ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT MỞ RỘNG

KD-TREE AND HASHING

GVHD: Thầy NGUYỄN ĐỨC DŨNG
SV thực hiện: Lê Minh Nghĩa – 2010445

Tp. Hồ Chí Minh, Tháng 12/2021

Mục lục

Lời mở đầu:

Với sự xuất hiện của Internet, con người ngày càng sản sinh ra nhiều dữ liệu trong quá trình sống và sinh hoạt của mình. Điều này dẫn đến việc các chương trình ngày nay phải lưu trữ và làm việc với một lượng dữ liệu lớn và đa chiều. Điều này dẫn đến nhu cầu trong việc tổ chức cũng như xử lý dữ liệu với số lượng lớn một cách nhanh chóng.

Các hệ quản trị cơ sở dữ liệu không gian (spatial database management systems) được tổ chức với mục đích là có thể trả lời các truy vấn dựa trên đặc tính không gian của dữ liệu một cách nhanh chóng. Ví dụ, một cơ sở dữ liệu không gian có thể tổ chức vị trí của các tòa nhà thu được từ hình ảnh vệ tinh thành các ô vuông nhằm hỗ trợ cho việc tìm kiếm, truy xuất của một địa chỉ cụ thể.[4]

Bài báo cáo này sẽ tập trung vào dữ liệu là các điểm trong không gian ba chiều với độ chính xác là 0.001, tầm giá trị trong khoảng [0,100] trên mỗi chiều. Các cấu trúc dữ liệu sẽ được sử dụng là KD-Tree và Hashing nhằm hỗ trợ trong việc truy vấn ra điểm gần nhất với điểm đã cho (Nearest Neighbor Search) dựa vào khoảng cách Euclid.

1 Định nghĩa - Giới thiệu

1.1 KD-Tree

KD-Tree là một cây nhị phân mà mỗi node là một điểm trong không gian k-chiều. Mỗi node sẽ được xem như là một mặt phẳng chia không gian ra thành hai phần riêng biệt rồi từ đó chia các điểm về hai nhánh của cây. Sau khi thực hiện một cách đệ quy bằng một cách chọn nào đó, chương trình sẽ chia không gian ra thành các ô riêng biệt.[5]

1.1.1 Tạo cây KD-Tree

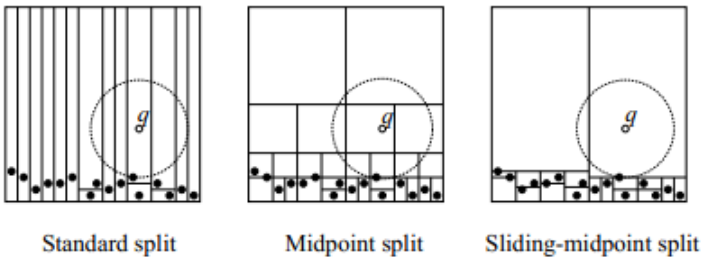
Do có nhiều cách chọn chiều không gian cũng như chọn điểm phân tách không gian, vậy nên cũng sẽ có rất nhiều cách để dựng KD-Tree.

Chọn chiều phân tách trong không gian: cách đơn giản nhất là chương trình sẽ đi lần lượt từ trục x, trục y và trục z rồi sau đó quay lại trục x. Ngoài ra, một cách chọn khác chính là chương trình sẽ chọn theo chiều có độ phân tán dữ liệu cao nhất.

Cách chọn giá trị phân cách: giá trị phân cách có thể chọn dựa vào trung vị của các điểm trên chiều đó hoặc dựa vào điểm chính giữa trên trục trong ô đang xét.

Một số cách phân tách không gian phổ biến:

- Standard split (chọn chiều không gian có độ phân tán dữ liệu cao nhất và giá trị phân tách sẽ là trung vị của các điểm trên chiều ấy)
- Midpoint split (chiều phân tách sẽ là chiều dài nhất và mặt phân tách sẽ luôn đi qua tâm của ô đang xét)
- Sliding midpoint split (mặt phân tách chọn giống như midpoint split. Tuy nhiên, khi các điểm chỉ có ở một phía của mặt phân tách thì mặt phân tách sẽ dịch về hướng đó sao cho điểm đầu tiên trở thành node lá)



Hình 1: *Các cách phân tách khi xử lý cụm dữ liệu[3]*

Trong bài báo cáo này, chương trình sẽ đảo qua chiều phân tách lần lượt là các trục x,y,z,x,... và giá trị phân tách là giá trị trung vị của các điểm đang xét trên chiều đã chọn. Do chương trình sẽ phải sắp xếp lại các điểm để chọn ra giá trị trung vị nên thời gian để tạo nên cây KD-Tree sẽ có độ phức tạp thời gian là $O(nlog(n)^2)$ để có thể tạo ra cây KD-Tree cân bằng (balanced).

1.1.2 Tìm kiếm trên cây KD-Tree

Chương trình sẽ tìm được điểm trên cây có khoảng cách gần nhất với một điểm được đưa vào một cách hiệu quả nhờ vào việc tìm kiếm có chọn lọc các node trên cây nhờ vào đặc tính của KD-Tree. Quá trình tìm kiếm sẽ diễn ra như sau với một điểm đầu vào:

- Bắt đầu từ gốc của cây, chương trình sẽ dịch chuyển dần xuống các node lá như một cây BST bình thường với giá trị kiểm tra là tọa độ tại chiều đang xét ở bậc đó.
- Tại node lá, chương trình sẽ kiểm tra khoảng cách giữa điểm ban đầu với điểm tại đó có phải là nhỏ nhất hay không.

- Trong quá trình quay trở lại của hàm đệ quy, chương trình sẽ phải kiểm tra xem các điểm ở nhánh còn lại có phải điểm gần nhất hay không? Tuy nhiên, quá trình này có thể bỏ qua bằng cách so sánh khoảng cách ngắn nhất với khoảng cách từ điểm đầu vào đến mặt phân cách. Nếu khoảng cách đến mặt phân cách là lớn hơn thì sẽ không có điểm nào ở nhánh còn lại gần hơn điểm hiện tại nên ta có thể bỏ qua.
- Điểm gần nhất sẽ được tìm ra khi chương trình quay trở lại node gốc.

Độ phức tạp về thời gian của việc tìm kiếm sẽ là O(log(n)). Tuy nhiên, trong trường hợp tệ nhất thì chương trình có thể chạy trong O(n). Điều này xảy ra khi chương trình không thể loại được nhánh nào.

1.1.3 Độ chính xác

Do chương trình sẽ gần như tìm kiếm hết tất cả các điểm và sẽ chỉ không xét khi chắc chắn các điểm bị loại sẽ không thỏa là điểm gần nhất nên ta có thể nói rằng độ chính xác gần như là 100%.

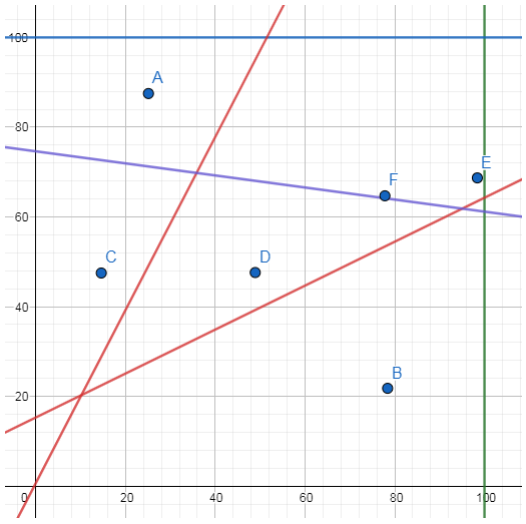
1.2 Hàm Hashing bằng cách chia không gian thành các vùng (cơ bản)

Với k hàm hash dưới dạng (p1,p2,p3,p4), ta sẽ tính được giá $y = p1 * x + p2 * y + p3 * z + p4$ của một điểm. Tùy vào giá trị $y >= 0$ hay $y < 0$ thì ta sẽ thành công trong việc tách không gian thành hai phần và xác định được điểm vừa chọn sẽ nằm trong phần nào. Lặp lại điều này với k hàm hash thì chương trình sẽ chia không gian làm 2^k phần. Tuy nhiên, do có thể xảy ra trường hợp hai điểm gần nhất sẽ nằm ở hai bên của một hàm hash, cho nên chương trình sẽ tạo ra L bảng hash để tăng độ chính xác.

1.2.1 Tạo bảng Hash

Đầu tiên, chương trình sẽ tạo ra k hàm hash bằng cách chọn ngẫu nhiên các giá trị thực trong khoảng từ (-10,10) theo phân phối đều và phần tử p4 sẽ trong khoảng (-1000,1000). Lặp lại với L bảng hash. Sau đó, chương trình sẽ đưa các điểm ban đầu trong không gian vào trong bảng hash. Vị trí của mỗi điểm sẽ được xác định bằng cách: Nếu hàm hash thứ n cho ra giá trị lớn hơn hoặc bằng 0 thì bit lớn thứ n sẽ là 1, ngược lại thì sẽ là 0. Sau đó chương trình sẽ lấy giá trị đó mod cho kích thước của bảng là một nửa số điểm, thu được vị trí của điểm trên bảng.

Vậy nên độ phức tạp thời gian khi tạo bảng hash sẽ là O(n) khi các giá trị L và k là hằng số.



Hình 2: *Hash không gian thành các vùng*

1.2.2 Search bằng bảng hash

Khi nhận một điểm đầu vào, chương trình sẽ xác định vị trí của điểm đó trong L bảng hash bằng cách sử dụng k hàm hash như khi gán các điểm ban đầu vào bảng. Tại vị trí ta thu được khi hash điểm đầu vào

sẽ tồn tại các điểm cùng nằm trong khu vực với điểm hash. Sau đó, chương trình sẽ tiến hành so sánh với các điểm trong vị trí đó trên bảng băm để tìm điểm gần nhất với điểm đầu vào. Điều này sẽ được lặp lại trong L bảng băm để tăng độ chính xác.

Do có giới hạn trong việc tìm kiếm thế nên thời gian tìm kiếm sẽ có độ phức tạp là O(1).

1.2.3 Độ chính xác

Trong phương pháp này, không gian sẽ bị chia thành nhiều phần và có thể tồn tại 2 điểm rất gần nhau nhưng bị tách ra bởi hai mặt phẳng. Vì vậy nên, ta sẽ tăng số lượng bảng băm nhằm hạn chế việc bỏ sót điểm. Tuy nhiên, để đảm bảo thời gian truy xuất, số lần so sánh sẽ bị giới hạn. Do các yếu tố trên nên độ chính xác của việc tìm điểm gần nhất bằng hash sẽ không được đảm bảo.

1.3 Hashing bằng cách sử dụng LSH(Locality-sensitive Hashing) dựa vào p-Stable Distributions

Mục đích của việc Hashing theo LSH chính là đưa các điểm gần nhau trong không gian vào chung một bucket để giảm thời gian truy vấn.

Đầu tiên, hàm hash chương trình sẽ sử dụng chính là

$$\frac{\langle z\,,x\rangle + t}{w}$$

- z: là một vector 3 chiều, mỗi giá trị được tạo ra một cách ngẫu nhiên từ phân phối chuẩn.
- x: vector thể hiện điểm trong không gian
- t: là một giá trị được tạo ngẫu nhiên từ phân phối đều
- w: giá trị giúp rút ngắn lại và gom lại các điểm

Việc nhân vô hướng vector x của một điểm với một vector ngẫu nhiên z nhằm thể hiện việc chiếu vector x lên vector z cộng thêm một giá trị t. Khi 2 điểm được chiếu lên một đường thì khoảng cách mới giữa 2 điểm sẽ nhỏ hơn hoặc bằng khoảng cách cũ giữa chúng trong không gian 3 chiều. Nói cách khác, thông qua việc nhân tích vô hướng với một vector ngẫu nhiên và chia w. Các điểm gần nhau sẽ được đưa vào cùng chung một tập hợp (lấy phần nguyên của kết quả làm vị trí của tập hợp).[1]

Sau k phép hash như thế, ta sẽ biết được vị trí của một điểm trong một ô nhất định trong không gian k-chiều tạo thành từ k vector trong phép hash cũng như giá trị t tương ứng. **Sau đó, chương trình sẽ ghép các tọa độ lại thành một string và hash, rồi mod theo một số nguyên tố lớn và theo độ lớn của bảng hash để tìm vị trí tương ứng trong bảng hash.**(phép Hash từ $Z^n \to Z$)

Tuy nhiên, do vẫn có thể tồn tại điểm gần nhất so với điểm đầu vào nhưng lại nằm ở 2 ô khác nhau nên chương trình sẽ làm việc với L hàm hash để tăng độ chính xác.

1.3.1 Tạo bảng Hash

Để thiết lập nên bảng hash dựa vào LSH, đầu tiên chương trình sẽ tạo k hàm hash cơ bản là các vector có giá trị của từng chiều là ngẫu nhiên theo phân phối chuẩn. Sau đó, chương trình sẽ tạo một dãy các số t để làm offset cho tích vô hướng và giá trị w. Sau đó, chương trình sẽ xây dựng hàm hash tăng cường cho L bảng hash.

Hàm hash tăng cường ứng với mỗi bảng hash sẽ là một hoán vị từ 0 đến k-1, nhằm làm cho các một hàm hash nhất định ở các bảng khác nhau sẽ có offset khác nhau.

Sau đó, ta sẽ hash lần lượt các điểm có sẵn trong không gian vào từng bảng hash trong L bảng hash.

Vậy nên độ phức tạp thời gian để tạo bảng Hash sẽ là O(n) khi L và k là hằng số.

1.3.2 Tìm kiếm trong bảng Hash theo phương pháp LSH

Để tìm kiếm điểm gần nhất với điểm đầu vào trong bảng hash LSH, chương trình sẽ tính ra vị trí của điểm đó trong bảng hash. Sau đó, chương trình sẽ truy xuất một số điểm tại vị trí đó để tìm điểm gần nhất. Quá trình này sẽ lặp lại với L bảng hash.

Độ phức tạp thời gian khi tìm kiếm trên bảng hash sẽ là O(1) do có giới hạn trong số lần tìm kiếm.

1.3.3 Độ chính xác

Do bảng hash hoạt động theo cách chia không gian thành các phần, thế nên vẫn sẽ tồn tại trường hợp 2 điểm gần nhau nhưng nằm ở 2 vị trí khác nhau. Giá trị w càng lớn sẽ giúp cải thiện độ chính xác, tuy nhiên do số điểm trong một ô hash sẽ tăng lên nên sẽ ảnh hưởng đến tốc độ truy vấn.

2 So sánh hiệu năng của các phương pháp

2.1 Các biến của chương trình

2.1.1 KD-Tree

- root: Biến chứa con trỏ dẫn đến cây KD-Tree của chương trình.

2.1.2 Hàm Hash cơ bản:

- L_PosNeg: Biến chứa tổng số bảng Hash của hàm.
- k_PosNeg: Biến chưa tổng số hàm hash (Tổng số mặt phẳng sẽ dùng để tách không gian thành các vùng(bin))
- mainHashTable_PosNeg: Bảng chính dùng để chứa L bảng hash, mỗi bảng chứa 1e5 ô.
- base2: Vector gồm L phần tử chứa các hàm hash cho L bảng hash.

2.1.3 Hàm Hash bằng LSH:

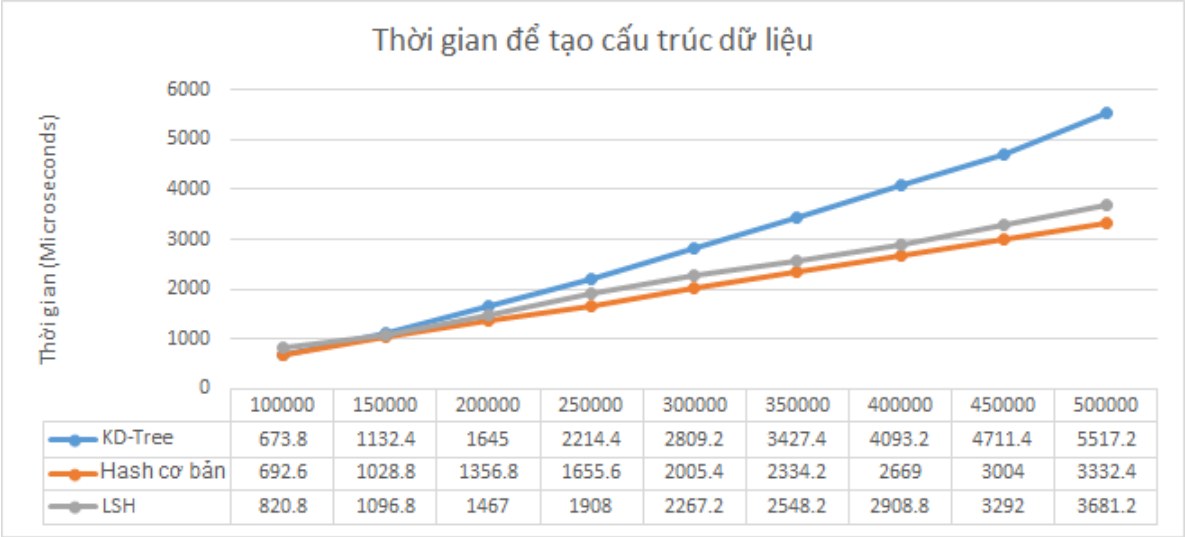
- L_LSH: Biến chứa số lượng bảng Hash.
- k_LSH: Biến chứa số lượng hàm Hash của mỗi bảng.
- mau_LSH: Biến w dùng để gom các giá trị vào các ô trên bảng Hash.
- lim: Giới hạn tìm kiếm cho mỗi bảng Hash trong L bảng.
- mainHashTable_LSH: Bảng chính chứa L bảng hash, mỗi bảng sẽ có số ô là một nửa tổng số điểm trong tập mẫu.
- outerHash: Bảng chứa hàm Hash tăng cường
- hashFuncs_LSH: Bảng chứa k phép hash sẽ được sử dụng qua hàm Hash tăng cường.

2.2 So sánh hiệu năng

Lưu ý:

- Hàm Hash cơ bản sẽ có số bảng Hash(L) là 3 và số hàm Hash(k) là 50.
- Hàm Hash bằng LSH sẽ có số bảng Hash(L) là 3, số hàm Hash(k) là 4, giá trị w là 500 và giới hạn tìm kiếm sẽ là 10 điểm trên mỗi bảng.

2.2.1 Tạo cơ sở dữ liệu(Building):

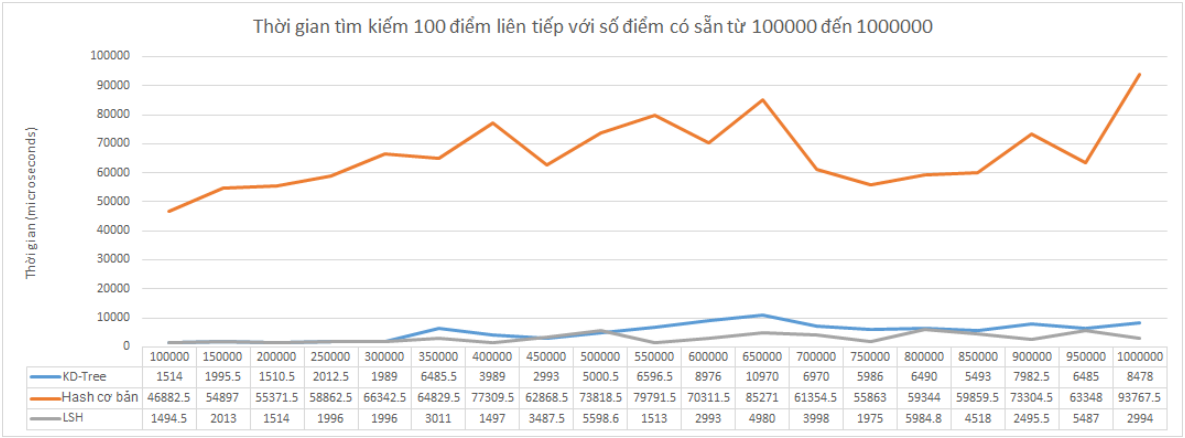


	100000	150000	200000	250000	300000	350000	400000	450000	500000
KD-Tree	673.8	1132.4	1645	2214.4	2809.2	3427.4	4093.2	4711.4	5517.2
Hash cơ bản	692.6	1028.8	1356.8	1655.6	2005.4	2334.2	2669	3004	3332.4
LSH	820.8	1096.8	1467	1908	2267.2	2548.2	2908.8	3292	3681.2

Hình 3: *Thời gian cần thiết để tạo cấu trúc dữ liệu(micro giây)*

Nhận xét: Độ phức tạp thời gian khi tạo một cây KD-Tree là O(nlog(n)[2]). Còn đối với các hàm hash sẽ là O(n). Điều này được thể hiện rõ ràng khi số điểm trong tập mẫu tăng lên. Khi số lượng điểm nhỏ thì thời gian cần thiết để xây dựng cấu trúc dữ liệu cần thiết giữa ba phương pháp không có sự cách biệt quá lớn. Nhưng khi số điểm trong tập mẫu tăng đến 200000 điểm thì dần xuất hiện sự khác biệt và khác biệt ngày càng lớn khi tăng số điểm trong tập mẫu.

2.2.2 Thời gian tìm kiếm Nearest Neighbor của một điểm đầu vào

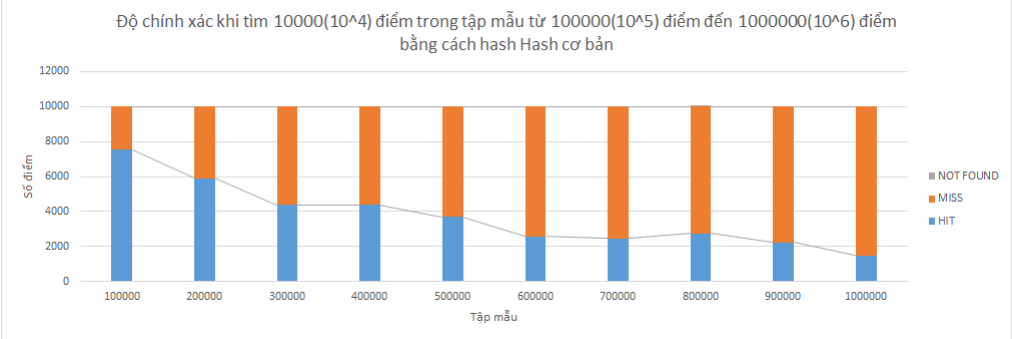


	100000	150000	200000	250000	300000	350000	400000	450000	500000	550000	600000	650000	700000	750000	800000	850000	900000	950000	1000000
KD-Tree	1514	1995.5	1510.5	2012.5	1989	6485.5	3989	2993	5000.5	6596.5	8976	10970	6970	5986	6490	5493	7982.5	6485	8478
Hash cơ bản	46882.5	54897	55371.5	58862.5	66342.5	64829.5	77309.5	62868.5	73818.5	79791.5	70311.5	85271	61354.5	55863	59344	59859.5	73304.5	63348	93767.5
LSH	1494.5	2013	1514	1996	1996	3011	1497	3487.5	5598.6	1513	2993	4980	3998	1975	5984.8	4518	2495.5	5487	2994

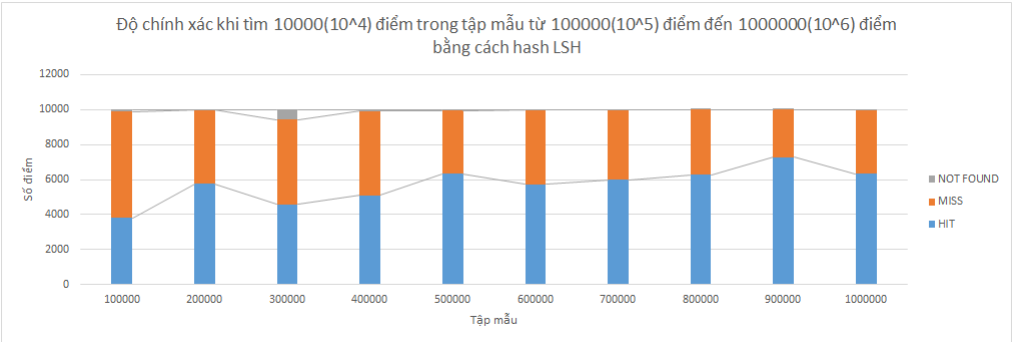
Hình 4: *Thời gian cần thiết đề tìm kiếm 100 điểm liên tiếp với tập điểm mẫu khác nhau (micro giây)*

Nhận xét: Qua đồ thị, thời gian tìm kiếm của cấu trúc KD-Tree và hàm Hash bằng LSH là gần như tương tự nhau, tuy thời gian dao động khá lớn. Đối với hàm Hash cơ bản, thì thời gian tìm kiếm là rất lớn so với hai cấu trúc còn lại.

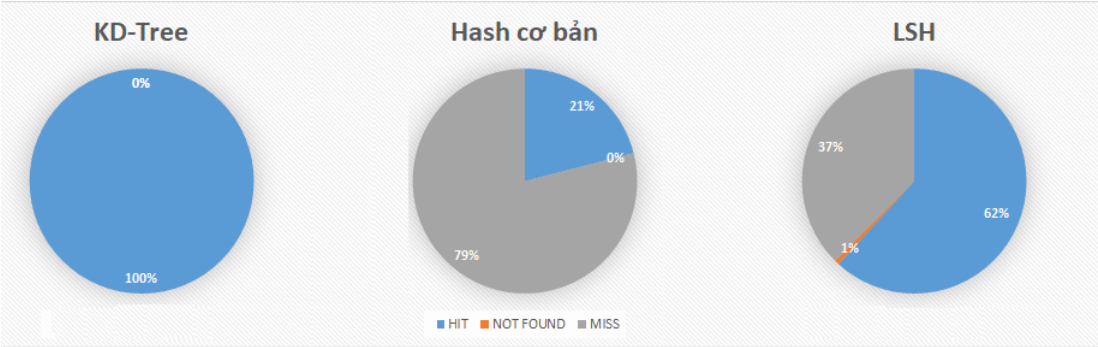
2.2.3 Độ chính xác



Hình 5: *Độ chính xác khi tìm kiếm 10000 lần với hàm Hash cơ bản(so với cách tìm kiếm Naive)*



Hình 6: *Độ chính xác khi tìm kiếm 10000 lần với hàm Hash LSH(so với cách tìm kiếm Naive)*



Hình 7: *Độ chính xác khi tìm kiếm 10000 lần với tập mẫu 10^6 điểm(so với cách tìm kiếm Naive)*

Độ lệch chuẩn:

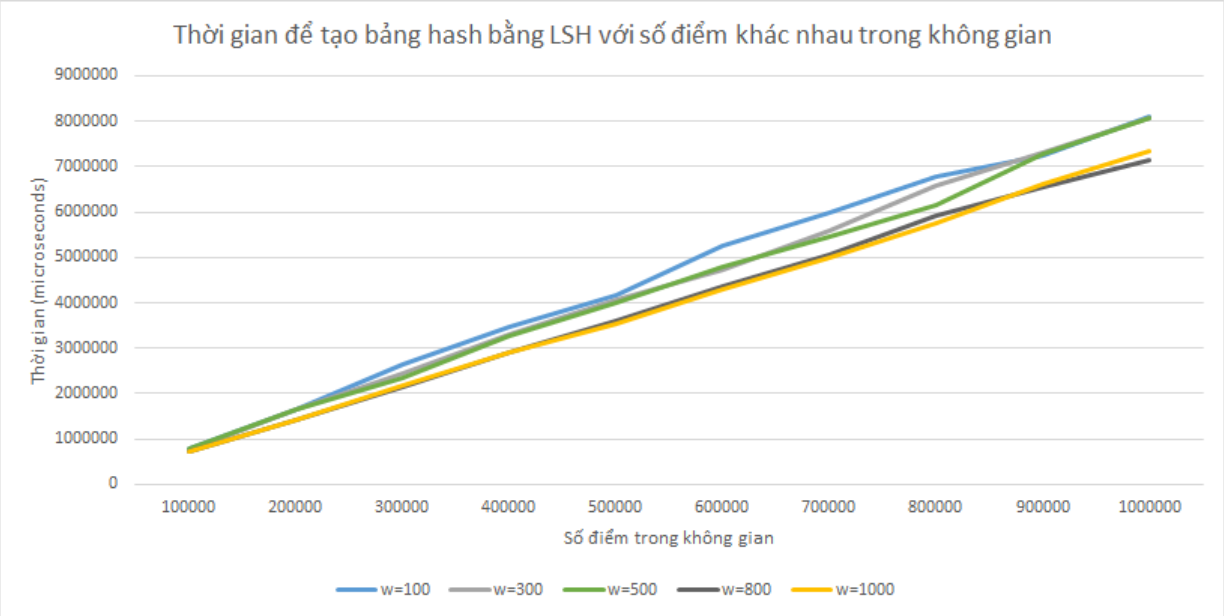
- KD-Tree: 0
- Hash cơ bản: 1.345104
- Hash LSH: 2.051865

Nhận xét: Qua đồ thị, ta có thể thấy được rằng cây KD-Tree có độ chính xác tốt hơn hẳn so với 2 cấu trúc dữ liệu còn lại, đạt đến 100%. Tiếp theo là hàm Hash bằng LSH với độ chính xác lên đến 62% với độ chính xác tăng dần khi tăng số điểm trong tập mẫu. Còn lại thì hàm hash cơ bản có độ chính xác thấp nhất trong 3 cấu trúc dữ liệu, ở 21% và giảm dần khi tăng số điểm trong tập mẫu. Trong trường hợp kết quả bị lệch so với phương pháp tìm kiếm từng điểm thì độ lệch chuẩn của các kết quả sẽ chỉ nằm trong khoảng từ 1-2 đơn vị.

2.3 So sánh hiệu năng của hàm LSH khi w thay đổi

Giá trị L và k của các hàm Hash sẽ lần lượt là 3 và 4. Giá trị w sẽ thay đổi trong khoảng từ 100 đến 1000.

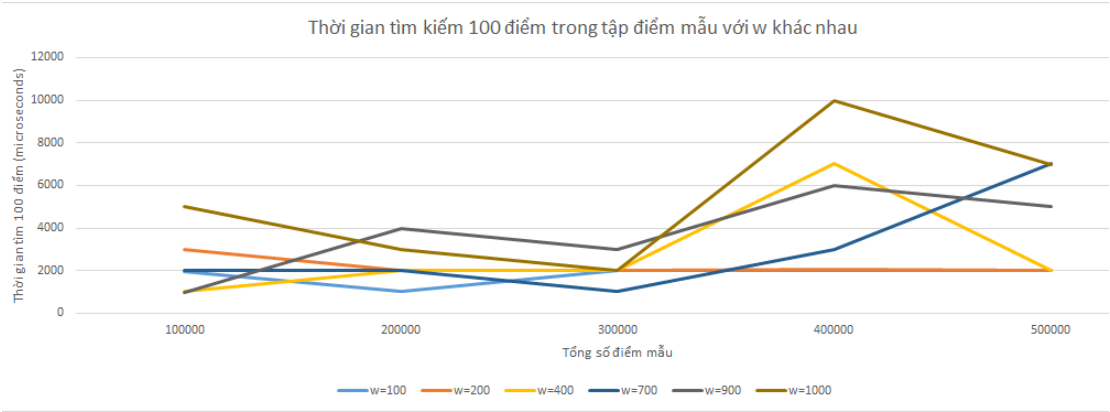
2.3.1 Tạo bảng Hash:



Hình 8: *Thời gian để tạo cấu trúc dữ liệu với giá trị w thay đổi*

Nhận xét: Với giá trị w tăng dần thì thời gian cần thiết để tạo bảng Hash sẽ không có quá nhiều sự thay đổi và nhìn chung vẫn theo độ phức tạp O(n).

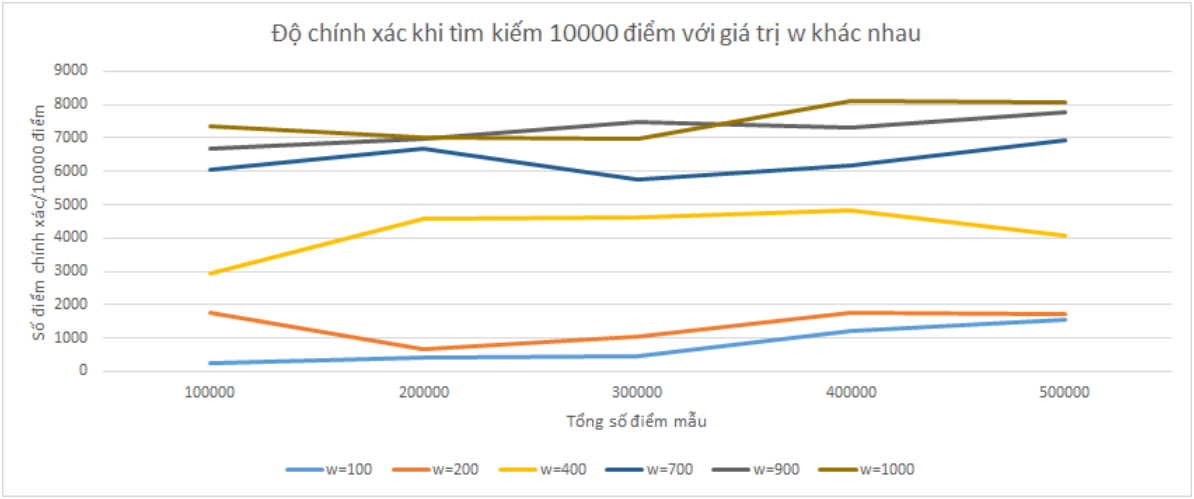
2.3.2 Thời gian tìm kiếm Nearest Neighbor:



Hình 9: *Thời gian để truy vấn 100 điểm liên tiếp khi w thay đổi*

Nhận xét: Với giá trị w tăng dần thì việc truy vấn Nearest Neighbor của một điểm, tuy có chênh lệch, nhưng nhìn chung sẽ không có thay đổi gì đáng kể. Sự truy vấn vẫn sẽ có độ phức tạp thời gian là O(1) do vẫn bị giới hạn số lần so sánh.

2.3.3 Độ chính xác:



Hình 10: *Độ chính xác của phương pháp với w thay đổi khi so với phương pháp Naive*

Nhận xét: Khi chương trình tăng dần giá trị w thì độ chính xác của quá trình tìm kiếm điểm gần nhất với một điểm cho trước cũng sẽ tăng dần khi đem so sánh với phương pháp kiểm tra tất cả các điểm trong tập mẫu.

Tuy nhiên, khi giá trị w tăng đến 900 thì và 1000 thì độ chính xác sẽ không có sự cái thiện đáng kể, do số điểm được chọn để so sánh trong một bảng Hash là có giới hạn để giữ tốc độ truy vấn là tốt nhất.

3 Kết luận

3.1 Về cây KD-Tree và hàm Hash bằng LSH

Qua kết quả từ phần so sánh hiệu năng của báo cáo, thời gian cần thiết để tạo nên cấu trúc dữ liệu của KD-Tree sẽ tăng một cách nhanh chóng khi so sánh với hàm Hash LSH khi số điểm trong tập mẫu tăng lên.

Tuy nhiên, về việc truy vấn Nearest Neighbor của một điểm đầu vào thì cả hai cấu trúc dữ liệu đều có thời gian tìm kiếm gần như là bằng nhau. Về độ chính xác khi so sánh với phương pháp tìm kiếm từng điểm thì ta thấy rằng truy vấn bằng cây KD-Tree sẽ có độ chính xác hơn hẳn so với tìm kiếm bằng Hash LSH.

Về việc thêm vào, xóa đi và thay đổi dữ liệu thì hàm Hash LSH có một lợi thế rõ ràng hơn so với cây KD-Tree. KD-Tree có mặt phân tách ở một bậc trên cây là khác nhau, nên việc điều chỉnh một node ở trên cây sẽ gặp rất nhiều khó khăn (không thể điều chỉnh bằng các phương pháp như rotation được). Trong khi đó, do hàm Hash LSH được xây dựng dựa trên phương pháp Hash nên việc chuyển đổi để chứa dữ liệu động sẽ diễn ra dễ dàng hơn.[1]

Tổng quan:

- Khi xét đến các điểm trong không gian 3 chiều, tuy có bất lợi về mặt tạo ra cấu trúc dữ liệu, KD-Tree có lợi thể hơn hẳn so với phương pháp truy vấn bằng hàm Hash LSH ở độ chính xác. Vì vậy nên, đối với các dữ liệu cơ bản ở không gian 3 chiều thì cây KD-Tree hiệu quả hơn hẳn trong việc xử lý.
- Tuy nhiên, khi xử lý dữ liệu động (dữ liệu có sự thay đổi theo thời gian) thì hàm Hash LSH sẽ có ưu thế hơn hẳn nhờ vào khả năng có thể thêm và xóa điểm một cách dễ dàng. Vì vậy nên, phương pháp này sẽ được ứng dụng vào các hệ thống xử lý dữ liệu thời gian thực.

3.2 The curse of dimensionality

Khi số chiều của dữ liệu tăng lên thì việc tìm kiếm trên cây KD-Tree sẽ không còn hiệu quả nữa. Điều này là do khi số chiều trong không gian tăng lên thì trọng số của mỗi chiều là như nhau, nên khoảng cách giữa 2 điểm khi xét trên một chiều sẽ ít ảnh hưởng đến khoảng cách thực tế khi xét trong không gian. Vậy nên, phương pháp tìm kiếm của cây KD-Tree dựa vào khoảng cách từ điểm đang xét sẽ không còn hiệu quả trong việc loại bỏ được các điểm phải xét để ra được kết quả. Vì vậy nên độ phức tạp của việc tìm kiếm sẽ tiến đến O(n) khi số chiều tăng lên.[2][6, Chapter 39]

Tuy nhiên, đối với phương pháp Hash LSH thì sẽ không gặp phải vấn đề như trên khi xét các điểm trong không gian nhiều chiều. Phương pháp này sẽ chỉ tìm kiếm điểm gần nhất trong phần(bucket) mà điểm đầu vào được hash vào. Nhờ vậy nên mà hàm Hash này sẽ tránh được việc thời gian tìm kiếm tiến đến thời gian O(n) như những cấu trúc dữ liệu cây, phân tách không gian khác. [1]

3.3 Về độ chính xác của phương pháp Hash LSH

Theo như thống kê ở trên, một điểm yếu của phương pháp Hash LSH chính là độ chính xác của việc truy vấn Nearest Neighbor của một điểm không cao như của KD-Tree. Tuy nhiên, độ lệch của những điểm này chỉ nằm ở trung bình là 2 đơn vị. Vậy nên, có thể xem các điểm đó hoàn toàn có thể thay thế điểm chính xác. Trong trường hợp mà những điểm có sự sai lệch nhỏ như thế mà có chất lượng giảm hơn hẳn so với điểm gần nhất với đầu vào, thì điểm gần nhất sẽ không ổn định và việc tìm kiếm điểm gần nhất đó có thể sẽ không có tác dụng đáng kể.[1]

3.4 So sánh hiệu quả khi thay đổi độ lớn của bin trong phương pháp Hash LSH

Đối với giá trị L và k cố định:

- Khi giá trị w là rất nhỏ thì các điểm trong tập mẫu sẽ được phân bố một cách dàn trải trên bảng băm. Điều này sẽ dẫn đến việc là các điểm nằm chung ô Hash với điểm đầu vào sẽ là những trường hợp xảy ra đụng độ. Điều này giải thích vì sao độ chính xác của những bảng Hash có giá trị w nhỏ thường rất thấp và độ lệch cao.
- Khi giá trị w tăng dần lên thì các điểm gần nhau sẽ có xu hướng được hash vào cùng một ô. Vậy nên, độ chính xác của việc truy xuất sẽ tăng dần theo giá trị w.
- Tuy nhiên, khi giá trị w tăng lên quá cao thì các điểm sẽ có sự phân bố tập trung vào một số ô nhất định mà không dàn trải ra. Vì vậy nên, nếu như chương trình không đặt ra giới hạn cho số điểm để xét trong một bucket thì thời gian truy vấn sẽ tăng lên đáng kể mà không có sự cái thiện đáng kể về độ chính xác.

Tài liệu

[1] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262, 2004.

[2] Sariel Har-Peled, Piotr Indyk, and Rajeev Motwani. Approximate nearest neighbor: Towards removing the curse of dimensionality. *Theory of computing*, 8(1):321–350, 2012.

[3] Songrit Maneewongvatana and David M Mount. It's okay to be skinny, if your friends are fat. In *Center for geometric computing 4th annual workshop on computational geometry*, volume 2, pages 1–8, 1999.

[4] Apostolos N Papadopoulos and Yannis Manolopoulos. *Nearest Neighbor Search:: A Database Perspective*. Springer Science & Business Media, 2006.

[5] Martin Skrodzki. The kd tree data structure and a proof for neighborhood computation in expected logarithmic time. *arXiv preprint arXiv:1903.04936*, 2019.

[6] Csaba D Toth, Joseph O'Rourke, and Jacob E Goodman. *Handbook of discrete and computational geometry*. CRC press, 2017.