Geometrische Datenstrukturen
Seminar über Algorithmen

Haozhe Chen // Aaron Richardson

June 30, 2005

1 Range-trees

1D Range Tree: Ein balanzierter Binärbaum, jedes Blatt repräsentiert einen Punkt, jeder Innerknoten markiert den größten Punkt an seinem linken Teilbaum. Alle Elemente in dem rechten Teilbaum sind größer als die Stamminnerknoten.

Konstruieren:

-Sortieren die Punkte. Fügen die Punkte in die Blätter.

-Suchen das Median aus, als der Punkt teilt die Bereich zu 2 Teilen. Speichern linken Teil im linken Teilbaum, rechtes Teil im rechten Teilbaum.

-Speichern die Wert des größten Punkts an dem linken Teilbaum in diesem Stamminnerknoten.

Speicherbedarf: $O(n)$

Konstruierungszeit: $O(n \log n)$

Laufzeit der Abfrage nach die punkte in einem gegebenen Bereich: $O(k + \log n)$

2D Range Tree: Hauptbaum ist ein mit x-Koordinate konstruierter 1D Range Tree. Zugehöriger Baum ist ein mit y-Koordinate konstruierter 1D Range Tree. Jeder Innerknoten repräsentiert einen zubehörigen Baum mit gleichen Blättern von seinem Teilbaum.

Queryzeit: $O((\log n)^2 + k)$

Speicherbedarf: $O(n \log n)$

Konstruierungszeit: $O(n \log n)$

2 kd-Tree

Aufgabe: Suche nach einer Menge von Punkten innerhalb eines Quadrats $[x, x'] \times [y, y']$,$x < x', y < y'$. Wir nehmen an, die Punkte haben paarweise unterschiedliche x- und y-Koordinaten. Ein Punkt $p := (p_x, p_y)$ liegt im Quadrat genau dann wenn

$$p_x \in [x, x'] \text{ und } p_y \in [y, y']$$

2.1 Aufbau des Baumes

Um eine Datenstruktur aufzustellen bauen wir einen Baum. Erst teilen wir nach der x-Koordinate in zwei (fast) gleiche kleinere Mengen, und dann teilen die zwei Mengen in vier kleinere Mengen nach der y-Koordinate, usw. Der Algorithmus sieht so aus:

	Algorithm: BuildKdTree (P, Depth)
Input:	A set of points p and the current depth *depth*
Output:	The root of a kd-Tree storing P.
1:	**if** P contains only one point
2:	**then return** leaf with this point.
3:	**else if** *depth* is even
4:	**then** ℓ ist the median x-coordinate in P, then split P into two subsets with all points in P_1 smaller or equal to ℓ in the x-coordinate and P_2 the set of point with x-coordinate larger than ℓ.
5:	**else** ℓ ist the median y-coordinate in P, then split P into two subsets with all points in P_1 smaller or equal to ℓ in the y-coordinate and P_2 the set of point with y-coordinate larger than ℓ.
6:	$\nu_{left} := BuildKDTree(P_1,\ depth + 1)$
7:	$\nu_{right} := BuildKDTree(P_2,\ depth + 1)$
8:	create node ν storing ℓ, make ν_{left} left child of ν and make ν_{right} right child of ν.
9:	**return** ν

(De Berg 98-99)

So ist die Aufbauzeit von n Punkten (nachdem sie nach x oder y einsortiert sind) ist

$$T(n) = \begin{cases} O(1) & \text{if } n = 1. \\ O(n) + 2T(\lceil n/2 \rceil) & \text{if } n > 1. \end{cases}$$

was am Ende $O(n \log n)$ ist. Da auch das Sortieren $O(n \log n)$ ist, hat man eine Laufzeit von $O(n \log n)$.

2.2 Abfrage des Baumes

	Algorithm: SearchKDTree (ν, R)
Input:	root of a kd-tree and range $R := [x, x'] \times [y, y']$
Output:	points at leaves below ν that lie in range
1:	**if** ν is leaf
2:	**then** report point stored at ν if it lies in R.
3:	**else if** *region((lc(ν))* is fully contained in R.
4:	**then** ReportSubtree(lc((**ν**)))
5:	**else if** *region((lc(ν))* intersects R.
6:	**then** SearchKDTree(lc(ν))
7:	**if** *region((rc(ν))* is fully contained in R.
8:	**then** ReportSubtree(rc((**ν**)))
9:	**else if** *region((rc(ν))* intersects R.
10:	**then** SearchKDTree(rc(ν))

(De Berg 101)

Die Komplexität des Algorithmus ist die Zahl der zurückgegeben Punkte, k, plus die Zahl der Nodes der Bäume, die zwar besucht werden, aber deren Unterbaüme nicht ganz im Inputrange sind, d.h. eine Grenze des Gebiets durchläuft das zugehörige Gebeit von Node. Eine senkrechte Gerade kann nur zwei von vier Untergebieten durchlaufen. So ist die Zahl der Nodes, die diese senkrechte Gerade treffen genau

$$Q(n) = \begin{cases} O(1) & \text{if } n = 1. \\ 2 + 2T(n/4) & \text{if } n > 1. \end{cases}$$

oder nach Induktion $Q(n) = O(\sqrt{n})$. $Q(\sqrt{n})$ gilt auch für waagerechte Geraden und allgemein für die Grenze eines Quadrats. (Die Grenzen müssen parellel mit den Achsen sein.) So ist die Komplexität der Abfrage $O(\sqrt{n} + k)$.

3 Intervallbäume

Staat Punkte wollen wir nun Segmente archivieren und abfragen. Die Segmente sind hier eindimensional. Jedes Node des Baumes hat einen Wert x_{mid} Am jedem Node eines Baume speichern wir nur die Intervalle, die sog. Mittelintervalle, die den Punkt x_{mid} hat. Alle Intervalle links und rechts von x_{mid} werden in den Unternodes gespeichert. So wird den Baum gebaut:

	Algorithm: ConstructIntervalTree (ν, R)
Input:	set I of intervals on the real line
Output:	root of an internal tree for I
1:	**if** $I = \emptyset$
2:	**then return** empty leaf
3:	**else**
4:	create node ν. Compute x_{mid}, the median of the set of interval endpoints, and store x_{mid} with ν.
5:	Sort intervals into I_{mid} (intervals with point x_{mid}), I_{left} (intervals with points completly less than x_{mid}) and I_{right} (intervals with points completly more than x_{mid}) and constuct two sorted lists for I_{mid}: a list $L_{left}(\nu)$ sorted on left endpoint and list $L_{right}(\nu)$ sorted on right endpoint.
6:	$lc(\nu)$:= ConstructIntervalTree(I_{left})
7:	$rc(\nu)$:= ConstructIntervalTree(I_{right})
8:	**return** ν

(De Berg 211)

Um die Listen $L_{left}(\nu)$ und $L_{right}(\nu)$ zu sortieren brauchen wir eine Gesamtzeit von $O(n_{mid} \log n_{mid})$ fr jedes Node oder $O(n \log n)$ für alle Nodes, da jedes Intervall ist nur in einem Node zu finden.

Zum Queren brauchen wir den folgenden Algorithmus:

	Algorithm: QueryIntervalTree (ν, R)
Input:	root ν of interval tree and query point q_x
Output:	All intervals that contain q_x
1:	**if** ν is not a leaf
2:	**then if** ($q_x < x_{mid}(\nu)$)
3:	**then** walk along list $L_{left}(\nu)$, starting at leftmost endpoint, reporting all the intervals that contain q_x. Stop as soon intervall does not contain q_x.
4:	QueryIntervallTree ($lc(\nu), q_x$)
5:	**else** walk along list $L_{right}(\nu)$, starting at rightmost endpoint, reporting all the intervals that contain q_x. Stop as soon intervall does not contain q_x.
6:	QueryIntervallTree ($rc(\nu), q_x$)

(De Berg 212)

Ein Query an jedem Node braucht $O(1+k_\nu)$ Zeit (k_ν sind die I_{mid} Intervallen am Node ν zurückgegeben) und für alle Nodes braucht man dann $O(\log n + k)$ Zeit.

4 Priorität-Suchbäume

Wir nehmen wieder an, die Punkte sind zweidimensional und haben paarweise verschiedene x- und y-Koordinaten. Wir suchen nach allen Punkten in einem Region $((-\infty, q_x] \times [q_y, q'_y]$. Der Baum $\mathcal{T}$ wird so gebaut:

- wenn $P = 0$ dann der Baum hat ein leeres Blatt
- sonst sei p_{min} der Punkt in P mit kleinster x-Koordinate. Es sei y_{mid} Medien von den y-Koordinaten. Dann ist

$$P_{below} := \{p \in P \setminus \{p_{min}\} : p_y < y_{mid}\} \quad (1)$$
$$P_{above} := \{p \in P \setminus \{p_{min}\} : p_y > y_{mid}\} \quad (2)$$

  So jeder (Unter)baum hat Node ν und die x- und y-Koordinate vom Node sind gespeichert.
- linker Unterbaum von ν is Priorität-Suchbaum von P_{below}
- rechter Unterbaum von ν is Priorität-Suchbaum von P_{above}

Der Algorithmus braucht $O(n)$ Platz (jeder Punkt einmal gespeichert) und kann in $O(n \log n)$ aufgebaut sein. Um ein Query durchzuführen benutzen wir die Funktion:

	Algorithm: QueryPrioSearchTree ($\mathcal{T}, (-\infty, q_x] \times [q_y, q'_y]$)
Input:	A priority search tree and a range, unbounded to the left.
Output:	All points lying in the range.
1:	Search with q_y and q'_y in $\mathcal{T}$. Let ν_{split} be the node where the two search paths split.
2:	**for** each node ν on the search path of q_y or q'_y
3:	**do if** $p(\nu) \in (-\infty, q_x]x[q_y, q'_y]$**then report**$p(v)$
4:	**for** each node ν on the path of q_y in the left subtree of v_{split}
5:	**do if** the search path goes left at ν
6:	**then** ReportInSubtree($rc(\nu), q_x$)
7:	**for** each node ν on the path of q_y in the right subtree of v_{split}
8:	**do if** the search path goes left at ν
9:	**then** ReportInSubtree($rc(\nu), q_x$)

Mit der Funktion:

	Algorithm: ReportInSubtree (ν,q_x)
Input:	The root ν of a subtree of a priority search tree and value q_x
Output:	All points in the subtree with x-coordinate at most q_x
1:	**if** ν is not leaf and $(p(\nu))_x \leq q_x$
2:	**then report** $P(\nu)$)
3:	ReportInSubtree ($lc(\nu), q_x$)
4:	ReportInSubtree ($rc(\nu), q_x$)

(DeBerg 219-220)
Für eine Anfrage brauchen wir $O(\log n + k)$ Zeit.

5 Segment-Bäume

Elementares Segment

$\exists$ n *Intervalle* $\Rightarrow$ m *Endpunkte*$(n+1 \leq m \leq 2n)$; $\forall$ i $i \in \{1, ..., m-1\}$ [i,i+1] ist ein elementares Segment.

Segment-Bäume sind vollständige Binärbäume. Jeder Blatt repräsentiert ein elementares Intervall. Jeder innere Knoten repräsentiert eine Vereinigung (der Folge) der elementaren Segmente an den Blattern im Teilbaum dieses Knotens.

$\exists$ $[a, b]$ *mit* $a, b \in \{1, ..., n\}$

$\Rightarrow$ [a,b] ist eine Folge von elementaren Segmenten $[i, i+1], 1 \leq i < n$

Konstruieren:

1. Man baut zunächst einen vollständigen Binärbaum, wobei die Anzahl der Blätter grösser als m (die Anzahl der Endpunkte Menge) kleiner als 2m.
2. Fügen die Elementare Segmente in die Blätter ein.
3. Fügen die Vereinigung der elementaren Segmente an die inneren Knoten.
4. Fügen die Intervalle in die Intervalllist der Knoten ein.

procedure Einfügen (I: Intervall; p; Knoten); {anfangs ist p die Wurzel des Segment-Baumes}

if $I(p) \subseteq I$
then
füge I in die Intervall-Liste von p ein und fertig
else
begin
if(p hat linken Sohn p_l) and ($I(p_l) \cap$I $\neq \phi$)
then Einfügen(I, p_l)
if(p hat linken Sohn p_r) and ($I(p_r) \cap$I $\neq \phi$)
then Einfügen(I, p_r)
end

Lemma1: Ein Segment-Baum von einer Menge der n Intervalle hat den Speicherbedarf $O(n \log n)$

Lemma2: Ein Segment-Baum von einer Menge der n Intervalle mit dem Speicherbedarf $O(n \log n)$ kann in $O(n \log n)$ aufgebaut werden.

References

[1] Mark de Berg, Otfried Schwarzkopf, Marc van Kreveld, Mark Overmars *Computational Geometry: Algorithms and Applications* (Heidelburg: Springer Verlag, 2000).

[2] Rolf Klein *Algorithmische Geometrie* (Addison-Wesley, 1997).