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 12/2025

Đường dẫn liên quan:

GitHub Repository: dsa-advanced-assignment-hnsw

Google Colab Notebook 1: Compare Brute-force vs HNSW

Google Colab Notebook 2: Compare 4 model CLIP

Web App: HNSW Search Engine

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 10,000 vector với 128 chiều. Kết quả thực nghiệm cho thấy HNSW (phiên bản HNSWLib) đạt độ trễ truy vấn trung bình khoảng 258.9 – 267.4 micro giây, nhanh hơn đáng kể so với brute-force (573.9 – 600.9 micro giây). Ngoài ra, còn một số biến thể của HNSW, trong HNSW-PQ cho tốc độ truy vấn nhanh nhất (khoảng 100 s), nhưng độ chính xác Recall@1 bị giảm đáng kể (chỉ đạt 0.17), do đó hệ thống ưu tiên sử dụng thuw viện HNSWlib hoặc cấu hình HNSW-SQ hoặc HNSW-Flat để đảm bảo chất lượng tìm kiếm.

Nghiên cứu cũng phân tích ảnh hưởng của các tham số HNSW (M, $efConstruction$, $efSearch$) đến hiệu suất. Hệ thống đã được triển khai thành công với ba dịch vụ backend độc lập và giao diện frontend thống nhất, chứng minh tính khả thi cao cho các ứng dụng thực tế quy mô lớn.

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 ≥ 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.

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

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

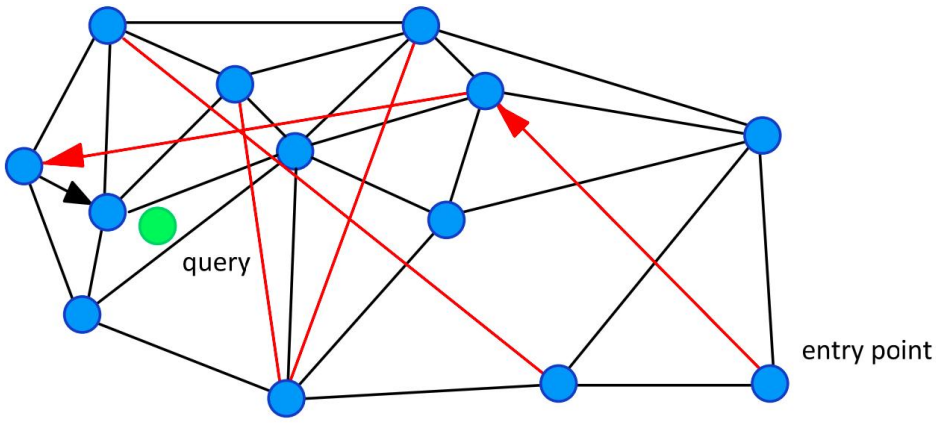
$$d(p, q) = 1 - \frac{\sum_{i=1}^{n} p_i \times q_i}{\sqrt{\sum_{i=1}^{n} p_i^2} + \sqrt{\sum_{i=1}^{n} q_i^2}} \tag{2.3}$$

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

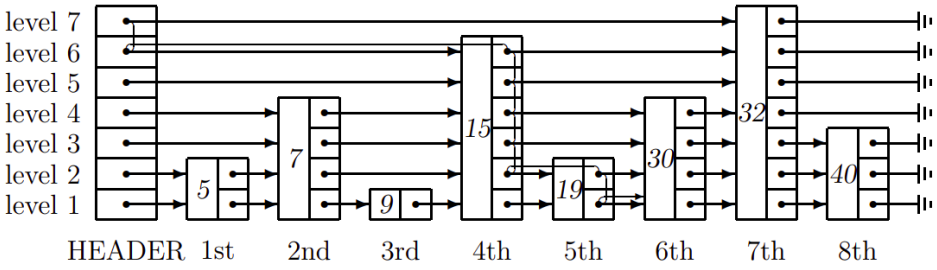
```
1:  procedure K-NNSEARCH(q, m, k)
2:      TreeSet tempRes, candidates, visitedSet, result
3:      for i ← 1 to m do
4:          Put a random entry point into candidates
5:          tempRes ← ∅
6:          loop
7:              c ← get element from candidates closest to q
8:              Remove c from 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

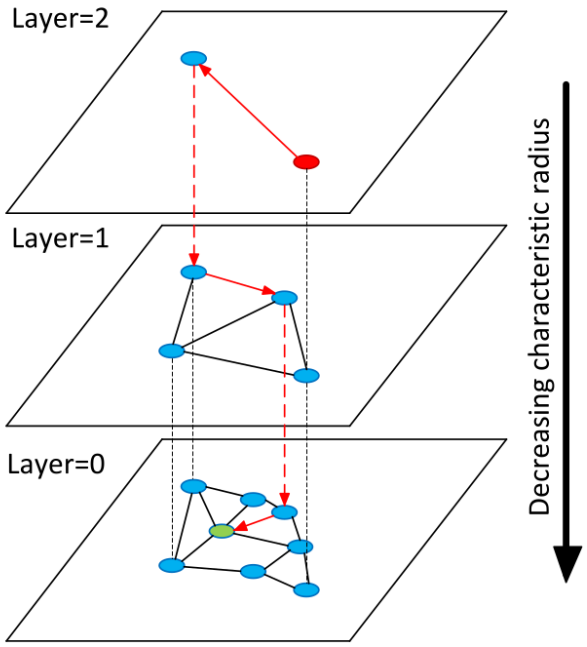
a Đồ thị phân tầng - nền tảng của HNSW

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($hnsw, q, K, ef$)

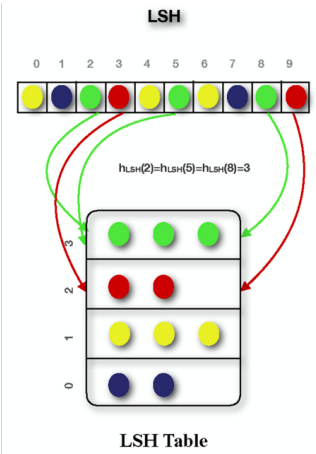
procedure K-NN-SEARCH($hnsw, q, K, ef$)
  $W \leftarrow \emptyset$ ▷ Set for the current nearest elements
  $ep \leftarrow$ get enter-point for $hnsw$
  $L \leftarrow$ level of ep ▷ Top layer for $hnsw$
  for $l_c \leftarrow L$ **down to** 1 **do**
    $W \leftarrow$ SEARCH-LAYER$(q, ep, ef = 1, l_c)$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

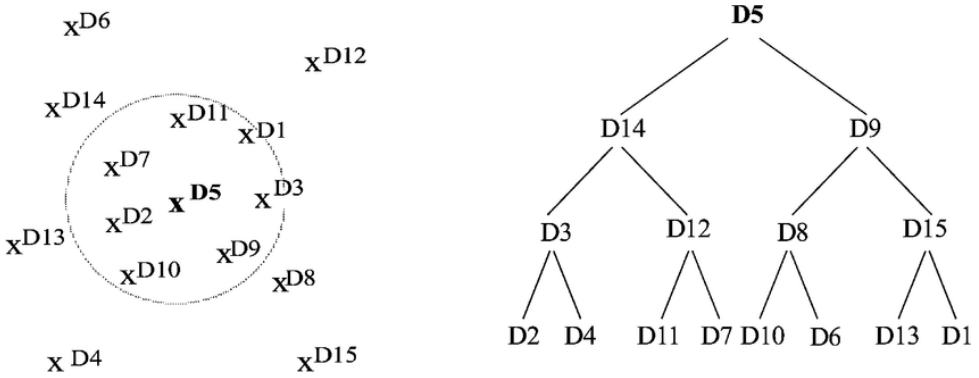


Hình 2.4: Minh họa cho các buckets trong thuật toán LSH [**?**]

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.

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: `https://huynguyen6906-image-server.hf.space/`
- Paper Search API: `https://huynguyen6906-paper-server.hf.space/`
- Medical Search API: `https://huynguyen6906-medical-server.hf.space/`

2 Thành phần Backend

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

a Image Search Service (`https://huynguyen6906-image-server.hf.space/`)

- **Mô hình**: OpenAI CLIP (ViT-B/32)
- **Embedding dimension**: 512
- **Dataset**: Open Images V7, 1.502.477 hình ảnh
- **Storage**: HDF5 file (`Image_Embedded.h5`)
- **Features**: Tìm kiếm hình ảnh bằng từ khóa (string) hay ảnh (.png, .jpg)

b Paper Search Service (`https://huynguyen6906-paper-server.hf.space/`)

- **Mô hình**: Sentence Transformers (all-roberta-large-v1)
- **Embedding dimension**: 1024
- **Dataset**: arXiv papers, 1.000.000 tài liệu
- **Storage**: HDF5 file (`Papers_Embedbed_0-1000000.h5`)
- **Features**: Tìm kiếm bằng văn bản hoặc tải lên file PDF/TXT

c Medical Search Service (`https://huynguyen6906-medical-server.hf.space/`)

- **Mô hình**: BiomedCLIP (microsoft/BiomedCLIP-PubMedBERT_256-vit_base_patch16_224)
- **Embedding dimension**: 512
- **Dataset**: FracAtlas bone fractures, 3.366 hình ảnh X-quang
- **Storage**: HDF5 file (`Medical_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 1,5M+ hình ảnh được lấy từ nguồn 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 1.000.000 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)**: 20 - 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
}
```

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: 3GB cho 1,5M hình ảnh (đã chuyển đổi thành vector 512 chiều), 4GB cho 1M tài liệu (đã chuyển đổi thành vector 1024 chiều).

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

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 nền tảng đám mây Google Colab (phiên bản miễn phí) với cấu hình chi tiết như sau:

- **Môi trường**: Google Colab (Free Tier)
- **CPU**: Intel Xeon (2 vCPUs @ 2.20GHz)
- **RAM**: $\approx$ 12.7 GB
- **GPU**: Không sử dụng (Chế độ CPU-only)
- **Hệ điều hành**: Linux (Ubuntu 22.04 LTS)
- **Python**: 3.10+
- **Thư viện**: hnswlib 0.8.0, faiss-cpu, numpy 1.24.0, PyTorch 2.0+

2 Cấu hình thí nghiệm

- **Số lượng vector**: 10,000
- **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, efSearch=100

3 Các phương pháp và triển khai được đánh giá

Trong nghiên cứu này, chúng tôi so sánh hiệu năng của giải pháp tìm kiếm chính xác (Exact Search) với **các triển khai khác nhau của thuật toán HNSW** (HNSW-based Implementations). Các phương pháp được chia thành hai nhóm chính:

- **Baseline**: Sử dụng thuật toán tìm kiếm vét cạn (Brute-force) để làm chuẩn so sánh về độ chính xác.

– **HNSW Implementations**: Bao gồm thư viện `hnswlib` (triển khai gốc) và các cấu hình tối ưu hóa trong thư viện FAISS (`HNSW-Flat`, `SQ`, `PQ`) để đánh giá khả năng cân bằng giữa tốc độ, bộ nhớ và độ chính xác.

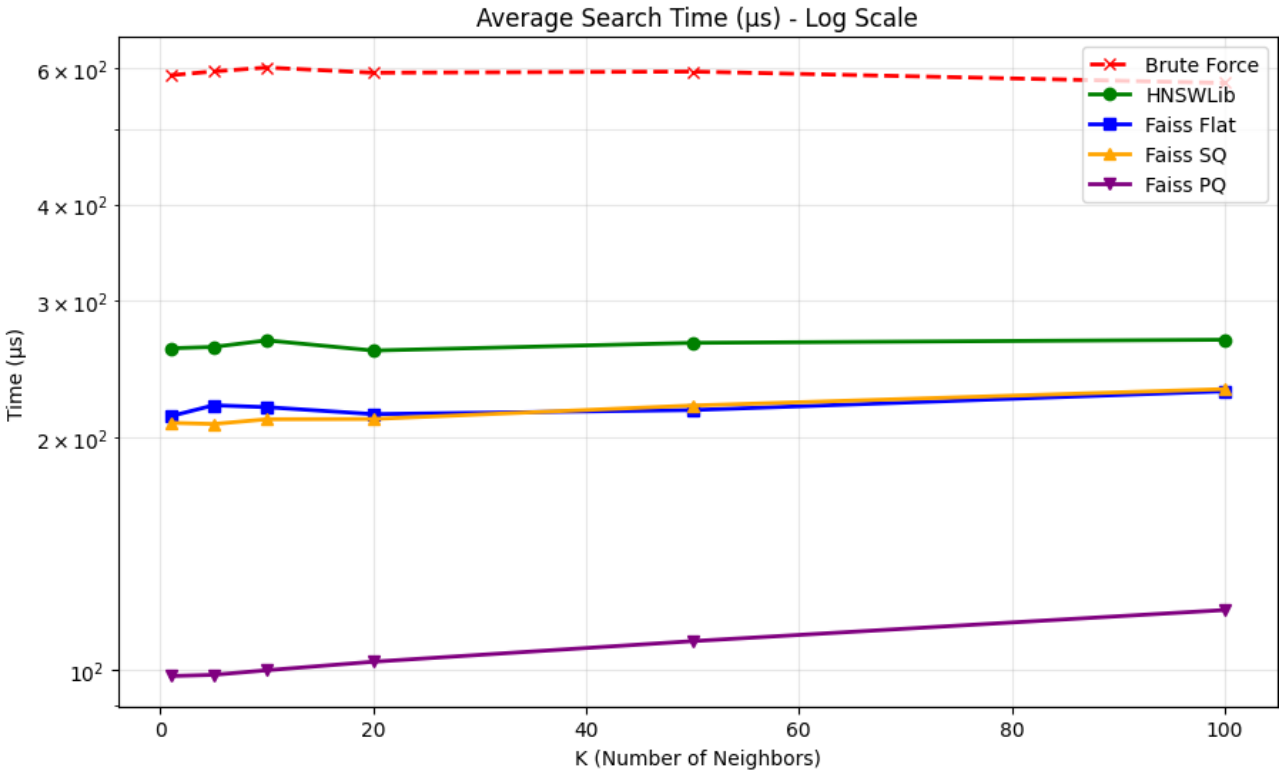
II CÁC CHỈ SỐ ĐÁNH GIÁ

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	Brute-force (µs)	HNSWlib (µs)	HNSW-Flat (µs)	HNSW-SQ (µs)	HNSW-PQ (µs)
1	587.3	260.7	213.0	208.8	98.3
5	593.9	261.7	220.0	208.1	98.7
10	600.9	266.9	218.8	211.0	100.0
20	591.6	258.9	214.3	211.2	102.6
50	593.7	264.9	216.9	219.8	109.1
100	573.9	267.4	229.3	230.9	119.6



Hình 4.1: Biểu đồ so sánh độ trễ giữa HNSW và Brute-force

Kết quả cho thấy các thuật toán dựa trên HNSW cải thiện đáng kể tốc độ truy vấn so với

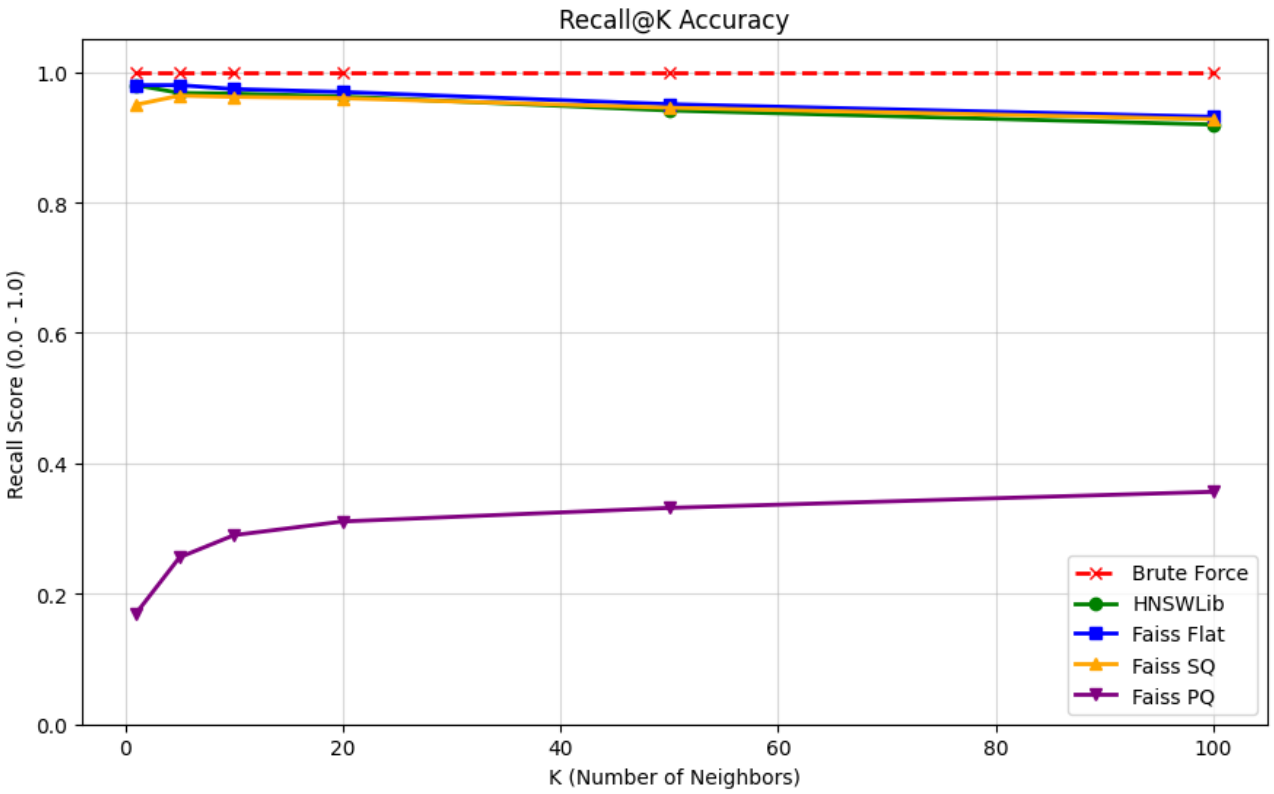
phương pháp vét cạn (Brute-force). Cụ thể, thuật toán HNSW-PQ cho hiệu năng tốt nhất với độ trễ trung bình khoảng 98 µs, nhanh gấp 6 lần so với Brute-force (590 µs). Các biến thể khác như HNSWLib, HNSW-Flat và HNSW-SQ cũng duy trì độ trễ ổn định trong khoảng 200-260 µs, nhanh hơn Brute-force từ 2.2 đến 2.8 lần.

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			
	HNSWLib	HNSW-Flat	HNSW-SQ	HNSW-PQ
1	0.9800	0.9800	0.9500	0.1700
5	0.9680	0.9800	0.9640	0.2560
10	0.9670	0.9740	0.9620	0.2900
20	0.9625	0.9695	0.9600	0.3110
50	0.9412	0.9508	0.9454	0.3318
100	0.9196	0.9315	0.9277	0.3565



Hình 4.2: Biểu đồ so sánh độ chính xác Recall@K của HNSW

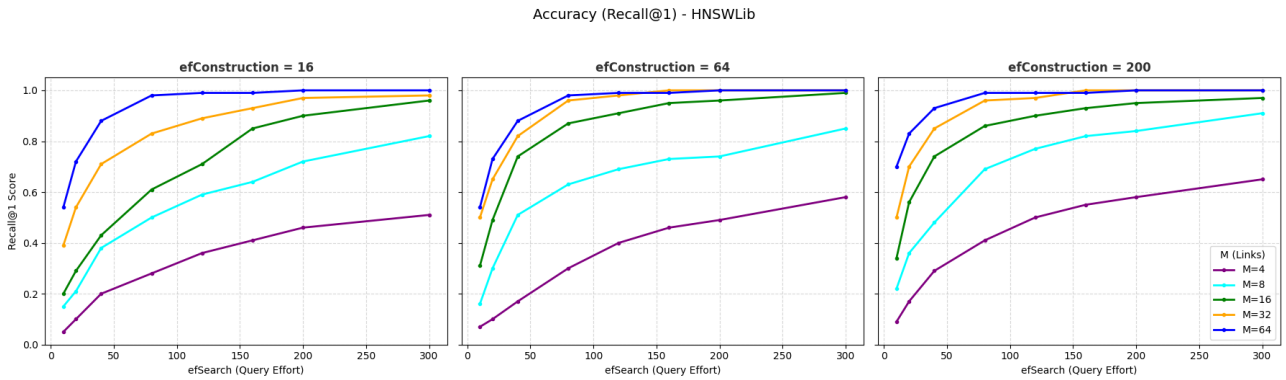
– **Độ chính xác cao nhất (HNSWLib & HNSW-Flat):** Hai đường biểu diễn nằm sát đường cơ sở (*Brute Force*) với Recall@1 đạt 0.98. Đây là lựa chọn tối ưu khi ưu tiên độ chính xác tuyệt đối và không bị giới hạn về bộ nhớ RAM.

– **Cân bằng tối ưu (HNSW-SQ):** Mặc dù sử dụng lượng tử hóa vô hướng (*Scalar Quantization*) để nén dữ liệu, độ chính xác vẫn rất cao (Recall@1 đạt 0.95), chỉ thấp hơn bản Flat không đáng kể. Đây là phương pháp hiệu quả nhất xét trên tỷ lệ *hiệu năng/tài nguyên*.

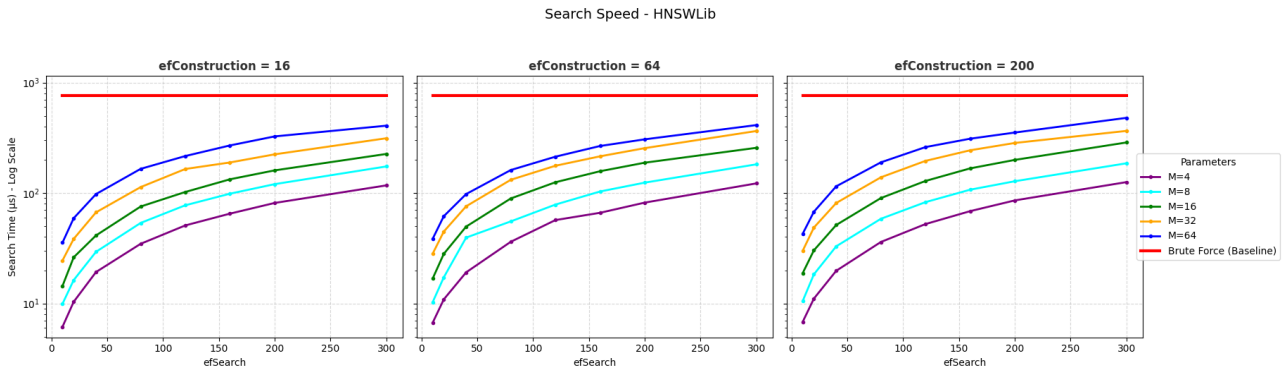
– **Hiệu năng thấp (HNSW-PQ):** Kỹ thuật *Product Quantization* trong trường hợp này làm mất mát quá nhiều thông tin, dẫn đến độ chính xác rất thấp (Recall@1 chỉ đạt 0.17). Không khuyến nghị sử dụng cấu hình này cho tập dữ liệu hiện tại.

Kết luận: Dựa trên biểu đồ, **HNSW-SQ** là giải pháp tốt nhất để triển khai thực tế nhờ khả năng tiết kiệm bộ nhớ trong khi vẫn duy trì độ chính xác cao (96%).

3 Ảnh hưởng của các thành phần đến độ chính xác và độ trễ



Hình 4.3: Ảnh hưởng của các tham số đến độ chính xác Recall@1 (HNSWLib)



Hình 4.4: Ảnh hưởng của các tham số đến độ trễ khi truy vấn của thuật toán (HNSWLib)

a Phân tích độ chính xác (Accuracy Analysis)

– **Vai trò của tham số M:** Độ chính xác tỉ lệ thuận với số lượng liên kết M. Các cấu hình $M \geq 32$ (đường màu xanh dương và cam) nhanh chóng đạt mức bão hòa (Recall ≈ 0.99) ngay tại các giá trị $efSearch$ thấp. Ngược lại, với $M = 4$, cấu trúc đồ thị quá thưa khiến thuật toán không thể đạt độ chính xác chấp nhận được.

– **Tác động của efConstruction:** Việc tăng $efConstruction$ từ 16 lên 200 giúp đường cong Recall tiệm cận mức 1.0 nhanh hơn. Điều này chứng tỏ chất lượng đồ thị tốt hơn (được xây dựng kỹ hơn) sẽ giảm bớt gánh nặng tính toán trong giai đoạn tìm kiếm.

b Phân tích tốc độ tìm kiếm (Latency Analysis)

Biểu đồ tốc độ sử dụng thang đo logarit (Log Scale) để làm nổi bật sự chênh lệch lớn về hiệu năng:

– **Hiệu quả so với Brute-force:** Tất cả các cấu hình HNSW đều có thời gian truy vấn thấp hơn đáng kể so với phương pháp vét cạn (đường màu đỏ - Baseline). Ngay cả ở cấu hình chậm nhất ($M = 64, efSearch = 300$), HNSW vẫn nhanh hơn Brute-force gần một bậc độ lớn (order of magnitude).

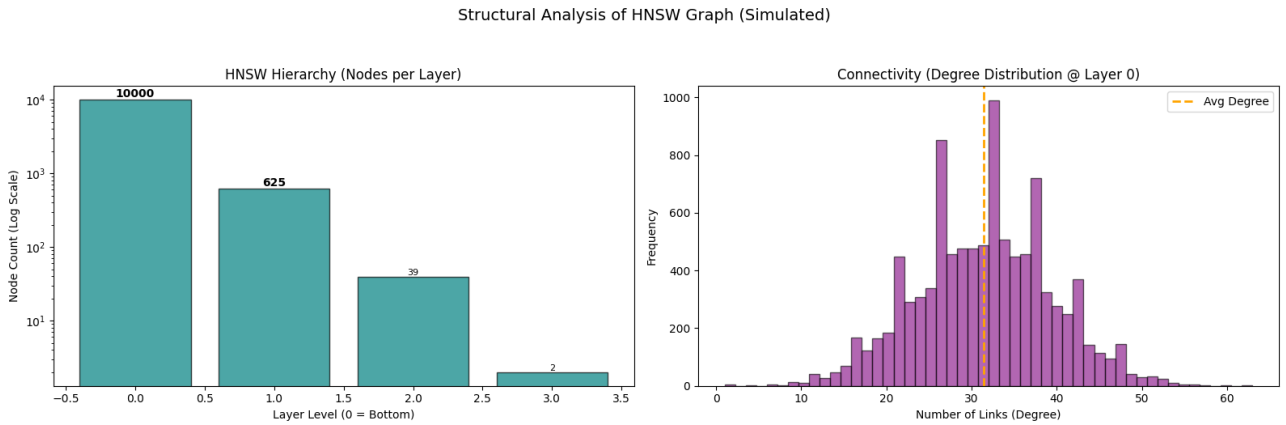
– **Chi phí thời gian của M:** Tăng M làm tăng thời gian tìm kiếm do số lượng phép tính khoảng cách tại mỗi bước nhảy tăng lên. Đường $M = 64$ (xanh dương) nằm cao nhất trên biểu đồ thời gian, trong khi $M = 4$ (tím) là nhanh nhất nhưng lại không đảm bảo độ chính xác.

c Kết luận về sự đánh đổi (Trade-off Conclusion)

Từ sự đối chiếu giữa hai biểu đồ, ta rút ra các nhận định quan trọng để lựa chọn cấu hình:

1. **Điểm bão hòa hiệu năng:** Tại mức $efSearch \approx 100 - 150$, độ chính xác của các mô hình $M \geq 16$ đã đạt đỉnh. Việc tiếp tục tăng $efSearch$ sau điểm này (ví dụ lên 300) làm thời gian truy vấn tăng tuyến tính nhưng độ chính xác gần như không đổi. Đây là vùng lãng phí tài nguyên cần tránh.
2. **Cấu hình tối ưu (Sweet Spot):** Cấu hình $M = 32$ với $efConstruction = 200$ mang lại sự cân bằng tốt nhất.
   – So với $M = 64$: Độ chính xác gần như tương đương nhưng tốc độ nhanh hơn khoảng 30-40%.
   – So với $M = 16$: Tốc độ chậm hơn không đáng kể nhưng độ ổn định của Recall cao hơn nhiều.

4 Phân tích cấu trúc đồ thị HNSW mô phỏng



Hình 4.5: Thống kê cấu trúc đồ thị HNSW mô phỏng

Dựa trên kết quả mô phỏng với kích thước tập dữ liệu $N = 10,000$, cấu trúc đồ thị HNSW hình thành như sau:

a Phân cấp số lượng tầng

Đồ thị được tổ chức thành 4 tầng riêng biệt (từ Layer 0 đến Layer 3), tuân theo quy luật giảm dần theo hàm mũ để tối ưu hóa đường dẫn tìm kiếm:

- **Tầng 0 (Layer 0):** Chứa toàn bộ $10,000$ vector dữ liệu. Đây là tầng cơ sở lưu trữ đầy đủ thông tin.
- **Các tầng định hướng (Layer 1 - 3):** Số lượng nút giảm mạnh để tạo thành cấu trúc "Small World":
  - Layer 1: 625 nút.
  - Layer 2: 39 nút.
  - Layer 3: 2 nút.

Tỷ lệ giảm giữa các tầng xấp xỉ $1/16$, cho thấy tham số xác suất xây dựng tầng ($probability\_factor$) hoạt động ổn định ở mức 6.25%.

b Độ kết nối và bậc trung bình

Tại tầng 0, tính chất liên kết của các vector được thể hiện qua phân phối bậc (Degree Distribution):

- **Bậc trung bình:** Mỗi nút có trung bình khoảng 32 liên kết với các nút láng giềng.

– **Phân phối:** Biểu đồ tần suất cho thấy sự phân bố tập trung (dạng hình chuông), với đa số các nút có từ 25 đến 45 cạnh. Điều này đảm bảo tính đồng nhất của đồ thị, hạn chế sự xuất hiện của các nút cô lập hoặc các "siêu nút"(hubs) gây mất cân bằng tải khi duyệt.

c Điểm vào (Entry Point)

Điểm khởi đầu cho thuật toán tìm kiếm nằm tại tầng cao nhất (Layer 3). Với chỉ duy nhất 2 nút tại tầng này, hệ thống có thể xác định hướng đi ban đầu gần như tức thời trước khi điều hướng xuống các tầng chi tiết hơn.

III 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

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

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

IV 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 THUẬT TOÁN

I MẪU TRỰC QUAN HÓA

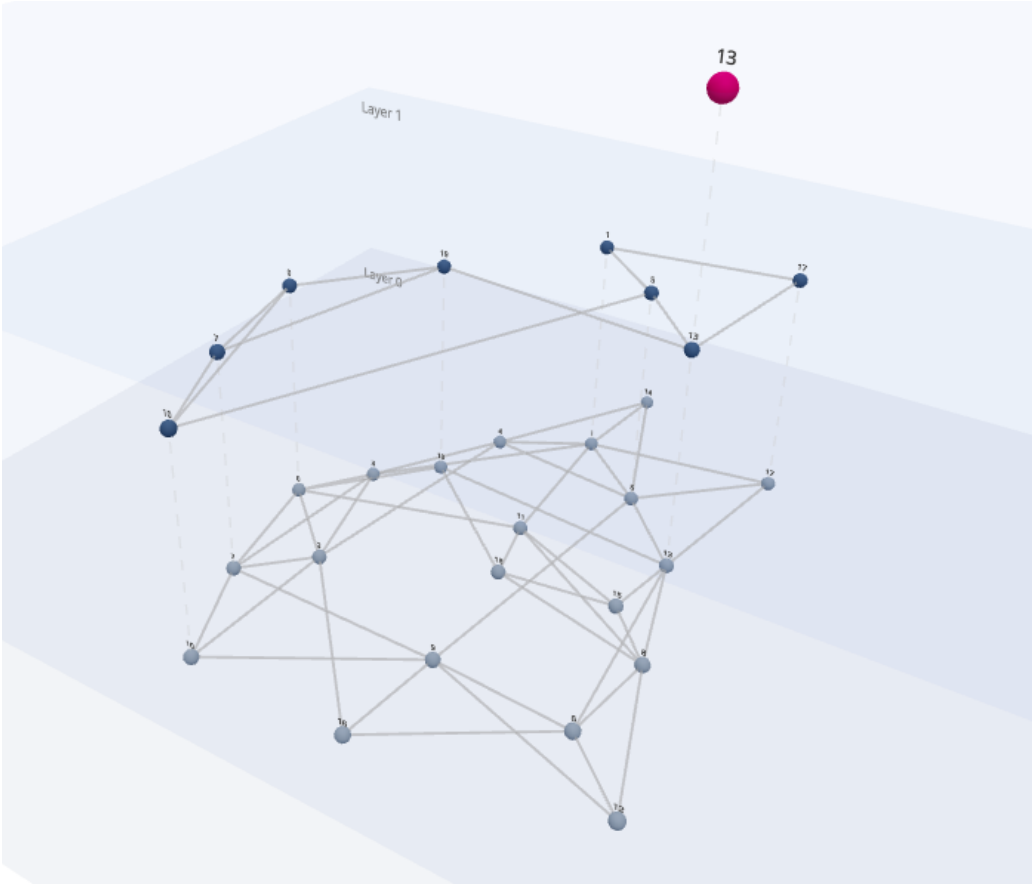
Các thí nghiệm được thực hiện trên nền tảng đám mây Google Colab (phiên bản miễn phí) với cấu hình chi tiết như sau:

- **Số lượng phần tử**: 20
- **M (Số liên kết tối đa)**: 3
- **efConstruction**: 3
- **efSearch**: 5
- **k (số láng giềng gần nhất)**: 10

II CẤU TRÚC ĐỒ THỊ 3D

?? minh họa cấu trúc không gian 3 chiều của đồ thị HNSW, cung cấp cái nhìn trực quan về sự phân tầng và mạng lưới liên kết giữa các vector. Việc trực quan hóa này giúp làm rõ cách thức tổ chức dữ liệu trong không gian nhiều chiều:

- **Tầng cơ sở (Layer 0):** Có mật độ nút cao nhất, chứa toàn bộ các vector dữ liệu.
- **Các tầng trung gian:** Mật độ giảm dần, đóng vai trò cầu nối giúp thuật toán định hướng nhanh.
- **Tầng cao nhất:** Cấu trúc thưa thớt, chứa duy nhất nút điểm vào (entry point - nút màu đỏ).

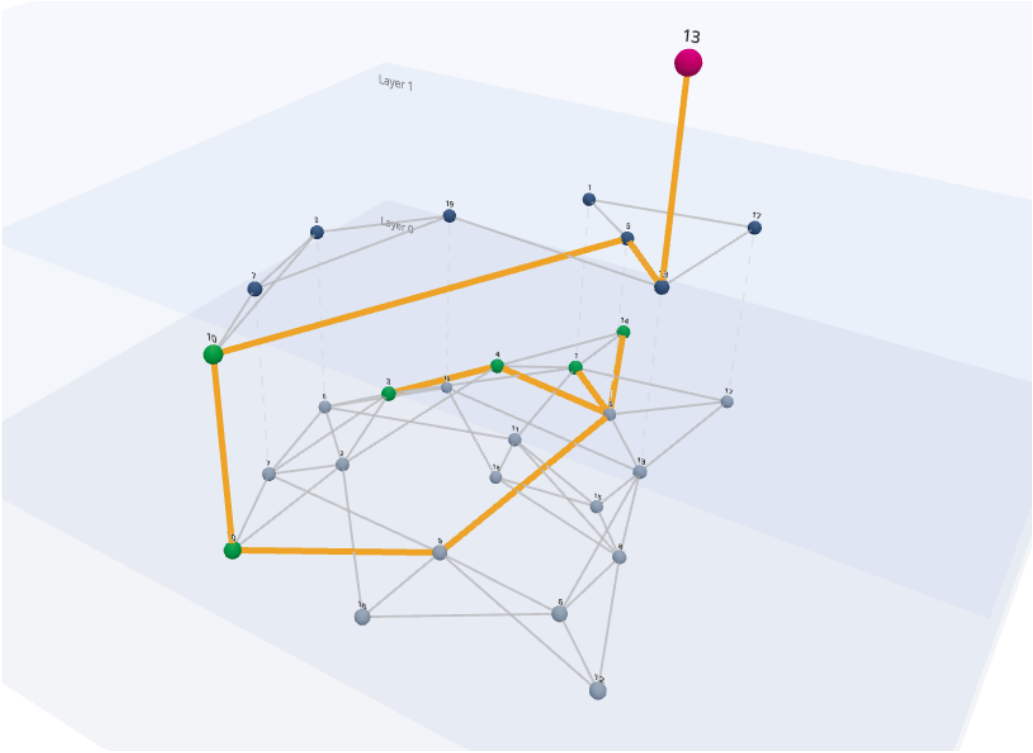


Hình 5.1: Mô hình 3D thể hiện sự phân tầng của HNSW

III MÔ PHỎNG QUÁ TRÌNH TÌM KIẾM

?? minh họa trực quan lộ trình di chuyển của giải thuật Greedy Search trong không gian đồ thị phân tầng. Quá trình này mô phỏng cơ chế "zoom-in", cho phép hệ thống thu hẹp nhanh chóng vùng tìm kiếm từ quy mô toàn cục xuống cục bộ. Các giai đoạn thực thi cụ thể bao gồm:

- **Khởi tạo (Initialization):** Thuật toán bắt đầu tại điểm vào (entry point - nút màu đỏ) nằm ở tầng cao nhất, nơi đồ thị thưa thớt nhất để thực hiện các bước nhảy lớn.
- **Duyệt tầng (Layer Navigation):** Tại mỗi tầng, thuật toán thực hiện tìm kiếm tham lam bằng cách so sánh khoảng cách và chọn nút láng giềng có độ tương đồng cao nhất với vector truy vấn (query vector).
- **Chuyển tầng (Hierarchical Descent):** Khi không còn láng giềng nào gần truy vấn hơn nút hiện tại, thuật toán hạ xuống tầng tiếp theo ngay tại vị trí đó để tinh chỉnh kết quả.
- **Kết thúc (Convergence):** Quy trình lặp lại cho đến khi đạt trạng thái tối ưu cục bộ tại tầng cơ sở (Layer 0), trả về vector mục tiêu (màu xanh lá) là kết quả tìm kiếm gần đúng nhất.

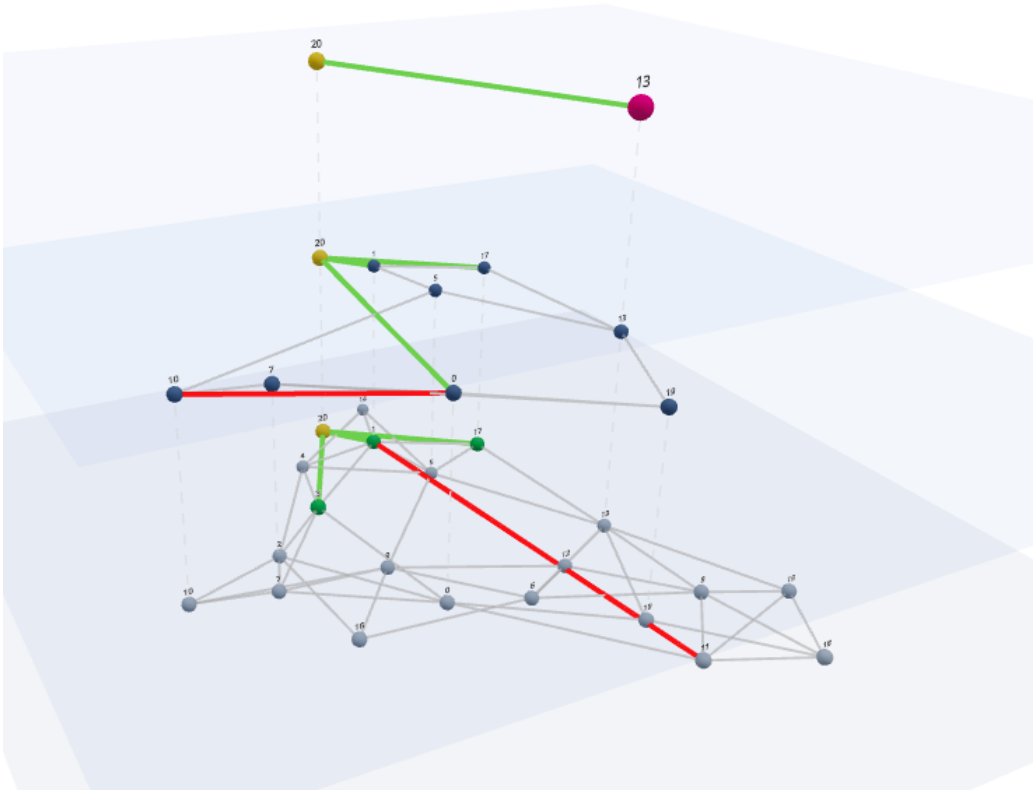


Hình 5.2: Đường dẫn tìm kiếm vector mục tiêu trong không gian HNSW

IV CƠ CHẾ THÊM PHẦN TỬ MỚI

?? minh họa chi tiết quy trình cập nhật cấu trúc đồ thị khi chèn một phần tử mới (nút màu vàng)[cite: 1487]. Tầng cao nhất (l) của phần tử này được xác định dựa trên hàm phân phối xác suất cấp độ, giúp duy trì cấu trúc phân tầng tối ưu cho đồ thị. Quy trình thiết lập liên kết được thực hiện qua các giai đoạn sau:

- **Tìm kiếm láng giềng (Neighbor Search):** Tại mỗi tầng từ l xuống tầng 0, thuật toán thực hiện tìm kiếm tập hợp M láng giềng gần nhất với nút mới thông qua giải thuật tham lam.
- **Thiết lập liên kết (Connection Establishment):** Các cạnh hai chiều mới (màu xanh) được khởi tạo để tích hợp phần tử mới vào mạng lưới hiện tại, đảm bảo tính kết nối và khả năng điều hướng của đồ thị.
- **Tối ưu hóa cấu trúc (Heuristic Pruning):** Trong trường hợp số lượng kết nối của một nút vượt quá ngưỡng M_{max}, thuật toán sẽ kích hoạt cơ chế tái cấu trúc (SELECT-NEIGHBORS-HEURISTIC).
- **Cắt tỉa cạnh (Edge Pruning):** Thuật toán thực hiện đánh giá lại các liên kết dựa trên cả khoảng cách và sự đa dạng về hướng, sau đó loại bỏ các cạnh thừa (màu đỏ) nhằm kiểm soát độ phức tạp bộ nhớ và duy trì hiệu suất truy vấn logarit.



Hình 5.3: Quá trình thêm nút và cập nhật cạnh trong HNSW

CHƯƠNG 6: 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, đạt độ trễ dưới 1ms
- **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**: 1,500,000+ hình ảnh từ Open Images V7
- **Tìm kiếm tài liệu**: 1,000,000+ bài báo khoa học từ arXiv
- **Tìm kiếm y tế**: 3,300+ 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 khoảng 285–490 MB RAM cho 100K vector, và khoảng 2.8–4.9 GB 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.