Capitole Speciale de Informatică

Arbori kd

1 Noiembrie 2018

Un *arbore kd* (prescurtare de la „arbore k-dimensional") este o structură de date arborescentă pentru o colecţie de puncte dintr–un spaţiu k-dimensional $S = \underbrace{A_0 \times \ldots \times A_{k-1}}$ unde $A_0, \ldots, A_{k-1}$ sunt mulţimi total ordonate[1] de valori, care poate fi folosită pentru a rezolva problema următoare:

Se dă un punct $X \in S$ şi o mulţime finită de puncte $M \subset S$

Să se găsească punctul $P \in M$ care este cel mai apropiat de punctul X.

Un arbore kd pentru M este un arbore binar care memorează câte un punct din M în fiecare nod, şi satisface condiţiile următoare:

- Pentru fiecare nod X la nivelul n, dacă $(a_0, \ldots, a_{k-1})$ este punctul reţinut în nodul X şi $i = n \mod k$, atunci
  - $a'_i \leq a_i$ pentru toate punctele $(a_0, \ldots, a_{k-1})$ stocate în subarborele stâng al lui X,
  - $a_i < a'_i$ pentru toate punctele $(a'_0, \ldots, a'_{k-1})$ stocate în subarborele drept al lui X.

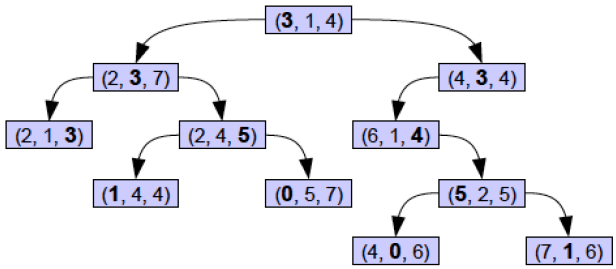
Deci, un arbore kd este o generalizare a unui arbore binar de căutare în care poziţia cheii de căutare dintr-un punct depinde de nivelul nodului care reţine punctul respectiv: cheia de căutare dintr-un punct X stocat într-un nod de la nivelul n este coordonata $(n \mod k)$ a lui X; presupunem că coordonatele punctelor sunt indexate pornind de la 0.

De examplu, dacă $k = 3$, $A_0 = A_1 = A_2 = \mathbb{R}$ şi

$$S = \{\ (0,5,7), (1,4,4), (2,1,3), (2,3,7), (2,4,5), \\ (3,1,4), (4,0,6), (4,3,4), (5,2,5), (6,1,4), (7,1,6)\}$$

atunci un arbore kd care reţine nodurile mulţimii S este

[1] O mulţime A este total ordonată dacă există o relaţie binară $\leq$ pe A care este **totală** (pentru toţi $a, b \in A$, fie $a \leq b$ sau $b \leq a$), **reflexivă** ($a \leq a$ pentru toţi $a \in A$), **antisimetricıa** (pentru toţi $a, b \in A$, dacă $a \leq b$ şi $b \leq a$ atunci $a = b$) şi **tranzitivă** (pentru toţi $a, b, c \in A$, dacă $a \leq b$ şi $b \leq c$ atunci $a \leq c$).



1 Construcţia unui arbore kd balansat

Fie L o listă de puncte k-dimensionale şi o axă $i \in \{0, 1, \ldots, k-1\}$. Putem defini recursiv arborele kd $T(L, i)$ pentru lista de puncte L şi axa iniţială i, în felul următor:

Cazul de bază: Dacă L este lista vidă atunci $T(L, i)$ este arborele vid.

Cazul recursiv: Fie $L' = [P_1, \ldots, P_n]$ rezultatul sortării listei L în ordine creascătoare a valorilor coordonatei i a punctelor, $m_0 = \lfloor n/2 \rfloor$, şi

$$m_1 := \max\{j \mid m_0 \leq j \leq n \text{ şi } P_{m_0}, P_j \text{ au aceeaşi valoare a coordonatei } i\}.$$

Atunci $T(L, i)$ re

- punctul P_{m_1} stocat în nodul rădăcină,
- subarborele stâng $T([P_1, \ldots, P_{m_1-1}], (i+1) \mod k)$, şi
- subarborele drept $T([P_{m_1+1}, \ldots, P_n], (i+1) \mod k)$.

Arborele kd construit în acest fel este balansat dacă $m_0 = m_1$ în toţi paşii recursivi.

2 Găsirea unui nod în un arbore kd

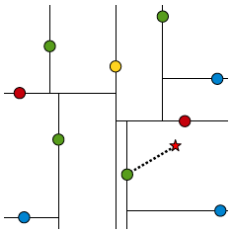
Presupunem că T este un arbore kd cu axa iniţială i. Pseudocodul algoritmului de găsire a nodului din T care reţine un punct $A = (a_0, \ldots, a_{k-1})$ este:

FindPoint(T, i, A)
if T este gol **then return** `'nod negasit'`
$(a'_0, \ldots, a'_{k-1})$:=punctul din rădăcina lui T
if $a'_j = a_j$ for $0 \leq j < k$ **then return** rădăcina lui T
if $a_i \leq a'_i$ **then**
  return FindPoint$(T.\text{left}, (i+1) \mod k, A)$
else /* $a_i > a'_i$ */
  return FindPoint$(T.\text{right}, (i+1) \mod k, A)$

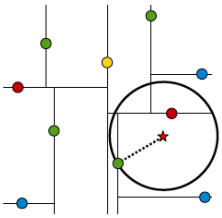
3 Găsirea celui mai apropiat vecin al unui punct

Presupunem că T este un arbore kd cu axa iniţială i, şi că vrem să găsim punctul stocat în T care este cel mai aproape de un punct test $A = (a_0, \ldots, a_{k-1})$.

Intuiţia din spatele algoritmului prezentat aici este următoarea. Să presupunem că avem o bănuială care este nodul din T cel mai apropiat de punctul A. De exemplu, să presupunem că punctul A (punctul de test) este cel indicat de o steluţă, şi că credem că vecinul lui cel mai apropiat este cel conectat la steluţă cu line întreruptă.



Remarcă: Dacă ar exista un punct mai apropiat de A decât punctul cu care este conectat prin linie întreruptă, atunci acel punct ar trebui să fie în cercul cu centrul în A care trece prin bănuiala curentă:



Observaţiii

- În un spaţiu k-dimensional am avea o hipersferă în loc de cerc, pe care am numi-o **hipersferă candidat**.
- Remarca precedentă ne permite să eliminăm subarbori ai arborelui kd din spaţiul de căutare a celui mai apropiat vecin: Dacă toată hipersfera este într-o singură parte a unui hiperplan de separare, atunci cel mai apropiat vecin nu poate fi in partea cealaltă a hiperplanului de separare, şi prin urmare putem omite căutarea unui vecin în subarborele pentru punctele din cealaltă parte a hiperplanului.

  De pildă, în exemplul ilustrat cercul se află complet în dreapta liniei verticale care trece prin rădăcina arborelui. Prin urmare, orice punct stocat în subarborele stâng al rădăcinii nu poate fi mai aproape decât nodul cu care este conectat A prin linie întreruptă.
- Criteriul d detecţie dacă o hipersferă cu centrul $(a_0, \ldots, a_{k-1})$ şi raza d nu este complet într-o parte a hiperplanului definit de ecuaţia $x_i = \text{curr}_i$ este foarte simplu:
  $|\text{curr}_i - a_i| < d$.
- Dacă coordonatele punctelor sunt valori reale, se obişnuieşte să se presupună că distanţa dintre două puncte $X = (x_0, \ldots, x_{k-1})$ şi $Y = (y_0, \ldots, y_{k-1})$ este distanţa euclidiană,

adică:

$$distance(X, Y) = \sqrt{(x_0 - y_0)^2 + \ldots + (x_{k-1} - y_{k-1})^2}.$$

Pe baza acestor remarci putem defini algoritmul următor (pseudocod) de găsire a punctului din T care este cel mai aproape de A:

1NN(T, i, A)
Maintain a global best estimate of the nearest neighbor, called `guess`
Maintain a global value of the distance to that neighbor, called `bestDist`
`guess` := `Null`
`bestDist` := ∞
1NNAUX(T.`root`, i, A)
return `guess`

where

1NNAUX(`curr`, i, A)
if `curr` is empty **then return**
if $distance(A,$ `curr`$) <$ `bestDist` **then**
  `guess` := `curr`
  `bestDist` :=$distance(A,$ `curr`$)$
/* recursive search of A in current kd-tree `curr` */
if $a_i \leq \text{curr}_i$ **then**
  `search` :=′ `left`′
  1NNAUX(`curr`.left, $(i + 1) \mod k, A$)
else
  `search` :=′ `right`′
  1NNAUX(`curr`.right, $(i + 1) \mod k, A$)
/* Check if the candidate hypersphere crosses this splitting hyperplane */
if $|\text{curr}_i - a_i| <$ `bestDist` **then**
  if `search` =′ `left`′ **then**
    1NNAUX(`curr`.right, $(i + 1) \mod k, A$)
  else
    1NNAUX(`curr`.left, $(i + 1) \mod k, A$)

4 Găsirea celor mai apropiaţi K vecini ai unui punct

Algoritmul 1NN poate fi generalizat pentru a găsi cei mai apropiaţi K vecini ai unui punct test $A = (a_0, \ldots, a_{k-1})$ într-un arbore kd T cu axa iniţială i: în loc să ţinem evidenţa unei singure bănuieli globale `guess`, putem reţine intr-o structură globală cei mai apropiaţi K vecini descoperiţi până la momentul respectiv. O structură de date potrivită pentru acest lucru este o coadă cu priorităţi şi capacitate limitată (engl. *bounded priority queue*, BPQ), al cărei comportament este descris în Apendix.

Algoritmul generalizat se numeşte kNN (prescurtare de la „k nearest neighbors"). Pseudocodul acestui algoritm este indicat în figura 4.

kNN(T, i, A)
Maintain a BPQ of the candidate nearest neighbors, called `bpq`
Set the maximum size of `bpq` to K
kNNAUX(T.`root`, i, A)
return `bpq`

where

kNNAUX(`curr`, i, A)
if `curr` is empty **then return**
/* add `curr` to `bpq` */
enqueue `q` into `bpq` with priority $distance(A, \texttt{curr})$
/* recursively search the half of the tree that contains the test point A */
if $a_i < \texttt{curr}_i$ **then**
  `search` :=′ `left`′
  kNNAUX(`curr.left`, $(i+1) \mod k, A$)
else
  `search` :=′ `right`′
  kNNAUX(`curr.right`, $(i+1) \mod k, A$)
p :=priority of max.-priority element of `bpq`
if `bpq` is not full OR $|\texttt{curr}_i - a_i| < p$ **then**
  if `search` =′ `left`′ **then**
    kNNAUX(`curr.right`, $(i+1) \mod k, A$)
  else
    kNNAUX(`curr.left`, $(i+1) \mod k, A$)

Figure 1: Algoritmul kNN.

Temă de laborator

1. (O structură de date pentru arbori kd) Folosiţi `C++` pentru a implementa `kdTree`, o structură de date pentru arbori kd care reţin o mulţime de puncte implementate ca instanţe ale clasei

   ```
   struct Point {
       float x[3]; // coordonatele punctelor
   }
   ```

   şi implementaţi metoda

   - `makeKdTree(vector<Point> v)` care crează un arbore balansat kd din punctele reţinute în un vector `v`

2. (Problema kNN) Scrieţi un program care face următoarele:

   (a) Crează un arbore kd balansat T pentru o mulţime de n puncte, ale căror coordonate sunt citite din un fişier text `points.txt` cu conţinutul următor:

Are n linii, şi fiecare linie conţine trei numere în virgulă flotantă separate prin spaţiu, care reprezintă coordonatele unui punct.

De exemplu, `points.txt` ar putea avea conţinutul următor:

```
1.0 4.0 5.14
-17.3 25 6.42
0 0 0.3
3.0 4.0 5.0
```

(b) Aşteaptă ca utilizatorul să introducă de la consolă

- valoarea lui K (un număr întreg pozitiv) şi
- coordonatele punctului de test A (trei numere `float`)

şi returnează K noduri din T care sunt cele mai apropiate de A.

Utilizaţi structurile `Point` şi `BPQ` disponibile de pe site-ul cursului, şi structura de date `kdTree` implementată pentru prima temă de laborator.

A Cozi cu priorităţi si capacitate limitată

O coadă cu priorităţi si capacitate limitată (BPQ) este asemănătoare cu o coadă cu priorităţi obişnuită, cu excepţia faptului că are o capacitate maximă care reprezintă numărul de elemente care pot fi reţinute în ea. Ori de câte ori se adaugă un element nou la ea, dacă coada este plină, se elimină din ea elementul care are prioritatea cea mai mare. De exemplu, să presupunem că avem o BPQ cu capacitatea de 5 elemente, care reţine elementele următoare:

Valoare	A	B	C	D	E
Prioritate	0.1	0.25	1.33	3.2	4.6

Ce se întâmplă dacă vrem să inserăm în ea elementul F cu prioritatea 0.4? Deoarece coada este plină, se va insera elementul F după ce se elimină elementul cu cea mai mare prioritate (E). În final, conţinutul cozii va fi

Valoare	A	B	F	C	D
Prioritate	0.1	0.25	0.4	1.33	3.2

Apoi, să presupunem că vrem să inserăm elementul G cu prioritatea 4.0 în această coadă. Deoarece prioritatea lui G este mai mare decât cea mai mare prioritate a unui element din coadă, coada de priorităţi rămâne neschimbată (nu se va întâmpla nimic).

Observaţie: o implementare a BPQ, concepută să reţină instanţe ale clasei `Point` care reprezintă puncte din $\mathbb{R}^3$, este dată în `BPQ.cpp` and `BPQ.h`, şi poate fi descărcată de pe site-ul acestui curs.