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

Cấu trúc dữ liệu và giải thuật - CO2003

Bài tập lớn 2

CẢI TIẾN GIẢI THUẬT kNN BẰNG CẤU TRÚC DỮ LIỆU kD-TREE

TP. HỒ CHÍ MINH, THÁNG 04/2024

ĐẶC TẢ BÀI TẬP LỚN

Phiên bản 1.1

1 Chuẩn đầu ra

Sau khi hoàn thành bài tập lớn này, sinh viên ôn lại và sử dụng thành thục:

- Lập trình hướng đối tượng.
- Các cấu trúc dữ liệu cây.
- Giải thuật sắp xếp.

2 Dẫn nhập

Trong bài tập lớn 1, sinh viên được yêu cầu hiện thực giải thuật phân loại k-nearest neighbors(kNN) đơn giản bằng giải thuật Brute-force. Tuy nhiên, độ phức tạp về thời gian của giải thuật này là O(k * n * d) vì ta cần vòng lặp d lần để tính chiều dài, n vòng lặp để tính khoảng cách giữa các điểm trong tập dữ liệu và k lần để xác định k điểm gần nhất.

Trong bài tập lớn này, sinh viên được yêu cầu hiện thực cấu trúc dữ liệu dạng cây được gọi là kD-Tree để giảm độ phức tạp thời gian cho thuật toán kNN. Từ đó, sinh viên được yêu cầu hiện thực lại class kNN mới sử dụng cấu trúc dữ liệu kD-Tree đã hiện thực và ứng dụng trên bộ dữ liệu MNIST từ bài tập lớn 1.

Các thông tin về tập dữ liệu MNIST, thuật toán k-nearest neighbors, khoảng cách Euclidean, quá trình huấn luyện và dự đoán đã được mô tả cụ thể ở bài tập lớn 1.

3 k-D Tree

Trước khi giới thiệu về k-D Tree, ta cần nhắc lại về cây tìm kiếm nhị phân (BST). Cây tìm kiếm nhị phân (BST) là một cấu trúc dữ liệu phân cấp bao gồm các node, trong đó mỗi node có tối đa hai con, được gọi là con trái và con phải. Trong BST, con trái của một node chứa một giá trị nhỏ hơn giá trị của node và con phải chứa một giá trị lớn hơn giá trị của node. Đặc tính này làm cho các thao tác tìm kiếm, chèn và xóa trở nên hiệu quả, thường có độ phức tạp thời gian là O(log n) đối với các cây cân bằng.

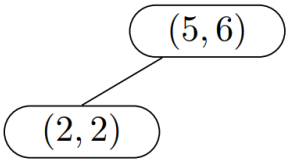
"k" trong k-D Tree đại diện cho số chiều của dữ liệu. Trong khi BST là một cấu trúc một chiều, k-D Tree là một sự mở rộng nhiều chiều của BST, được thiết kế để xử lý dữ liệu với nhiều chiều một cách hiệu quả.

Trong một cây k-D, mỗi node đại diện cho một điểm k chiều trong không gian. Giống như trong BST, mỗi node có một con trái và một con phải, nhưng việc xác định node con nằm ở bên trái hay bên phải được xem xét dựa trên một chiều cụ thể ở mỗi tầng của cây. Ví dụ, trong một cây 2-D (k-D Tree với k = 2): Khi chúng ta di chuyển từ tầng node gốc xuống tầng tiếp theo, chúng ta sẽ đi sang trái hoặc sang phải tùy thuộc vào giá trị x (chiều đầu tiên). Đối với tầng tiếp theo, chúng ta sẽ đi sang trái hoặc sang phải tùy thuộc vào giá trị y (chiều thứ hai). Sau đó, chúng ta chuyển lại sang x, sau đó là y, và tiếp tục như vậy. Giá trị để xác định tầng hiện tại sẽ chia theo chiều nào còn được loại là mặt phân chia. Các phần sau sẽ giới thiệu rõ hơn vì sao nó được gọi là mặt phân chia.

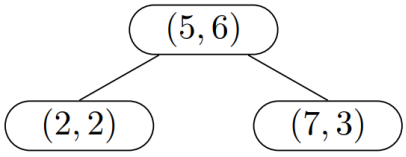
Để dễ hiểu, ta sẽ tham khảo ví dụ sau. Hãy chèn tất cả các điểm được liệt kê ở đây vào cây 2-D:

$$(5, 6), (2, 2), (7, 3), (2, 8), (8, 7), (8, 1), (9, 4), (3, 5)$$

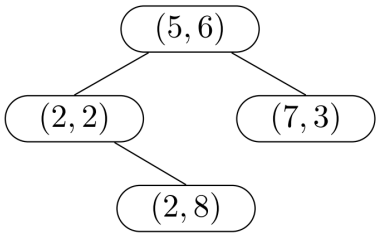
node gốc là (5, 6). Tầng gốc này chia theo giá trị x, vì vậy khi chúng ta chèn (2, 2), chúng ta đi về phía trái:



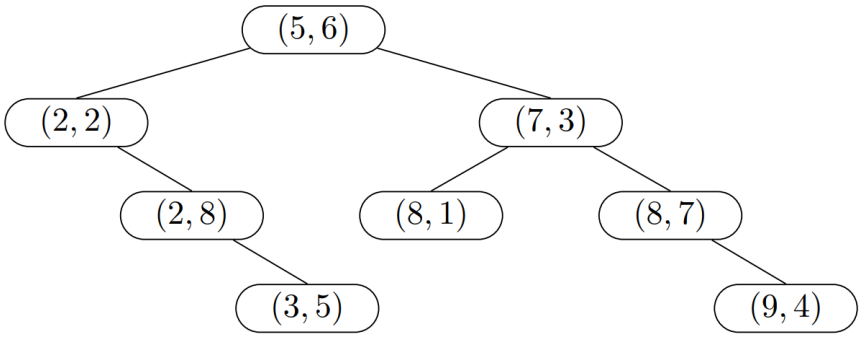
Khi chúng ta chèn (7, 3) vào, trước tiên chúng ta hỏi xem giá trị x của nó so với (5, 6) như thế nào. Nó lớn hơn, vì vậy chúng ta đi về phía phải và đặt nó ở đó:



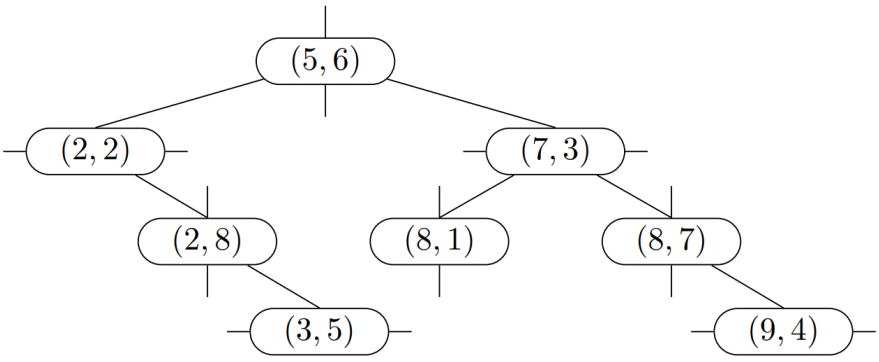
Bây giờ khi chúng ta chèn (2, 8) vào, trước tiên chúng ta hỏi xem giá trị x của nó so với (5, 6) như thế nào. Nó nhỏ hơn nên chúng ta đi về phía trái và đến (2, 2). Sau đó, chúng ta hỏi xem giá trị y của nó so với (2, 2) như thế nào. Nó lớn hơn, vì vậy chúng ta đi về phía phải và đặt nó ở đó:



Nếu chúng ta tiếp tục quá trình này, chúng ta sẽ có được cây cuối cùng:

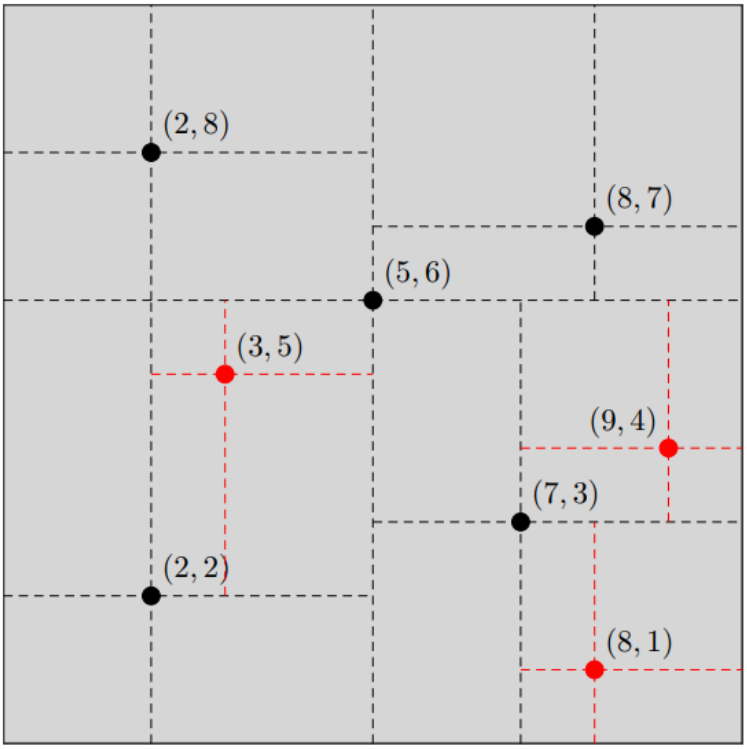


Thông thường, để theo dõi các mặt phân chia, chúng ta sẽ kí hiệu các node như sau. Chúng ta đặt một dấu gạch dọc để chỉ ra rằng chúng ta đang phân chia theo chiều x và một dấu gạch ngang để chỉ ra rằng chúng ta đang phân chia theo chiều y:



Quá trình xây dựng một cây k-D liên quan đến việc phân chia không gian theo các chiều khác nhau cho đến khi tất cả các điểm được đặt đúng. Trong quá trình tìm kiếm, cây được duyệt dựa trên các chiều, cho phép tìm kiếm hiệu quả trong nhiều chiều. Điều này làm cho cây k-D trở nên đặc biệt hữu ích trong các ứng dụng liên quan đến dữ liệu không gian, chẳng hạn như tìm kiếm láng giềng gần nhất, tìm kiếm phạm vi và tìm kiếm k láng giềng gần nhất.

Một cách biểu diễn khác của cây k-D Tree 2 chiều có thể như sau:



Khi ta đặt điểm (5, 6) là node gốc, node này chia mặt phẳng 2 chiều ra làm 2 bên trái phải theo trục dọc (trục x). Từ đó, những điểm nào có giá trị theo trục dọc bé hơn 5 thì sẽ nằm bên trái của trục được chia bởi điểm (5, 6); ngược lại sẽ nằm phía bên phải. Tương tự, xét điểm (2, 2) là con của (5, 6), ta thấy:

- Điểm này chia không gian là hình chữ nhật bên trái của điểm (5, 6)
- Điểm này cũng chia không gian bởi một trục nằm ngang (2, 2) mà con của nó sẽ là trục (3, 5).

Vì vậy nên, ta có thể gọi các điểm trong cây k-D Tree là các mặt phẳng chia vì nó chia không gian của các điểm trên một mặt phẳng.

Không chỉ là 2 chiều: Ở ví dụ trên, ta chỉ ví dụ với cây 2D. k-D Tree có thể hoạt động với dữ liệu nhiều chiều. Ví dụ, khi quản lý các điểm trong không gian 3D, chúng ta có thể liên tục lặp lại thứ tự phân chia của cây bằng x, sau đó y, sau đó z, sau đó x, sau đó y, và tiếp tục như vậy.

Trong bài tập lớn này, dữ liệu ảnh là 784 chiều, vì vậy ta sẽ phân chia cây k-D từ chiều đầu tiên tương ứng với cột 1x1, chiều thứ hai là cột 1x2, ..., và tiếp tục như vậy cho đến khi lặp lại.

Tọa độ bằng nhau: Theo quy ước, nếu chúng ta đang đi xuống cây để chèn một node và chúng ta đang ở một node nơi chúng ta phân chia bởi trục α và node mà chúng ta muốn chèn có cùng giá trị tọa độ thì chúng ta đi về bên phải.

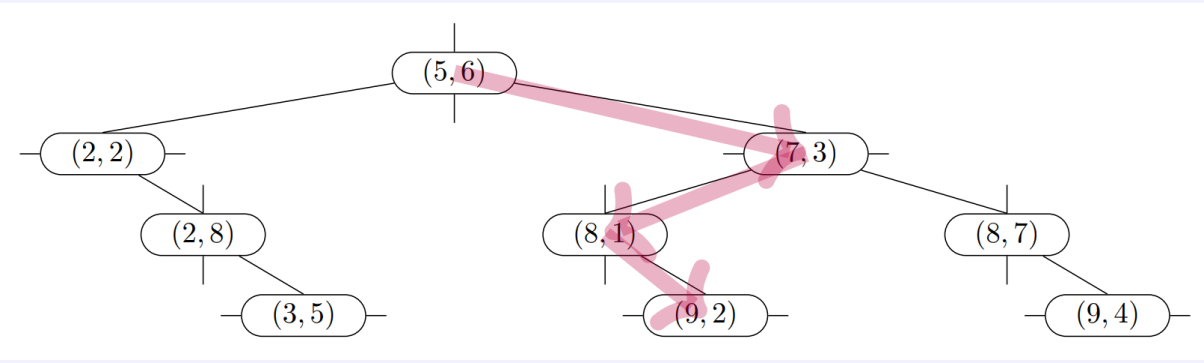
Ví dụ, nếu chúng ta đang chèn (20, 30) và chúng ta gặp node (20, 40) nơi chia theo trục x, thì chúng ta đi về phía bên phải.

3.1 Chèn

Việc chèn trong một cây k-d hoạt động chính xác như một cây nhị phân tìm kiếm với sự điều chỉnh là chúng ta trước tiên cần phải xác định mặt phẳng chia hiện tại. Từ đó, ta xác định nhánh của node cần thêm vào là bên trái hay bên phải của mặt phẳng chia. Sau khi di chuyển tới node lá (không thể đi xuống nhánh nào nữa), tùy thuộc vào mặt phẳng chia của node này để đặt node mới vào.

Ví dụ 3.1

Ví dụ, việc chèn (9, 2) vào cây k-d của chúng ta từ trước đó sẽ tiến hành như sau. Chúng ta bắt đầu tại node gốc và so sánh theo mặt phẳng chia x và đi về bên phải. Tại (7, 3), chúng ta so sánh với mặt phẳng chia y và đi về bên trái. Tại (8, 1), chúng ta so sánh với mặt phẳng chia x và chèn vào bên phải. Lưu ý rằng node được chèn sẽ có mặt phẳng chia phù hợp với tầng mà nó đang ở.



3.2 Xây cây

Ở phần giới thiệu, việc xây cây được thể hiện qua chèn các điểm vào cây rỗng theo thứ tự từ điểm đầu tiên tới điểm cuối cùng. Tuy nhiên, phương pháp này có thể khiến cho cây bị mất cân bằng.

Để có thể xây một cây k-D cân bằng, ta tuân theo giải thuật sau:

1. Khi di chuyển xuống cây, xác định mặt phẳng chia hiện tại.
2. Sắp xếp lại danh sách các điểm theo chiều của mặt phẳng chia hiện tại.
3. Xác định điểm tiếp theo được chèn vào là trung vị của danh sách sau khi được sắp xếp
4. Cây con bên trái của điểm mới được thêm vào sẽ là cây được xây bằng danh sách các điểm bé hơn điểm trung vị. Cây con bên phải của điểm mới được thêm vào sẽ là cây được xây bằng danh sách các điểm lớn hơn hoặc bằng điểm trung vị.

Chú ý rằng, để tìm ra điểm trung vị, ta có thể sử dụng nhiều phương pháp khác nhau. BTL này yêu cầu sinh viên phải tìm điểm trung vị bằng cách sử dụng **Mergesort** để sắp xếp lại danh sách, sau đó chọn điểm chính giữa là điểm trung vị. Nếu có 2 điểm chính giữa, ta sẽ chọn điểm đầu tiên trong 2 điểm đó.

Ví dụ 3.2

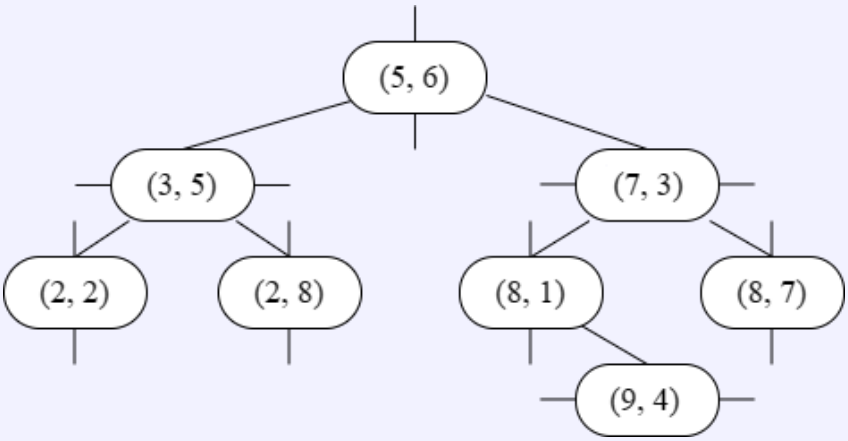
Với giải thuật được trình bày ở trên và danh sách các điểm được thêm vào là:

(5, 6), (2, 2), (7, 3), (2, 8), (8, 7), (8, 1), (9, 4), (3, 5)

Chiều đầu tiên là chiều x, ta tiến hành sắp xếp các điểm theo chiều x và thu được danh sách:

(2, 2), (2, 8), (3, 5), (5, 6), (7, 3), (8, 1), (8, 7), (9, 4)

Trung vị theo chiều x sẽ là điểm (5, 6). Điểm (5, 6) sẽ là node gốc của cây. Ta tiếp tục xây cây con bên trái của node gốc với danh sách là (2, 2), (2, 8), (3, 5), còn cây con bên phải node gốc sẽ được xây từ danh sách (7,3), (8, 1), (8, 7), (9, 4). Chiều tiếp theo của 2 cây con sẽ là chiều y.
Tiếp tục như vậy, đến cuối cùng ta sẽ thu được cây như sau:



3.3 Xóa

3.3.1 Tìm kiếm node thay thế

Khi thực hiện xóa trong các cây BST, chúng ta thường tìm kiếm thay thế bằng cách tìm node nhỏ nhất mà lớn hơn node bị xóa. Phần tử này dễ tìm - đi về phía bên phải (nếu có thể) sau đó đi về phía trái xa nhất có thể. Tuy nhiên, thuật toán này dựa trên việc chúng ta luôn phân chia theo một mặt phẳng chia duy nhất và với cây k-d, điều này không còn đúng nữa.

Quan sát cho thấy rằng khi chúng ta đang tìm kiếm node thay thế, chúng ta đang tìm kiếm node nhỏ nhất trong cây con bên phải của node bị xóa. Vì vậy, hãy tưởng tượng rằng chúng ta chỉ đơn giản có một cây k-d và muốn tìm một node có giá trị nhỏ nhất theo chiều α tương ứng với mặt phẳng chia của node bị xóa.

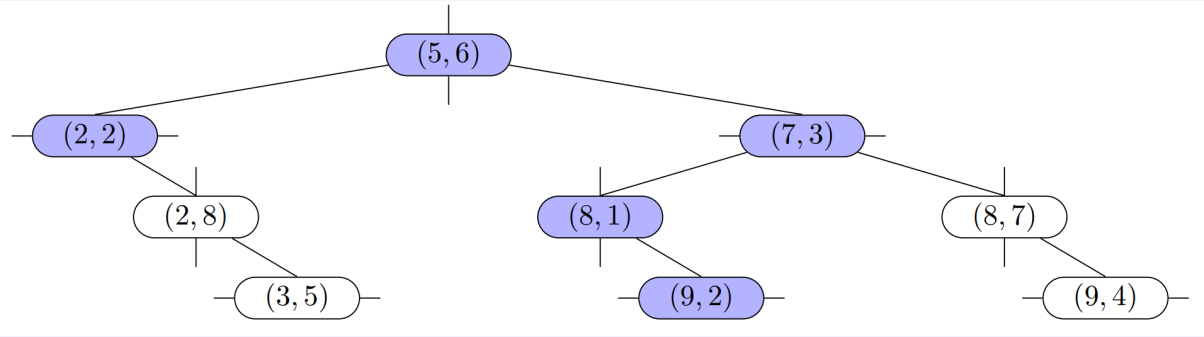
Giải thuật tìm kiếm node thay thế được mô tả một cách đệ quy như sau:

- Nếu chúng ta đang ở một node mà bị chia bởi α thì mục tiêu sẽ ở trong cây con bên trái. Nếu cây con bên trái đó là NULL thì chỉ cần trả về node hiện tại. Nếu không, tiếp tục đệ quy đến cuối cùng của cây con bên trái.
- Nếu chúng ta đang ở một node mà phân chia bởi chiều khác α thì mục tiêu có thể ở trong cả hai cây con hoặc ở chính node đó. Chúng ta đệ quy đến cả hai cây con, lấy kết quả từ cả hai cây con cũng như từ node chính nó và chọn một node có giá trị theo chiều α nhỏ nhất. Nếu có nhiều hơn một thì chúng ta chọn với độ ưu tiên là node chính nó, cây con trái và cuối cung là cây con phải.

Ví dụ 3.3

Giả sử bằng cây ở mục Chèn, chúng ta muốn tìm node có giá trị chiều y nhỏ nhất. Chúng ta bắt đầu từ (5, 6) nhưng vì nó phân chia theo tọa độ x nên chúng ta phải kiểm tra cả hai cây con.
Chúng ta đệ quy đến cây con bắt đầu từ (2, 2), lưu ý rằng (2, 2) phân chia theo tọa độ y nên chúng ta cần đi về phía bên trái. Tuy nhiên, chúng ta không thể đi về phía bên trái, vì vậy chúng ta đơn giản trả về (2, 2).
Chúng ta đệ quy đến cây con bắt đầu từ (7, 3), lưu ý rằng (7, 3) phân chia theo tọa độ y nên chúng ta cần đi về phía bên trái. Chúng ta đi về phía bên trái đến (8, 1) nhưng vì nó phân chia theo tọa độ x nên chúng ta phải kiểm tra cả hai cây con nhưng chỉ có một cây con, và nó trả về (9, 2).
Vì vậy, cây con của (8, 1) trả về giá trị nhỏ nhất giữa (8, 1) và (9, 2), trong đó tối thiểu

có nghĩa là tọa độ y nhỏ nhất, nên đó là (8, 1). Cây con của (7, 3) cũng trả về giá trị này. Cây con của (5, 6) trả về giá trị nhỏ nhất giữa (2, 2), (8, 1) và (5, 6), vì vậy đó là (8, 1). Điều này có thể được minh họa trong biểu đồ dưới đây, trong đó các node được tô màu là các node mà chúng ta thực sự cần phân tích.



3.3.2 Giải thuật xóa

Trong trường hợp 1-D (BST), quá trình này đơn giản - đi về phía bên phải và sau đó đi về phía bên trái xa nhất có thể để tìm phần tử kế tiếp theo theo thứ tự, thay thế node bị xóa bằng phần tử kế tiếp theo đó, sau đó hoặc nối lại cây (nếu phần tử kế tiếp theo không phải là lá) hoặc chỉ đơn giản là cắt bỏ node phần tử kế tiếp theo cũ (nếu nó là lá).

Điều này sẽ không ổn đối với k-D Tree vì nối lại cây sẽ làm cho phần dưới của nối lại có tất cả các mặt phẳng chia bị lộn xộn. Vậy chúng ta có thể làm gì?

Trước khi trình bày giải thuật xóa, chúng ta quy ước u_α có nghĩa là giá trị chiều α của node u. Ví dụ: $(17, 42, 100)_x = 17$, $(5, 6)_y = 6$ và cứ thế.

Quá trình xóa của chúng ta sẽ hoạt động như sau. Quá trình này là đệ quy, giả định node cần xóa là u:

(a) Nếu u là một lá: Chỉ cần xóa nó và kết thúc.

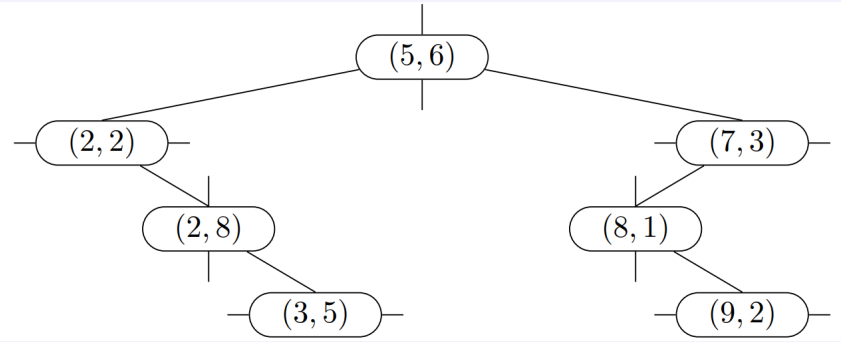
(b) Nếu u có cây con bên phải: Đặt α là chiều mà u phân chia. Tìm một node thay thế r trong $u.right$ (cây con bên phải của u) sao cho giá trị theo chiều α của r là nhỏ nhất. Chúng ta copy điểm r vào u (ghi đè lên điểm của u). Sau đó, chúng ta gọi đệ quy xóa trên node r cũ.

(c) Nếu u không có cây con bên phải thì nó sẽ có cây con bên trái: Đặt α là chiều mà u phân chia. Tìm một node thay thế r trong $u.left$ (cây con bên trái) sao cho giá trị theo chiều α của r là nhỏ nhất. Chúng ta copy điểm r vào u (ghi đè lên điểm của u) và sau đó, chúng ta di chuyển cây con bên trái của u để trở thành cây con bên phải của u. Sau đó, chúng

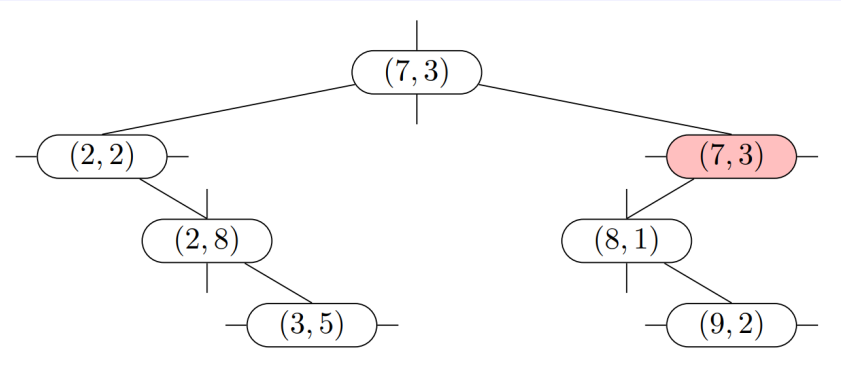
ta gọi đệ quy xóa trên node r cũ.

Ví dụ 3.4

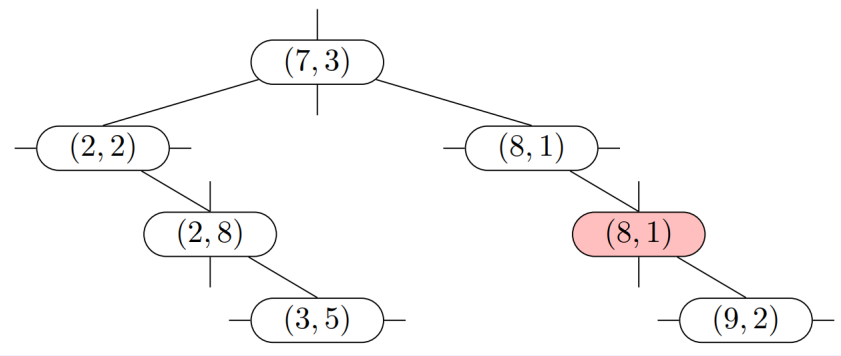
Hãy xóa (5, 6) từ cây này:



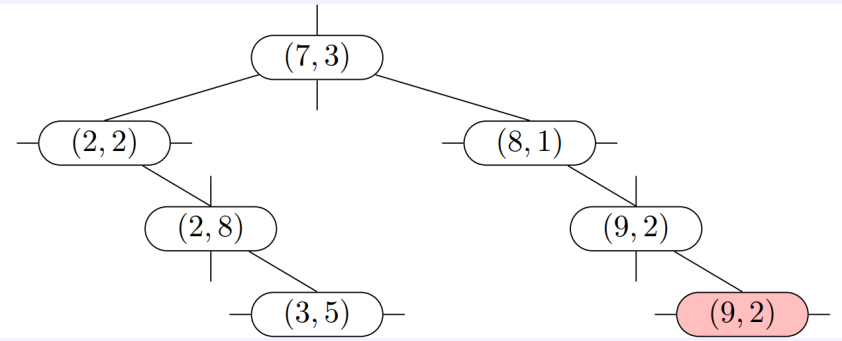
Vì (5, 6) phân chia theo chiều x và có một cây con bên phải, chúng ta tìm node trong cây con bên phải có tọa độ x nhỏ nhất, đó là (7, 3). Chúng ta thay thế (5, 6) bằng (7, 3). Tạm thời có hai bản sao của (7, 3), bản sao màu đỏ là bản sao cần phải được xóa.



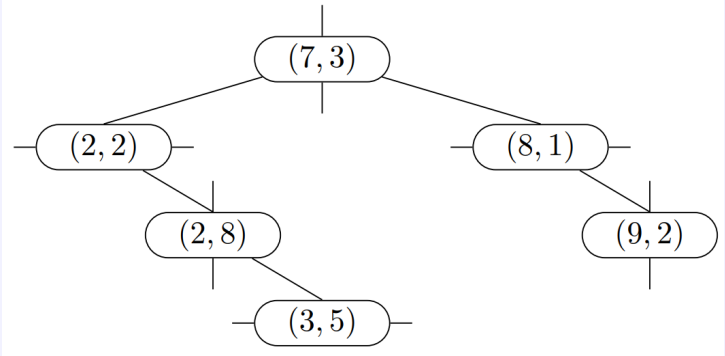
Vì (7, 3) màu đỏ phân chia theo chiều y nhưng không có cây con bên phải, chúng ta chọn node trong cây con bên trái có chiều y nhỏ nhất, đó là (8, 1). Chúng ta thay thế (7, 3) bằng (8, 1) và sau đó di chuyển cây con bên trái cũ sang bên phải: Tạm thời có hai bản sao của (8, 1), bản sao màu đỏ bản sao cần phải được xóa.



Vì (8, 1) màu đỏ phân chia theo chiều x và có một cây con bên phải, chúng ta chọn node trong cây con bên phải có chiều x nhỏ nhất, đó là (9, 2). Chúng ta thay thế (8, 1) bằng (9, 2). Tạm thời có hai bản sao của (9, 2), bản sao màu đỏ bản sao cần phải được xóa.



Vì (9, 2) màu đỏ là node lá nên chúng ta xóa nó đi và kết thúc:



3.4 Tìm kiếm

Tìm kiếm trong một cây k-d hoạt động chính xác như một cây nhị phân tìm kiếm với sự điều chỉnh là chúng ta đi theo nhánh theo mặt phẳng chia cho mỗi node.

3.5 Láng giềng gần nhất (nearest neighbor)

Tìm kiếm điểm láng giềng gần nhất (nearest neighbord) nhằm tìm điểm trong cây mà gần nhất với một điểm đầu vào cụ thể (gọi là điểm **mục tiêu**). Việc tìm kiếm này có thể được thực hiện một cách hiệu quả bằng cách sử dụng các thuộc tính của cây để loại bỏ nhanh chóng các phần lớn của không gian tìm kiếm.

Việc tìm kiếm điểm láng giềng gần nhất trong một cây k-d diễn ra như sau:

1. Bắt đầu tại node gốc của cây.

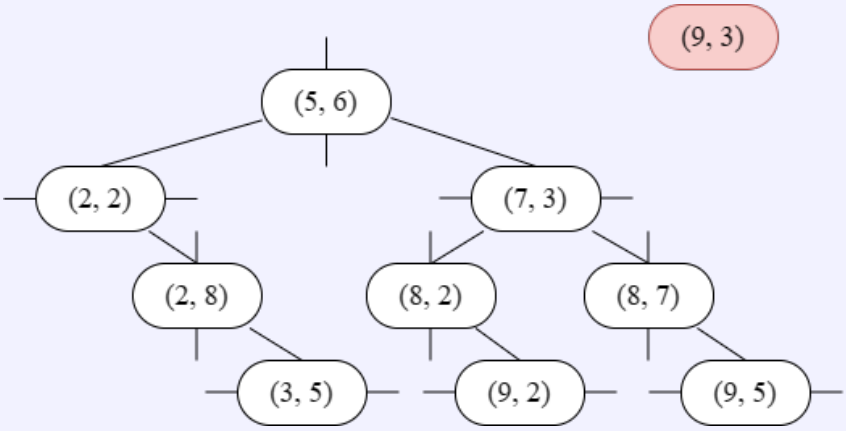
2. Di chuyển xuống cây theo đệ quy, tương tự như khi chèn một điểm, bằng cách so sánh tọa độ của điểm hiện tại với mặt phẳng chia tại mỗi node.
3. Khi đến một node lá (hoặc là không di chuyển được nữa), lưu trữ nó như là láng giềng gần nhất tiềm năng.
4. Đi ngược trở lại lên trên cây, kiểm tra từng node:
    - Nếu node hiện tại gần hơn láng giềng gần nhất đã lưu trữ, cập nhật láng giềng gần nhất.
    - Kiểm tra xem liệu có thể có các điểm gần hơn ở phía bên kia của mặt phẳng tách:
        - Hãy tưởng tượng một hình cầu xung quanh điểm tìm kiếm có bán kính bằng khoảng cách đến láng giềng gần nhất hiện tại. (1)
        - Nếu hình cầu này cắt mặt phẳng tách tại node hiện tại, hãy khám phá phía bên kia của cây từ node đó. (2)
        - Nếu không, tiếp tục đi ngược lên trên cây, loại bỏ nhánh ở phía bên kia.
5. Lặp lại quá trình này cho đến khi đến node gốc, hoàn thành tìm kiếm.

(1) Để dễ hiểu, ta gọi bán kính này là R, là khoảng cách từ láng giềng gần nhất hiện tại đến điểm tìm kiếm.

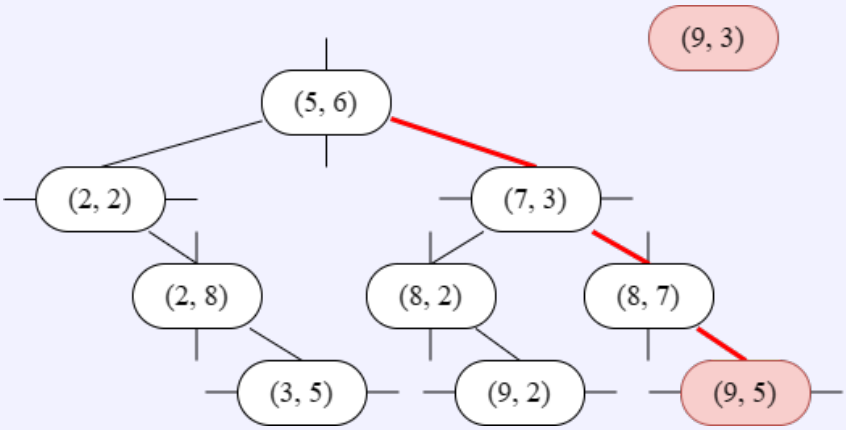
(2) Để dễ hiểu, gọi r là khoảng cách theo chiều hiện tại của điểm đang xét hiện tại với điểm tìm kiếm. Ví dụ, điểm tìm kiếm là (9, 3), điểm đang xét hiện tại là (8, 5) và mặt cắt của nó là chiều y. Khi này, r = 2. Hình cầu cắt mặt phẳng tách của node hiện tại khi R >= r.

Ví dụ 3.5

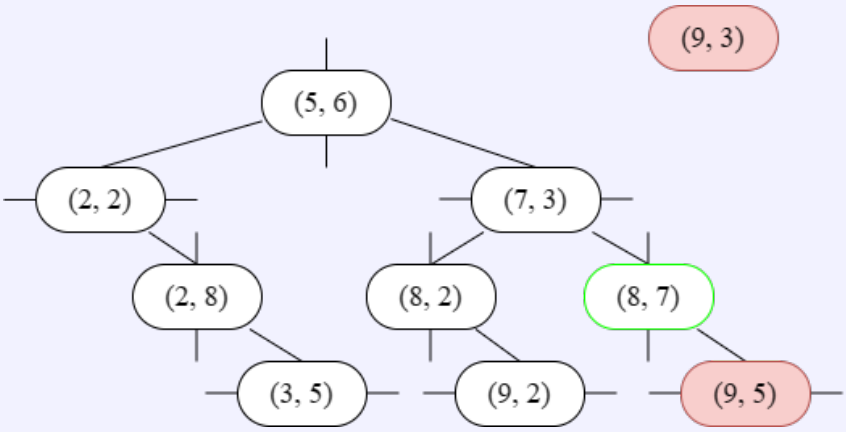
Xét ví dụ với cây k-D 2 chiều như sau và điểm tìm kiếm là (9, 3):



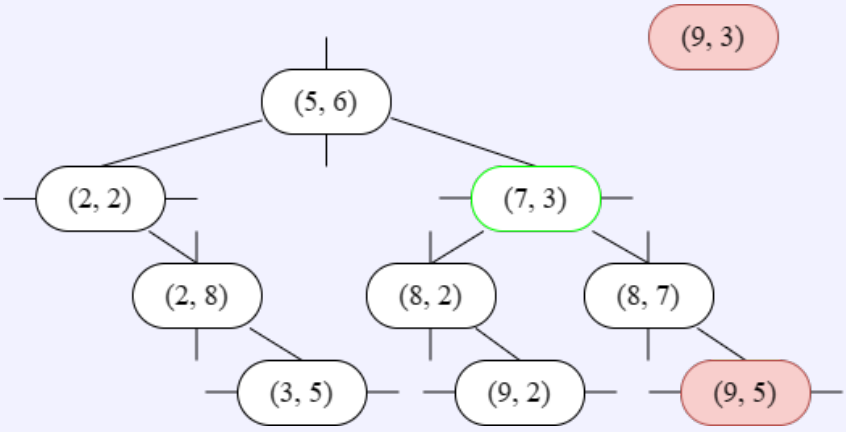
Theo giải thuật đến bước 3, ta có điểm gần nhất với điểm tìm kiếm là (9, 5):



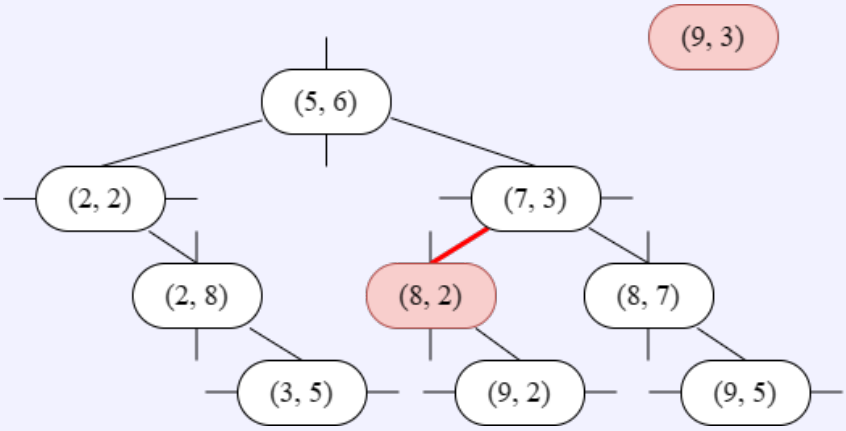
Từ điểm này, bắt đầu đi ngược lên điểm (8, 7), t có R = 2, khoảng cách giữa (8, 7) và (9, 3) lớn hơn R và r = 1 < R. Tuy nhiên nhánh trái của (8, 7) là NULL nên không làm gì cả, tiếp tục lên node phía trên:



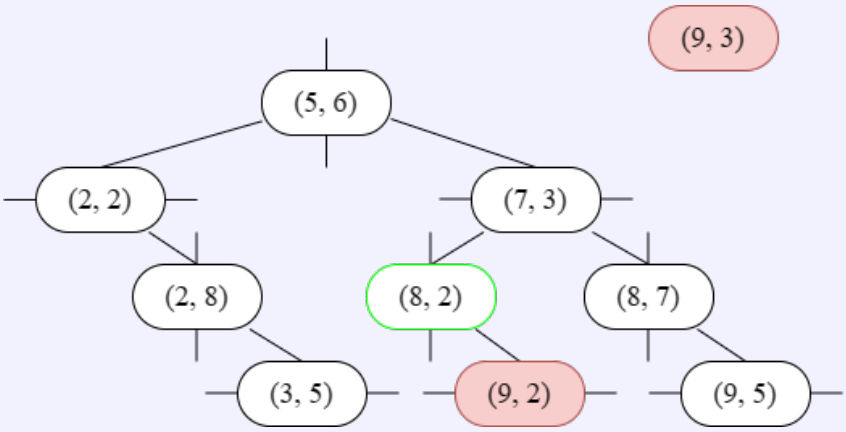
Xét điểm (7, 3), R = 2, khoảng cách giữa (7, 3) và (9, 3) không bé hơn R. r = 0 < R, ta sẽ xét nhánh còn lại của (7, 3), cụ thể là thực hiện tìm kiếm láng giềng gần nhất trên cây có đỉnh là (8, 2):



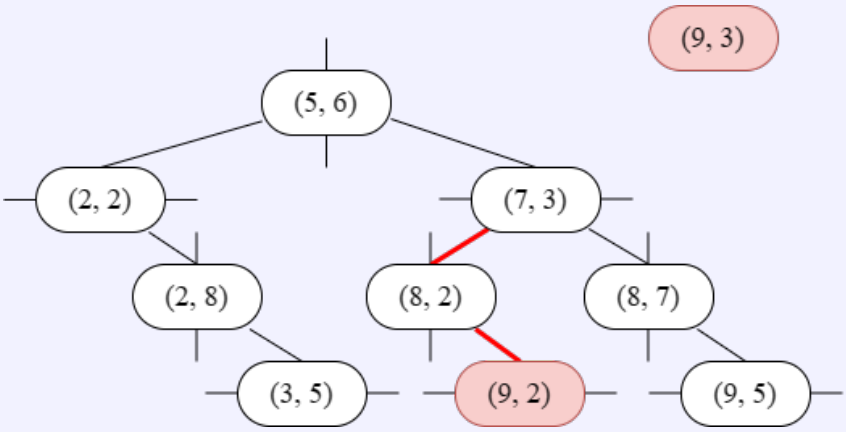
Thực hiện thao tác như bước 1 đến bước 3 cho cây có đỉnh là (8,2). Sau khi di chuyển tới điểm (9, 2), khoảng cách giữa (9, 2) và (9, 3) là $1 < R$ nên láng giềng gần nhất bây giờ là (9,2):



Đi ngược lên từ node (9, 2) là node (8, 2), R = 1, khoảng cách giữa (8, 2) và (9, 3) là $\sqrt{2} > 1$. r = 1 == R nhưng (8, 2) không có cây con bên trái nên không xét tiếp, tiếp tục quay ngược lên trên:



Ta bỏ qua điểm (7, 3) vì đã tìm kiếm đầy đủ các nhánh của điểm này, tiếp tục đi ngược lên điểm (5, 6). Dễ dàng tính được giải thuật đã dừng ở điểm (5, 6):



Giải thuật nearest neighbor trả về điểm (9, 2).

4 Yêu cầu

Trong bài tập lớn này, sinh viên được yêu cầu phải thực hiện các class bên dưới để mô tả quá trình xử lí dữ liệu và hiện thực thuật toán k-nearest neighbor sử dụng cấu trúc k-D Tree, bao gồm: class kDTree và class kNN.

4.1 class kDTree

Class kDTree là class dùng để lưu trữ dữ liệu theo cấu trúc cây k-D và cung cấp cho người dùng các thao tác cơ bản trên cây.

Mô tả cho class kDTree sẽ như sau:

```
struct kDTreeNode
{
    vector<int> data;
    kDTreeNode *left;
    kDTreeNode *right;
    kDTreeNode(vector<int> data, kDTreeNode *left = nullptr, kDTreeNode *
   right = nullptr)
    {
        this->data = data;
        this->left = left;
        this->right = right;
    }

    friend ostream &operator<<(ostream &os, const kDTreeNode &node)
    {
        os << "(";
        for (int i = 0; i < node.data.size(); i++)
        {
            os << node.data[i];
            if (i != node.data.size() - 1)
            {
                os << ", ";
            }
        }
        os << ")";
```

```
        return os;
    }
};

class kDTree {
private:
    int k;
    kDTreeNode* root;
public:
    kDTree(int k = 2);
    ~kDTree();

    const kDTree& operator=(const kDTree& other);
    kDTree(const kDTree& other);

    void inorderTraversal() const;
    void preorderTraversal() const;
    void postorderTraversal() const;
    int height() const;
    int nodeCount() const;
    int leafCount() const;

    void insert(const vector<int>& point);
    void remove(const vector<int>& point);
    bool search(const vector<int>& point);
    void buildTree(const vector<vector<int>>& pointList);
    void nearestNeighbour(const vector<int>& target, kDTreeNode* &best);
    void kNearestNeighbour(const vector<int>& target, int k, vector<
   kDTreeNode*> &bestList);
};
```

Dưới đây là thông tin chi tiết về các phương thức và thuộc tính trong class kDTree:

1. `int k;`
    - Thuộc tính thể hiện số chiều của cây k-D
2. `kDTreeNode* root;`
    - Node gốc của cây
3. `kDTree();`
    - Constructor cho class kDTree.
    - root được gán bằng NULL;

4. `~kDTree();`
    - Destructor cho class kDTree.
5. `kDTree& operator=(const kDTree& other);`
    - Assignment operator.
6. `kDTree(const kDTree& other);`
    - Copy constructor.
7. `void inorderTraversal() const;`
    - Duyệt cây theo thứ tự In-order và in từng node theo format: Các node cách nhau một khoảng trắng, không có khoảng trắng dư ở cuối dòng.
    - Format in một node:

      $$(a_1, a_2, a_3, ..., a_n)$$

      Trong đó a_i là giá trị của chiều thứ i của điểm dữ liệu, và n là số chiều dữ liệu.
8. `void preorderTraversal() const;`
    - Duyệt cây theo thứ tự Pre-order và in từng node theo format: Các node cách nhau một khoảng trắng, không có khoảng trắng dư ở cuối dòng.
    - Format in một node: Tương tự như inorder
9. `void postorderTraversal() const;`
    - Duyệt cây theo thứ tự Post-order và in từng node theo format: Các node cách nhau một khoảng trắng, không có khoảng trắng dư ở cuối dòng.
    - Format in một node: Tương tự như inorder
10. `int height() const;`
    - Trả về chiều cao hiện tại của cây. Nếu cây không có phần tử thì chiều cao là 0.
11. `int nodeCount() const;`
    - Trả về số node hiện tại trong cây.
12. `int leafCount() const;`
    - Trả về số node lá trong cây. Node lá là node không có cây con bên trái lẫn bên phải.
13. `void insert(const vector<int>& point);`
    - Chèn một điểm có giá trị được chứa trong điểm *point* vào cây k-D.
    - Nếu chiều dài của data khác chỉ số k, hàm không thêm data vào cây.
    - Phương pháp chèn được đề cập ở mục trên.

14. `void remove(const vector<int>& point);`
    - Xóa một điểm dữ liệu trong cây bằng với điểm `point` truyền vào. Hai điểm được xem là bằng nhau khi danh sách các giá trị của chúng là như nhau và cùng số chiều dữ liệu.
    - Nếu không tìm thấy điểm dữ liệu cần xóa, hàm không làm gì cả.
    - Phương pháp xóa được đề cập ở mục trên.
15. `bool search(const vector<int>& point);`
    - Tìm kiếm điểm `point` có nằm trong cây hay không. Nếu có, trả về `true`, ngược lại trả về `false`.
16. `void buildTree(const vector<vector<int»& pointList);`
    - Phương thức được dùng để xây cây từ danh sách các điểm dữ liệu `pointList` cho trước.
    - Sinh viên tham khảo phương pháp xây cây ở mục trên.
17. `void nearestNeighbour(const vector<int>& target, kDTreeNode *best);`
    - Phương thức được dùng để tìm láng giềng gần nhất của điểm `target`. Điểm láng giềng gần nhất này sẽ được lưu vào con trỏ `best`
    - Nếu có 2 láng giềng gần nhất có khoảng cách với `target` như nhau, láng giềng gần hơn là láng giềng được tìm thấy đầu tiên.
    - Sinh viên tham khảo phương pháp tìm láng giềng gần nhất ở mục trên.
18. `void kNearestNeighbour(const vector<int>& target, int k, vector<kDTreeNode*> &bestList);`
    - Phương thức được dùng để tìm k láng giềng gần nhất của điểm `target`. k điểm láng giềng gần nhất này sẽ được lưu vào danh sách các con trỏ `bestList`. Lưu ý rằng chỉ số k này là số lượng các điểm láng giềng gần nhất. Sinh viên không nên nhầm lẫn với k trong k-D Tree, tức số chiều của dữ liệu.
    - Nếu có 2 láng giềng gần nhất có khoảng cách với `target` như nhau, láng giềng gần hơn là láng giềng được tìm thấy đầu tiên.
    - Sinh viên tham khảo phương pháp tìm láng giềng gần nhất ở mục trên.

4.2 class kNN

Class kNN là class dùng để hiện thực các chức năng của mô hình dự đoán sử dụng thuật toán kNN bằng cấu trúc dữ liệu k-D Tree. Thiết kế cho class kNN sẽ như sau:

```
class kNN {
private:
    int k;
public:
    kNN(int k = 5);
    void fit(const Dataset& X_train, const Dataset& y_train);
    Dataset predict(const Dataset& X_test);
    double score(const Dataset& y_test, const Dataset& y_pred);
};
```

Dưới đây là thông tin chi tiết về các trường dữ liệu và phương thức trong class kNN:

1. `int k;`
    - Chỉ số k thể hiện số láng giềng gần nhất. Sinh viên tránh nhầm lẫn với chỉ số k của k-D Tree.
2. `void fit(const Dataset& X_train, const Dataset& y_train);`
    - Thực hiện tải dữ liệu để huấn luyện, trong đó:
        - X_train: Bảng dữ liệu chứa các đặc trưng không phải là nhãn. Số hàng của X_train là số lượng ảnh được sử dụng để huấn luyện, và số cột là số đặc trưng.
        - y_train: Bảng dữ liệu chứa các nhãn tương ứng với các đặc trưng. Vì chỉ là các nhãn, nên số hàng của y_train là số lượng ảnh, và số cột chỉ duy nhất là 1.
    - Gợi ý: Sinh viên sẽ xây cây k-D Tree ở phương thức này.
3. `Dataset predict(const Dataset& X_test);`
    - Là phương thức thực thi việc dự đoán dựa vào thuật toán kNN.
    - X_test là bảng dữ liệu chứa đặc trưng của các ảnh đem dự đoán. Số hàng của X_test là số lượng ảnh được sử dụng để dự đoán, và số cột là số đặc trưng.
    - Lưu ý rằng, sinh viên cần phải hiện thực dự đoán cho nhiều ảnh mới, tương ứng với số hàng trong bảng dữ liệu X_test.
    - Kết quả trả về là bảng dữ liệu gồm nhiều hàng và 1 cột. Trong đó, số hàng là số ảnh được đem dự đoán. Nội dung của từng ô trong cột là nhãn (lớp) sau khi được dự đoán bởi thuật toán kNN.
    - Gợi ý: Sinh viên sẽ thực hiện tìm k láng giềng gần nhất trong cây k-D Tree đã xây từ bộ huấn luyện.
4. `double score(const Dataset& y_test, const Dataset& y_pred);`
    - Là phương thức đo đạc độ chính xác của quá trình dự đoán.

- y_test là bảng dữ liệu chứa nhãn thực sự các ảnh đem dự đoán. Số hàng của y_test là số lượng ảnh được sử dụng để dự đoán, và số cột là 1.
- y_pred là bảng dữ liệu chứa nhãn sau khi dự đoán bởi giải thuật kNN (thu được từ hàm predict). Số hàng của y_pred là số lượng ảnh được sử dụng để dự đoán, và số cột là 1.
- Kết quả trả về là một số thực với cách tính như sau:

$$\text{Độ chính xác} = \frac{\text{Số ảnh dự đoán đúng}}{\text{Tổng ảnh dự đoán}}$$

Trong đó, Số ảnh dự đoán đúng là số ảnh có nhãn dữ đoán trùng với nhãn thực sự.

4.3 class Dataset và hàm train_test_split

BTL này đã **hiện thực sẵn** class Dataset và hàm train_test_split, sinh viên đọc hiểu về class và hàm này để hiện thực các testcase và các class phía trên. Mô tả về class Dataset có chút thay đổi so với BTL 1.

Class Dataset là class dùng để lưu trữ và xử lí dữ liệu dạng bảng, cụ thể trong bài tập lớn này là bộ dữ liệu MNIST. Mô tả cho class Dataset sẽ như sau:

```
class Dataset {
private:
    list<list<int>> data;
    vector<string> columnName;
public:
    Dataset();
    ~Dataset();
    bool loadFromCSV(const char* fileName);
    void printHead(int nRows = 5, int nCols = 5) const;
    void printTail(int nRows = 5, int nCols = 5) const;
    void getShape(int& nRows, int& nCols) const;
    void columns() const;
    bool drop(int axis = 0, int index = 0, std::string columns = "");
    Dataset extract(int startRow = 0, int endRow = -1, int startCol = 0, int
    endCol = -1) const;
    Dataset(const Dataset& other);
    Dataset& operator=(const Dataset& other);
    friend void train_test_split(Dataset& X, Dataset& y, double test_size,
                        Dataset& X_train, Dataset& X_test, Dataset& y_train,
    Dataset& y_test);
    friend class kNN;
};
```

```
21
22 void train_test_split(Dataset& X, Dataset& y, double test_size,
23                         Dataset& X_train, Dataset& X_test, Dataset& y_train,
    Dataset& y_test);
```

Dưới đây là thông tin chi tiết về các trường dữ liệu và phương thức trong class Dataset:

1. `list<list<int» data;`
   - Là trường thông tin chính để lưu giữ dữ liệu ở dạng bảng.
2. `vector<string> data;`
   - Là trường thông tin để lưu trữ thông tin các cột của dữ liệu
3. `Dataset();`
   - Default constructor.
4. `~Dataset();`
   - Destructor.
5. `Dataset(const Dataset& other);`
   - Copy constructor.
6. `Dataset& operator=(const Dataset& other);`
   - Assignment operator.
7. `bool loadFromCSV(const char* fileName);`
   - Phương thức được sử dụng để tải dữ liệu từ file `fileName`, cụ thể trong bài tập lớn này là file `mnist.csv`. Các thông tin trong file sẽ được lưu trữ vào biến `data` và các b iến khác do sinh viên đề xuất.
   - Hàm trả về true nếu việc tải dữ liệu thành công, ngược lại false.
8. `void printHead(int nRows = 5, int nCols = 5) const;`
   - In ra `nRows` dòng đầu tiên, và chỉ in `nCols` cột đầu tiên của bảng dữ liệu.
   - Format in:
     - Dòng đầu tiên, in ra tên các cột của bảng dữ liệu.
     - Từ dòng thứ 2 trở đi, in giá trị của mỗi ô trong bảng, mỗi phần tử cách nhau bằng khoảng trắng, không có khoảng trắng dư ở cuối dòng.
   - Ngoại lệ: Nếu `nRows` lớn hơn số lượng dòng trong bảng dữ liệu, in hết dòng trong bảng dữ liệu. Nếu `nCols` lớn hơn số lượng cột trong bảng dữ liệu, in hết cột trong bảng dữ liệu. Nếu `nRows` hoặc `nCols` nhỏ hơn 0, không in gì cả.

9. `void printTail(int nRows = 5, int nCols = 5) const;`
    - In ra `nRows` dòng cuối cùng, và chỉ in `nCols` cột cuối cùng của bảng dữ liệu.
    - Format in:
        - Dòng đầu tiên, in ra tên các cột của bảng dữ liệu.
        - Từ dòng thứ 2 trở đi, in giá trị của mỗi ô trong bảng, mỗi phần tử cách nhau bằng khoảng trắng, không có khoảng trắng dư ở cuối dòng.
    - Ngoại lệ: Nếu `nRows` lớn hơn số lượng dòng trong bảng dữ liệu, in hết dòng trong bảng dữ liệu. Nếu `nCols` lớn hơn số lượng cột trong bảng dữ liệu, in hết cột trong bảng dữ liệu. Nếu `nRows` hoặc `nCols` nhỏ hơn 0, không in gì cả.
10. `void getShape(int& nRows, int& nCols) const;`
    - Phương thức trả về số lượng dòng và số lượng cột tổng cộng hiện tại của đối tượng thông qua hai tham số tham khảo `nRows` và `nCols`.
11. `void columns() const;`
    - In ra tên của tất cả các cột của bảng dữ liệu. Mỗi tên cách nhau một khoảng trắng, không có khoảng trắng dư cuối cùng.
12. `bool Dataset::drop(int axis = 0, int index = 0, std::string columnName = "");`
    - Xóa một hàng hoặc cột của bảng dữ liệu.
    - Nếu `axis` khác 0 hoặc 1, hàm không làm gì cả và trả về false.
    - Nếu `axis` == 0, thực hiện xóa một hàng tại vị trí `index`, sau đó trả về true. Nếu index >= số dòng hoặc < 0, hàm không làm gì cả và trả về false.
    - Nếu `axis` == 1, thực hiện xóa một cột có tên trùng với `columnName`, sau đó trả về true. Nếu columnName không nằm trong danh sách tên các cột, hàm không làm gì cả và trả về false.
13. `Dataset extract(int startRow = 0, int endRow = -1, int startCol = 0, int endCol = -1) const;`
    - Phương thức dùng để trích xuất một phần của bảng dữ liệu, sau đó trả về bảng dữ liệu đã trích xuất.
    - Trong đó: `startRow` là hàng bắt đầu, `endRow` là hàng cuối cùng, `startCol` là cột bắt đầu, `endCol` là cột cuối cùng.
    - Nếu endRow == -1, ta sẽ lấy tất cả các hàng. Tương tự với endCol == -1.
    - Các testcase sẽ đảm bảo startRow, endRow, startCol và endCol nằm trong khoảng giá trị hợp lệ (từ 0 đến số hàng/số cột) và start $\leq$ end.

Hàm train_test_split là hàm phụ trợ dùng để chia bộ dữ liệu ra thành các bảng dữ liệu nhỏ hơn phục vụ cho quá trình huấn luyện và dự đoán. Prototype của hàm như sau:

```
void train_test_split(Dataset& X, Dataset& y, double test_size, Dataset&
    X_train, Dataset& X_test, Dataset& y_train, Dataset& y_test);
```

Trong đó:

- Input:
  - Dataset& X: Bảng dữ liệu chứa các đặc trưng của ảnh. Số hàng là số lượng ảnh, Số cột là số lượng các đặc trưng.
  - Dataset& y: Bảng dữ liệu chứa nhãn của ảnh. Số hàng là số lượng ảnh, Số cột là 1.
  - double test_size: Số thực xác định tỉ lệ của tập dự đoán trên tổng thể dữ liệu. Phần còn lại sẽ là tập huấn luyện.
- Output:
  - Dataset& X_train: Bảng dữ liệu huấn luyện chứa các đặc trưng của ảnh. Số hàng là số lượng ảnh huấn luyện, Số cột là số lượng các đặc trưng.
  - Dataset& y_train: Bảng dữ liệu huấn luyện chứa nhãn của ảnh. Số hàng là số lượng ảnh huấn luyện, Số cột là 1.
  - Dataset& X_test: Bảng dữ liệu dự đoán chứa các đặc trưng của ảnh. Số hàng là số lượng ảnh, Số cột là số lượng các đặc trưng.
  - Dataset& y_test: Bảng dữ liệu dự đoán chứa nhãn thực sự của ảnh. Số hàng là số lượng ảnh, Số cột là 1.

5 Hướng dẫn

Để hoàn thành bài tập lớn này, sinh viên phải:

1. Đọc toàn bộ tập tin mô tả này.
2. Tải xuống tập tin initial.zip và giải nén nó. Sau khi giải nén, sinh viên sẽ nhận được các tập tin: main.cpp, main.hpp, kDTree.hpp, kDTree.cpp, Dataset.hpp, Dataset.o và file mnist.csv. Sinh viên sẽ chỉ nộp 2 tập tin là kDTree.hpp và kDTree.cpp nên không được sửa đổi tập tin main.hpp khi chạy thử chương trình.
3. Sinh viên sử dụng câu lệnh sau để biên dịch:
   g++ -o main main.cpp kDTree.cpp Dataset.o -I . -std=c++11
   Câu lệnh trên được dùng trong command prompt/terminal để biên dịch chương trình. Nếu sinh viên dùng IDE để chạy chương trình, sinh viên cần chú ý: thêm đầy đủ các

tập tin vào project/workspace của IDE; thay đổi lệnh biên dịch của IDE cho phù hợp. IDE thường cung cấp các node (button) cho việc biên dịch (Build) và chạy chương trình (Run). Khi nhấn Build IDE sẽ chạy một câu lệnh biên dịch tương ứng, thông thường, chỉ biên dịch phải main.cpp. Sinh viên cần tìm cách cấu hình trên IDE để thay đổi lệnh biên dịch: thêm file kDTree.cpp, Dataset.o, thêm option -std=c++11, -I .

4. Chương trình sẽ được chấm trên nền tảng UNIX. Nền tảng và trình biên dịch của sinh viên có thể khác với nơi chấm thực tế. Nơi nộp bài trên BKeL được cài đặt giống với nơi chấm thực tế. Sinh viên phải chạy thử chương trình trên nơi nộp bài và phải sửa tất cả các lỗi xảy ra ở nơi nộp bài BKeL để có đúng kết quả khi chấm thực tế.
5. Sửa đổi các file kDTree.hpp, kDTree.cpp để hoàn thành bài tập lớn này và đảm bảo hai yêu cầu sau:

    - Tất cả các phương thức trong mô tả này đều phải được hiện thực để việc biên dịch được thực hiện thành công. Nếu sinh viên chưa thể hiện thực được phương thức nào, hãy cung cấp một hiện thực rỗng cho phương thức đó. Mỗi testcase sẽ gọi một số phương thức đã mô tả để kiểm tra kết quả trả về.
    - Chỉ có 2 lệnh **include** trong tập tin kDTree.hpp là **#include "main.hpp"** và **#include "Dataset.hpp"**. Chỉ có một include trong tập tin kDTree.cpp là **#include "kDTree.hpp"**. Ngoài ra, không cho phép có một **#include** nào khác trong các tập tin này.

6. Trong tập tin main.cpp có cung cấp một số testcases đơn giản trong các hàm có định dạng **"tc<x>()"**.
7. Sinh viên được phép viết thêm các phương thức và thuộc tính khác trong các class được yêu cầu hiện thực. Sinh viên được phép viết thêm các class khác ngoài các yêu cầu hiện thực.
8. Sinh viên được yêu cầu thiết kế và sử dụng các cấu trúc dữ liệu dựa trên các loại danh sách đã học.
9. Sinh viên phải giải phóng toàn bộ vùng nhớ đã xin cấp phát động khi chương trình kết thúc.

6 Nộp bài

Sinh viên chỉ nộp 2 tập tin: kDTree.hpp và kDTree.cpp, trước thời hạn được đưa ra trong đường dẫn "Assignment 2 - Submission". Có một số testcase đơn giản được sử dụng để kiểm tra bài làm của sinh viên nhằm đảm bảo rằng kết quả của sinh viên có thể biên dịch và chạy

được. Sinh viên có thể nộp bài bao nhiêu lần tùy ý nhưng chỉ có bài nộp cuối cùng được tính điểm. Vì hệ thống không thể chịu tải khi quá nhiều sinh viên nộp bài cùng một lúc, vì vậy sinh viên nên nộp bài càng sớm càng tốt. Sinh viên sẽ tự chịu rủi ro nếu nộp bài sát hạn chót. Khi quá thời hạn nộp bài, hệ thống sẽ đóng nên sinh viên sẽ không thể nộp nữa. Bài nộp qua các phương thức khác đều không được chấp nhận.

7 Một số quy định khác

- Sinh viên phải tự mình hoàn thành bài tập lớn này và phải ngăn không cho người khác đánh cắp kết quả của mình. Nếu không, sinh viên sẽ bị xử lý theo quy định của trường vì gian lận.
- Mọi quyết định của giảng viên phụ trách bài tập lớn là quyết định cuối cùng.
- Sinh viên không được cung cấp testcase sau khi chấm bài mà chỉ được cung cấp thông tin về chiến lược thiết kế testcase và phân bố số lượng sinh viên đúng theo từng testcase.
- Nội dung Bài tập lớn sẽ được Harmony với một câu hỏi trong bài kiểm tra với nội dung tương tự.

8 Thay đổi so với phiên bản trước

- Cập nhật struct kDTreeNode: Thêm overloading cho operator «
- Chỉnh sửa prototype của phương thức nearestNeighbor của class kDTree
- Ví dụ 3.2 phần Xây cây: Thay đổi kết quả (đổi vị trí của node (7,3) và node (9,4).

——————————HẾT——————————