. 545–546. Springer US, Boston, MA, 2009.

[22] CHEN, Z.; WANG, Y.; NARASAYYA, V.; CHAUDHURI, S. **Customizable and scalable fuzzy join for big data**. *Proc. VLDB Endow.*, 12(12):2106–2117, aug 2019.

[23] CHEW, L. P. **Constrained delaunay triangulations**. In: *Proceedings of the Third Annual Symposium on Computational Geometry*, SCG '87, p. 215–222, New York, NY, USA, 1987. Association for Computing Machinery.

[24] CHRISTIANI, T.; PAGH, R.; SIVERTSEN, J. **Scalable and robust set similarity join**. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, p. 1240–1243, 2018.

[25] CHU, X.; ILYAS, I. F.; KRISHNAN, S.; WANG, J. **Data cleaning: Overview and emerging challenges**. In: *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, p. 2201–2206, New York, NY, USA, 2016. Association for Computing Machinery.

[26] COHEN, W. W. **Integration of heterogeneous databases without common domains using queries based on textual similarity**. In: *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, p. 201–212, New York, NY, USA, 1998. Association for Computing Machinery.

[27] DE OLIVEIRA NOVAES, J. V.; SANTOS, L. F. D.; TRAINA, A. J. M.; TRAINA JR, C. **Ortree: Tuning diversified similarity queries by means of data partitioning**. In: *Advances in Databases and Information Systems: 26th European Conference, ADBIS 2022, Turin, Italy, 2022, Proceedings*, p. 165–178. Springer, 2022.

[28] DENG, D.; KIM, A.; MADDEN, S.; STONEBRAKER, M. **Silkmoth: An efficient method for finding related sets with maximum matching constraints**. *Proc. VLDB Endow.*, 10(10):1082–1093, jun 2017.

[29] DENG, D.; LI, G.; WEN, H.; FENG, J. **An efficient partition based method for exact set similarity joins**. *Proc. VLDB Endow.*, 9(4):360–371, dec 2015.

[30] DEVLIN, J.; CHANG, M.; LEE, K.; TOUTANOVA, K. **BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding**. In: *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, p. 4171–4186, 2019.

[31] DO CARMO OLIVEIRA, D. J.; BORGES, F. F.; RIBEIRO, L. A.; CUZZOCREA, A. **Set similarity joins with complex expressions on distributed platforms**. In: Benczúr, A.; Thalheim, B.; Horváth, T., editors, *Advances in Databases and Information Systems - 22nd European Conference, ADBIS 2018, Budapest, Hungary, September 2-5, 2018, Proceedings*, volume 11019 de **Lecture Notes in Computer Science**, p. 216–230. Springer, 2018.

[32] ECHIHABI, K.; TSANDILAS, T.; GOGOLOU, A.; BEZERIANOS, A.; PALPANAS, T. **Pros: data series progressive k-nn similarity search and classification with probabilistic quality guarantees**. *The VLDB Journal*, 32(4):763–789, Jul 2023.

[33] ECHIHABI, K.; ZOUMPATIANOS, K.; PALPANAS, T. **High-dimensional similarity search for scalable data science**. In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, p. 2369–2372, 2021.

[34] ECHIHABI, K.; ZOUMPATIANOS, K.; PALPANAS, T.; BENBRAHIM, H. **The lernaean hydra of data series similarity search: An experimental evaluation of the state of the art**. *Proc. VLDB Endow.*, 12(2):112–127, oct 2018.

[35] ECHIHABI, K.; ZOUMPATIANOS, K.; PALPANAS, T.; BENBRAHIM, H. **Return of the lernaean hydra: Experimental evaluation of data series approximate similarity search**. *Proc. VLDB Endow.*, 13(3):403–420, nov 2019.

[36] EMINAGAOGLU, M. **A new similarity measure for vector space models in text classification and information retrieval**. *J. Inf. Sci.*, 48(4):463–476, aug 2022.

[37] FAN, G.; WANG, J.; LI, Y.; ZHANG, D.; MILLER, R. J. **Semantics-aware dataset discovery from data lakes with contextualized column-based representation learning**. *Proc. VLDB Endow.*, 16(7):1726–1739, may 2023.

[38] FASOLIN, K.; FILETO, R.; KRUGERY, M.; KASTERZ, D. S.; FERREIRAX, M. R. P.; CORDEIROX, R. L. F.; TRAINAX, A. J. M.; TRAINA, C. **Efficient execution of conjunctive complex queries on big multimedia databases**. In: *2013 IEEE International Symposium on Multimedia*, p. 536–543, 2013.

[39] FAUZI, M. A.; UTOMO, D. C.; SETIAWAN, B. D.; PRAMUKANTORO, E. S. **Automatic essay scoring system using n-gram and cosine similarity for gamification based e-learning**. In: *Proceedings of the International Conference on Advances in Image Processing*, ICAIP '17, p. 151–155, New York, NY, USA, 2017.

[40] FIER, F.; AUGSTEN, N.; BOUROS, P.; LESER, U.; FREYTAG, J.-C. **Set similarity joins on mapreduce: An experimental survey**. *Proc. VLDB Endow.*, 11(10):1110–1122, jun 2018.

[41] FOSTER, C.; SEVILMIS, B.; KIMIA, B. **Generalized relative neighborhood graph (grng) for similarity search**. In: Skopal, T.; Falchi, F.; Lokoč, J.; Sapino, M. L.; Bartolini, I.; Patella, M., editors, *Similarity Search and Applications*, p. 133–149, Cham, 2022. Springer International Publishing.

[42] FU, C.; XIANG, C.; WANG, C.; CAI, D. **Fast approximate nearest neighbor search with the navigating spreading-out graph**. *Proc. VLDB Endow.*, 12(5):461–474, jan 2019.

[43] GOLDBERG, Y. **A primer on neural network models for natural language processing**. *Journal of Artificial Intelligence Research*, 57:345–420, 2016.

[44] GOMAA, W. H.; FAHMY, A. A. **Article: A survey of text similarity approaches**. *International Journal of Computer Applications*, 68(13):13–18, April 2013. Full text available.

[45] GOWANLOCK, M.; KARSIN, B. **Accelerating the similarity self-join using the gpu**. *Journal of Parallel and Distributed Computing*, 133:107–123, 2019.

[46] GUO, R.; SUN, P.; LINDGREN, E.; GENG, Q.; SIMCHA, D.; CHERN, F.; KUMAR, S. **Accelerating large-scale inference with anisotropic vector quantization**. In: III, H. D.; Singh, A., editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 de **Proceedings of Machine Learning Research**, p. 3887–3896. PMLR, 13–18 Jul 2020.

[47] HABERT, B.; ADDA, G.; ADDA-DECKER, M.; DE MAREÜIL, P. B.; FERRARI, S.; FERRET, O.; ILLOUZ, G.; PARAUBECK, P. **Towards tokenization evaluation.** In: *Lrec*, p. 427–432, 1998.

[48] HUANG, Q.; FENG, J.; ZHANG, Y.; FANG, Q.; NG, W. **Query-aware locality-sensitive hashing for approximate nearest neighbor search**. *Proc. VLDB Endow.*, 9(1):1–12, sep 2015.

[49] JAFARI, O.; MAURYA, P.; NAGARKAR, P.; ISLAM, K. M.; CRUSHEV, C. **A survey on locality sensitive hashing algorithms and their applications**. *arXiv preprint arXiv:2102.08942*, 2021.

[50] JIANG, Y.; LI, G.; FENG, J.; LI, W.-S. **String similarity joins: An experimental evaluation**. *Proc. VLDB Endow.*, 7(8):625–636, apr 2014.

[51] JOHNSON, J.; DOUZE, M.; JÉGOU, H. **Billion-scale similarity search with gpus**. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[52] JÉGOU, H.; DOUZE, M.; SCHMID, C. **Product quantization for nearest neighbor search**. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[53] KAGGLE. **2017 the state of data science & machine learning**, 2017.

[54] KAGGLE. **2019 kaggle machine learning & data science survey**, 2019.

[55] KHATTER, H.; GOEL, N.; GUPTA, N.; GULATI, M. **Movie recommendation system using cosine similarity with sentiment analysis**. In: *2021 Third International Conference on Inventive Research in Computing Applications (ICIRCA)*, p. 597–603, 2021.

[56] KLEINBERG, J. **The small-world phenomenon: An algorithmic perspective**. In: *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, STOC '00, p. 163–170, New York, NY, USA, 2000. Association for Computing Machinery.

[57] LEE, D.; PARK, J.; SHIM, J.; LEE, S.-G. **An efficient similarity join algorithm with cosine similarity predicate**. In: Bringas, P. G.; Hameurlain, A.; Quirchmayr, G., editors, *Database and Expert Systems Applications*, p. 422–436, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[58] LEVCHENKO, O.; KOLEV, B.; YAGOUBI, D.-E.; AKBARINIA, R.; MASSEGLIA, F.; PALPANAS, T.; SHASHA, D.; VALDURIEZ, P. **Bestneighbor: efficient evaluation of knn queries on large time series databases**. *Knowledge and Information Systems*, 63(2):349–378, Feb 2021.

[59] LEVENSHTEIN, V. **Binary Codes Capable of Correcting Deletions, Insertions and Reversals**. *Soviet Physics Doklady*, 10:707, 1966.

[60] LI, G.; DENG, D.; WANG, J.; FENG, J. **Pass-join: A partition-based method for similarity joins**. *Proc. VLDB Endow.*, 5(3):253–264, nov 2011.

[61] LI, W.; ZHANG, Y.; SUN, Y.; WANG, W.; LI, M.; ZHANG, W.; LIN, X. **Approximate nearest neighbor search on high dimensional data — experiments, analyses, and improvement**. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1475–1488, 2020.

[62] LI, Y.; YU, X.; KOUDAS, N. **Les3: Learning-based exact set similarity search**. *Proc. VLDB Endow.*, 14(11):2073–2086, jul 2021.

[63] LI, Y.; LI, J.; SUHARA, Y.; DOAN, A.; TAN, W.-C. **Effective entity matching with transformers**. *The VLDB Journal*, Jan 2023.

[64] LI, Y.; LI, J.; SUHARA, Y.; DOAN, A.; TAN, W.-C. **Deep entity matching with pre-trained language models**. *Proc. VLDB Endow.*, 14(1):50–60, sep 2020.

[65] MALKOV, Y. A.; YASHUNIN, D. A. **Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs**. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(4):824–836, 2020.

[66] MANN, W.; AUGSTEN, N.; BOUROS, P. **An empirical evaluation of set similarity join techniques**. *Proc. VLDB Endow.*, 9(9):636–647, may 2016.

[67] MIKOLOV, T.; YIH, W.-T.; ZWEIG, G. **Linguistic regularities in continuous space word representations**. *Proceedings of NAACL-HLT*, p. 746–751, 01 2013.

[68] MIKOLOV, T.; CHEN, K.; CORRADO, G.; DEAN, J. **Efficient Estimation of Word Representations in Vector Space**. In: Bengio, Y.; LeCun, Y., editors, *International Conference on Learning Representations*, 2013.

[69] MIKOLOV, T.; GRAVE, E.; BOJANOWSKI, P.; PUHRSCH, C.; JOULIN, A. **Advances in pre-training distributed word representations**. In: *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[70] MOUNTANTONAKIS, M.; TZITZIKAS, Y. **Large-scale semantic integration of linked data: A survey**. *ACM Comput. Surv.*, 52(5), sep 2019.

[71] MUDGAL, S.; LI, H.; REKATSINAS, T.; DOAN, A.; PARK, Y.; KRISHNAN, G.; DEEP, R.; ARCAUTE, E.; RAGHAVENDRA, V. **Deep Learning for Entity Matching: A Design Space Exploration**. In: *Proceedings of the SIGMOD Conference*, p. 19–34, 2018.

[72] MUENNIGHOFF, N.; TAZI, N.; MAGNE, L.; REIMERS, N. **MTEB: Massive text embedding benchmark**. In: *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, p. 2014–2037, Dubrovnik, Croatia, May 2023. Association for Computational Linguistics.

[73] MUJA, M.; LOWE, D. G. **Scalable nearest neighbor algorithms for high dimensional data**. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, Nov 2014.

[74] NARGESIAN, F.; ZHU, E.; MILLER, R. J.; PU, K. Q.; AROCENA, P. C. **Data lake management: Challenges and opportunities**. *Proc. VLDB Endow.*, 12(12):1986–1989, aug 2019.

[75] NOBARI, A. D.; RAFIEI, D. **Efficiently transforming tables for joinability**. In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, p. 1649–1661, 2022.

[76] O. NOVAES, J. V.; F. D. SANTOS, L.; OLMES CARVALHO, L.; DE OLIVEIRA, D.; V. N. BEDO, M.; J. M. TRAINA, A.; TRAINA JR., C. **J-eda: A workbench for tuning similarity and diversity search parameters in content-based image retrieval**. *Journal of Information and Data Management*, 12(2), Sep. 2021.

[77] PAIK, J. H. **A novel tf-idf weighting scheme for effective ranking**. In: *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, p. 343–352, New York, NY, USA, 2013.

[78] PAPADAKIS, G.; SKOUTAS, D.; THANOS, E.; PALPANAS, T. **Blocking and filtering techniques for entity resolution: A survey**. *ACM Comput. Surv.*, 53(2), mar 2020.

[79] PAPARRIZOS, J.; EDIAN, I.; LIU, C.; ELMORE, A. J.; FRANKLIN, M. J. **Fast adaptive similarity search through variance-aware quantization**. In: *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, p. 2969–2983, 2022.

[80] PENNINGTON, J.; SOCHER, R.; MANNING, C. **GloVe: Global vectors for word representation**. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, p. 1532–1543, Doha, Qatar, Oct. 2014.

[81] QIN, J.; WANG, W.; LU, Y.; XIAO, C.; LIN, X. **Efficient exact edit similarity query processing with the asymmetric signature scheme**. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, p. 1033–1044, New York, NY, USA, 2011. Association for Computing Machinery.

[82] QUIRINO, R. D.; RIBEIRO-JÚNIOR, S.; RIBEIRO, L. A.; MARTINS, W. S. **fgssjoin: A gpu-based algorithm for set similarity joins**. In: Hammoudi, S.; Smialek, M.; Camp, O.; Filipe, J., editors, *ICEIS 2017 - Proceedings of the 19th International Conference on Enterprise Information Systems, Volume 1, Porto, Portugal, April 26-29, 2017*, p. 152–161. SciTePress, 2017.

[83] RAFFEL, C.; SHAZEER, N.; ROBERTS, A.; LEE, K.; NARANG, S.; MATENA, M.; ZHOU, Y.; LI, W.; LIU, P. J. **Exploring the limits of transfer learning with a unified text-to-text transformer**, 2019.

[84] RAMOS, J. S.; WATANABE, C. Y.; TRAINA, C.; TRAINA, A. J. **How to speed up outliers removal in image matching**. *Pattern Recognition Letters*, 114:31–40, 2018. Data Representation and Representation Learning for Video Analysis.

[85] REIMERS, N.; GUREVYCH, I. **Sentence-bert: Sentence embeddings using siamese bert-networks**. *CoRR*, abs/1908.10084, 2019.

[86] RIBEIRO, L.; BORGES, F.; OLIVEIRA, D. **A framework for set similarity join on multi-attribute data**. In: *Anais do XXXV Simpósio Brasileiro de Bancos de Dados*, p. 61–72, Porto Alegre, RS, Brasil, 2020. SBC.

[87] RIBEIRO, L. A.; HÄRDER, T. **Generalizing Prefix Filtering to Improve Set Similarity Joins**. *Information Systems*, 36(1):62–78, 2011.

[88] RIBEIRO-JÚNIOR, S.; QUIRINO, R. D.; RIBEIRO, L. A.; MARTINS, W. S. **Fast parallel set similarity joins on many-core architectures**. *J. Inf. Data Manag.*, 8(3):255–270, 2017.

[89] ROBERTSON, S. E.; JONES, K. S. **Relevance weighting of search terms**. *Journal of the American Society for Information Science*, 27(3):129–146, 1976.

[90] SALTON, G.; WONG, A.; YANG, C. **A Vector Space Model for Automatic Indexing**. *Communications of the ACM*, 18(11):613–620, 1975.

[91] SATULURI, V.; PARTHASARATHY, S. **Bayesian locality sensitive hashing for fast similarity search**. *Proc. VLDB Endow.*, 5(5):430–441, jan 2012.

[92] SHIMOMURA, L. C.; OYAMADA, R. S.; VIEIRA, M. R.; KASTER, D. S. **A survey on graph-based methods for similarity searches in metric spaces**. *Information Systems*, 95:101507, 2021.

[93] SILVA, L.; RIBEIRO, L. **Junções por similaridade usando processamento distribuído e paralelismo massivo**. In: *Anais do XXXVII Simpósio Brasileiro de Bancos de Dados*, p. 421–426, Porto Alegre, RS, Brasil, 2022. SBC.

[94] SURI, S.; ILYAS, I. F.; RÉ, C.; REKATSINAS, T. **Ember: No-code context enrichment via similarity-based keyless joins**. *Proc. VLDB Endow.*, 15(3):699–712, nov 2021.

[95] T. CAZZOLATO, M.; C. SCABORA, L.; F. ZABOT, G.; A. GUTIERREZ, M.; TRAINA JR., C.; J. M. TRAINA, A. **Featset+: Visual features extracted from public image datasets**. *Journal of Information and Data Management*, 13(1), Aug. 2022.

[96] TAO, W.; DENG, D.; STONEBRAKER, M. **Approximate string joins with abbreviations**. *Proc. VLDB Endow.*, 11(1):53–65, sep 2017.

[97] TATA, S.; PATEL, J. M. **Estimating the selectivity of tf-idf based cosine similarity predicates**. *SIGMOD Rec.*, 36(2):7–12, jun 2007.

[98] TURIAN, J.; RATINOV, L.; BENGIO, Y. **Word representations: A simple and general method for semi-supervised learning**. In: *Proceedings of the 48th Annual*

Meeting of the Association for Computational Linguistics, ACL '10, p. 384–394, USA, 2010. Association for Computational Linguistics.

[99] UKEY, N.; YANG, Z.; LI, B.; ZHANG, G.; HU, Y.; ZHANG, W. **Survey on exact knn queries over high-dimensional data space**. *Sensors*, 23(2), 2023.

[100] UKKONEN, E. **Approximate string-matching with q-grams and maximal matches**. *Theoretical Computer Science*, 92(1):191–211, 1992.

[101] VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L.; GOMEZ, A. N.; KAISER, L.; POLOSUKHIN, I. **Attention is All you Need**. In: *Annual Conference on Neural Information Processing Systems*, p. 5998–6008, 2017.

[102] WANG, J.; YI, X.; GUO, R.; JIN, H.; XU, P.; LI, S.; WANG, X.; GUO, X.; LI, C.; XU, X.; YU, K.; YUAN, Y.; ZOU, Y.; LONG, J.; CAI, Y.; LI, Z.; ZHANG, Z.; MO, Y.; GU, J.; JIANG, R.; WEI, Y.; XIE, C. **Milvus: A Purpose-Built Vector Data Management System**. In: *Proceedings of the SIGMOD Conference*, p. 2614–2627, 2021.

[103] WANG, J.; FENG, J.; LI, G. **Trie-join: Efficient trie-based string similarity joins with edit-distance constraints**. *Proc. VLDB Endow.*, 3(1–2):1219–1230, sep 2010.

[104] WANG, J.; LI, G.; FE, J. **Fast-join: An efficient method for fuzzy token matching based string similarity join**. In: *2011 IEEE 27th International Conference on Data Engineering*, p. 458–469, 2011.

[105] WANG, J.; LI, G.; FENG, J. **Can we beat the prefix filtering? an adaptive framework for similarity join and search**. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, p. 85–96, New York, NY, USA, 2012. Association for Computing Machinery.

[106] WANG, J.; YANG, X.; WANG, B.; LIU, C. **Ls-join: Local similarity join on string collections (extended abstract)**. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, p. 1779–1780, 2018.

[107] WANG, J.; LIN, C.; ZANIOLO, C. **Mf-join: Efficient fuzzy string similarity join with multi-level filtering**. In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, p. 386–397, April 2019.

[108] WANG, M.; XU, X.; YUE, Q.; WANG, Y. **A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search**. *Proc. VLDB Endow.*, 14(11):1964–1978, jul 2021.

[109] WANG, W.; QIN, J.; XIAO, C.; LIN, X.; SHEN, H. T. **Vchunkjoin: An efficient algorithm for edit similarity joins**. *IEEE Transactions on Knowledge and Data Engineering*, 25(8):1916–1929, 2013.

[110] WANG, X.; QIN, L.; LIN, X.; ZHANG, Y.; CHANG, L. **Leveraging set relations in exact set similarity join**. *Proc. VLDB Endow.*, 10(9):925–936, may 2017.

[111] XIAO, C.; WANG, W.; LIN, X. **Ed-join: An efficient algorithm for similarity joins with edit distance constraints**. *Proc. VLDB Endow.*, 1(1):933–944, aug 2008.

[112] XIAO, C.; WANG, W.; LIN, X.; SHANG, H. **Top-k set similarity joins**. In: *2009 IEEE 25th International Conference on Data Engineering*, p. 916–927, 2009.

[113] XIAO, C.; WANG, W.; LIN, X.; YU, J. X.; WANG, G. **Efficient similarity joins for near-duplicate detection**. *ACM Trans. Database Syst.*, 36(3), aug 2011.

[114] YAN, D.; WANG, Y.; WANG, J.; WANG, H.; LI, Z. **K-nearest neighbor search by random projection forests**. *IEEE Transactions on Big Data*, 7(1):147–157, 2021.

[115] YANG, K.; SHAHABI, C. **A pca-based similarity measure for multivariate time series**. In: *Proceedings of the 2nd ACM International Workshop on Multimedia Databases*, MMDB '04, p. 65–74, New York, NY, USA, 2004.

[116] YUJIAN, L.; BO, L. **A normalized levenshtein distance metric**. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1091–1095, June 2007.

[117] ZEAKIS, A.; SKOUTAS, D.; SACHARIDIS, D.; PAPAPETROU, O.; KOUBARAKIS, M. **Tokenjoin: Efficient filtering for set similarity join with maximum weighted bipartite matching**. *Proc. VLDB Endow.*, 16(4):790–802, dec 2022.

[118] ZHAI, J.; LOU, Y.; GEHRKE, J. **Atlas: A probabilistic algorithm for high dimensional similarity search**. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, p. 997–1008, New York, NY, USA, 2011. Association for Computing Machinery.

[119] ZHANG, Y.; LI, X.; WANG, J.; ZHANG, Y.; XING, C.; YUAN, X. **An efficient framework for exact set similarity search using tree structure indexes**. In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, p. 759–770, 2017.

[120] ZHU, X.; LI, X.; ZHANG, S.; XU, Z.; YU, L.; WANG, C. **Graph pca hashing for similarity search**. *IEEE Transactions on Multimedia*, 19(9):2033–2044, 2017.

APÊNDICE A

Algoritmo L2AP

Para facilitar a compreensão do algoritmo L2AP, é importante entender sua base no algoritmo AllPairs. Portanto, a seguir, serão apresentados os pseudo-códigos e descrições das implementações de ambos os algoritmos.

Serão destacadas as diferenças entre suas implementações, bem como as principais otimizações realizadas em cada um. Para uma melhor compreensão das notações utilizadas, é possível consultar a Tabela A.1.

Tanto AllPairs como L2AP utilizam um esquema de índice invertido e o constroem de forma incremental, vetor por vetor. O índice invertido é um conjunto de n listas associadas, $\{I = I_1, I_2, ... I_n\}$, onde cada lista I_j está relacionada a um token e contém todos os pares $(\mathbf{x}, x_j)$, $onde\ x_j \neq 0$, sendo que $\mathbf{x}$ representa o vetor a qual o token está presente e x_j o valor deste token no referido vetor.

Métodos são utilizados para minimizar o tamanho do índice invertido para otimizar a computação, como será detalhado nas Seções A.1.1 e A.2.1, e assim apenas uma parte dos tokens de cada vetor é necessária ser adicionada desde que garantam que este vetor e todos os seus semelhantes compartilhem pelo menos de um token comum. A parte não indexada é definida como prefixo do vetor ou x', e a parte indexada como sufixo ou x''.

A execução é iniciada com o índice vazio e para cada vetor x do conjunto D, é computado os pares semelhantes, de x com todos vetores y presentes no índice invertido, a partir do *threshold* pré-determinado. A estrutura dos algoritmos pode ser divida em três etapas:

- Construção do índice invertido: Constrói o índice invertido I a partir da inclusão de tokens dos novos vetores vistos;
- Geração de candidatos: Gera pares de candidatos entre o vetor que está sendo processado com todos os demais vetores incluídos no índice invertido. Pode conter falsos-positivos, a serem confirmados na fase de verificação;
- Verificação de candidatos: Finaliza o cálculo da similaridade dos pares candidatos, exclui os falso-positivos e retorna os pares semelhantes a partir do limite de similaridade pré-determinado.

Notação	Definição
D	Coleção de vetores de entrada
x	Representa um vetor da coleção D
x_j	Representa um token no vetor x
I	índice invertido
I_j	índice invertido associada ao token j
τ	Valor do *threshold*
C	Array de similaridade acumulada para pares candidatos
$C[y]$	Similaridade parcial para par de vetores candidatos (x, y'')
$b1, b3$	Pontuação de prefixo que captura um limite superior de similaridade
rw_x	Maior valor entre os tokens do vetor x
cw_j	Maior valor para o token j na coleção D
$\|x\|$	Tamanho ou número de não zeros no vetor x
$\|\|x\|\|$	Magnitude do vetor x
x'	Prefixo do vetor x, ou seja, elementos que não foram indexados
y'	Prefixo do vetor candidato y
y''	Sufixo do vetor candidato y (elementos que foram indexados)
rw'_x	Maior valor de x'
P	Conjunto de pares com similaridade $\geq \tau$. Equivale ao $\mathcal{A}_{sj}$, Def. 2.3
sz_1	Minsize: Tamanho mínimo para o vetor candidato
rs_1, rs_3, rs_4	Remscore: Limite superior de pontuação atingível
$ps[x]$	*Pscore*: Limite superior de similaridade para o vetor x
$\hat{cw}$	Mapa com maior valor de cada token j dentre os vetores em I
rw'_{xj}	Maior valor de x' até o token j

Tabela A.1: Notações utilizadas nos algoritmos.

Os vetores de entrada para ambos os algoritmos são normalizados para uma unidade de comprimento, ou seja $||x|| = 1$. Assim, como o cálculo da similaridade entre dois vetores é o produto interno entre eles (Definição A-1).

$$sim(x, y) = dot(x, y) = x \cdot y = \sum_{j=1}^{d} x_j \cdot y_j, \tag{A-1}$$

O que implica que o produto interno é igual a similaridade do cosseno (Definição A-2):

$$sim(x, y) = cos(x, y) \tag{A-2}$$

Nas seções A.1 e A.2 são detalhadas as otimizações em cada fase nos algoritmos AllPairs e L2AP respectivamente.

A.1 AllPairs

O algoritmo desenvolvido por Bayardo *et al.* trouxe novas estratégias para resolução da junção por similaridade. O mesmo se aproveita de uma ordem pré-definida no conjuntos de vetores para minimizar o tamanho do índice invertido e realizar podas de candidatos. Diversos métodos, desenvolvidos posteriormente para o problema de junção por similaridade, seguem a mesma estrutura do AllPairs.

O pseudo-código do AllPairs, dividido nas fases de indexação, geração de candidatos e verificação, é apresentado a seguir nos Algoritmos A.1, A.2 e A.3 e a discussão do mesmo nas seções correspondentes.

A.1.1 Indexação - AllPairs

Em um método simples de construção do índice invertido, são incluídos todos os tokens do vetor que está sendo indexado no índice invertido, para que os tokens dos próximos vetores sejam comparados com os do índice invertido.

Na indexação do AllPairs, este processo é otimizado através da redução do tamanho do índice invertido. É indexado somente alguns tokens de um vetor x, tokens estes que garantam que este vetor e todos os seus semelhantes y compartilhem pelo menos de um token comum. Devido o conjunto de vetores de entrada estarem em ordem decrescente do maior valor entre os tokens de cada vetor (rw_x), é possível manter um valor superior de similaridade atingível (*pscore*), armazenado na variável $b1$, combinando os primeiros tokens de x com qualquer outro vetor no conjunto de dados. Para calcular o valor do *pscore*, é utilizado além do rw_x, o maior valor do token no conjunto de dados (cw_j) (Linha 8).

Enquanto o *pscore* é menor que τ, os tokens processados podem ser desconsiderados da indexação sem o risco de perder algum par similar, pois a similaridade de x com qualquer outro vetor não poderá exceder τ. Somente a partir que o *pscore* ultrapassar τ, os demais tokens do vetor x são adicionados no índice invertido (Linha 10) e assim definido o prefixo x' (tokens não adicionados) e sufixo x'' (tokens adicionados no índice invertido) do vetor x (Linha 11).

O pseudo-código da fase de indexação, que também é o método que aciona as demais fases do AllPairs, pode ser consultado no algoritmo A.1.

Algoritmo A.1: Index-AllPairs

Entrada: Conjunto de vetores D e *threshold* τ;
Vetores de D em ordem decrescente de rw_x;
Colunas de D em ordem decrescente de frequência dos *tokens*.
Resultado: Conjunto de Pares P em D com similaridade $\geq \tau$

```
1  P ← ∅ // Inicializa resultado
2  I ← ∅ // Inicializa lista invertida
3  para cada x ∈ D faça
4      C ← CandGen-AllPairs(x, I, τ)
5      P ← P ∪ Verify-AllPairs(x, C, I, τ)
6      b1 ← 0
7      para cada j = 1...d, x_j > 0 ∈ x faça
8          b1 ← b1 + Min(cw_j, rw_x) · x_j
9          se b1 ≥ τ então
10             I_j ← I_j ∪ {(x, x_j)}
11             x_j ← 0
12 retorna P
```

A.1.2 Geração de Candidatos - AllPairs

Na etapa de geração de candidatos (Algoritmo A.2) é calculado a similaridade entre o vetor x e os seus candidatos, ou seja, com vetores y do índice invertido que compartilham dos mesmos tokens que x. Perceba que os vetores y possuem somente algumas características no índice invertido e por isso a similaridade com x é parcial, sendo finalizado o cálculo na fase de verificação. Nesta etapa, duas otimizações são incluídas pelo AllPairs a fim de reduzir o número de pares candidatos e eliminar falsos-positivos.

A primeira se refere ao limite de tamanho do vetor candidato, armazenado em sz_1. Desta forma pode-se eliminar os candidatos que, mesmo que compartilhem tokens em comum com o vetor x, não possuem quantidade suficiente de tokens diferentes de 0 para atingir τ (Linhas 2 e 5). Como D está em ordem decrescente de rw_x, é garantido que se y não atingir o limite de tamanho mínimo, não atingirá com próximos vetores e assim pode ser eliminado de I_j.

A segunda é o limite superior de similaridade possível entre o vetor x com qualquer vetor y presente em I. Esta informação é armazenada na variável rs_1, que inicialmente recebe o produto escalar entre os tokens do vetor x com o maior valor dos respectivos tokens na coleção D (Linha 3). A cada processamento dos itens de I, rs_1 é atualizado, retirando o valor do produto escalar do token corrente (Linha 9). Quando

rs_1 se torna menor que τ, temos que os próximos vetores y não conseguirão atingir a similaridade τ com x, e portanto podem ser ignorados (Linha 7). Ao final, temos o mapa C que contém os vetores candidatos de x e o respectivo valor da similaridade parcial.

Algoritmo A.2: CandGen-AllPairs

Entrada: Vetor x, Lista invertida I e *threshold* τ

Resultado: Valores de similaridade acumulada para pares de vetores candidatos C, onde $C = \{y \in D, C[y] > 0\}$

1 $C \leftarrow \varnothing$
2 $sz_1 \leftarrow \tau / rw_x$
3 $rs_1 \leftarrow \sum_{j=1}^{d} x_j \cdot cw_j$
4 **para cada** $j = d...1, x_j > 0 \in x$ **faça** `// Em ordem reversa`
5   Remover (y, y_j) de I_j quando $|y| < sz_1$
6   **para cada** $(y, y_j) \in I_j$ **faça**
7   **se** $C[y] > 0$ ***ou*** $rs_1 \geq \tau$ **então**
8             $C[y] \leftarrow C[y] + x_j \cdot y_j$
9     $rs_1 \leftarrow rs_1 - x_j \cdot cw_j$
10 **retorna** C

A.1.3 Verificação - AllPairs

Temos que $sim(x, y) = sim(x, y') + sim(x, y'')$. Assim, na geração de candidatos, foi calculada a similaridade de x com y'', ou seja, de x com a parte indexada de y, e o resultado armazenado em $C[y]$. Na etapa de Verificação (Algoritmo A.3) é calculado a similaridade final de x com seus candidatos, portanto, de x com a parte não indexada y'.

Nesta etapa uma otimização é realizada através da estimativa da similaridade de x com y (Linha 3). Esse limite chamado de *dpscore*, permite passar para próximo candidato se a estimativa ficar abaixo de τ, caso contrário é calculado a similaridade entre x e y' e adicionado a similaridade parcial para obter a similaridade final (Linha 4). Ao final, o par que ainda exceder τ é incluído no resultado (Linhas 5 e 6).

Algoritmo A.3: Verify-AllPairs

Entrada: Vetor x, Lista invertida I, vetores candidatos C e *threshold* τ

Resultado: Conjunto de pares similares $P = \{(x, y) \mid sim(x, y) \geq \tau$

```
1 P ← ∅
2 para cada y, C[y] > 0 faça
3     se C[y] + Min(|x|, |y'|) · rw_x · rw'_y ≥ τ então
4         s ← C[y] + calculaVerificacao(x, y')
5         se s ≥ τ então
6             P ← P ∪ {(x, y, s)}
7 retorna P
```

A.2 L2AP

O algoritmo L2AP se baseia na mesma arquitetura do AllPairs e através do princípio da desigualdade de Cauchy-Schwarz, que afirma que $sim(x, y) \leq ||x|| \cdot ||y||$, é obtido limites mais apertados. Como todos os vetores são normalizados para 1, temos que $||y|| \leq 1$, portanto o mesmo limite se aplica ao considerar o prefixo x' do vetor, desta forma: $sim(x', y) \leq ||x'|| \cdot ||y|| \leq ||x'||$, portanto $sim(x', y) \leq ||x'||$.

Com a aplicação destes novos limites, doravante nomeados de norma L2, torna-se possível a construção de um índice invertido menor e a eliminação de um maior número de candidatos. Outras otimizações incluídas no L2AP são discutidas nas Seções A.2.1, A.2.2 e A.2.3. O pseudo-código do L2AP, também dividido nas fases de indexação, geração de candidatos e verificação, é apresentado nos algoritmos A.4, A.5 e A.6.

A.2.1 Indexação - L2AP

Com base nos limites da norma L2, se $||x'|| < \tau$ significa que nenhum token em x' poderá levar a novos candidatos cuja $sim(x, y) \geq \tau$. Esse novo limite para o *pscore* possui, na maioria dos casos, valor mais restrito que o $b1$ do AllPairs, e assim consegue reduzir o tamanho do índice invertido. O novo limite do *pscore* ($||x'||$) é armazenado na variável $b3$ (linha 10).

A fim de obter o menor limite para o *pscore*, L2AP utiliza o mínimo entre o novo limite definido pela norma L2 e o anterior proposto por Bayardo *et al.* (Linha 11). Além disso, o *pscore* do vetor x é armazenado em um array ps para ser utilizado na fase de verificação (Linha 12).

A magnitude do vetor x até a posição j também é indexada no índice invertido (Linha 13). Esta informação será necessária nas fases de geração de candidatos e

verificação, para aplicar os filtros pela norma L2. Assim, no índice invertido terá a tripla (x, x_j, $||x'_j||$). O algoritmo A.4 apresenta o pseudo-código da fase de indexação do L2AP, que mantém a mesma estrutura do AllPairs com acréscimo das otimizações citadas anteriormente.

Algoritmo A.4: Index-L2AP

Entrada: Conjunto de vetores D e *threshold* τ;
Vetores de D em ordem decrescente de rw_x;
Colunas de D em ordem decrescente de frequência dos *tokens*.
Resultado: Conjunto de Pares P em D com similaridade $\geq \tau$

```
1  P ← ∅;  I ← ∅
2  ĉw_j ← 0, para j = 1...d
3  para cada x ∈ D faça
4  |  C ← CandGen-L2AP(x, I, ĉw, τ)
5  |  P ← P ∪ Verify-L2AP(x, ps, C, I, τ)
6  |  b1 ← 0;  bt ← 0;  b3 ← 0
7  |  para cada j = 1...d, x_j > 0 ∈ x faça
8  |  |  pscore ← Min(b1, b3)
9  |  |  b1 ← b1 + Min(cw_j, rw_x) · x_j
10 |  |  bt ← bt + x_j^2; b3 ← √bt
11 |  |  se Min(b1, b3) ≥ τ então
12 |  |  |  ps[x] ← pscore se ps[x] = 0
13 |  |  |  I_j ← I_j ∪ {(x, x_j, ||x'_j||)}
14 |  |  |  x_j ← 0
15 retorna P
```

A.2.2 Geração de Candidatos - L2AP

Assim como no AllPairs, a varredura do vetor x na fase de geração de candidatos do L2AP (Algoritmo A.5) é realizada em ordem reversa, ou seja, inversa à usada durante a indexação. A estrutura do algoritmo é similar ao AllPairs, com a inclusão das otimizações detalhadas a seguir.

Limite remscore rs_3: Uma alteração no limite de remscore rs_1 (valor de similaridade máxima atingível entre x e o candidato y presente em I) do AllPairs é realizada, alterando o cw_j, maior valor do token em D, por $\hat{cw}_j$, e por isso este novo remscore foi nomeado de rs_3 (Linha 3). O $\hat{cw}_j$ é o maior valor do token entre os tokens incluídos no índice invertido. Devido a similaridade de x, na geração de candidatos, ser calculada somente com vetores incluídos em I, obtém-se um valor mais restrito ao utilizar

o $\hat{cw}_j$ do que cw_j. O mapa $\hat{cw}$ que contém os tokens e respectivos valores é atualizado ao final do processamento do vetor atual de consulta x (Linha 14).

Algoritmo A.5: CandGen-L2AP

Entrada: Vetor x, Lista invertida I, Mapa $\hat{cw}$ e *threshold* τ

Resultado: Valores de similaridade acumulada para pares de vetores candidatos C, onde $C = \{y \in D, C[y] > 0\}$

```
1  C ← ∅
2  sz1 ← τ/rw_x
3  rs3 ← Σ_{j=1}^{d} x_j · ĉw_j
4  rs_t ← 1; rs4 ← 1
5  para cada j = d...1, x_j > 0 ∈ x faça // Em ordem reversa
6      Remover (y, y_j) de I_j quando |y| < sz1 // Filtro de tamanho
7      para cada (y, y_j, ||y'_j||) ∈ I_j faça
8          se C[y] > 0 ou Min(rs3, rs4) ≥ τ então
9              C[y] ← C[y] + x_j · y_j
10             se C[y] + ||x'_j|| · ||y'_j|| < τ então // Poda de candidatos pela
               norma L2
11                 C[y] ← 0 // Parar de acumular y e passar para próximo
                   candidato.
12     rs3 ← rs3 − x_j · ĉw_j
13     rs_t ← rs_t − x_j^2; rs4 ← √rs_t
14 ĉw_j ← Max(x_j, ĉw_j), ∀ x_j > 0
15 retorna C
```

Limite remscore rs_4: Um novo limite de remscore é criado a partir da norma L2. Como o maior valor de similaridade possível é 1, este novo limite, armazenado na variável rs_4, inicia-se com o valor 1 (Linha 4), sendo reduzido a cada iteração pelo valor da magnitude do vetor na posição $j-1$ ($||x_{j-1}||$) (Linha 13). Desta forma, é obtido um limite mais restrito e que elimina um maior número de candidatos falsos-positivos. Quando $||x_{j-1}||$ não atinge τ, pode-se ignorar o candidato y pois sabemos que o mesmo não atingirá a similaridade maior que τ com o vetor x.

O L2AP utiliza o mínimo entre $rs3$ e $rs4$ para o remscore (Linha 8), porém, conforme verificado em experimentos realizados durante a pesquisa, em 99,82% dos casos o $rs4$ se apresentou menor que $rs3$ e, desta forma, demonstra sua eficácia.

Poda de candidatos pela norma L2: Outra melhoria se refere a inclusão de uma verificação, para poda de candidatos, nesta fase. A partir da norma L2, é estimada a similaridade de x' com y' e adicionado a similaridade parcial já acumulada para verificar se conseguirá atingir τ (Linha 10). O valor de $||y'_j||$, incluído em I na fase de indexação é utilizado nesse momento. Caso a verificação não atinja ou ultrapasse o valor de τ, a acumulação de similaridade é interrompida e o próximo candidato é considerado.

A.2.3 Verificação - L2AP

Na última fase do L2AP (Algoritmo mostrado em A.6) é calculado a semelhança final de x com os vetores y candidatos. A estrutura segue a mesma do AllPairs, porém com a inclusão de mais filtros e podas de candidatos conforme relatado a seguir:

Filtro *pscore*: Na fase de indexação foi armazenado o limite *pscore*, valor superior de similaridade atingível para y' com qualquer outro vetor em D, ou seja $sim(*, y')$. Já na geração de candidatos, $C[y]$ armazena a similaridade parcial $sim(x, y'')$. Portanto, como na verificação é calculado $sim(x, y')$, caso $C[y] + ps[y]$ for menor que τ, o candidato pode ser seguramente descartado (Linha 4).

Filtro e poda *dpscore*: Novos limites de *dpscore* são incluídos para estimar a similaridade possível de ser atingida e caso não ultrapasse τ, o candidato poderá ser descartado. Para tanto, Anastasiu e Karypis se basearem em Awekar e Samatova [7] e também na ideia de filtragem posicional introduzida por Xiao [113] para desenvolver esses novos limites, bem como alterações no limite *dpscore* de Bayardo *et al*.

A aplicação dos limites *dpscore* é realizada antes de encontrar o primeiro token comum (Linha 6) e depois do encontrar o primeiro token comum entre o vetor x e y' (Linha 12). Das 8 equações da Tabela A.2 para o limite *dpscore*, segundo Anastasiu e Karypis, as equações 1 a 4 fornecem melhores limites, mas possuem maior custo computacional para cálculos e armazenamentos das somas de prefixo vetorial. Ao analisar resultados dos experimentos realizados durante a pesquisa, pode ser observado que, ao utilizar os demais filtros principalmente os da norma L2 na fase de geração de candidatos e verificação, o filtro e poda pelo *dpscore* possui baixa efetividade em relação ao tempo de execução.

Equações para o limite *dpscore*	
(1) $sim(x,y) \leq C[y] + min(rw_x \cdot \Sigma'_y, rw'_y \cdot \Sigma_x)$	(5) $sim(x,y) \leq C[y] + min(\|x\|, \|y'\|) \cdot rw_x \cdot rw'_y)$
(2) $sim(x,y) \leq C[y] + min(rw'_{x_p} \cdot \Sigma'_{y_p}, rw'_y \cdot \Sigma_x)$	(6) $sim(x,y) \leq C[y] + min(\|x\|, \|y'_p\|) \cdot rw'_{x_p} \cdot rw'_{y_p})$
(3) $sim(x,y) \leq C[y] + min(rw_x \cdot \Sigma'_{y_p}, rw'_y \cdot \Sigma'_{x_p})$	(7) $sim(x,y) \leq C[y] + min(\|x'_p\|, \|y'_p\|) \cdot rw_x \cdot rw'_{y_p})$
(4) $sim(x,y) \leq C[y] + min(rw'_{x_p} \cdot \Sigma'_{y_p}, rw'_y \cdot \Sigma'_{x_p})$	(8) $sim(x,y) \leq C[y] + min(\|x'_p\|, \|y'_p\|) \cdot rw'_{x_p} \cdot rw'_{y_p})$

Tabela A.2: Equações para o limite *dpscore*.

Algoritmo A.6: Verify-L2AP

Entrada: Vetor x, vetores candidatos C, Mapa de pscore ps e *threshold* τ

Resultado: Conjunto de pares similares $P = \{(x, y) \mid sim(x, y) \geq \tau$

1 $P \leftarrow \varnothing$
2 $rw'_{xj} \leftarrow Max(x'_j)$; $\Sigma'_{x_j} \leftarrow \sum_{i=1}^{j-1} x_i$
3 **para cada** $y,\ C[y] > 0$ **faça**
4   **se** $C[y] + ps[y] < \tau$ **então** `// Filtro pscore`
5   **continue**
6   **se** $C[y] + Min(rw_x \cdot \Sigma'_y,\ rw'_y \cdot \Sigma_x) < \tau$ **então**
7   **continue** `// Poda Dpscore, antes do primeiro token comum`
8   $s \leftarrow C[y]$
9   **para cada** $y_p \in y', y_p = x_p$ **faça** `// Calcular verificação`
10     $s \leftarrow s + x_p \cdot y_p$
11   **se** $p = 1$ **então** `// Verifica se primeiro token comum`
12   **se** $s + Min(rw'_{x_p} \cdot \Sigma'_{y_p},\ rw'_y \cdot \Sigma'_{x_p}) < \tau$ **então**
13   **break** `// Poda Dpscore, depois do primeiro token comum. Parar de acumular e passar para próximo candidato`
14   **senão**
15   **se** $s + ||x'_p|| \cdot ||y'_p|| < \tau$ **então** `// Poda com base no limite L2`
16   **break** `// Parar de acumular e passar para próximo candidato`
17   **se** $s \geq \tau$ **então**
18     $P \leftarrow P \cup \{(x, y, s)\}$
19 **retorna** P

Poda pela norma L2: A cada acumulação de similaridade que passar nas verificações anteriores, é aplicado a norma L2, conforme detalhado na seção A.2, que estima a similaridade de prefixo e assim podar candidatos que ainda não ultrapassam τ (Linha 15). Ao final de todas as verificações, os pares que ainda excederem τ, são incluídos no resultado (Linhas 17 e 18).

APÊNDICE B

Algoritmo HNSW

Este apêndice apresenta uma descrição do funcionamento do HNSW e seus cinco algoritmos específicos que desempenham papéis na construção e busca da estrutura.

Na Seção B.1, é descrito os algoritmos responsáveis pela construção do grafo. O algoritmo INSERT é responsável por inserir um novo elemento no grafo, estabelecendo conexões com os elementos existentes em diferentes camadas. Os algoritmos SELECT-NEIGHBORS-SIMPLE e SELECT-NEIGHBORS-HEURISTIC são utilizados para selecionar os vizinhos mais próximos de um elemento, levando em consideração diferentes critérios de seleção.

Na Seção B.2, é abordado o algoritmo de busca do HNSW. O algoritmo K-NN-SEARCH é responsável por encontrar os k elementos mais próximos de um elemento de busca no grafo. O algoritmo SEARCH-LAYER é utilizado pelo K-NN-SEARCH, bem como pelo INSERT, para buscar elementos em uma determinada camada do grafo. Esse algoritmo visita os elementos candidatos a partir de um ponto de entrada, expandindo-se para os vizinhos na camada específica.

Por fim, na Seção B.3 é apresentado uma breve análise da complexidade e consumo de memória do algoritmo.

B.1 Construção do Grafo

No método de construção do grafo HNSW, a inserção dos elementos ocorre de forma consecutiva e é realizada pelo INSERT (Algoritmo B.1). A cada elemento inserido, é selecionada aleatoriamente uma camada máxima na qual o elemento estará presente (parâmetro l). Essa seleção é feita com base em uma distribuição de probabilidade com decaimento exponencial, que é normalizada pelo parâmetro m_L (Linha 4). Nesse contexto, 'unif(0..1)' é uma função que gera números pseudoaleatórios no intervalo [0, 1), ou seja, um número aleatório uniformemente distribuído no intervalo semiaberto de 0 a 1.

Essa abordagem de seleção aleatória da camada máxima para cada elemento garante uma distribuição diversificada dos elementos ao longo das camadas do grafo.

Algoritmo B.1: INSERT

Entrada: Grafo multicamadas $hnsw$, novo elemento q, número de conexões estabelecidas M, número máximo de conexões para cada elemento por camada M_{max}, tamanho da lista dinâmica de candidatos $efConstruction$, fator de normalização para geração de nível m_L

Saída: atualização do $hnsw$ inserindo o elemento q

1. $W \leftarrow \varnothing$ `// lista dos elementos mais próximos atualmente encontrados`
2. $ep \leftarrow$ obter ponto de entrada do $hnsw$
3. $L \leftarrow$ nível do ep `// camada superior do` $hnsw$
4. $l \leftarrow \lfloor -\ln(\text{unif}(0..1)) \cdot m_L \rfloor$ `// nível do novo elemento`
5. **para** $l_c \leftarrow L...l+1$ **faça**
6. $\quad W \leftarrow$ SEARCH-LAYER$(q, ep, ef = 1, l_c)$
7. $\quad ep \leftarrow$ obter o elemento mais próximo em W de q
8. **para** $l_c \leftarrow \min(L, l) \ldots 0$ **faça**
9. $\quad W \leftarrow$ SEARCH-LAYER$(q, ep, efConstruction, l_c)$
10. $\quad neighbors \leftarrow$ SELECT-NEIGHBORS(q, W, M, l_c) `// alg. B.2 ou alg. B.3`
11. $\quad$ adicionar conexões bidirecionais de $neighbors$ para q na camada l_c
12. $\quad$ **para cada** $e \in neighbors$ **faça**
    `// reduza as conexões, se necessário`
13. $\quad\quad eConn \leftarrow vizinhos(e)$ na camada l_c
14. $\quad\quad$ **se** $|eConn| > M_{max}$ **então**
    `// reduzir conexões de` e
    `// se` $l_c = 0$ `então` $M_{max} = M_{max0}$
15. $\quad\quad\quad eNewConn \leftarrow$ SELECT-NEIGHBORS$(e, eConn, M_{max}, l_c)$ `// alg. B.2 ou alg. B.3`
16. $\quad\quad\quad$ definir $vizinhos(e)$ na camada l_c para $eNewConn$
17. $\quad ep \leftarrow W$
18. **se** $l > L$ **então**
19. $\quad$ definir ponto de entrada do $hnsw$ para q

A utilização de uma distribuição com decaimento exponencial implica que a maioria dos elementos estará presente em camadas inferiores, enquanto apenas alguns elementos estarão presentes em camadas mais altas.

Para reduzir a sobreposição entre vizinhos de diferentes camadas no grafo HNSW, é necessário diminuir o parâmetro m_L. No entanto, essa redução resulta em um aumento no número médio de saltos durante a busca em cada camada, o que pode afetar o desempenho do algoritmo. Portanto, é crucial encontrar um valor ótimo para m_L que equilibre a sobreposição entre vizinhos e o número de saltos. De acordo com Malkov e Yashunin, uma escolha simples para o valor de m_L é $\frac{1}{\ln(M)}$. Essa definição é amplamente adotada como padrão e foi utilizada nas implementações realizadas neste

trabalho de pesquisa.

A inserção de um novo elemento q começa na camada superior do grafo e percorre o grafo para encontrar os ef vizinhos mais próximos do elemento q na camada em questão. Depois disso, o algoritmo continua a busca da próxima camada usando os vizinhos mais próximos encontrados da camada anterior como pontos de entrada, e o processo se repete (Linhas 5-7). Os vizinhos mais próximos em cada camada são encontrados utilizando o algoritmo de busca SEARCH-LAYER (Linhas 6 e 9).

Para obter os ef vizinhos mais próximos em uma determinada camada l_c, que serão conectados ao elemento que está sendo inserido, o algoritmo HNSW utiliza uma lista dinâmica W de ef elementos encontrados mais próximos. Essa lista é inicialmente preenchida com os pontos de entrada do grafo e é atualizada a cada passo da busca.

Durante a primeira fase da busca, o parâmetro ef é definido como 1 (Linha 6). Em cada camada, o elemento mais próximo à consulta é encontrado e definido como o ponto de entrada para a próxima camada (Linha 7).

Quando a busca atinge a camada igual ou menor que l, inicia-se a segunda fase do algoritmo de construção (Linha 8). Nessa fase, o parâmetro ef recebe o valor de $efConstruction$, que controla o procedimento de recuperação da busca (Linha 9). Os vizinhos mais próximos encontrados em cada camada também são utilizados como candidatos para as conexões do elemento inserido (Linha 10).

Exceto na camada superior, o número total de conexões que cada elemento pode ter é definido pelo parâmetro M. Existem duas formas de selecionar os M vizinhos a partir dos candidatos (Linhas 10 e 15):

- Conexão simples aos elementos mais próximos (Algoritmo B.2).
- Heurística que leva em consideração as distâncias entre os elementos candidatos para criar conexões em diferentes direções (Algoritmo B.3).

As conexões são estabelecidas de forma bidirecional entre os vizinhos selecionados e o novo elemento na camada correspondente (Linha 11). Caso o número de conexões exceda o máximo permitido (M_{max}), é realizado um processo de redução de conexões para cada vizinho, mantendo apenas as M_{max} conexões mais relevantes (Linhas 14-16). O procedimento de inserção termina quando se realiza as conexões dos elementos na camada zero.

Após as conexões serem estabelecidas, o ponto de entrada é atualizado para os elementos encontrados (Linha 17). Ao final, se o nível do novo elemento for maior que o nível da camada superior, o ponto de entrada do HNSW é definido para o novo elemento (Linhas 18-19).

O algoritmo SELECT-NEIGHBORS-SIMPLE (Algoritmo B.2) adota uma abordagem simples e direta. Ele recebe como entrada o elemento base q e um conjunto de

Algoritmo B.2: SELECT-NEIGHBORS-SIMPLE

Entrada: elemento base q, elementos candidatos C, número de vizinhos para retornar M
Saída : M elementos mais próximos de q

```
1  retorna M elementos mais próximos em C de q
```

Algoritmo B.3: SELECT-NEIGHBORS-HEURISTIC

Entrada: elemento base q, candidatos C, número de vizinhos a retornar M, número da camada l_c, indicador se deve ou não estender a lista de candidatos $extendCandidates$, indicador de adição de elementos descartados $keepPrunedConnections$
Saída : M elementos selecionados pela heurística

```
1  R ← ∅ // conjunto de elementos selecionados
2  W ← C // fila de trabalho para os candidatos
3  se extendCandidates então
       // estender candidatos por seus vizinhos
4      para cada e ∈ C faça
5          para cada e_adj ∈ vizinhos(e) na camada l_c faça
6              se e_adj ∉ W então
7                  W ← W ∪ e_adj
8  W_d ← ∅ // fila para os candidatos descartados
9  enquanto |W| > 0 e |R| < M faça
10     e ← extrair o elemento mais próximo em W de q
11     se e está mais próximo de q em comparação com qualquer elemento em R
        então
12         R ← R ∪ e
13     senão
14         W_d ← W_d ∪ e
15 se keepPrunedConnections então
       // adicione algumas das conexões descartadas de W_d
16     enquanto |W_d| > 0 e |R| < M faça
17         R ← R ∪ extrair o elemento mais próximo de W_d para q
18 retorna R
```

candidatos C, e retorna os M elementos mais próximos de q em C. Essa abordagem não leva em consideração heurísticas adicionais ou estratégias de otimização.

Por outro lado, o algoritmo SELECT-NEIGHBORS-HEURISTIC (Algoritmo B.3) utiliza uma heurística mais sofisticada para criar conexões em diversas direções. Além dos parâmetros de entrada, ele também recebe o número da camada l_c, o indicador *extendCandidates* (definido como *false* por padrão) e o indicador *keepPrunedConnections*. O *extendCandidates* é útil apenas para dados extremamente agrupados. O *keepPru-*

nedConnections permite obter um número fixo de conexões por elemento. A heurística utilizada envolve a avaliação da proximidade dos candidatos em relação a q e a possibilidade de estender a lista de candidatos pelos vizinhos dos candidatos. Além disso, o algoritmo também considera a adição de elementos descartados, caso indicado. Desta forma ele também leva em consideração conexões de longo alcance no grafo, visando garantir a propriedade de navegação do mundo pequeno e melhorar a eficiência da busca. A figura a seguir ilustra essa heurística para selecionar vizinhos em dois *clusters* isolados.

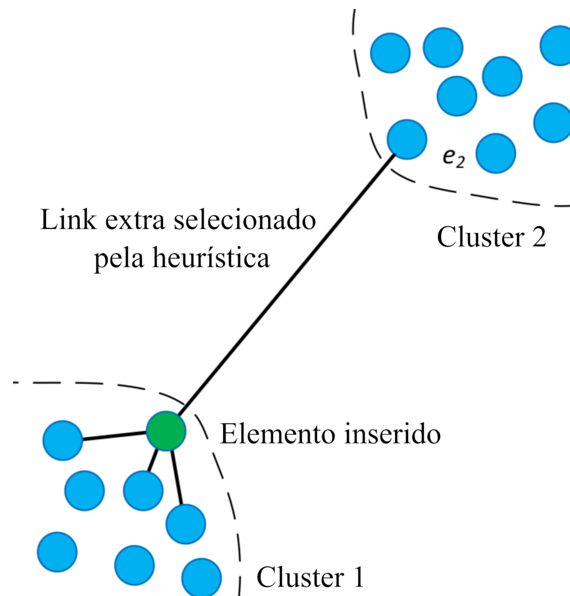


Figura B.1: Ilustração da heurística utilizada para selecionar os vizinhos do grafo HNSW para dois clusters isolados.

B.2 Busca no Grafo

A busca realizada pelo algoritmo é semelhante ao processo de inserção de um item na camada zero do grafo. No entanto, ao contrário da inserção, a busca retorna os vizinhos mais próximos como resultado, em vez de conectar o elemento buscado. A qualidade da busca é determinada pelo parâmetro ef (efSearch), que corresponde ao $efConstruction$ utilizado durante a etapa de construção do grafo. Aumentar o valor de ef resulta em um maior número de vizinhos retornados, o que pode melhorar a precisão dos resultados. No entanto, é importante notar que um valor maior de ef também pode levar a um aumento no tempo de execução da busca.

O algoritmo K-NN-SEARCH (Algoritmo B.4) inicia na camada mais alta do grafo e percorre cada camada até chegar à camada 0. O parâmetro ef controla o número de elementos mais próximos a serem retornados em cada camada. Nas camadas superiores, o valor de ef é definido como 1 (Linha 5), o que permite uma navegação mais rápida pelo

Algoritmo B.4: K-NN-SEARCH

Entrada: grafo multicamadas $hnsw$, elemento de busca q, número de vizinhos mais próximos para retornar K, tamanho da lista dinâmica de candidatos ef

Saída : K elementos mais próximos de q

```
1  W ← ∅ // conjunto para os elementos atuais mais próximos
2  ep ← obter nó de entrada do hnsw
3  L ← nível do ep // camada superior do hnsw
4  para l_c ← L ... 1 faça
5      W ← SEARCH-LAYER(q, ep, ef = 1, l_c)
6      ep ← obter o elemento mais próximo em W de q
7  W ← SEARCH-LAYER(q, ep, ef, l_c = 0)
8  retorna K elementos mais próximos em W de q
```

Algoritmo B.5: SEARCH-LAYER

Entrada: elemento de busca q, ponto de entrada ep, número de elementos mais próximo de q pra retornar ef, número da camada l_c

Saída : ef vizinhos mais próximos de q

```
1  v ← ep // conjunto de elementos visitados
2  C ← ep // conjunto de candidatos
3  W ← ep // lista dinâmica de vizinhos mais próximos encontrados
4  enquanto |C| > 0 faça
5      c ← extrair o elemento mais próximo em C de q
6      f ← obter o elemento mais distante em W de q
7      se distance(c, q) > distance(f, q) então
8          break // todos os elementos em W foram avaliados
9      para cada e ∈ vizinhos(c) na camada l_c faça
10         se e ∉ v então
11             v ← v ∪ e
12             f ← obter o elemento mais distante em W de q
13             se distance(e, q) < distance(f, q) ou |W| < ef então
14                 C ← C ∪ e
15                 W ← W ∪ e
16                 se |W| > ef então
17                     remova o elemento mais distante em W de q
18 retorna W
```

grafo nessas camadas. Durante esse percurso, é acionado o algoritmo SEARCH-LAYER para encontrar os elementos mais próximos em cada camada. A cada camada, o elemento mais próximo é definido como ponto de entrada para próxima camada (Linha 3). Ao alcançar a camada 0, o parâmetro ef recebe o valor definido na entrada do algoritmo, para encontrar os elementos mais próximos nessa camada (Linha 7). Ao final da busca, o

algoritmo retorna os k elementos mais próximos encontrados (Linha 8).

O algoritmo SEARCH-LAYER (Algoritmo B.5) realiza a busca em uma determinada camada do grafo. Ele mantém um conjunto de elementos visitados (v) e um conjunto de candidatos (C). Inicialmente, ambos são definidos como o ponto de entrada (ep) (Linhas 1-3). O algoritmo seleciona o elemento mais próximo em C em relação ao elemento de busca (q) e o elemento mais distante em W em relação a q (Linhas 5-6). Em seguida, percorre os vizinhos do elemento selecionado e verifica se eles já foram visitados (Linhas 9-10). Se um vizinho não foi visitado, ele é adicionado a v e avaliado em relação a q (Linhas 10-11). Se a distância do vizinho para q for menor que a distância do elemento mais distante em W para q ou o tamanho de W for menor que ef, o vizinho é adicionado a C e W (Linhas 13-15). Se o tamanho de W exceder ef, o elemento mais distante em W é removido (Linhas 16-17). O processo continua até que não haja mais candidatos em C ou é interrompido quando se encontra um mínimo local (Linha 7). Por fim, o algoritmo retorna a lista W contendo os ef elementos mais próximos encontrados (Linha 18).

B.3 Análise da Complexidade e Consumo de Memória

O HNSW possui uma complexidade de busca de $O(\log(N))$ ao usar grafos de Delaunay exatos. A média de passos para encontrar o elemento mais próximo em cada camada é constante, independente do tamanho do conjunto de dados. A adoção da aproximação do grafo de Delaunay no HNSW rompe com a suposição de um grafo exato, mas não afeta a escalabilidade do algoritmo de forma significativa. Contudo, Malkov e Yashunin destacam a necessidade de mais evidências para verificar se essa resiliência se mantém em espaços de dimensões superiores. A construção do grafo no HNSW é feita por meio de inserções iterativas de todos os elementos. Cada inserção envolve uma sequência de buscas *k*NN em diferentes camadas, seguidas do uso de uma heurística com complexidade fixa (definida por $efConstruction$). O número médio de camadas para adicionar um elemento é constante e depende de m_L. Portanto, a complexidade de inserção segue a mesma escala da busca, o que significa que o tempo de construção do grafo é $O(N \dot{\log}(N))$.

O consumo de memória do HNSW é determinado pelo armazenamento das conexões do grafo. O número de conexões por elemento é o parâmetro M_{max0} para a camada zero e o M_{max} para as outras camadas, ambos definidos a partir do parâmetro M. Portanto, o consumo médio de memória por elemento é dado pela Fórmula B-1.

$$(M_{max0} + m_L \cdot M_{max}) \cdot bc, \tag{B-1}$$

onde m_L representa o fator de normalização para definição da camada máxima do elemento e bc a quantidade de bytes por conexão