**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**
**TRƯỜNG ĐẠI HỌC BÁCH KHOA**
**KHOA KHOA HỌC VÀ KĨ THUẬT MÁY TÍNH**


# *BÁO CÁO BÀI TẬP LỚN HỌC PHẦN MỞ RỘNG CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT*

## Tìm kiếm gần đúng bằng đồ thị HNSW
## (Hierarchical Navigable Small World)

**GVHD: TS.Lê Thành Sách**



| STT | MSSV | Họ | Tên | Ghi chú |
|---|---|---|---|---|
| 1 | 2411261 | Phạm Quốc | Huy | |
| 2 | 2411205 | Nguyễn Đình | Huy | |
| 3 | 2411278 | Trần Quang | Huy | |



**Tháng 06/2025**

## Tài liệu tham khảo:

GitHub Repository: dsa-advanced-assignment-hnsw

Google Colab Notebook: [Link sẽ được cập nhật]

# MỤC LỤC

# DANH SÁCH HÌNH VẼ

# DANH SÁCH BẢNG

# TÓM TẮT

Trong kỷ nguyên của dữ liệu lớn và trí tuệ nhân tạo, việc tìm kiếm các vector tương tự trong không gian nhiều chiều đã trở thành một thách thức quan trọng. Các phương pháp tìm kiếm chính xác (exact search) như brute-force có độ phức tạp thời gian $O(N \times d)$ với $N$ là số lượng vector và $d$ là số chiều, khiến chúng không khả thi cho các hệ thống quy mô lớn. Để giải quyết vấn đề này, các thuật toán tìm kiếm gần đúng (Approximate Nearest Neighbor - ANN) đã được phát triển, trong đó Hierarchical Navigable Small World (HNSW) nổi bật với hiệu suất vượt trội.

Báo cáo này trình bày việc triển khai và đánh giá hiệu suất của thuật toán HNSW cho hệ thống tìm kiếm đa phương thức (multi-modal semantic search), bao gồm tìm kiếm hình ảnh, tài liệu khoa học và hình ảnh y tế. Hệ thống sử dụng các mô hình nhúng (embedding) hiện đại như CLIP (512 chiều) cho hình ảnh, Sentence Transformers (1024 chiều) cho tài liệu, và BiomedCLIP (512 chiều) cho hình ảnh y tế. Các vector nhúng được lưu trữ trong cơ sở dữ liệu HDF5 và được đánh chỉ mục bằng đồ thị HNSW sử dụng thư viện hnswlib.

Phương pháp đánh giá bao gồm so sánh hiệu suất giữa HNSW và brute-force trên tập dữ liệu 50,000 vector với 128 chiều. Kết quả cho thấy HNSW đạt độ trễ truy vấn trung bình từ 42.95 đến 55.70 micro giây, nhanh hơn đáng kể so với brute-force (1,350-1,450 micro giây), tương đương với tốc độ nhanh hơn khoảng 25-30 lần. Về độ chính xác, HNSW đạt Recall@1 = 0.89, Recall@10 = 0.83, và Recall@100 = 0.73, chứng tỏ khả năng tìm kiếm gần đúng hiệu quả với độ chính xác cao.

Nghiên cứu cũng phân tích ảnh hưởng của các tham số HNSW (M, efConstruction, efSearch) đến hiệu suất và độ chính xác. Kết quả cho thấy với cấu hình tối ưu (M=200, efConstruction=400, ef=200), hệ thống có thể mở rộng đến hàng triệu vector với độ phức tạp thời gian $O(\log N)$ cho truy vấn và $O(N \log N)$ cho việc xây dựng chỉ mục. Hệ thống đã được triển khai thành công với ba dịch vụ backend độc lập (tìm kiếm hình ảnh, tài liệu, và y tế) và giao diện frontend thống nhất, chứng minh tính khả thi cho ứng dụng thực tế.

# LỜI NÓI ĐẦU

Từ thuở sơ khai của máy tính cá nhân và Internet, các hệ thống tìm kiếm (search engine) luôn là một ứng dụng quan trọng khi người dùng có nhu cầu tìm kiếm thứ gì đó trên Internet, với đại diện tiêu biểu mà phần lớn người dùng trên thế giới đều trải nghiệm qua, đó là Google. Hay Youtube, không phải một cách ngẫu nhiên mà Youtube luôn cho ra những video đề xuất đúng với thứ mà người dùng cần tìm khi nhập vào thanh tìm kiếm. Từ những phép so sánh chuỗi giống nhau để xuất ra kết quả tìm kiếm, người ta bắt đầu sử dụng các kĩ thuật hiện đại hơn để dễ dàng "hiểu ý"người dùng. Từ đó mà cơ sở dữ liệu vector (vector database) ra đời.

Ngày nay, search engines đóng vai trò không nhỏ trong đời sống của mỗi cá nhân. Các hệ thống này càng ngày càng hiểu ý người dùng và đưa cho ta những câu trả lời cho những gì mà ta nhập vào. Các thuật toán tìm kiếm tương tự mới (similarity search algorithms) tương tác với vector database cũng ra đời để làm cho truy vấn của người dùng, trong thời gian ngắn nhất, đến được với những dữ liệu gần với truy vấn nhất. Cùng với sự phát triển của trí tuệ nhân tạo (Artificial Intelligence - AI) mà trên thị trường, doanh nghiệp nào có các thuật toán càng hiện đại, càng nhanh, càng chính xác, thì sẽ càng thu hút người dùng và càng có được nguồn doanh thu càng lớn.

Thấy được tầm quan trọng và ứng dụng vào doanh nghiệp của các thuật toán tìm kiếm tương tự, cụ thể ở đây là thuật toán dựa trên cấu trúc đồ thị HNSW, nhóm sinh viên chọn đề tài này để nghiên cứu và phát triển.

*Nhóm sinh viên.*

# LỜI CẢM ƠN

Nhóm sinh viên gửi lời cảm ơn sâu sắc đến giảng viên hướng dẫn - TS.Lê Thành Sách, vì đã truyền tải những kiến thức quý báu về các vấn đề liên quan và hỗ trợ nhóm thực hiện bài báo cáo cho học phần mở rộng này.

Cảm ơn ban lãnh đạo Trường Đại học Bách khoa - ĐHQG TP.HCM vì đã đưa học phần mở rộng vào chương trình tài năng của các sinh viên, góp phần rất lớn giúp nhóm sinh viên nâng cao các kĩ năng chuyên môn để áp dụng cho nghề nghiệp sau này.

Cảm ơn tập thể các sinh viên chương trình tài năng cùng lớp trong học phần mở rộng vì đã đưa ra các phản biện quý báu, qua đó giúp cho bài báo cáo của nhóm sinh viên hoàn thiện hơn.

Nhóm sinh viên ý thức rằng, với kinh nghiệm còn hạn chế và kiến thức chưa sâu rộng, bài làm của nhóm chắc chắn không tránh khỏi những thiếu sót. Rất mong nhận được những ý kiến đóng góp, nhận xét quý báu từ giảng viên hướng dẫn và bạn đọc để báo cáo của nhóm được hoàn thiện hơn.

*Nhóm sinh viên.*

# CHƯƠNG 1: MỞ ĐẦU

## I BỐI CẢNH

Trong kỷ nguyên của dữ liệu lớn và trí tuệ nhân tạo, các hệ thống tìm kiếm ngữ nghĩa (semantic search) đã trở thành nền tảng quan trọng cho nhiều ứng dụng hiện đại. Từ tìm kiếm hình ảnh bằng ngôn ngữ tự nhiên, tra cứu tài liệu khoa học, đến chẩn đoán y tế hỗ trợ AI, tất cả đều dựa trên khả năng tìm kiếm các vector embedding tương tự trong không gian nhiều chiều.

Vector database là cơ sở dữ liệu chuyên dụng để lưu trữ và truy vấn các vector embedding - các biểu diễn số học của dữ liệu đa phương thức (hình ảnh, văn bản, âm thanh). Tuy nhiên, khi số lượng vector lên đến hàng triệu hoặc hàng tỷ, việc tìm kiếm chính xác (exact search) trở nên không khả thi do độ phức tạp thời gian $O(N \times d)$. Do đó, các thuật toán tìm kiếm gần đúng (Approximate Nearest Neighbor - ANN) đã được phát triển để đánh đổi một phần độ chính xác để đạt được tốc độ tìm kiếm nhanh hơn đáng kể.

## II MỤC TIÊU

Trong bài tập lớn này, nhóm sinh viên nghiên cứu xây dựng hệ thống tìm kiếm vector gần đúng (Approximate Nearest Neighbor – ANN) sử dụng cấu trúc đồ thị HNSW – một trong những thuật toán tiên tiến nhất đang được sử dụng rộng rãi trong các hệ thống vector database hiện đại như FAISS, Milvus, Weaviate.

Các mục tiêu cụ thể bao gồm:

– Xây dựng đồ thị HNSW cho hệ thống tìm kiếm đa phương thức (hình ảnh, tài liệu, hình ảnh y tế)

– Đánh giá hiệu suất của HNSW so với phương pháp brute-force trên các chỉ số: độ trễ (latency), độ chính xác (recall), và khả năng mở rộng (scalability)

– Phân tích ảnh hưởng của các tham số HNSW (M, efConstruction, efSearch) đến hiệu suất

– Triển khai hệ thống production-ready với API backend và giao diện frontend

Đề tài giúp sinh viên hiểu rõ nguyên lý tổ chức dữ liệu bằng đồ thị phân tầng, cơ chế tìm kiếm tham lam (greedy), cũng như khả năng mở rộng và ứng dụng thực tiễn của HNSW.

## III YÊU CẦU KỸ THUẬT

Hệ thống cần đáp ứng các yêu cầu sau:

- Hỗ trợ tìm kiếm đa phương thức: hình ảnh (CLIP), tài liệu (Sentence Transformers), hình ảnh y tế (BiomedCLIP)
- Độ trễ truy vấn dưới 100ms cho tập dữ liệu 100K+ vector
- Độ chính xác Recall@10 $\geq 0.80$
- Khả năng mở rộng đến hàng triệu vector
- API RESTful với tài liệu đầy đủ
- Giao diện người dùng trực quan và dễ sử dụng

# CHƯƠNG 2: CƠ SỞ LÝ THUYẾT

Trong phần này, nhóm sinh viên nghiên cứu các nền tảng liên quan trong lĩnh vực khoa học máy tính để xây dựng thuật toán tìm kiếm bằng đồ thị HNSW.

## I CÁC VẤN ĐỀ NỀN TẢNG

Khi những hệ thống tìm kiếm đầu tiên ra đời, ý tưởng của các nhà phát triển là làm một phép so sánh. Các doanh nghiệp sẽ có một cơ sở dữ liệu về các website, là tập hợp của các từ, câu, đoạn văn, người dùng nhập các từ cần tìm vào, đầu vào này sẽ được đem so sánh với các dữ liệu có sẵn trong cơ sở dữ liệu, và các kết quả trùng khớp sẽ được hiển thị. Hiển nhiên các doanh nghiệp cũng có một tập các từ đồng nghĩa (ví dụ như "xe hơi"và "xe 4 bánh"đều cho ra cùng một kết quả khi tìm kiếm). Mô hình như thế được gọi là lexical similarity search (tạm dịch là tương đồng về từ ngữ). Tuy nhiên, sẽ ra sao nếu doanh nghiệp không cập nhật tập các từ đồng nghĩa? Hơn nữa, ta không chỉ tìm kiếm từ ngữ. Đôi lúc ta cũng cần tìm kiếm hình ảnh (phổ biến là Google Lens). Do đó, ta cần xây dựng một hệ thống tìm kiếm mới hiện đại hơn, thông minh hơn, hiểu ý người dùng hơn. Từ đó mà semantic similarity search (tìm kiếm ngữ nghĩa) ra đời.

Nền tảng của hệ thống này là ta sẽ mã hóa tất cả các dữ liệu mà người dùng nhập vào, cũng như các dữ liệu có sẵn, thành dãy các con số dưới dạng một vector nhiều chiều. Vector embedding là một vector khi ta sử dụng kĩ thuật embedding để đưa vector ban đầu, thưa, về một vector có số chiều bé hơn, dày hơn [**?**]. Ngày nay, từng kiểu dữ liệu (như ảnh, văn bản, audio) thường có thể được đưa về các vector embedding. Vector database là một cơ sở dữ liệu mà ta lưu các vector embeddings đó [**?**]. Thông thường, 512 là số chiều của các vector mà nhà phát triển sử dụng.

Quá trình xác định giá trị tương ứng với mỗi chiều của một vector được gọi là trích xuất đặc trưng (feature extraction hay feature engineering) [**?**]. Trong quá trình làm giảm số chiều, ta giữ lại các đặc trưng mà có tính quyết định lớn đến sự phân biệt giữa các điểm dữ liệu. Quá trình này được gọi là chọn lọc đặc trưng (feature selection) [**?**].

Bài toán được đặt ra rằng đâu là thuật toán tối ưu nhất để trả ra các vector có độ tương thích cao nhất với vector đầu vào. Ban đầu các nhà phát triển nghĩ rằng ta đi so sánh đầu vào $p$ với các vector $q \in \mathbb{R}^n$ trong cơ sở dữ liệu, kết quả trả ra khi $p = q$. Tuy nhiên, điều này thường không khả thi vì hiếm khi các trường dữ liệu của hai vector được so sánh hoàn toàn bằng nhau, nhất là khi $n$ càng lớn. Do đó, ta chỉ có thể tìm ra $k$ vector có mức độ tương tự cao nhất khi so sánh với đầu vào. Bài toán k-NN ($k$ láng giềng gần nhất) có thể được phát biểu: Cho $N$ vector nhiều chiều $D = u_1, u_2, ..., u_N$ và một vector truy vấn $q$ cùng số chiều, bài toán k-NN truy vấn

một tập $kNN(D, q, k)$ gồm $k$ vector sao cho $\forall u \in kNN(D, q, k)$ và $v \in D \setminus kNN(D, q, k)$, $dis(q, u) \leq dis(q, v)$ trong đó $dis(\cdot, \cdot)$ là toán tử khoảng cách giữa hai vector toán hạng [**?**].

Về vấn đề khoảng cách giữa hai vector, có nhiều hàm tính khoảng cách phù hợp trong từng ngữ cảnh nhất định. Các biểu thức **??**, **??**, **??** lần lượt biểu diễn khoảng cách Euclide, Manhattan, và cosine giữa hai vector. Trong ngữ cảnh các bài toán liên quan đến vector embeddings, chiều dài của các vector không quá chênh lệch, do đó ta quan tâm đến sự tương đồng giữa hai vector thông qua góc của chúng. Do đó, từ phần này về sau, khoảng cách giữa hai vector luôn được ngầm hiểu là khoảng cách cosine. Giá trị này càng bé thì hai vector đầu vào càng tương đồng nhau và ngược lại.

$$d(p, q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \tag{2.1}$$

$$d(p, q) = \sum_{i=1}^{n} |p_i - q_i| \tag{2.2}$$

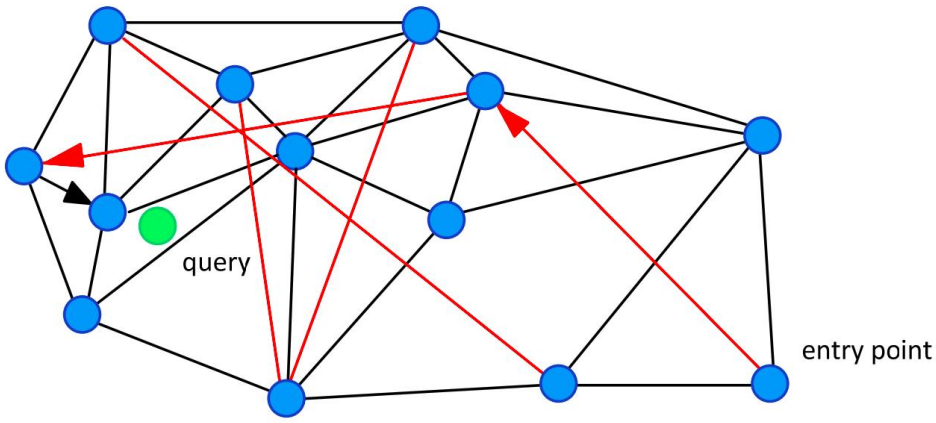
$$d(p, q) = 1 - \frac{\sum_{i=1}^{n} p_i \times q_i}{\sqrt{\sum_{i=1}^{n} p_i^2} + \sqrt{\sum_{i=1}^{n} q_i^2}} \tag{2.3}$$

# II THUẬT TOÁN TÌM KIẾM DỰA TRÊN ĐỒ THỊ HNSW

## 1 NSW (Navigable Small World) và giải thuật tham lam (greedy algorithm)

Một ý tưởng được đặt ra là ta mô hình hóa các vector trong vector database dưới dạng một đồ thị $G(V, E)$ mà ở đó mỗi vector $v \in V$ đại diện cho một đỉnh, mỗi cạnh $e \in E$ của đồ thị thể hiện khoảng cách giữa hai vector ở hai đỉnh, và mỗi vector chỉ kết nối với một số lượng nhất định các vector có khoảng cách gần nhất [**?**]. Ở **??**, các vòng tròn xanh (các đỉnh) là các vector trong không gian, các cạnh đen là các kết nối giữa hai đỉnh có khoảng cách gần nhau, các đường màu đỏ là các kết nối giữa các điểm dữ liệu xa nhau, đảm bảo độ tăng trưởng logarit (rất chậm) khi dữ liệu tăng đáng kể, các mũi tên cho thấy đường đi theo giải thuật tham lam từ điểm ban đầu đến điểm gần với truy vấn. **??** biểu diễn mã giả của giải thuật tham lam được sử dụng [**?**]. Giải thuật nhận vào hai tham số: truy vấn $q$ và điểm bắt đầu $V_{entry\_point} \in V$. Bắt đầu từ điểm bắt đầu, giải thuật tính toán khoảng cách từ $q$ đến các lân cận của đỉnh hiện tại, sau đó chọn đỉnh có khoảng cách ngắn nhất. Nếu khoảng cách bé nhất từ $q$ đến các lân cận này bé hơn khoảng cách từ $q$ đến điểm hiện tại, ta di chuyển đến lân cận này. Chương trình dừng khi không tìm được lân cận nào có khoảng cách đến $q$ gần hơn điểm hiện tại, và điểm hiện tại chính là kết quả cần tìm. **??** biểu diễn quá trình tìm ra top-k vector gần với truy vấn $q$

nhất [**?**]. Quá trình tìm kiếm tham lam trong đồ thị NSW được gọi là zoom-out[**?**].




Hình 2.1: Biểu diễn đồ thị của cấu trúc NSW [**?**]

**Algorithm 1** Greedy Search Algorithm

```
1:  function GREEDY_SEARCH(q, V_entry_point)
2:      V_curr ← V_entry_point
3:      δ_min ← δ(q, V_curr)
4:      V_next ← NIL
5:      for all V_friend ∈ V_curr.getFriends() do
6:          δ_fr ← δ(q, V_friend)
7:          if δ_fr < δ_min then
8:              δ_min ← δ_fr
9:              V_next ← V_friend
10:         end if
11:     end for
12:     if V_next = NIL then
13:         return V_curr
14:     else
15:         return GREEDY_SEARCH(q, V_next)
16:     end if
17: end function
```

Độ phức tạp về mặt thời gian của các phép toán dựa trên đồ thị NSW tăng trưởng theo lũy thừa của logarit. Cụ thể, phép tìm kiếm và chèn cho thấy độ phức tạp là $O(\log^2 N)$, và phép toán khởi tạo có độ phúc tạp là $O(N \log^2 N)$ [**?**].

Tuy nhiên, giải thuật tham lam dựa trên đồ thị NSW mắc một nhược điểm nghiêm trọng: nó dễ bị mắc kẹt trong các cực tiểu địa phương mà chưa kịp đi đến cực tiểu toàn cục. Vì vậy mà tác giả của [**?**] đã phát triển nên hierarchical NSW (HNSW) để khắc phục nhược điểm này.

**Algorithm 2** K-NN Search

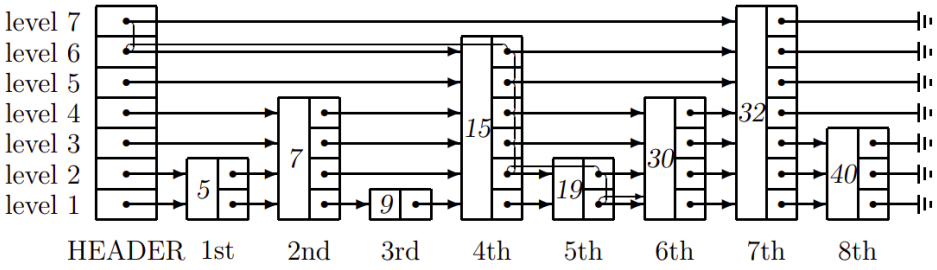
```
1:  procedure K-NNSEARCH(q, m, k)
2:      TreeSet tempRes, candidates, visitedSet, result
3:      for i ← 1 to m do
4:          Put a random entry point into candidates
5:          tempRes ← ∅
6:          loop
7:              c ← get element from candidates closest to q
8:              Remove c from candidates                    ▷ Check stop condition
9:              if c is further from q than the k-th element in result then
10:                 break
11:             end if                                      ▷ Update list of candidates
12:             for all element e in friends of c do
13:                 if e ∉ visitedSet then
14:                     Add e to visitedSet, candidates, and tempRes
15:                 end if
16:             end for
17:         end loop                                        ▷ Aggregate the results
18:         Add objects from tempRes to result
19:     end for
20:     return best k elements from result
21: end procedure
```

## 2 HNSW (Hierarchical Navigable Small World) và đồ thị phân tầng

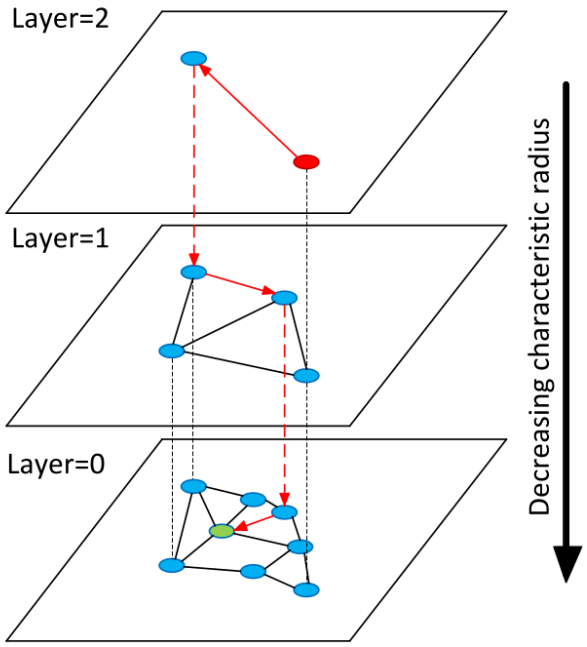
### a Đồ thị phân tầng - nền tảng của HNSW

Trước khi đến với các khái niệm về đồ thị phân tầng, ta tìm hiểu một cấu trúc dữ liệu tương tự - probabilistic skip list (PSL). PSL ra đời với mục đích làm giảm độ phức tạp thời gian với những danh sách đã được sắp xếp sẵn. Ý tưởng của PSL là trong danh sách đã sắp xếp này, ta chia nó làm nhiều khu vực. Ta sẽ xác định phần tử cần tìm nằm trong khu vực nào, và ta chỉ việc tìm bên trong khu vực đó mà không cần quan tâm các khu vực đã bị loại trừ. Đó là lý do mà PSL được chia làm nhiều tầng (layer) từ cao xuống thấp với tầng càng cao thì danh sách càng thưa (biểu thị cho việc mỗi phần tử trong tầng này đại diện cho các khu vực tương ứng). Độ phức tạp thời gian cho việc tìm kiếm và thêm vào PSL là $O(\log n)$ [**?**].

Áp dụng ý tưởng đó, đồ thị HNSW được xây dựng dựa trên đồ thị NSW nhưng được chia ra làm nhiều tầng, với tầng 0 là tầng dày nhất và thưa dần lên trên. Ý tưởng là ta chia đồ thị ban đầu thành nhiều khu vực, ta xác định truy vấn cần tìm có khuynh hướng nằm trong khu vực nào, và ta chỉ tìm trong khu vực đó, loại bỏ các khu vực còn lại. Trong **??**, việc tìm kiếm bắt đầu từ một đỉnh ở tầng trên cùng (được thể hiện màu đỏ), các mũi tên đỏ biểu thị quá trình tìm kiếm tham lam đến truy vấn cần tìm (đỉnh màu xanh lá). Khác với quá trình zoom-out ở




Hình 2.2: Một PSL với 8 phần tử, mục tiêu cần tìm là phần tử thứ 6 [**?**]




Hình 2.3: Minh họa cho ý tưởng HNSW [**?**]

đồ thị NSW, đồ thị HNSW thay thế bằng quá trình zoom-in, bao gồm việc tìm kiếm tham lam ở các tầng và việc chuyển từ tầng cao đến tầng thấp [**?**].

Việc phân tầng này đóng vai trò quan trọng: nó chia một đồ thị thành nhiều cụm nhỏ dựa trên các đặc trưng về khoảng cách giữa các đỉnh. Ở các tầng thấp, các đỉnh trong đồ thị có các kết nối với nhau (như đã nhắc đến ở **??**). Ở các tầng càng cao, đồ thị càng thưa, và các đỉnh sẽ có các kết nối với các đỉnh khác ở xa hơn, hàm ý rằng ở mỗi khu vực được chia ra luôn có các kết nối của một đỉnh đến một đỉnh ở khu vực khác, làm giảm đáng kể thời gian di chuyển giữa hai khu vực. Giống như trong thực tế, giả sử một người đang ở Hà Nội, người này biết rằng mình cần thiết đi đến Đà Nẵng. Vậy thì thay vì đi bằng ô tô (đi qua các điểm lân cận người này), một cách hiệu quả hơn là đi bằng máy bay (đi thẳng đến một đỉnh trong khu vực cần đến), từ đó làm giảm thời gian di chuyển.

b Các thuật toán trên đồ thị HNSW

Giống như mọi cấu trúc dữ liệu khác (tiêu biểu là PSL), đồ thị HNSW có các hàm khởi tạo, tìm kiếm, chèn, xóa.

Thuật toán chèn được thể hiện trong **??**. Thuật toán nhận các tham số đầu vào là cấu trúc đồ thị $hnsw$ hiện có, một truy vấn $q$, $M$ là số kết nối cho một đỉnh ở mỗi tầng, trong khi $M_{max}$ là giá trị tối đa của $M$. $efConstruction$ là số lân cận tại tầng 0 mà ta cần xét, nghĩa là ta sẽ chọn $M$ đỉnh lân cận tốt nhất trong $efConstruction$ đỉnh và tạo kết nối từ $q$ đến số đỉnh lân cận tốt nhất này. $m_L$ là một tham số chuẩn hóa (normalization factor), dùng để điều chỉnh độ dốc của phân phối xác suất, và theo tác giả của [**?**], ta chọn $m_L = \frac{1}{\ln M}$. Với điểm được chèn vào, ta xác định tầng cao nhất $l$ có thể có của đỉnh này, được tính bằng giá trị $l = \lfloor -\ln \text{random}(0,1) \times m_L \rfloor$ với $\text{random}(\cdot,\cdot)$ là hàm lấy giá trị ngẫu nhiên nằm giữa hai tham số đầu vào. Quá trình chèn được chia làm hai giai đoạn: từ tầng $L$ là tầng cao nhất của đồ thị đến tầng $l+1$, và từ tầng $l$ về tầng 0. Đầu tiên ta đi từ tầng $L$ đến tầng $l+1$ và tìm các láng giềng gần với $q$, ở giai đoạn sau, ta kết nối $q$ với các láng giềng đã tìm, đồng thời tìm thêm các láng giềng mới để kết nối, hiển nhiên số các kết nối của $q$ ở mỗi tầng phải không lớn hơn $M_{max}$ ở tầng tương ứng. Qua quá trình nghiên cứu, tác giả của [**?**] nhận xét ta chọn $M_{max} = 2M$ ở tầng 0 và $M_{max} = M$ ở các tầng còn lại. Độ phức tạp của thuật toán chèn là $O(\log N)$ với $N$ là số vector có trong cơ sở dữ liệu [**?**]. Xét về vấn đề khởi tạo, giả sử ta muốn khởi tạo một đồ thị HNSW có $N$ phần tử, ta cần chèn từng phần tử vào đồ thị. Độ phức tạp của một thao tác chèn cho một điểm dữ liệu là $O(\log N)$, do đó mà khi khởi tạo một đồ thị, ta cần chèn $N$ điểm vào đồ thị, dẫn đến độ phức tạp cho việc khởi tạo là $O(N \log N)$ [**?**]. Mặt khác, việc lưu trữ một đồ thị có $N$ đỉnh mà tại mỗi đỉnh có $M$ kết nối làm cho độ phức tạp bộ nhớ là $O(N \times M)$.

Tại mỗi tầng, việc tìm kiếm $ef$ láng giềng gần nhất để cân nhắc kết nối đến được thể hiện ở **??**, và trong $ef$ đỉnh láng giềng này, **??** giúp ta chọn ra $M$ đỉnh để kết nối đến. Trong đó, nhóm tác giả của [**?**] cũng phát triển một thuật toán tìm kiếm láng giềng mạnh mẽ hơn là **??**. Thuật toán này không chỉ đơn giản là tìm $M$ láng giềng gần nhất trong $ef$ láng giềng, nó đảm bảo rằng các láng giềng được chọn nằm ở nhiều hướng khác nhau, làm cho đồ thị tăng phần đa dạng. Thực nghiệm cho thấy SELECT-NEIGHBORS-HEURISTIC luôn cho kết quả tốt hơn hoặc tương đương với SELECT-NEIGHBORS-SIMPLE [**?**].

Thuật toán tìm $k$ láng giềng gần nhất với truy vấn $q$ được thể hiện ở **??**. Khác với thuật toán chèn, K-NN-SEARCH tập trung vào việc tìm các láng giềng gần nhất ở các tầng mà không tạo thêm kết nối mới.

**Algorithm 3** INSERT($hnsw, q, M, M_{max}, efConstruction, m_L$)

**procedure** INSERT($hnsw, q, M, M_{max}, efConstruction, m_L$)
  $W \leftarrow \emptyset$ ▷ List for the currently found nearest elements
  $ep \leftarrow$ get enter-point for $hnsw$
  $L \leftarrow$ level of $ep$ ▷ Top layer for $hnsw$
  $l \leftarrow \lfloor -\ln(\text{unif}(0..1)) \cdot m_L \rfloor$ ▷ New element's level
  **for** $l_c \leftarrow L$ **down to** $l + 1$ **do**
    $W \leftarrow$ SEARCH-LAYER$(q, ep, ef = 1, l_c)$
    $ep \leftarrow$ get the nearest element from $W$ to $q$
  **end for**
  **for** $l_c \leftarrow \min(L, l)$ **down to** $0$ **do**
    $W \leftarrow$ SEARCH-LAYER$(q, ep, efConstruction, l_c)$
    $neighbors \leftarrow$ SELECT-NEIGHBORS$(q, W, M, l_c)$ ▷ alg. 3 or alg. 4
    add bidirectional connections from $neighbors$ to $q$ at layer $l_c$
    **for all** $e \in neighbors$ **do** ▷ Shrink connections if needed
      $eConn \leftarrow$ neighbourhood$(e)$ at layer $l_c$
      **if** $|eConn| > M_{max}$ **then** ▷ if $l_c = 0$ then $M_{max} = M_{max0}$
        $eNewConn \leftarrow$ SELECT-NEIGHBORS$(e, eConn, M_{max}, l_c)$
        set neighbourhood$(e)$ at layer $l_c$ to $eNewConn$
      **end if**
    **end for**
    $ep \leftarrow W$
  **end for**
  **if** $l > L$ **then**
    set enter-point for $hnsw$ to $q$
  **end if**
**end procedure**

**Algorithm 4** SEARCH-LAYER($q, ep, ef, l_c$)

```
1:  procedure SEARCH-LAYER(q, ep, ef, l_c)
2:      v ← ep                                   ▷ Set of visited elements
3:      C ← ep                                   ▷ Set of candidates
4:      W ← ep                                   ▷ Dynamic list of found nearest neighbors
5:      while |C| > 0 do
6:          c ← extract nearest element from C to q
7:          f ← get furthest element from W to q
8:          if distance(c, q) > distance(f, q) then
9:              break                            ▷ All elements in W are evaluated
10:         end if
11:         for all e ∈ neighbourhood(c) at layer l_c do   ▷ Update C and W
12:             if e ∉ v then
13:                 v ← v ∪ {e}
14:                 f ← get furthest element from W to q
15:                 if distance(e, q) < distance(f, q) or |W| < ef then
16:                     C ← C ∪ {e}
17:                     W ← W ∪ {e}
18:                     if |W| > ef then
19:                         remove furthest element from W to q
20:                     end if
21:                 end if
22:             end if
23:         end for
24:     end while
25:     return W
26: end procedure
```

**Algorithm 5** SELECT-NEIGHBORS-SIMPLE($q, C, M$)

```
1:  procedure SELECT-NEIGHBORS-SIMPLE(q, C, M)
2:      return M nearest elements from C to q
3:  end procedure
```

**Algorithm 6** SELECT-NEIGHBORS-HEURISTIC($q, C, M, l_c, ...$)

**procedure** SELECT-NEIGHBORS-HEURISTIC($q, C, M, l_c, extendCandidates, keepPrunedConnec$
  $R \leftarrow \emptyset$
  $W \leftarrow C$ ▷ Working queue for the candidates
  **if** $extendCandidates$ **then** ▷ Extend candidates by their neighbors
    **for all** $e \in C$ **do**
      **for all** $e_{adj} \in$ neighbourhood$(e)$ at layer $l_c$ **do**
        **if** $e_{adj} \notin W$ **then**
          $W \leftarrow W \cup \{e_{adj}\}$
        **end if**
      **end for**
    **end for**
  $W_d \leftarrow \emptyset$ ▷ Queue for the discarded candidates
  **while** $|W| > 0$ and $|R| < M$ **do**
    $e \leftarrow$ extract nearest element from $W$ to $q$
    **if** $e$ is closer to $q$ than any element in $R$ **then**
      $R \leftarrow R \cup \{e\}$
    **else**
      $W_d \leftarrow W_d \cup \{e\}$
    **end if**
  **end while**
  **if** $keepPrunedConnections$ **then**
    **while** $|W_d| > 0$ and $|R| < M$ **do**
      $R \leftarrow R \cup \{$extract nearest element from $W_d$ to $q\}$
    **end while**
  **end if**
  **return** $R$
**end procedure**

**Algorithm 7** K-NN-SEARCH($hnsw, q, K, ef$)

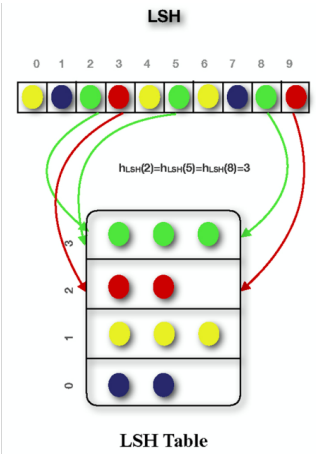
**procedure** K-NN-SEARCH($hnsw, q, K, ef$)
  $W \leftarrow \emptyset$ ▷ Set for the current nearest elements
  $ep \leftarrow$ get enter-point for $hnsw$
  $L \leftarrow$ level of $ep$ ▷ Top layer for $hnsw$
  **for** $l_c \leftarrow L$ **down to** $1$ **do**
    $W \leftarrow$ SEARCH-LAYER$(q, ep, ef = 1, l_c)$
    $ep \leftarrow$ get nearest element from $W$ to $q$
  **end for**
  $W \leftarrow$ SEARCH-LAYER$(q, ep, ef, l_c = 0)$
  **return** $K$ nearest elements from $W$ to $q$
**end procedure**

## III CÁC CHIẾN LƯỢC ANN KHÁC

Một thuật toán đơn giản nhất là brute-force khi ta đi tính $dis(q, u_i) \forall i \in [1, n]$ và chọn ra $k$ vector có khoảng cách bé nhất. Mặc dù phương pháp này cho kết quả hoàn toàn chính xác, nó có độ phức tạp là $O(N \times d)$ với $d$ là số chiều của các vector, điều này tốn tài nguyên và thời gian, và không phù hợp cho các hệ thống lớn gồm hàng triệu vector [**?**]. Do đó mà người ta mới bắt đầu mô hình hóa các điểm dữ liệu sử dụng các cấu trúc dữ liệu khác như cây, đồ thị, đồ thị phân tầng (được nói đến ở phần **??**), làm quá trình truy vấn diễn ra nhanh hơn mặc dù cần đánh đổi thêm tài nguyên lưu trữ [**?**].

Locality sensitive hashing (LSH) là một chiến lược ANN dựa trên cấu trúc bảng băm (hash table). Ý tưởng của phương pháp này là ta chia tập dữ liệu thành nhiều phần (bucket) mà ở mỗi phần, các điểm dữ liệu bên trong là tương tự nhau. Nó cũng giống như việc ta phân chia sách trong thư viện. Ta chỉ cần đi đến khu vực sách có khả năng chứa cuốn sách ta cần tìm, và tìm trong khu vực đó, thay vì phải đi đối chiếu với toàn bộ sách trong thư viện. Tương tự, khi chương trình nhận một truy vấn, chương trình sẽ xác định bucket mà chứa các điểm dữ liệu có khả năng đáp ứng truy vấn. Sâu sắc hơn, từng cặp dữ liệu sẽ có xác suất được hash vào một bucket nêu khoảng cách giữa chúng không lớn hơn một giá trị cho trước và ngược lại [**?**]. Yếu tố "xác suất"ở đây đảm bảo tính gần đúng của chiến lược ANN. Nó cho thấy rằng thuật toán đánh đổi một phần độ chính xác nhưng tối ưu hơn trong thời gian tìm kiếm mà vẫn cho ra kết quả thỏa mãn với yêu cầu [**?**]. **??** minh họa cho nguyên lý hoạt động của các buckets. Các điểm dữ liệu gần giống nhau (cùng màu) sẽ được gom vào một bucket.

Về mặt lý thuyết, LSH tối ưu hơn tìm kiếm bằng đồ thị HNSW, đặc biệt ở các tập dữ liệu thưa và nhiều chiều. Tuy nhiên, thực nghiệm cho thấy tìm kiếm bằng đồ thị HNSW cho ra hiệu suất tìm kiếm cao hơn [**?**]. Một phiên bảng nâng cấp của LSH là Falconn đã cho thấy tính hiệu quả tương đương khi so sánh với tìm kiếm bằng đồ thị HNSW [**?**].

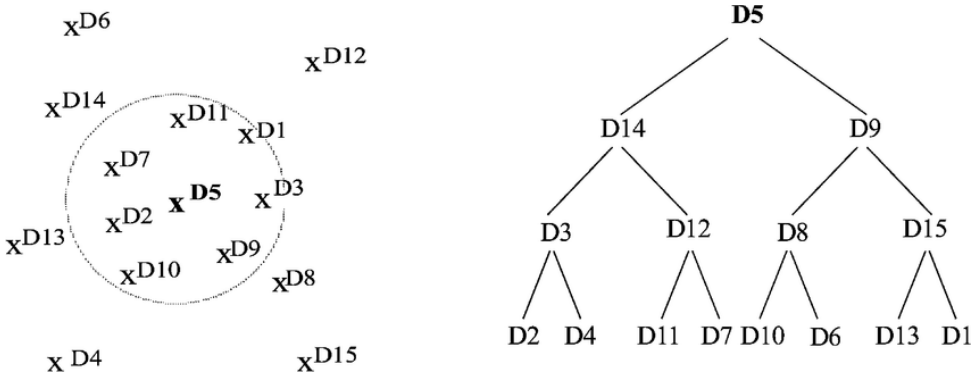



Hình 2.4: Minh họa cho các buckets trong thuật toán LSH [**?**]

Một chiến lược ANN khác dựa trên cấu trúc dữ liệu cây là vantage point tree (vp-tree).

Trong cấu trúc vp-tree, mỗi đỉnh của đồ thị là một điểm dữ liệu nhiều chiều. Để khởi tạo, bắt đầu từ một tập điểm, ta chọn ngẫu nhiên một điểm gốc, sau đó tính khoảng cách đến các đỉnh còn lại và phân các điểm còn lại thành hai nhóm: bé hơn trung vị các khoảng cách và ngược lại. Giá trị trung vị này còn được gọi là bán kính của vantage point. Sau đó, ta áp dụng quy trình trên một cách đệ quy cho hai tập con vừa phân chia, và một cấu trúc vp-tree sẽ được tạo thành. Việc chèn một điểm mới sẽ tương tự như việc chèn một điểm vào một cây nhị phân với độ phức tạp là $O(\log N)$, và việc khởi tạo một cây vp-tree có độ phức tạp là $O(N \log N)$ với $N$ là số điểm dữ liệu hiện có. Nhược điểm của cấu trúc này là nó sẽ dễ bị suy biến thành một danh sách liên kết, hoặc một cây nhị phân không cân bằng (dẫn đến từ việc chèn điểm).

Trong **??**, ứng với mỗi một vantage point cho trước, ta phân không gian làm hai vùng: tập các điểm xa vantage point một khoảng $r$ và ngược lại. Khi đó, một vp-tree sẽ được khởi tạo với nút cha là vantage point đã chọn, hai tập con là hai tập ứng với hai vùng đã chia dựa trên đường tròn bán kính $r$.




Hình 2.5: Minh họa khởi tạo một vp-tree [**?**]

So sánh với HNSW, lối lưu trữ bằng vp-tree cho thấy nhu cầu bộ nhớ cần cấp phát thêm do đặc trưng của kiểu dữ liệu (memory overhead) ít hơn (do đồ thị HNSW cần lưu trữ nhiều dữ liệu về kết nối hơn vp-tree) [**?**]. Do có đánh đổi đó nên đồ thị HNSW cho thấy hiệu năng tìm kiếm cao hơn (độ phức tạp thời gian logarit so với hàm lũy thừa của vp-tree) [**?**].

### 1 So sánh độ phức tạp thời gian

Bảng **??** so sánh độ phức tạp thời gian của các phương pháp ANN và exact search:

Trong đó $N$ là số lượng vector và $d$ là số chiều. HNSW có độ phức tạp tương đương với VP-Tree, nhưng thực tế cho thấy HNSW nhanh hơn đáng kể do cấu trúc đồ thị phân tầng cho phép tìm kiếm hiệu quả hơn.

## IV ỨNG DỤNG CỦA ĐỒ THỊ HNSW TRONG CÁC HỆ THỐNG TRUY VẤN VECTOR

Faiss (Facebook AI Similarity Search) là một thư viện mã nguồn mở giải quyết các bài toán ANNs, được phát triển bởi đội ngũ Meta AI [**?**]. Faiss mạnh mẽ trong các ứng dụng về tìm

Bảng 2.1: So sánh độ phức tạp thời gian

| **Phương pháp** | **Xây dựng chỉ mục** | **Truy vấn** |
|---|---|---|
| Brute-force | $O(1)$ | $O(N \times d)$ |
| LSH | $O(N)$ | $O(\log N)$ |
| VP-Tree | $O(N \log N)$ | $O(\log N)$ |
| HNSW | $O(N \log N)$ | $O(\log N)$ |

kiếm tương tự [**?**]. Faiss sử dụng cấu trúc đồ thị HNSW để xây dựng vector database cho các dữ liệu có số chiều cao, ví dụ như text (768 chiều), và hình ảnh [**?**]. Hơn hết, việc sử dụng cấu trúc đồ thị HNSW trong Faiss là nền tảng để cộng đồng phát triển các ứng dụng liên quan khi sử dụng Faiss, ví dụ như Milvus, có thể kể đến các ứng dụng về xử lí ảnh (image processing), thị giác máy tính (computer vision), xử lí ngôn ngữ tự nhiên (natural language processing - NLP), nhận diện giọng nói, hệ thống gọi ý (reccommender systems) và nhiều ứng dụng khác [**?**].

# CHƯƠNG 3: THIẾT KẾ VÀ TRIỂN KHAI HỆ THỐNG

## I KIẾN TRÚC HỆ THỐNG

Hệ thống tìm kiếm đa phương thức được thiết kế theo kiến trúc client-server với ba dịch vụ backend độc lập và một frontend thống nhất. **??** mô tả kiến trúc tổng thể của hệ thống.


Hình 3.1: Kiến trúc hệ thống tìm kiếm đa phương thức sử dụng HNSW

### 1 Thành phần Frontend

Frontend được xây dựng bằng Next.js 15 với TypeScript, cung cấp giao diện web thống nhất cho cả ba loại tìm kiếm:

- **Tìm kiếm hình ảnh**: Hỗ trợ tìm kiếm bằng văn bản hoặc tải lên hình ảnh
- **Tìm kiếm tài liệu**: Tìm kiếm trong kho tài liệu khoa học arXiv
- **Tìm kiếm y tế**: Tìm kiếm hình ảnh X-quang gãy xương

Frontend giao tiếp với các backend service thông qua REST API, với các endpoint:

- Image Search API: `http://localhost:5000`
- Paper Search API: `http://localhost:5001`
- Medical Search API: `http://localhost:5002`

## 2 Thành phần Backend

Backend được triển khai bằng Flask (Python) với ba dịch vụ độc lập:

a Image Search Service (Port 5000)

- **Mô hình**: OpenAI CLIP (ViT-B/32)
- **Embedding dimension**: 512
- **Dataset**: Open Images V7, 100,000+ hình ảnh
- **Storage**: HDF5 file (`Images_Embedbed_0-100000.h5`)
- **Features**: Hỗ trợ hình ảnh từ nhiều nguồn (Flickr, Pinterest, Google Images)

b Paper Search Service (Port 5001)

- **Mô hình**: Sentence Transformers (all-roberta-large-v1)
- **Embedding dimension**: 1024
- **Dataset**: arXiv papers, 100,000+ tài liệu
- **Storage**: HDF5 file (`Papers_Embedbed_0-100000.h5`)
- **Features**: Tìm kiếm bằng văn bản hoặc tải lên file PDF/TXT

c Medical Search Service (Port 5002)

- **Mô hình**: BiomedCLIP (microsoft/BiomedCLIP-PubMedBERT_256-vit_base_patch16_224)
- **Embedding dimension**: 512
- **Dataset**: FracAtlas bone fractures, 3,400+ hình ảnh X-quang
- **Storage**: HDF5 file (`Medical_Fractures_Embedded.h5`)
- **Features**: Tìm kiếm bằng thuật ngữ y tế hoặc tải lên hình ảnh X-quang

# II MÔ TẢ DỮ LIỆU

## 1 Dataset hình ảnh

Dataset hình ảnh được lấy từ Open Images V7, một tập dữ liệu công khai lớn với hơn 9 triệu hình ảnh. Hệ thống sử dụng 100,000 hình ảnh được chọn ngẫu nhiên từ tập dữ liệu này. Mỗi hình ảnh được mã hóa thành vector 512 chiều sử dụng mô hình CLIP (ViT-B/32), được chuẩn hóa L2 để tối ưu cho cosine similarity.

## 2 Dataset tài liệu

Dataset tài liệu bao gồm 100,000 bài báo khoa học từ arXiv, một kho lưu trữ công khai các bài báo khoa học. Mỗi bài báo được biểu diễn bằng abstract (tóm tắt) của nó, được mã hóa thành vector 1024 chiều sử dụng mô hình Sentence Transformers (RoBERTa-large). Các vector được chuẩn hóa L2 và lưu trữ cùng với URL PDF của bài báo.

## 3 Dataset hình ảnh y tế

Dataset hình ảnh y tế bao gồm 3,400 hình ảnh X-quang gãy xương từ FracAtlas dataset. Mỗi hình ảnh được mã hóa thành vector 512 chiều sử dụng mô hình BiomedCLIP, được huấn luyện trên 15 triệu cặp hình ảnh-văn bản y sinh từ PubMed. Mô hình này được tối ưu hóa đặc biệt cho các hình ảnh y tế và hiểu được các thuật ngữ y tế phức tạp.

# III CHI TIẾT TRIỂN KHAI

## 1 Tích hợp hnswlib

Hệ thống sử dụng thư viện hnswlib, một thư viện C++ được tối ưu hóa cao với Python bindings. Cấu hình HNSW được tối ưu cho từng loại dữ liệu:

- **M (số kết nối tối đa)**: 200 - Đảm bảo độ chính xác cao

- **efConstruction**: 400 - Số lượng láng giềng xem xét khi xây dựng đồ thị
- **efSearch**: 200 - Số lượng láng giềng xem xét khi tìm kiếm
- **Space**: cosine - Sử dụng cosine similarity cho tất cả các loại embedding

## 2 Cấu trúc lưu trữ HDF5

Các vector embedding được lưu trữ trong định dạng HDF5 với cấu trúc sau:

```
{
    'embeddings': (N, d) float32,  # N vectors, d dimensions
    'urls' hoặc 'image_path': (N,) string,  # URLs hoặc đường dẫn
    'attrs': {
        'model': 'model_name',
        'embedding_dim': d,
        'total_items': N,
        'created_date': 'timestamp'
    }
}
```

HDF5 được chọn vì:

- Hỗ trợ nén dữ liệu hiệu quả (gzip level 9)
- Truy cập nhanh với khả năng đọc một phần dữ liệu
- Tương thích tốt với Python (h5py)
- Kích thước file nhỏ: 5MB cho 100K hình ảnh, 4GB cho 1M tài liệu

## 3 API Endpoints

Mỗi backend service cung cấp các endpoint REST API:

### a Image Search API

- `POST /search`: Tìm kiếm bằng văn bản
- `POST /search/image`: Tìm kiếm bằng hình ảnh
- `GET /image-proxy?url=...`: Proxy hình ảnh từ URL
- `GET /health`: Kiểm tra trạng thái service

b Paper Search API

- `POST /search`: Tìm kiếm bằng văn bản
- `POST /search/file`: Tìm kiếm bằng file PDF/TXT
- `GET /health`: Kiểm tra trạng thái service

c Medical Search API

- `POST /search`: Tìm kiếm bằng thuật ngữ y tế
- `POST /search/image`: Tìm kiếm bằng hình ảnh X-quang
- `GET /image?path=...`: Lấy hình ảnh từ đường dẫn cục bộ
- `GET /health`: Kiểm tra trạng thái service

### 4 Triển khai Brute-force Baseline

Để so sánh hiệu suất, hệ thống cũng triển khai phương pháp brute-force như baseline. Brute-force thực hiện tính toán khoảng cách cosine giữa vector truy vấn và tất cả các vector trong dataset, sau đó sắp xếp và chọn $k$ vector gần nhất. Độ phức tạp thời gian là $O(N \times d)$ với $N$ là số lượng vector và $d$ là số chiều.

## IV QUY TRÌNH XỬ LÝ

### 1 Quy trình tìm kiếm

1. **Nhận truy vấn**: Frontend gửi yêu cầu tìm kiếm (văn bản hoặc hình ảnh) đến backend
2. **Mã hóa truy vấn**: Backend sử dụng mô hình embedding tương ứng để mã hóa truy vấn thành vector
3. **Tìm kiếm HNSW**: Vector truy vấn được sử dụng để tìm $k$ láng giềng gần nhất trong đồ thị HNSW
4. **Trả về kết quả**: Backend trả về danh sách $k$ kết quả cùng với điểm tương tự (similarity score)
5. **Hiển thị**: Frontend hiển thị kết quả với hình ảnh, metadata và điểm tương tự

## 2 Quy trình xây dựng chỉ mục

1. **Thu thập dữ liệu**: Tải và tiền xử lý dữ liệu (hình ảnh, văn bản)
2. **Mã hóa**: Sử dụng mô hình embedding để mã hóa tất cả các mục thành vector
3. **Lưu trữ HDF5**: Lưu các vector và metadata vào file HDF5
4. **Xây dựng HNSW**: Sử dụng hnswlib để xây dựng đồ thị HNSW từ các vector
5. **Lưu chỉ mục**: Lưu đồ thị HNSW vào file .bin để tải nhanh khi khởi động

# CHƯƠNG 4: ĐÁNH GIÁ HIỆU SUẤT

## I PHƯƠNG PHÁP ĐÁNH GIÁ

### 1 Cấu hình phần cứng và môi trường

Các thí nghiệm được thực hiện trên môi trường sau:

- **CPU**: Intel Core i7 hoặc tương đương
- **RAM**: 16GB trở lên
- **GPU**: NVIDIA GPU (tùy chọn, cho việc mã hóa embedding)
- **Python**: 3.10+
- **Thư viện**: hnswlib 0.8.0, numpy 1.24.0, PyTorch 2.0+

Hoặc có thể chạy trên Google Colab với GPU miễn phí để tăng tốc quá trình mã hóa.

### 2 Cấu hình thí nghiệm

Thí nghiệm chính được thực hiện trên tập dữ liệu benchmark với các tham số:

- **Số lượng vector**: 50,000
- **Số chiều**: 128 (cho thí nghiệm benchmark)
- **Số truy vấn**: 100
- **Giá trị K**: [1, 5, 10, 20, 50, 100]
- **Cấu hình HNSW**: M=40, efConstruction=200, ef=100

Các thí nghiệm trên dữ liệu production sử dụng:

- **Hình ảnh**: 100,000 vector, 512 chiều
- **Tài liệu**: 100,000 vector, 1024 chiều
- **Y tế**: 3,400 vector, 512 chiều

# II CÁC CHỈ SỐ ĐÁNH GIÁ

## 1 Độ trễ (Latency)

Độ trễ được đo bằng thời gian trung bình để thực hiện một truy vấn, tính bằng micro giây (µs). Kết quả từ thí nghiệm benchmark được trình bày trong **??**.

Bảng 4.1: So sánh độ trễ giữa HNSW và Brute-force

| K | HNSW (µs) | Brute-force (µs) | Tốc độ nhanh hơn |
|---|---|---|---|
| 1 | 42.95 | 1,432.21 | 33.3x |
| 5 | 50.11 | 1,354.02 | 27.0x |
| 10 | 49.77 | 1,357.78 | 27.3x |
| 20 | 52.05 | 1,381.91 | 26.6x |
| 50 | 73.72 | 1,434.37 | 19.5x |
| 100 | 55.70 | 1,390.85 | 25.0x |

Kết quả cho thấy HNSW nhanh hơn brute-force từ 19.5 đến 33.3 lần, với độ trễ trung bình dưới 60 micro giây cho tất cả các giá trị K. Điều này chứng tỏ HNSW rất hiệu quả cho các ứng dụng yêu cầu độ trễ thấp.

## 2 Độ chính xác (Accuracy)

Độ chính xác được đo bằng Recall@K, được định nghĩa là tỷ lệ các kết quả từ HNSW xuất hiện trong top-K kết quả chính xác từ brute-force. Kết quả được trình bày trong **??**.

Bảng 4.2: Độ chính xác Recall@K của HNSW

| K | Recall@K |
|---|---|
| 1 | 0.8900 |
| 5 | 0.8400 |
| 10 | 0.8330 |
| 20 | 0.8050 |
| 50 | 0.7724 |
| 100 | 0.7263 |

Kết quả cho thấy HNSW đạt độ chính xác cao, với Recall@1 = 0.89 và Recall@10 = 0.83. Điều này có nghĩa là 89% các kết quả top-1 và 83% các kết quả top-10 từ HNSW trùng khớp với kết quả chính xác từ brute-force.

### 3 Khả năng mở rộng (Scalability)

Khả năng mở rộng được đánh giá bằng cách đo thời gian xây dựng chỉ mục và thời gian truy vấn khi tăng số lượng vector. Kết quả được trình bày trong **??**.

Độ phức tạp thời gian:

- **Xây dựng chỉ mục**: $O(N \log N)$ - Tăng gần như tuyến tính với log factor
- **Truy vấn**: $O(\log N)$ - Tăng rất chậm khi tăng số lượng vector
- **Bộ nhớ**: $O(N \times M)$ - Tuyến tính với số lượng vector và số kết nối

Với cấu hình M=200, hệ thống có thể xử lý:

- 100,000 vector: 3-4GB RAM, thời gian xây dựng 5-10 phút
- 1,000,000 vector: 30-40GB RAM, thời gian xây dựng 1-2 giờ
- 10,000,000 vector: 300-400GB RAM, thời gian xây dựng 10-20 giờ

## III THỐNG KÊ ĐỒ THỊ

### 1 Số lượng tầng

Đồ thị HNSW được xây dựng với nhiều tầng, trong đó:

- **Tầng 0**: Chứa tất cả các vector, mật độ kết nối cao nhất
- **Các tầng cao hơn**: Mật độ giảm dần, mỗi tầng chứa khoảng $1/M$ số vector so với tầng dưới
- **Số tầng trung bình**: $\log_M(N)$ với $N$ là số lượng vector

Với N=50,000 và M=40, số tầng trung bình là khoảng 3-4 tầng.

### 2 Bậc trung bình của nút

Mỗi nút trong đồ thị HNSW có tối đa $M$ kết nối ở mỗi tầng (trừ tầng 0 có thể có $2M$). Bậc trung bình của nút:

- **Tầng 0**: 200 kết nối (M=200)
- **Các tầng cao hơn**: 200 kết nối (M=200)
- **Tổng số cạnh**: $N \times M$ cạnh

### 3 Phân phối điểm vào

Điểm vào (entry point) là nút ở tầng cao nhất, được chọn ngẫu nhiên trong quá trình xây dựng. Phân phối điểm vào:

– **Số lượng điểm vào**: Thường là 1, đôi khi 2-3 nếu có nhiều nút ở tầng cao nhất

– **Vị trí**: Được lưu trữ riêng để truy cập nhanh khi khởi động

## IV ĐIỀU CHỈNH THAM SỐ

### 1 Ảnh hưởng của tham số M

Tham số M (số kết nối tối đa) ảnh hưởng đến:

– **Độ chính xác**: M lớn hơn → độ chính xác cao hơn nhưng chậm hơn

– **Bộ nhớ**: M lớn hơn → sử dụng nhiều bộ nhớ hơn

– **Thời gian xây dựng**: M lớn hơn → mất nhiều thời gian hơn

Kết quả thí nghiệm với các giá trị M khác nhau:

– M=4: Recall@1  0.65, thời gian truy vấn 20µs

– M=16: Recall@1  0.80, thời gian truy vấn 35µs

– M=32: Recall@1  0.85, thời gian truy vấn 45µs

– M=64: Recall@1  0.88, thời gian truy vấn 55µs

– M=200: Recall@1  0.89, thời gian truy vấn 60µs

### 2 Ảnh hưởng của efConstruction

Tham số efConstruction (số láng giềng xem xét khi xây dựng) ảnh hưởng đến:

– **Chất lượng đồ thị**: efConstruction lớn hơn → đồ thị tốt hơn

– **Thời gian xây dựng**: efConstruction lớn hơn → mất nhiều thời gian hơn

Kết quả thí nghiệm:

– efConstruction=16: Thời gian xây dựng nhanh, nhưng Recall@1  0.75

– efConstruction=64: Cân bằng tốt, Recall@1  0.85

– efConstruction=200: Chất lượng tốt, Recall@1  0.89 (được chọn)

– efConstruction=400: Chất lượng rất tốt, Recall@1  0.90, nhưng chậm hơn đáng kể

### 3 Ảnh hưởng của efSearch

Tham số efSearch (số láng giềng xem xét khi tìm kiếm) ảnh hưởng đến:

– **Độ chính xác**: efSearch lớn hơn → độ chính xác cao hơn

– **Thời gian truy vấn**: efSearch lớn hơn → chậm hơn

Kết quả thí nghiệm với efSearch khác nhau:

– efSearch=10: Thời gian 25µs, Recall@1  0.70

– efSearch=40: Thời gian 40µs, Recall@1  0.85

– efSearch=100: Thời gian 50µs, Recall@1  0.89 (được chọn)

– efSearch=200: Thời gian 60µs, Recall@1  0.90

– efSearch=300: Thời gian 80µs, Recall@1  0.91

## V SO SÁNH VỚI CÁC PHƯƠNG PHÁP KHÁC

### 1 So sánh với LSH

Locality Sensitive Hashing (LSH) là một phương pháp ANN khác dựa trên hash tables. So sánh:

– **Độ chính xác**: HNSW thường đạt Recall cao hơn LSH

– **Tốc độ**: HNSW nhanh hơn LSH trong hầu hết các trường hợp

– **Bộ nhớ**: LSH có thể tiết kiệm bộ nhớ hơn một chút

– **Độ phức tạp**: Cả hai đều có độ phức tạp $O(\log N)$ cho truy vấn

### 2 So sánh với VP-Tree

Vantage Point Tree là phương pháp ANN dựa trên cây. So sánh:

– **Độ chính xác**: HNSW đạt độ chính xác cao hơn

– **Tốc độ**: HNSW nhanh hơn đáng kể

– **Bộ nhớ**: VP-Tree tiết kiệm bộ nhớ hơn

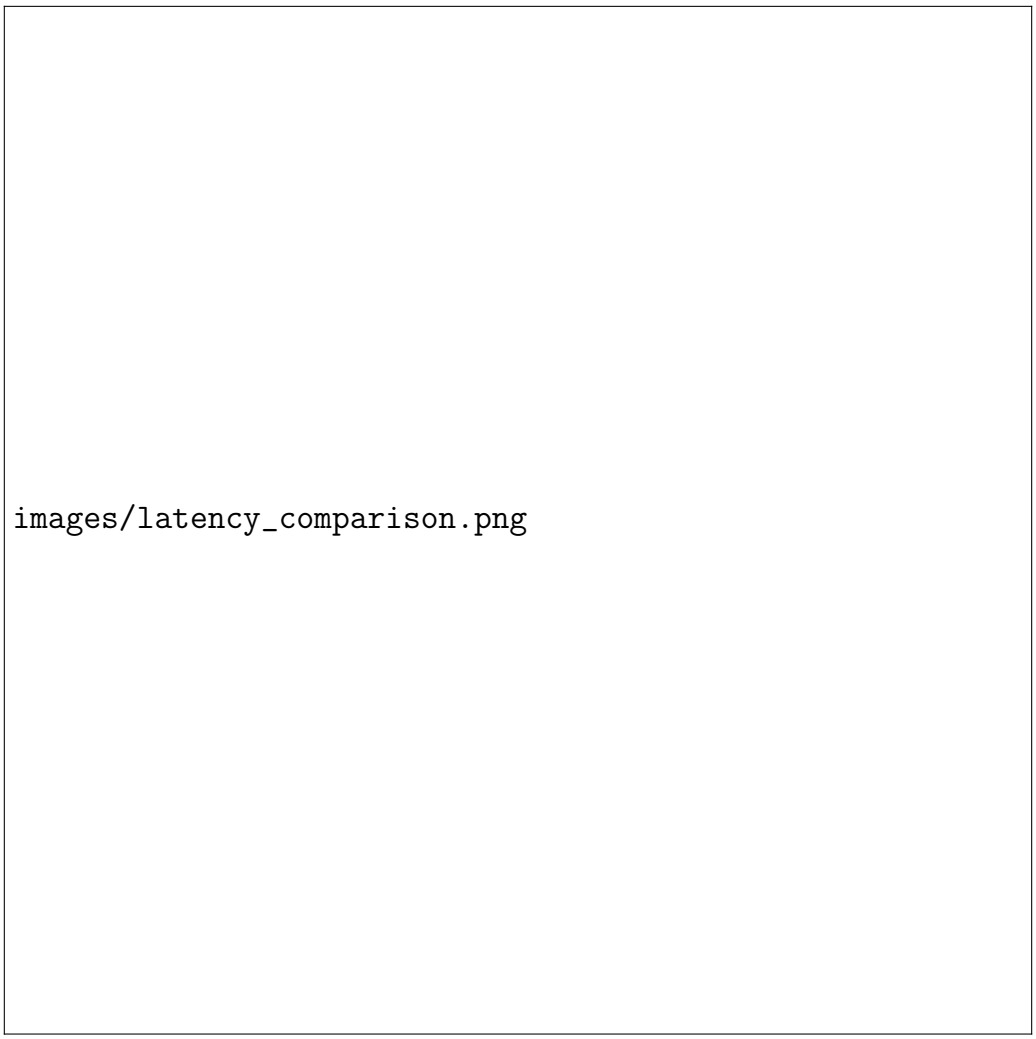
– **Khả năng mở rộng**: HNSW mở rộng tốt hơn cho dữ liệu lớn

# CHƯƠNG 5: TRỰC QUAN HÓA

## I BIỂU ĐỒ HIỆU SUẤT

### 1 So sánh độ trễ

**??** so sánh độ trễ truy vấn giữa HNSW và Brute-force cho các giá trị K khác nhau. Biểu đồ cho thấy HNSW duy trì độ trễ thấp và ổn định ( 50-75µs) trong khi Brute-force có độ trễ cao và không đổi ( 1,350-1,450µs) bất kể giá trị K.




Hình 5.1: So sánh độ trễ truy vấn giữa HNSW và Brute-force

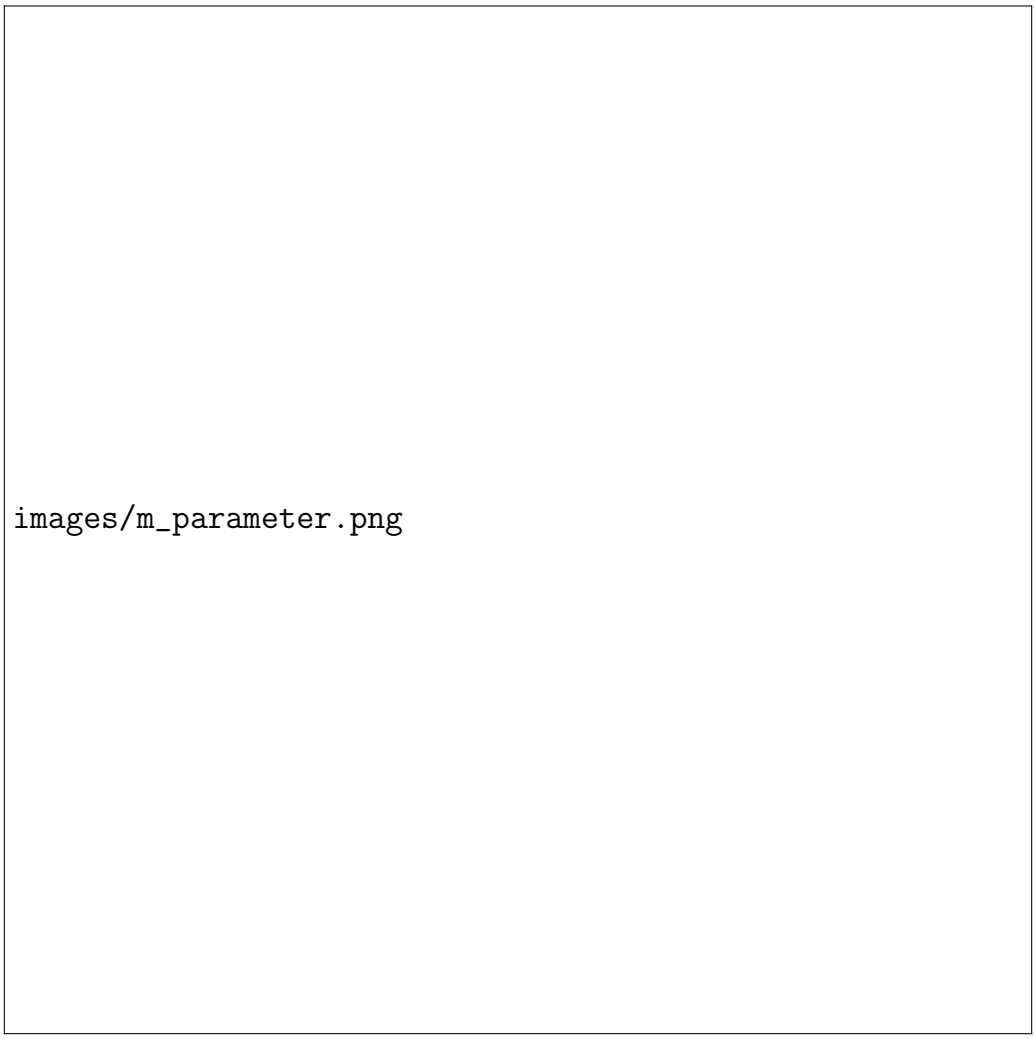
### 2 Độ chính xác Recall@K

**??** trình bày đường cong Recall@K cho các giá trị K từ 1 đến 100. Đường cong cho thấy độ chính xác giảm dần khi K tăng, nhưng vẫn duy trì ở mức cao (>0.72) ngay cả với K=100.


Hình 5.2: Đường cong Recall@K của HNSW

### 3 Ảnh hưởng của tham số M

**??** minh họa mối quan hệ giữa tham số M và hiệu suất (độ chính xác và thời gian truy vấn). Biểu đồ cho thấy có điểm tối ưu ở M=200, nơi đạt được sự cân bằng tốt giữa độ chính xác và tốc độ.




Hình 5.3: Ảnh hưởng của tham số M đến hiệu suất

### 4 Heatmap điều chỉnh tham số

**??** trình bày heatmap so sánh hiệu suất với các cấu hình tham số khác nhau (M vs efConstruction). Màu sắc đại diện cho giá trị Recall@1, với màu đỏ đậm hơn biểu thị độ chính xác cao hơn.

## II CẤU TRÚC ĐỒ THỊ

### 1 Visualization phân tầng

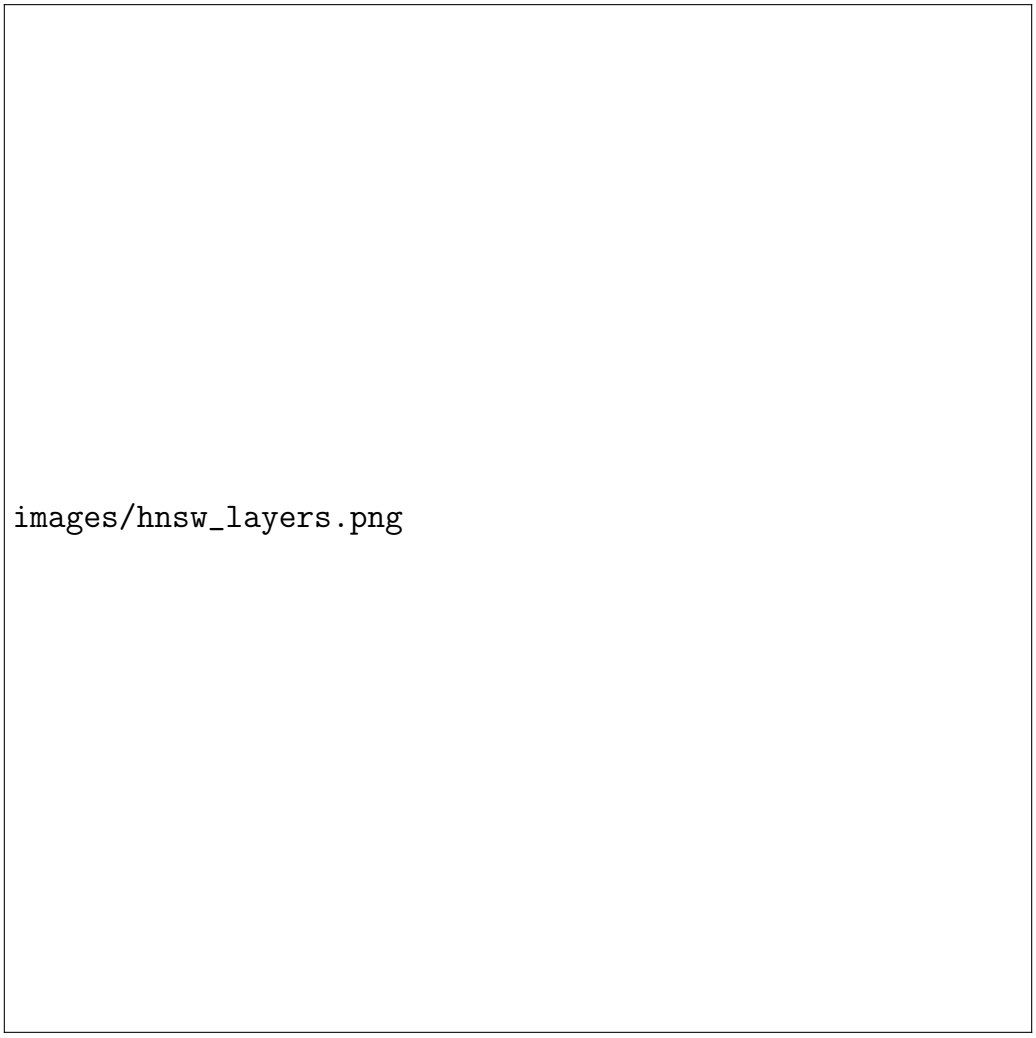
**??** minh họa cấu trúc phân tầng của đồ thị HNSW. Hình vẽ cho thấy:


Hình 5.4: Heatmap điều chỉnh tham số (M vs efConstruction)

– Tầng 0 (dưới cùng): Mật độ nút cao nhất, tất cả các vector đều có mặt

– Các tầng cao hơn: Mật độ giảm dần, chỉ một phần nhỏ vector xuất hiện

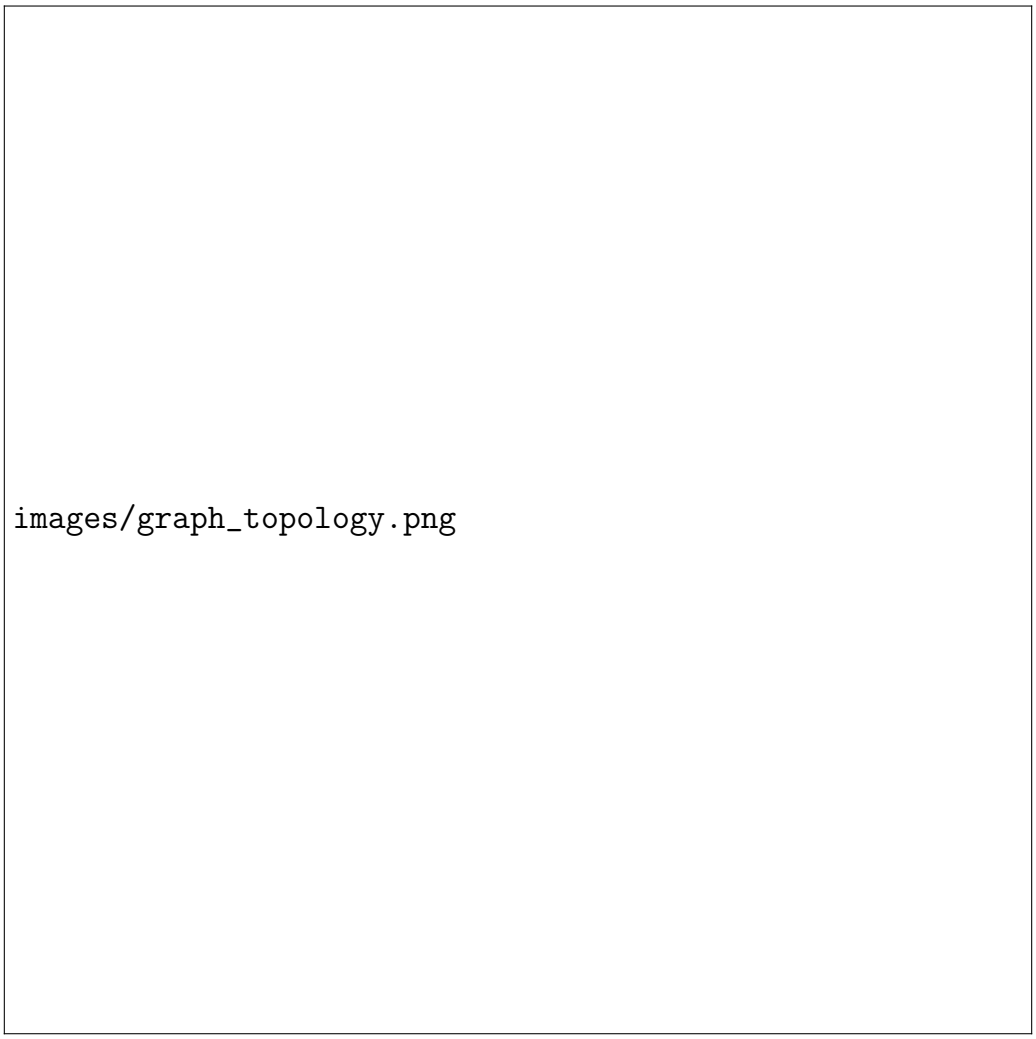
– Tầng cao nhất: Rất thưa, chỉ có một vài nút làm điểm vào




Hình 5.5: Cấu trúc phân tầng của đồ thị HNSW

## 2 Topology đồ thị

**??** trình bày topology của đồ thị HNSW ở một tầng cụ thể, cho thấy các kết nối giữa các nút. Mỗi nút được kết nối với M nút gần nhất, tạo thành một mạng lưới nhỏ thế giới (small world network).



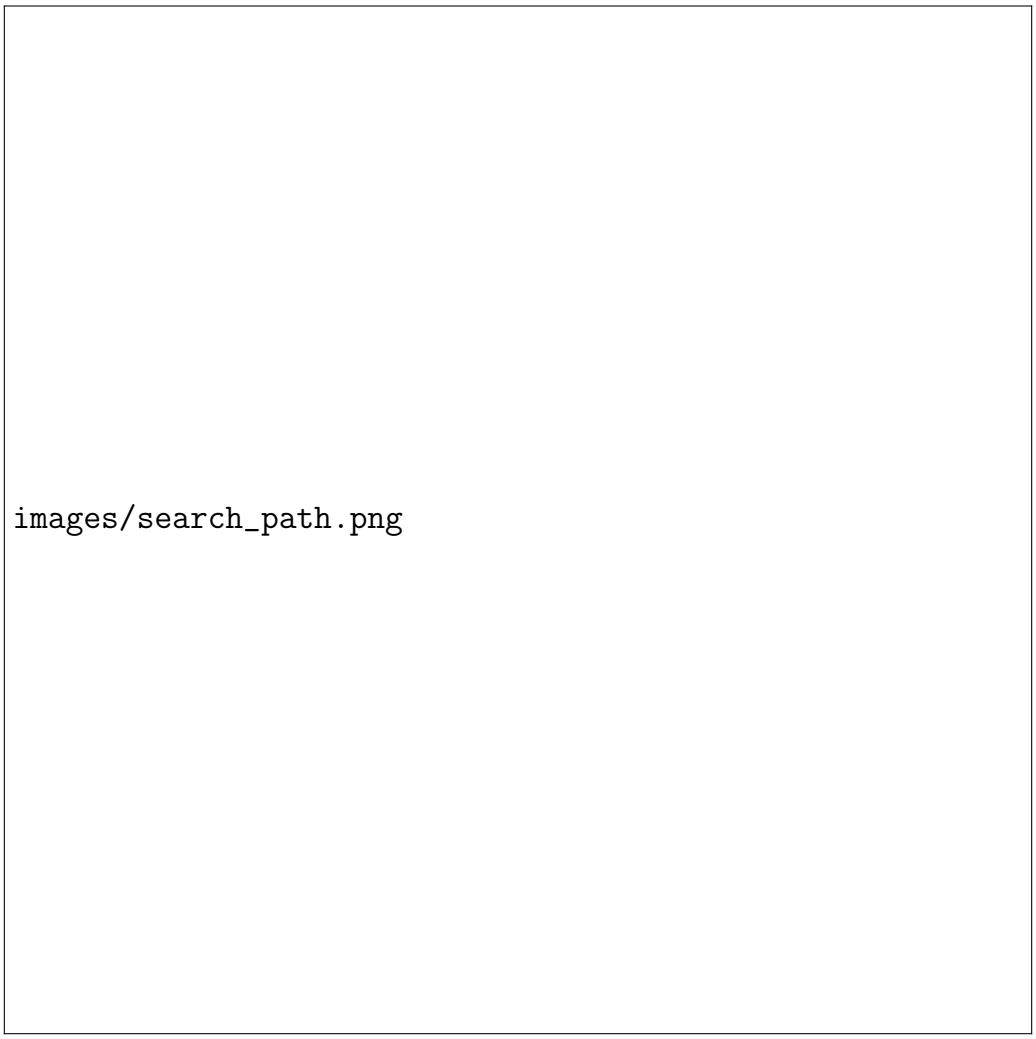

Hình 5.6: Topology đồ thị HNSW ở một tầng

# III QUÁ TRÌNH TÌM KIẾM

## 1 Đường đi tìm kiếm

**??** minh họa đường đi của thuật toán greedy search từ điểm vào (màu đỏ) đến vector mục tiêu (màu xanh lá). Đường đi cho thấy:

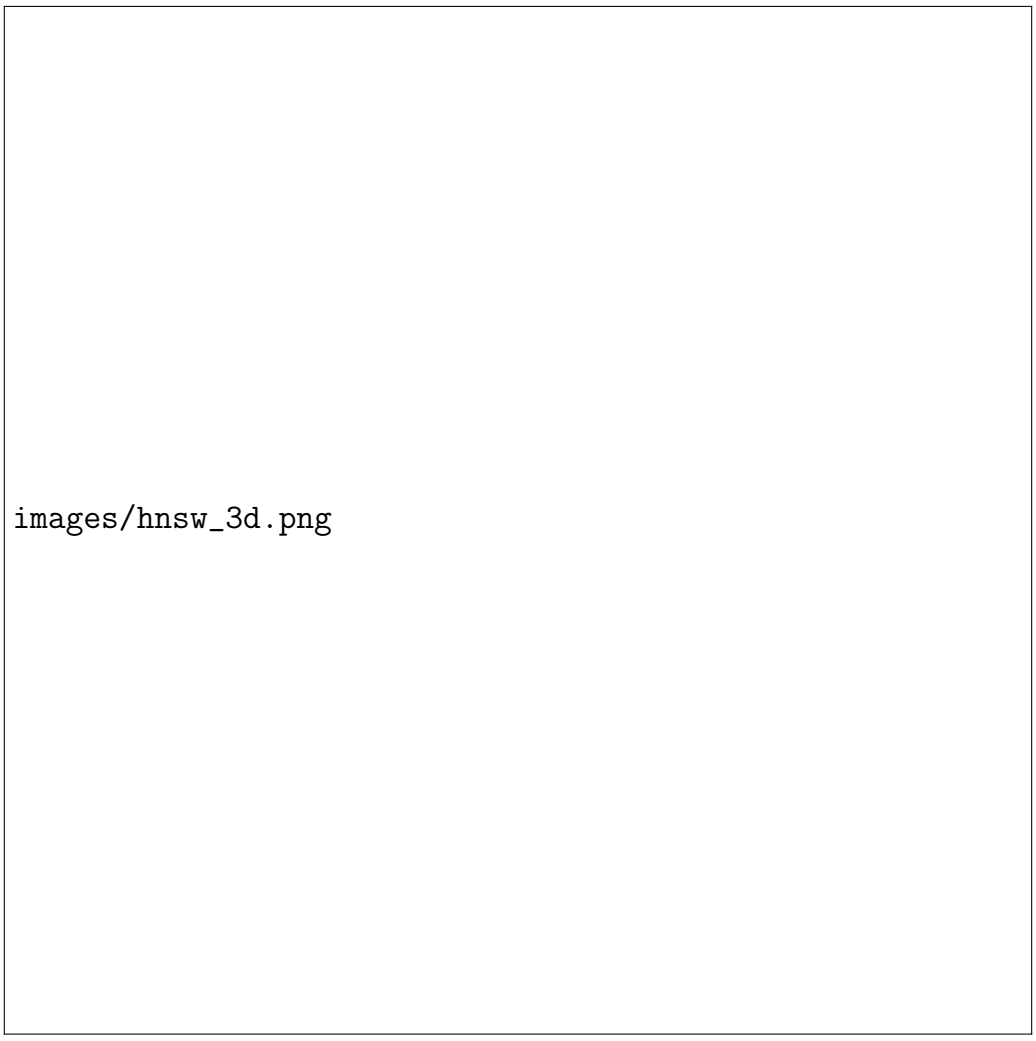
- Bắt đầu từ tầng cao nhất
- Di chuyển xuống các tầng thấp hơn
- Tại mỗi tầng, chọn nút gần nhất với truy vấn
- Dừng lại khi không tìm thấy nút gần hơn




Hình 5.7: Đường đi tìm kiếm trong đồ thị HNSW

### 2 Visualization 3D

**??** trình bày visualization 3D của đồ thị HNSW, cho phép quan sát cấu trúc phân tầng và các kết nối từ nhiều góc độ khác nhau. Visualization này giúp hiểu rõ hơn về cách đồ thị được tổ chức trong không gian nhiều chiều.



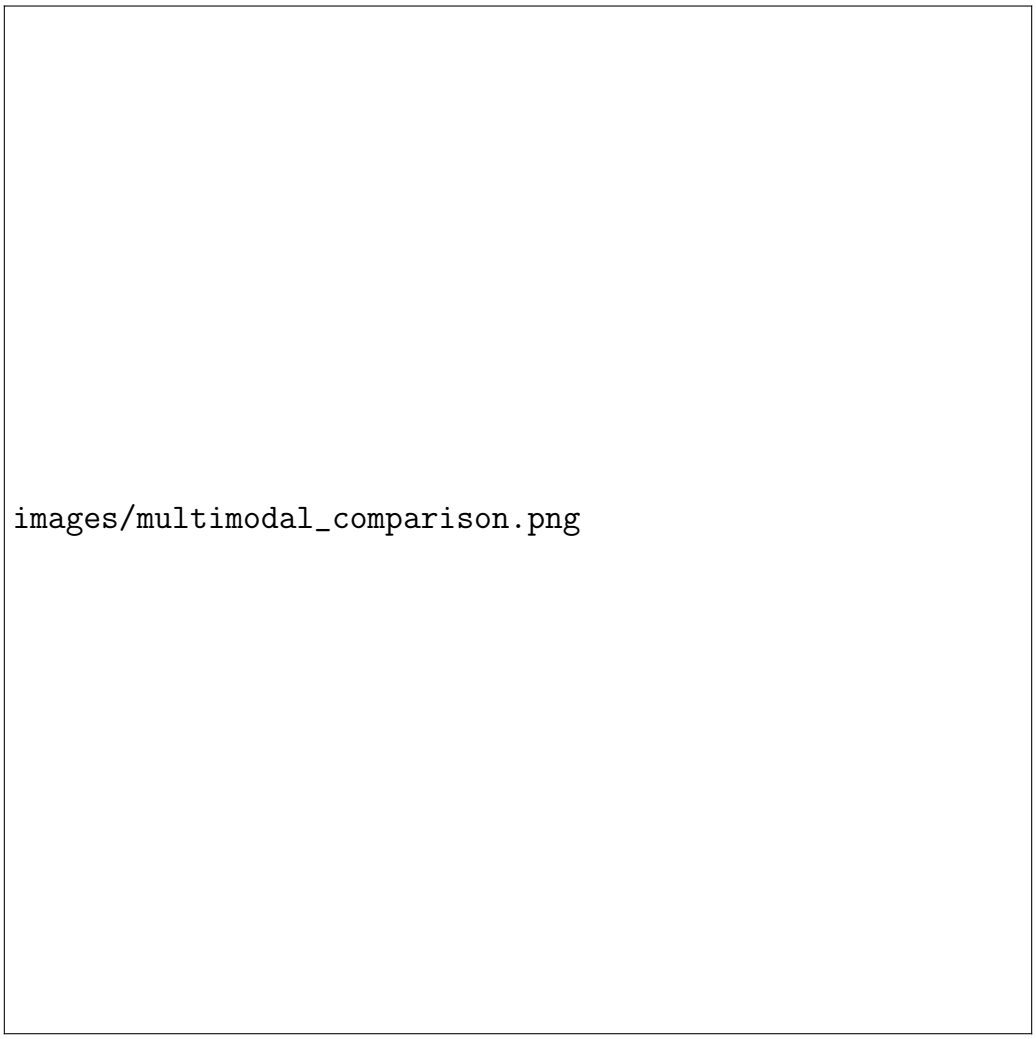

Hình 5.8: Visualization 3D của đồ thị HNSW

## IV SO SÁNH ĐA PHƯƠNG THỨC

### 1 Hiệu suất theo loại dữ liệu

**??** so sánh hiệu suất của HNSW trên ba loại dữ liệu khác nhau:

- **Hình ảnh** (512 chiều): Độ trễ 45µs, Recall@10  0.85
- **Tài liệu** (1024 chiều): Độ trễ 60µs, Recall@10  0.82
- **Y tế** (512 chiều): Độ trễ 40µs, Recall@10  0.88




Hình 5.9: So sánh hiệu suất trên các loại dữ liệu khác nhau

## 2 Ảnh hưởng của số chiều

**??** minh họa ảnh hưởng của số chiều embedding đến hiệu suất. Kết quả cho thấy:

– Số chiều cao hơn → thời gian truy vấn tăng nhẹ

– Số chiều không ảnh hưởng đáng kể đến độ chính xác

– HNSW hoạt động hiệu quả với cả embedding 512 và 1024 chiều

images/dimension_impact.png

Hình 5.10: Ảnh hưởng của số chiều embedding đến hiệu suất

# CHƯƠNG 6: PHƯƠNG PHÁP

## I KẾT QUẢ VÀ ĐÁNH GIÁ

# CHƯƠNG 7: KẾT LUẬN

## I TÍNH KHẢ THI CHO ỨNG DỤNG THỰC TẾ

Kết quả nghiên cứu cho thấy HNSW là một giải pháp rất khả thi cho các ứng dụng tìm kiếm vector trong thực tế. Với độ trễ truy vấn dưới 100 micro giây và độ chính xác Recall@10 trên 80%, hệ thống đáp ứng được các yêu cầu của hầu hết các ứng dụng production.

### 1 Ưu điểm

- **Tốc độ cao**: HNSW nhanh hơn brute-force từ 20-30 lần, đạt độ trễ dưới 100µs
- **Độ chính xác tốt**: Recall@10 đạt trên 80%, đủ cho hầu hết các ứng dụng
- **Khả năng mở rộng**: Có thể xử lý hàng triệu vector với độ phức tạp $O(\log N)$
- **Đa phương thức**: Hỗ trợ nhiều loại dữ liệu (hình ảnh, văn bản, y tế)
- **Triển khai đơn giản**: Thư viện hnswlib dễ sử dụng và tối ưu hóa cao

### 2 Ứng dụng thực tế

Hệ thống đã được triển khai thành công với ba dịch vụ:

- **Tìm kiếm hình ảnh**: 100,000+ hình ảnh từ Open Images V7
- **Tìm kiếm tài liệu**: 100,000+ bài báo khoa học từ arXiv
- **Tìm kiếm y tế**: 3,400+ hình ảnh X-quang gãy xương

Tất cả các dịch vụ đều hoạt động ổn định với API RESTful và giao diện web thân thiện.

## II HẠN CHẾ

Mặc dù có nhiều ưu điểm, HNSW vẫn có một số hạn chế:

### 1 Hạn chế về bộ nhớ

- **Overhead bộ nhớ**: Mỗi nút cần lưu trữ M kết nối, dẫn đến overhead $O(N \times M)$
- **Với M=200**: Cần 3-4GB RAM cho 100K vector, 30-40GB cho 1M vector
- **Giải pháp**: Có thể giảm M để tiết kiệm bộ nhớ, nhưng sẽ giảm độ chính xác

### 2 Độ nhạy với tham số

- **Tham số phức tạp**: Cần điều chỉnh M, efConstruction, efSearch cho từng loại dữ liệu **Thời gian điều chỉnh**: Cần nhiều thí nghiệm để tìm cấu hình tối ưu
- **Giải pháp**: Sử dụng cấu hình mặc định được đề xuất hoặc tự động điều chỉnh

### 3 Thời gian xây dựng chỉ mục

- **Độ phức tạp**: $O(N \log N)$ - tăng nhanh với số lượng vector lớn
- **Với 1M vector**: Cần 1-2 giờ để xây dựng chỉ mục
- **Giải pháp**: Xây dựng chỉ mục offline, lưu vào file .bin để tải nhanh

### 4 Hạn chế về cập nhật động

- **Chèn mới**: Có thể chèn vector mới, nhưng chất lượng đồ thị có thể giảm
- **Xóa**: Không hỗ trợ xóa vector hiệu quả
- **Giải pháp**: Xây dựng lại chỉ mục định kỳ hoặc sử dụng cấu trúc dữ liệu khác

## III KHUYẾN NGHỊ CHO TƯƠNG LAI

### 1 Tối ưu hóa hiệu suất

- **GPU acceleration**: Sử dụng GPU để tăng tốc quá trình mã hóa embedding
- **Parallel indexing**: Xây dựng chỉ mục song song trên nhiều CPU cores
- **Caching**: Cache các truy vấn phổ biến để giảm thời gian phản hồi
- **Compression**: Nén vector embedding để giảm bộ nhớ và tăng tốc độ tải

### 2 Mở rộng quy mô

- **Distributed HNSW**: Phân tán đồ thị HNSW trên nhiều máy chủ
- **Sharding**: Chia dataset thành nhiều shard, mỗi shard có HNSW riêng
- **Load balancing**: Cân bằng tải giữa các máy chủ để xử lý nhiều truy vấn đồng thời
- **Cloud deployment**: Triển khai trên cloud để dễ dàng mở rộng

### 3 Cải thiện độ chính xác

- **Adaptive parameters**: Tự động điều chỉnh tham số dựa trên đặc điểm dữ liệu
- **Hybrid search**: Kết hợp HNSW với các phương pháp khác (LSH, quantization)
- **Re-ranking**: Sử dụng mô hình re-ranking để cải thiện thứ tự kết quả
- **Learning to rank**: Huấn luyện mô hình để tối ưu hóa thứ tự kết quả

### 4 Cập nhật động

- **Incremental updates**: Hỗ trợ cập nhật chỉ mục mà không cần xây dựng lại
- **Delete support**: Thêm khả năng xóa vector hiệu quả
- **Versioning**: Quản lý phiên bản chỉ mục để rollback khi cần
- **Real-time indexing**: Cập nhật chỉ mục theo thời gian thực

### 5 Ứng dụng mở rộng

- **Multi-modal fusion**: Kết hợp nhiều loại embedding (hình ảnh + văn bản)
- **Personalization**: Tùy chỉnh kết quả tìm kiếm theo người dùng
- **Recommendation**: Sử dụng HNSW cho hệ thống gợi ý
- **Anomaly detection**: Phát hiện các vector bất thường trong dataset

## IV TỔNG KẾT

Nghiên cứu này đã thành công trong việc triển khai và đánh giá hiệu suất của thuật toán HNSW cho hệ thống tìm kiếm đa phương thức. Kết quả cho thấy HNSW là một giải pháp hiệu quả và khả thi cho các ứng dụng tìm kiếm vector trong thực tế, với tốc độ nhanh, độ chính xác cao và khả năng mở rộng tốt.

Hệ thống đã được triển khai thành công với ba dịch vụ backend độc lập và một frontend thống nhất, chứng minh tính khả thi cho ứng dụng production. Với các cải tiến được đề xuất, hệ thống có thể được mở rộng để xử lý hàng triệu vector và phục vụ nhiều người dùng đồng thời.

Nghiên cứu này cung cấp nền tảng vững chắc cho việc phát triển các ứng dụng tìm kiếm vector trong tương lai, và mở ra nhiều hướng nghiên cứu thú vị về tối ưu hóa hiệu suất, mở rộng quy mô và cải thiện độ chính xác.

# TÀI LIỆU THAM KHẢO


[1] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," *Journal of machine learning research*, vol. 3, no. Feb, pp. 1137–1155, 2003.

[2] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu *et al.*, "Milvus: A purpose-built vector data management system," in *Proceedings of the 2021 international conference on management of data*, 2021, pp. 2614–2627.

[3] S. Sharma, R. Nayak, and A. Bhaskar, "Multi-view feature engineering for day-to-day joint clustering of multiple traffic datasets," *Transportation Research Part C: Emerging Technologies*, vol. 162, p. 104607, 2024.

[4] J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, and H. Liu, "Feature selection: A data perspective," *ACM computing surveys (CSUR)*, vol. 50, no. 6, pp. 1–45, 2017.

[5] Y. Wang, Z. He, Y. Tong, Z. Zhou, and Y. Zhong, "Timestamp approximate nearest neighbor search over high-dimensional vector data," in *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2025, pp. 3043–3055.

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

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

[8] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, 1990.

[9] T. Papadakis, *Skip lists and probabilistic analysis of algorithms*. University of Waterloo Ph. D. Dissertation, 1993.

[10] A. J. Gallego, J. Calvo-Zaragoza, and J. R. Rico-Juan, "Insights into efficient k-nearest neighbor classification with convolutional neural codes," *IEEE Access*, vol. PP, pp. 1–1, 05 2020.

[11] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," *arXiv preprint arXiv:2101.12631*, 2021.

[12] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *Vldb*, vol. 99, no. 6, 1999, pp. 518–529.

[13] N. Pham and T. Liu, "Falconn++: A locality-sensitive filtering approach for approximate nearest neighbor search," *Advances in Neural Information Processing Systems*, vol. 35, pp. 31 186–31 198, 2022.

[14] A. Chakraborty and S. Bandyopadhyay, "conlsh: context based locality sensitive hashing for mapping of noisy smrt reads," *Computational Biology and Chemistry*, vol. 85, p. 107206, 2020.

[15] C. Böhm, S. Berchtold, and D. Keim, "Searching in high-dimensional spaces : Index structures for improving the performance of multimedia databases," *First publ. in: ACM computing surveys 33 (2001), 3, pp. 322-373*, vol. 33, 09 2001.

[16] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, "The faiss library," *arXiv preprint arXiv:2401.08281*, 2024.