KD-Trees & Quadtrees

Proseminar: Fortgeschrittene Algorithmen für Programmierwettbewerbe

Omar Chatila

2. Februar 2024

Zusammenfassung

In dieser Ausarbeitung werden KD-Trees und Quadtrees untersucht, zwei geometrische Datenstrukturen, die in der Verwaltung mehrdimensionaler Daten eingesetzt werden. Nach einer allgemeinen Einführung werden der Aufbau der beiden Datenstrukturen und grundlegende Operationen wie das Suchen, Einfügen und Löschen von Punkten behandelt. Anschließend werden Bereichsanfragen und k-Nearest-Neighbor-Search (k-NNS) behandelt. Die Ergebnisse der Speicher- und Laufzeitanalyse zeigen, dass KD-Trees die geringste Speicherbelegung haben und in k-NNS am schnellsten sind. In allen anderen Algorithmen schneiden Quadtress besser ab. Für Dimensionen $k > 2$ sind KD-Trees aufgrund der leichten Implementierung gegenüber Quadtrees zu bevorzugen, während bei Anwendungen wie Bildkomprimierung, in denen Ebenen in Regionen gleicher Farbe unterteilt werden, Quadtrees besser geeignet sind.

1 Einleitung

Die Speicherung und Verwaltung großer Datenmengen ist ein grundlegendes Problem in der Informatik. Effiziente geometrische Datenstrukturen, wie Quadtrees und KD-Trees, spielen hierbei eine wichtige Rolle. Sie sind nicht nur in Bereichen relevant, in denen es um geometrische Fragestellungen wie Bildkomprimierung, Kollisionserkennung oder Raytracing geht, sondern lassen sich grundsätzlich überall einsetzen, wo mehrdimensionale Daten quantifizierbar sind [SN15; OGR15].

KD-Trees, auch k-dimensionale Suchbäume genannt, dienen der Verwaltung einer Punktmenge im k-dimensionalen Raum $\mathbb{R}^k$ zur effizienten Beantwortung von Bereichsanfragen. Beispielsweise kann die Verwaltung einer Tabelle, in der die Mitarbeiter eines Unternehmens mit den Merkmalen Alter, Gehalt und Anzahl der Kinder erfasst werden, mithilfe eines 3D-Baums realiesiert werden. Jeder Datenpunkt bildet einen Punkt im dreidimensionalen Koordinatensystem, wobei das Alter auf der x-Achse, die Anzahl der Kinder auf der y-Achse und das Gehalt auf der z-Achse aufgetragen wird. Die Bereichsanfrage "Welche Arbeiter im Alter von 30-45 Jahren haben 2-3 Kinder und verdienen zwischen 50 und 60 Tausend Euro im Jahr?", kann in diesem Kontext als geometrische Fragestellung formuliert werden: „Welche Punkte liegen innerhalb des Quaders mit den Dimensionen $[30 : 45] \times [2 : 3] \times [50000 : 60000]$?“[De 00, S.95-96]

Ein Quadtree ist ein Baum, in dem jeder innere Knoten genau vier Kinder hat.

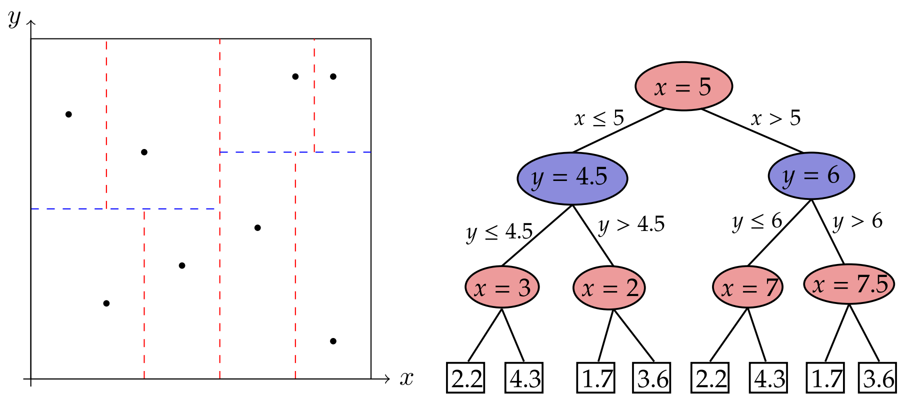
Es gibt zwei Haupttypen von Quadtrees, nämlich Point-Quadtrees und Region-Quadtrees. Beide Arten sind auf den 2-dimensionalen Raum beschränkt[De 00, S.294]. Point-Quadtrees sind besonders geeignet für Anwendungen wie Geoinformationssysteme [Sam84, S.191], bei denen Punktdaten auf einer Karte organisiert werden. Ein Beispiel hierfür ist die Lokalisierung von Standorten auf einer Stadtkarte. Die effiziente Speicherung von Punktinformationen ermöglicht es mittels einer „Nearest-Neighbor-Search “beispielsweise das nächstgelegene Restaurant zu finden. Region-Quadtrees eignen sich hervorragend zur Komprimierung von Bildern und zur Darstellung geografischer Regionen [Sam84, S.192-194]. Für eine bessere Vergleichbarkeit der beiden Datenstrukturen liegt der Fokus dieser Ausarbeitung auf der Verwaltung 2-dimensionaler Punktdaten. In Abschnitt 2 und Abschnitt 3 werden KD-Trees, gefolgt von Quadtrees, erklärt. Anschließend erfolgt in Abschnitt 4 ein Vergleich ihrer Performance und ihres Speicherbedarfs.

2 KD-Trees

KD-Trees sind eine Erweiterung des binären Suchbaums auf k-dimensionale Punktdaten. Zur Vereinfachung betrachten wir zunächst den 2-dimensionalen Fall. Gegeben sei eine Punktmenge P, wobei jeder Punkt eine x- und y-Koordinate hat. Zum Aufteilen der Punktmenge in zwei Teile wählen wir zuerst die *Splitkoordinate* X und den Median von P entlang dieser Koordinate als Splitwert s. Zur Konstruktion des KD-Trees wird P immer wieder abwechselnd nach seinen Splitkoordinaten X und Y in

$$P_{\leq s} = \{(x, y) \in P; x \leq s\} \text{ und } P_{>s} = \{(x, y) \in P; x > s\}$$

zerlegt, wobei der linke Teilbaum $P_{\leq s}$ und der rechte Teilbaum $P_{>s}$ verwaltet. Dieser Vorgang wird rekursiv für jeden Teilbaum wiederholt. Sobald die Punktmenge eines Knotens die Kardinalität eins erreicht, terminiert die Rekursion. Die einzelnen Splitwerte entsprechen den inneren Knoten des KD-Baums. Knoten auf derselben Ebene haben dieselbe Splitkoordinate und Knoten auf der Blattebene enthalten die einzelnen Punkte [Kle97, S.126].



(a) Punktmenge (b) KD-Tree

Abbildung 1: KD-Tree. Rot: Splitkoordinate X. Blau: Splitkoordinate Y

Ein Beispiel für einen KD-Baum zeigt Abb. 1b. Die Wurzel verwaltet die Punktmenge P aus Abb. 1a. Im ersten Schritt wird der Median der x-Koordinaten ($s = 5$) gewählt. Der linke Teilbaum verwaltet $P_{\leq 5}$ und der rechte Teilbaum verwaltet $P_{>5}$. Diese beiden Knoten werden nach der Splitkoordinate Y mit den Splitwerten ihrer Mediane unterteilt. Zuletzt wird erneut nach X unterteilt. Jeder Punkt hat nun seinen eigenen Knoten und der Algorithmus terminiert.

Algorithmus 1 BaueKDTree(P, d) [De 00, S.100]

Eingabe: Punktmenge P, aktuelle Tiefe d
Ausgabe: Knoten v als Wurzel des KD-Trees
  if P enthält nur einen Punkt **then**
    return Blatt, das diesen Punkt speichert
  end if
  if d ist gerade **then**
    sortiere P nach X-Koordinate
    teile P an seiner Medianstelle s in $P_{\leq s}$ und $P_{>s}$ auf
  else
    sortiere P nach Y-Koordinate
    teile P an seiner Medianstelle s in $P_{\leq s}$ und $P_{>s}$ auf
  end if
  $V_l \leftarrow$ BaueKDTree($P_{\leq s}$, $d+1$)
  $V_r \leftarrow$ BaueKDTree($P_{>s}$, $d+1$)
  erstelle Knoten v, der P und Split-Wert s speichert
  setze V_l als linkes Kind und V_r als rechtes Kind von v
  return v

Konstruktionszeit: Sei n die Anzahl der Elemente in P und d die Höhe des Baumes. Der zeitaufwändigste Schritt ist das Finden der Splitwerte. Die Sortierung der einzelnen Punktmengen hat eine Zeitkomplexität von $\mathcal{O}(n \log(n))$ während das Erstellen der Teilmengen von P $\mathcal{O}(n)$ benötigt. Für die restlichen Schritte kann eine konstante Laufzeit angenommen werden. Die Konstruktion des Baumes hat somit eine Zeitkomplexität von $\mathcal{O}(d \cdot n \log(n))$. [Kle97, S.101]. Dadurch, dass in jedem Schritt genau bei der Hälfte der Punktmengen geteilt wird, ist der durch BuildKDTree(P, d) aufgebaute KD-Baum ein ausgeglichener Binärbaum. Für die Höhe gilt $d = \lceil log(n) \rceil$ [De 00, S.130].

Theorem 1 (Komplexität von BuildKDTree($\boldsymbol{P}$, $\boldsymbol{d}$)) *Ein 2-dimensionaler KD-Baum, der eine Punktmenge der Größe n verwaltet, hat eine Höhe $\lceil log(n) \rceil$ und lässt sich in $\mathcal{O}(n \log(n)^2)$ aufbauen. Er hat eine Speicherkomplexität von $\mathcal{O}(n)$. [De 00, S.101]*

Suche nach einem Punkt: Zur Suche eines Punktes, muss man den Baum von der Wurzel beginnend traversieren und abwechselnd X- und Y-Koordinate des Punktes mit dem Splitwert der einzelnen Knoten auf dem Pfad von der Wurzel zum entsprechenden Blatt vergleichen. Aufgrund der Ausgewogenheit des Baumes ergibt sich eine Zeitkomplexität von $\mathcal{O}(\log(n))$ [Kle97, S.128].

Einfügen eines Punktes: Sei $R(v)$ das Rechteck eines Knotens v, das durch die Teilung der Ebene entlang des Pfads von der Wurzel zu v entsteht [Kle97, S.128]. Analog zur Suche wird der Baum so lange traversiert, bis der Blattknoten v mit dem entsprechenden Rechteck, welches den neuen Punkt umschließt, gefunden wird. Der neue Knoten wird zur Punktmenge von von v hinzugefügt, und es wird nach der Splitkoordinate von v gesplittet. Der Knoten v erhält nun zwei neue Kinder, mit jeweils einem der beiden Punkte. Es ist jedoch zu beachten, dass diese Art, Punkte hinzuzufügen, die Ausgewogenheit des Baumes aufheben kann [Kle97, S.130].

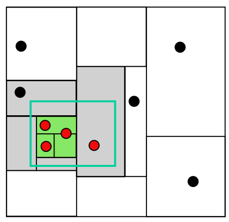
Löschen eines Punktes: Ein Punkt kann *schwach* gelöscht werden, indem analog zur Suche traversiert wird und der zu löschende Punkt markiert wird. Die schwache Entfernung bedeutet, dass die Größe des Baumes nach dem Löschen unverändert bleibt [De 00, S. 127, 130].

Theorem 2 (Operationen) *Das Suchen, Einfügen und Löschen von Punkten in einem 2-dimensionalem KD-Baum hat eine Zeitkomplexität von jeweils* $\mathcal{O}(log(n))$ *[Kle97, S.130].*

Bereichsanfrage: Eine Bereichsanfrage mit einem achsenparallelen Anfragerechteck q gibt alle Punkte zurück, die sich in q befinden. Hierfür genügt es, alle Knoten mit $R(v) \cap q \neq \emptyset$ zu besuchen. Wir unterscheiden 2 Typen von Knoten, die diese Bedingung erfüllen:

1. Knoten v mit $R(v) \subseteq q$ und
2. Knoten v mit $R(v) \nsubseteq q$ und $R(v) \cap q \neq \emptyset$

Abb. rechts: Rot: Punkte in q, grün: $R(v)$ aus Fall 1, grau: $R(v)$ aus Fall 2



Algorithmus 2 KD-Bereichsanfrage(v, q) [De 00, S.103]

Eingabe: KD-Tree v, Rechteck q
Ausgabe: Menge S von Punkten in q
  $S \leftarrow \emptyset$
  if v ist Blatt **then**
    if q enthält $P(v)$ **then**
      $S \leftarrow S \cup P(v)$
    end if
  else if $R(v) \subseteq q$ **then**
    $S \leftarrow S\cup$ Blätter von v
  end if
  if $q \cap R(lc(v)) \neq \emptyset$ **then**
    $S \leftarrow S \cup \text{KD-BEREICHSANFRAGE}(v, q)$
  end if
  if $q \cap R(rc(v)) \neq \emptyset$ **then**
    $S \leftarrow S \cup \text{KD-BEREICHSANFRAGE}(v, q)$
  end if
  return S

Im ersten Fall können einfach die Blätter des gesamten Teilbaums $T(v)$ von v zurückgegeben werden. Dabei sei $a(T(v))$ die Anzahl der Blätter von $T(v)$. Dadurch ergibt sich eine Laufzeit von $\mathcal{O}(a(T(v))$. Im zweiten Fall wird überprüft, ob die Rechtecke der Kinder von v einen nichtleeren Schnitt mit q bildet. Bei positivem Ergebnis wird der Suchalgorithmus auf den entsprechenden Teilbäumen aufgerufen. Falls es sich bei v um ein Blatt handelt und sein Punkt in q liegt, wird dieser zurückgegeben [Kle97, S.128]. Bemerkenswert ist, dass das Ergebnis von Algorithmus 2 nicht von der Form des Anfrageobjekts abhängt. Es ist folglich nicht nötig, dass q ein Rechteck ist [De 00, S.104].

Theorem 3 (Bereichsanfrage) *Eine rechteckige Bereichsanfrage für einen KD-Baum, der n Punkte verwaltet, lässt sich in* $\mathcal{O}(\sqrt{n} + a(v))$ *beantworten, wobei* $a(v)$ *die Größe der Antwort ist [Kle97, S.130].*

k-Nearest-Neighbor-Search Die k-Nearest-Neighbor-Search (k-NNS) gibt die k Punkte zurück, die am nächsten zu einem Anfragepunkt p liegen. Zur Vorbereitung auf Algorithmus 3 wird eine Prioritätswarteschlange *queue* initialisiert, die als Vergleichsfunktion den Abstand des Rechtecks eines Knotens zum Anfragepunkt p verwendet. Weiterhin wird eine Liste R übergeben, in die die k nächsten Nachbarn hinzugefügt werden. Die Funktion k-NNS startet mit der Wurzel *node*. Falls der aktuelle Knoten ein Blatt ist, wird dieser zu *queue* hinzugefügt. Ansonsten werden das linke und rechte Kind des Knotens zu *queue* hinzugefügt. Solange die Warteschlange nicht leer ist und R noch nicht k Elemente enthält, werden Knoten aus *queue* entfernt. Falls der zuletzt entfernte Knoten *current* ein Blatt ist, wird sein Punkt zu R hinzugefügt. Für den Fall, dass *current* ein innerer Knoten ist, wird K-NNS rekursiv auf diesen Knoten hinzugefügt. Nach Terminierung des Algorithmus, befinden sich die k nächsten Nachbarn von p in der Liste R. [Dal18]

Algorithmus 3 K-NNS(*node*, *k*, *queue*, *R*) [Dal18]

Eingabe: Wurzel *node*, Anzahl der Nachbarn k, Prioritätswarteschlange $queue{<}node, p{>}$, Liste R
if *node* ist Blattknoten **then**
  Füge *node* zur Prioritätswarteschlange *queue* hinzu
else
  for alle Kinder *child* von *node* **do**
    Füge *child* zur Prioritätswarteschlange *queue* hinzu
  end for
end if
while die Warteschlange *queue* nicht leer ist und $|R| < k$ **do**
  $current \leftarrow queue$.Top()
  queue.Pop()
  if *current* ist Blattknoten **then**
    Füge den Punkt von *current* zu R hinzu
  else
    K-NNS(*current*, *k*, *queue*, *R*)
  end if
end while

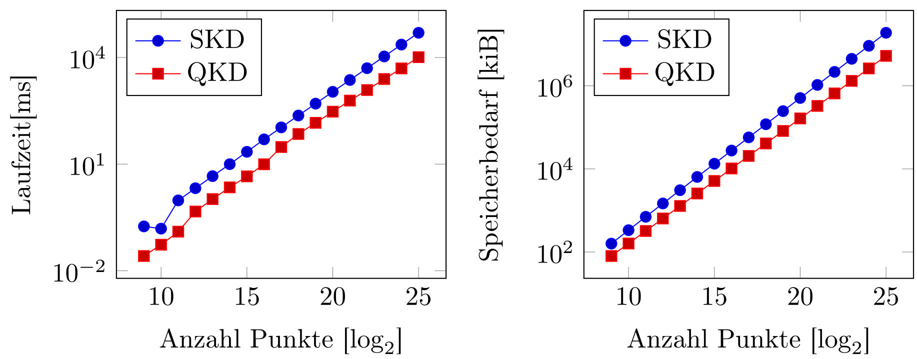
Theorem 4 (k-NNS) *Das Finden der k nächsten Punkte zu einem Anfragepunkt in einem KD-Tree hat eine Zeitkomplexität von* $\mathcal{O}(k \cdot \log(n))$.

2.1 Optimierungen

In der durch Algorithmus 5 definierten Variante von BuildKDTree ist angenommen worden, dass jeder Knoten eine Speicherkomplexität von $\mathcal{O}(1)$ hat. In der Praxis teilt jeder Knoten jedoch die Liste und seine Kinder manipulieren diese durch die Sortierung; demnach muss jeder Knoten die entsprechende Subliste speichern. Auf einer Ebene der Höhe d befinden sich 2^d Knoten mit Sublisten der Größe $\frac{n}{2^d}$. Es gibt laut Theorem 1 $\log(n)$ Ebenen, also beträgt die Speicherkomplexität bei dieser Implementierung tatsächlich $2^d \frac{n}{2^d} \log(n) = \mathcal{O}(n \log(n))$. Eine Speicherkomplexität von $\mathcal{O}(1)$ pro Knoten kann nur erzielt werden, wenn jeder Knoten auf die Ursprungsliste P zugreift. Dies lässt sich realisieren, indem man 2 Indizes $from$ und *to* einführt, die in jedem Knoten die Bereiche der Liste P, auf denen dieser Knoten operiert, eingrenzen. Des Weiteren lässt sich feststellen, dass eine Sortierung der Sublisten für den Algorithmus nicht nötig ist. Verwendet man stattdessen eine lineare Mediansuche, wie *quickselect*, so lässt sich der Zeitaufwand in jeder Ebene von $\mathcal{O}(n \log n)$ auf $\mathcal{O}(n)$ reduzieren. Die Gesamtlaufzeit beträgt nun $\mathcal{O}(n \log(n))$.

2.2 Vergleich beider Varianten

Alle Benchmarks dieser Ausarbeitung wurden auf einem 64-Bit EndeavourOS Linux Betriebssystem (Version 2023.11.17) mit einem AMD Ryzen 5 5500 Prozessor und 31,1 GiB RAM ausgeführt. Zur Implementierung der Datenstrukturen und Algorithmen wurde C++23 und zur Zeitmessung wurde Google Benchmarks verwendet. Pro Test wurden 100 Iterationen auf zufällig erzeugten, homogen verteilten Punkten durchgeführt, wobei die Ergebnisse aus dem arithmetischen Mittel der Einzelergebnisse ermittelt wurden. Sort-KD-Tree (SKD) bezeichnet hierbei die Variante aus Algorithmus 5 und Quickselect-KD-Tree (QKD) die Variante mit der linearen Median Suche.



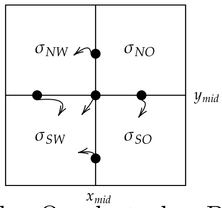
(a) Kontruktionszeit [ms] (b) Speicherbedarf [kiB]

Abbildung 2: KD-Tree Konstruktionszeiten und Speicherbedarf

Dem Ergebnissen in Abb. 2 zufolge ist der Aufbau des Baumes mit der linearen Mediansuche deutlich schneller. Ein weiterer Faktor ist, dass Punktmengen nicht mehr kopiert werden, sondern mit Indizes auf die für einen Knoten relevanten Intervalle verwiesen wird. Der Speicherbedarf ist ebenfalls geringer, was an der gemeinsamen Punktmenge für alle Knoten liegt.

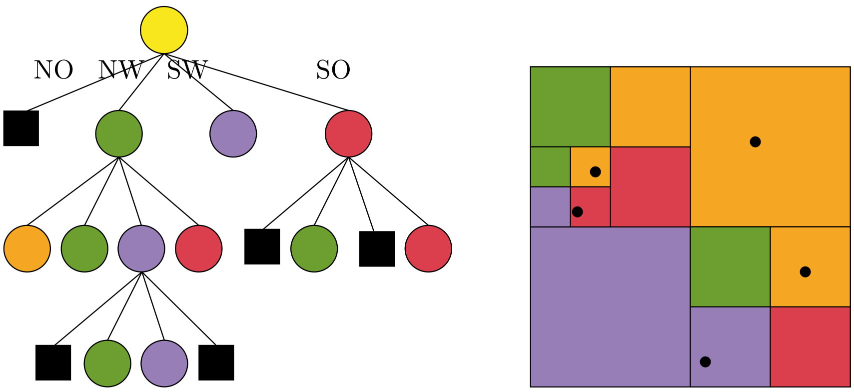
3 Quadtrees

Ein Quadtree ist ein Baum, bei dem jeder innere Knoten genau vier Kinder hat. Mit Quadtrees können 2-dimensionale Punktdaten verwaltet werden. Jeder Knoten repräsentiert ein Quadrat, wobei die Wurzel das Quadrat repräsentiert, das alle Punkte aus einer Punktmenge P enthält und die Blätter jeweils einen Punkt enthalten. Die vier Kinder eines Knotens repräsentieren die vier Quadranten (Nordost (NO), Nordwest (NW), Südwest (SW), Südost (SO), die durch die Unterteilung des Quadrats des Elternknotens entstehen. Wir bezeichnen das Quadrat $\sigma := [x_\sigma : x'_\sigma] \times [y_\sigma : y'_\sigma]$ mit $x_{mid} := (x_\sigma + x'_\sigma)/2$ und $y_{mid} := (y_\sigma + y'_\sigma)/2$ eines Knotens als $\sigma(v)$ und unterteilen die Punktmenge P eines Knotens in die vier Quadranten, wie folgt [De 00, S. 294-295] :



$$
\begin{aligned}
P_{\mathrm{NO}} &:= \{p \in P : p_x > x_{\mathrm{mid}} \land p_y > y_{\mathrm{mid}}\} \\
P_{\mathrm{NW}} &:= \{p \in P : p_x \leq x_{\mathrm{mid}} \land p_y > y_{\mathrm{mid}}\} \\
P_{\mathrm{SW}} &:= \{p \in P : p_x \leq x_{\mathrm{mid}} \land p_y \leq y_{\mathrm{mid}}\} \\
P_{\mathrm{SO}} &:= \{p \in P : p_x > x_{\mathrm{mid}} \land p_y \leq y_{\mathrm{mid}}\}
\end{aligned}
$$

Um einen Quadtree aufzubauen, unterteilt man das Quadrat, das P enthält in vier Quadranten. Für jede neu entstandene Punktmenge P_{NO} bis P_{SO} werden vier Kinder mit den entsprechenen Quadraten erzeugt. Diese werden rekursiv so lange unterteilt bis die Punktmenge eines Knotens nur noch die Kardinalität eins hat. Zur Vorbereitung ermittelt man das Quadrat der Wurzel, indem man für alle Punkte die kleinste und die größte x- und y-Koordinate berechnet. Diese Werte können in Linearzeit ermittelt werden.



(a) Quadtree

(b) Unterteilte Punktmenge

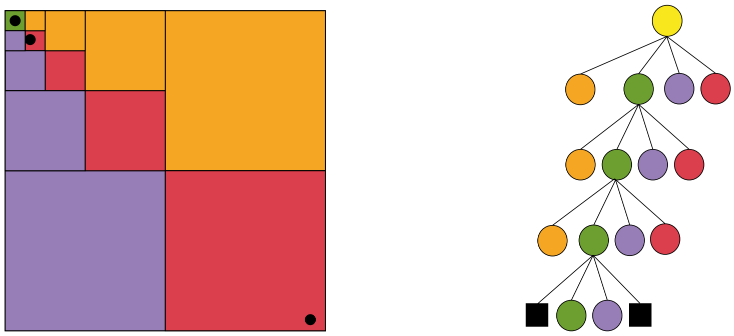
Abbildung 3: Beispiel: Quadtree. Orange: Nordost, Grün: Nordwest, Lila: Südwest, Rot: Südost

Algorithmus 4 BaueKDTree(P, d) [De 00, S.295]

Eingabe: Punktmenge P, aktuelle Tiefe d
Ausgabe: Knoten v als Wurzel des KD-Trees

> **if** P enthält nur einen Punkt **then**
>   **return** Blatt, das diesen Punkt speichert
> **end if**
> **if** d ist gerade **then**
>   sortiere P nach X-Koordinate
>   teile P an seiner Medianstelle s in $P_{\leq s}$ und $P_{>s}$ auf
> **else**
>   sortiere P nach Y-Koordinate
>   teile P an seiner Medianstelle s in $P_{\leq s}$ und $P_{>s}$ auf
> **end if**
> $V_l \leftarrow$ BaueKDTree($P_{\leq s}$, $d+1$)
> $V_r \leftarrow$ BaueKDTree($P_{>s}$, $d+1$)
> erstelle Knoten v, der P und Split-Wert s speichert
> setze V_l als linkes Kind und V_r als rechtes Kind von v
> **return** v

Das Erstellen von Sublisten erfolgt in Linearzeit. Die restlichen Schritte benötigen konstante Laufzeit. Beinhaltet ein Knoten mehr als einen Punkt, wird sein Quadrat in 4 Quadranten geteilt. Das bedeutet aber nicht, dass die Punktmenge $P(v)$ geteilt wird, da es möglich ist, dass alle Punkte von v im selben Quadranten liegen. Liegen Punkte sehr nah beieinander, kann es passieren, dass sehr oft geteilt werden muss, bis jeder Punkt sein eigenes Quadrat erhält.



(a) Tiefe 4 (b) Quadtree

Abbildung 4: Extremfall: Erst bei Tiefe 4 wird P vollständig unterteilt

Theorem 5 (Quadtree: Höhe) *Die Höhe d eines Quadtrees hängt von der kleinsten Distanz zweier Punkte c und von der Seitenlänge des Ursprungsquadrats s, jedoch nicht von der Anzahl der Punkte n ab. Sie ist beschränkt durch $\mathcal{O}(\log(\frac{s}{c}) + \frac{3}{2})$ [De 00, S.295]*

Lemma 6 *Die Quadtrate eine Quadtrees sind disjunkt. Pro Ebene gibt es höchstens n Quadtrate, die vereinigt das Ursprungsquadrat darstellen.*

Theorem 7 (Quadtree: Aufbau) *Das Erstellen eines Quadtrees der Tiefe d aus n Punkten, hat eine Speicher- und Laufzeitkomplexität von $\mathcal{O}((d+1) \cdot n)$*

Suche nach einem einzelnen Punkt: Ein Punkt kann lokalisiert werden indem der Baum beginnend bei der Wurzel so lange traversiert wird, bis man bei einem Blatt ankommt. In jedem Schritt des Pfades wählt man den Kindsknoten, dessen Quadrat den Punkt beinhaltet.

Einfügen eines Punktes: Analog zur Suche wird der entsprechende Blattknoten v_l gefunden. Dieser wird unterteilt und vier neue Knoten werden erzeugt. Da die Anzahl der Teilungen laut Theorem 5 von den Abstand zweier Punkte abhängt, ist es möglich, dass mehrfach geteilt werden muss.

Löschen eines Punktes: Das tatsächliche Löschen eines Punktes erweist sich als kompliziert, da man Knoten zusammenfassen müsste, falls ein Quadrat nur noch in einem Quadranten einen Punkt hat. Ein Punkt kann jedoch schwach gelöscht werden, indem analog zur Suche traversiert wird und der zu löschende Punkt markiert wird.

Theorem 8 (Operationen) *Das Suchen, Einfügen und Löschen von Punkten in Quadtree hat eine Zeitkomplexität von jeweils $\mathcal{O}(d)$, wobei d die Höhe des Baumes ist.*

Algorithmus 5 BAUEKDTREE(P, d)

Eingabe: Punktmenge P, aktuelle Tiefe d
Ausgabe: Knoten v als Wurzel des KD-Trees

```
if P enthält nur einen Punkt then
  return Blatt, das diesen Punkt speichert
end if
if d ist gerade then
  sortiere P nach X-Koordinate
  teile P an seiner Medianstelle s in P_{≤s} und P_{>s} auf
else
  sortiere P nach Y-Koordinate
  teile P an seiner Medianstelle s in P_{≤s} und P_{>s} auf
end if
V_l ← BaueKDTree(P_{≤s}, d + 1)
V_r ← BaueKDTree(P_{>s}, d + 1)
erstelle Knoten v, der P und Split-Wert s speichert
setze V_l als linkes Kind und V_r als rechtes Kind von v
return v
```

Finkel und Bentley [FB74] haben 1974 empirisch gezeigt, dass die Höhe d eines Quadtrees logarithmisch zur Eingabegröße n ist. Sei F die Anzahl der Knoten in q, dann beträgt die Laufzeitkomplexität für eine Bereichsanfage $\mathcal{O}(F + 2^d) = \mathcal{O}(F + n)$ [Sam06, S.40].

k-Nearest-Neighbor-Search Die k-NNS für Quadtrees funktioniert analog zur Algorithmus 3 für KD-Trees. Der Unterschied besteht darin, vier Kinder statt zwei zur Prioritätswarteschlange hinzuzufügen und anstelle der Überprüfung, ob der Knoten ein Blatt ist, zu prüfen ob der Knoten ein nicht leeres Blatt ist.

3.1 Optimierungen

Im Abschnitt 3 ist gezeigt worden, dass alle erwähnten Operationen auf Quadtrees von der Höhe des Baumes abhängen. Wir haben festgelegt, dass ein Quadrat unterteilt werden muss, wenn es mehr als einen Punkt enthält. Lockert man diese Bedingung und erlaubt beispielsweise 4 Punkte, so lässt sich die Anzahl der Unterteilungen und somit die Höhe des Baumes stark reduzieren. Diese Art von Quadtrees nennt sich Point-Region-Quadtree (PR-Quadtree). Für den Aufbau eines PR-Quadtrees überprüfen anstelle der Bedingung "**if** P enthält nur einen Punkt **then**"nun "**if** $|P| \leq$ *Kapazität*". Beim Einfügen eines Punktes in einen dynamischen Quadtree traversieren wir den Baum wie gehabt, teilen nun aber nur, sofern durch das Einfügen des Punktes in den Blattknoten die Kapazität überschritten wird [Ore82]. Für die Punktmenge aus Abb. 3b ergibt sich nun folgender PR-Quadtree:

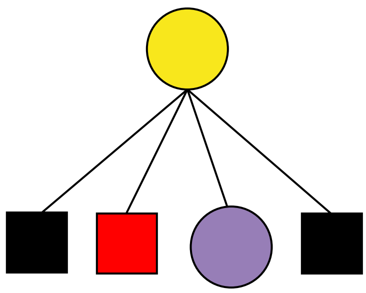


Abbildung 5: PR-Quadtree

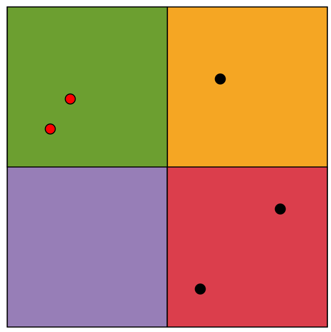
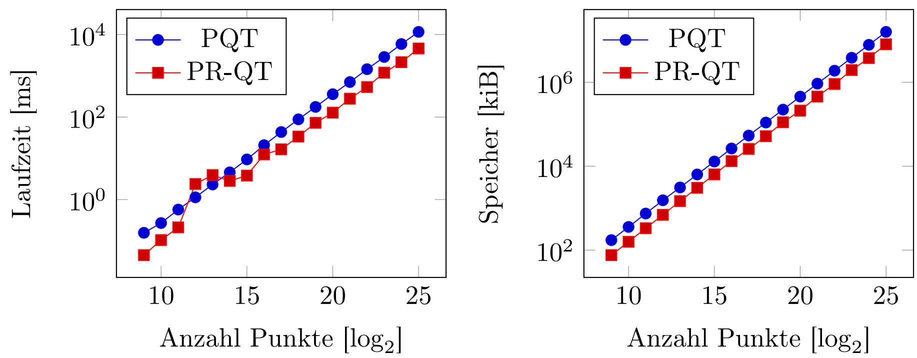


Abbildung 6: Unterteilte Punktmenge

Ein Nachteil gegenüber Point-Quadtrees ist, dass in einer Bereichsanfrage für alle Punkte in einem Blatt überprüft werden muss, ob diese im Rechteck liegen, anstatt einfach den einzelnen Punkt zurückzugeben. Je nach Anwendungsfall muss demnach entschieden werden, ob dieser Mehraufwand die verringerte Baumhöhe wert ist und gegebenenfalls die Kapazität eines Knotens angepasst werden.

3.2 Vergleich beider Varianten

Im Folgenden werden Point-Quadtrees mit PR-Quadtrees hinsichtlich ihrer Laufzeit verglichen. Beide Datenstrukturen operieren auf derselben zufällig erzeugten Punktmenge. Jeder Baum erhält bei einer Punktmenge der Kardinalität n als Wurzelquadrat das im ersten Quadranten liegende Quadrat mit der Seitenlänge n. Die Punkte sind homogen innerhalb dieses Quadrats platziert. Als Kapazität für die PR-Quadtrees legen wir $\log_{10}(n)$ fest, um die Höhe des Baumes zu beschränken, da mit mächtigeren Punktmengen zu erwarten ist, dass die kleinste Distanz zweier Punkte sinkt [Sam84, S.54].



(a) Kontruktionszeit [ms]

(b) Speicherbedarf [kiB]

Abbildung 7: Quadtrees Konstruktionszeiten und Speicherbedarf

Auffällig an Abb. 7a ist, dass PR-Quadtrees ungefähr doppelt so schnell aufgebaut werden. Dies lässt sich durch die geringere Anzahl an Teilungen und somit Rekursionsaufrufen begründen.

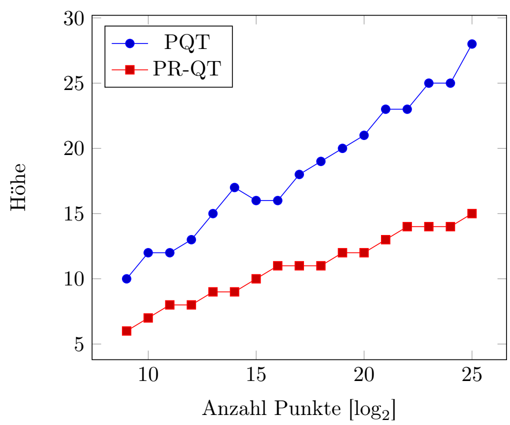


Abbildung 8: Höhen von Quadtrees

Wie erwartet ist die Höhe des PR-Quadtrees deutlich geringer (Abb. 8). Das liegt daran, dass aufgrund der höheren Blattkapazitäten, weniger Teilungen durchgeführt werden müssen. Hierdurch lässt sich der geringere Speicherbedarf für PR-Quadtrees in Abb. 7b erklären, da dieser im linearen Zusammenhang zur Baumhöhe steht.

4 Vergleich zwischen KD-Trees und Quadtrees

Im Folgenden werden alle in Abschnitt 2 und Abschnitt 3 definierten Variationen von Quadtrees und KD-Trees hinsichtlich Laufzeit und Speicherbedarf miteinander verglichen. Für jeden Test werden 100 Iterationen auf zufällig erzeugten Punktmengen ausgeführt. Das Rechteck der Wurzel wird bei einer Punktmenge mit $|P| = n$ als $r = [0 : n] \times [0 : n]$ definiert.

Konstruktion

Laufzeit

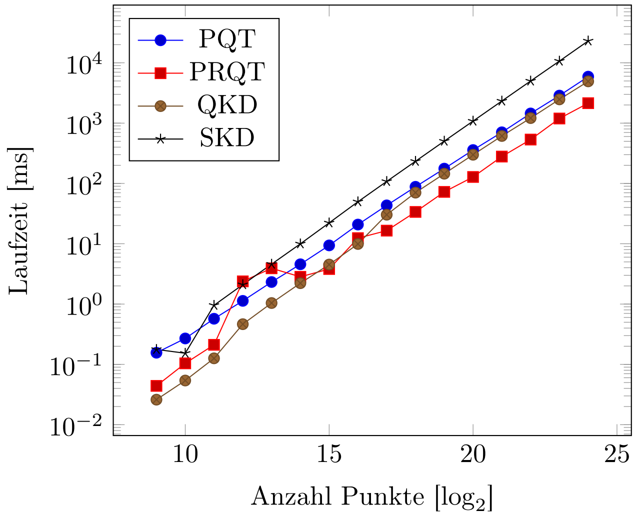


Abbildung 9: Laufzeit [ms]

Speicherbedarf

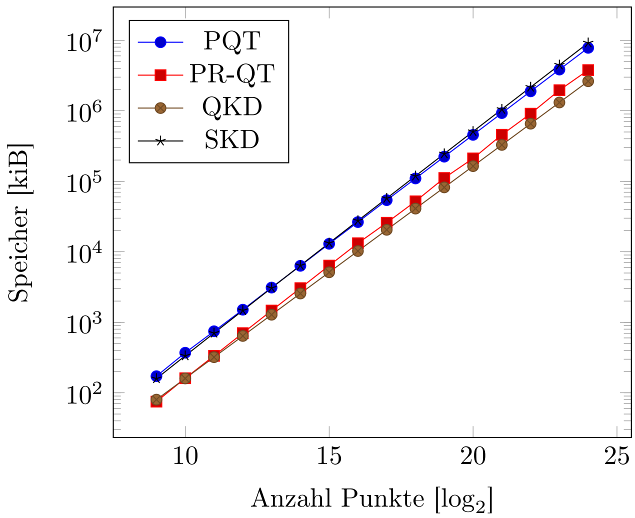


Abbildung 10: Speicherbedarf [kiB]

Für Punktmengen der Kardinalität kleiner als 2^{16} lassen sich laut Abb. 9 KD-Trees mit der linearen Mediansuche (QKD-Trees) am schnellsten konstruieren, während bei größeren Punktmengen PR-Quadtrees zu verwenden sind. Hinsichtlich des Speicherverbrauchs sind QKD-Trees unabhängig der Eingabegröße bevorzugt.

Bereichsanfragen

Für alle Bereichsanfragen wurde ein Anfragerechteck verwendet, das 2% des Wurzelrechtecks abdeckt.

Laufzeit

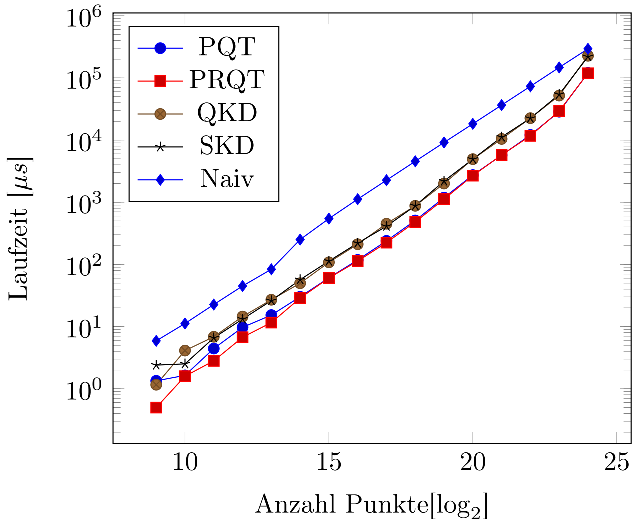


Abbildung 11: Laufzeit [μs]

Speicherbedarf

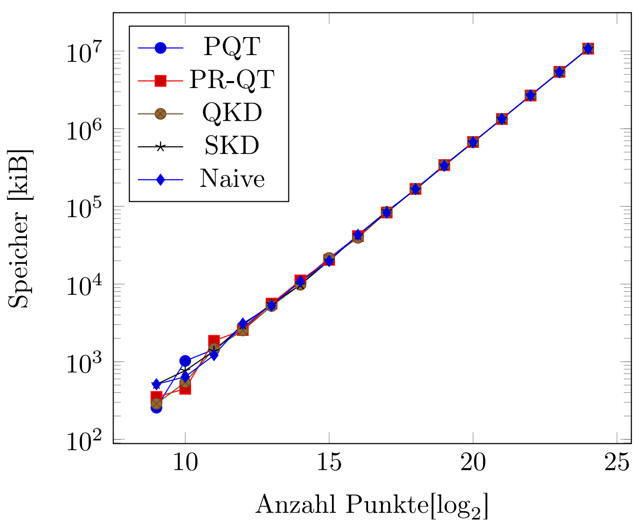
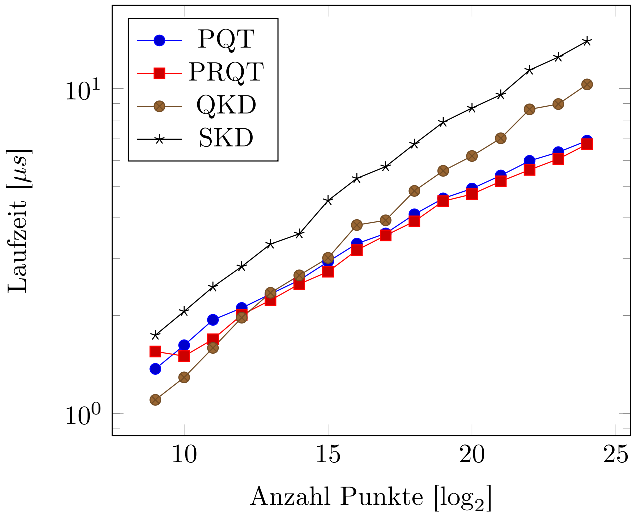


Abbildung 12: Speicherbedarf [kiB]

Aus Abb. 11 ist erkennbar, dass Quadtrees fast doppelt so schnell arbeiten wie KD-Trees. Der Speicherbedarf ist bei allen Datenstrukturen fast identisch. Insgesamt sind PR-Quadtrees für Bereichsanfragen zu bevorzugen, da sie die geringste Laufzeit haben.

Suche

Für die folgenden Tests wurden pro Eingabegröße 100 zufällige Punkte gesucht. Die Laufzeiten stellen die Gesamtlaufzeiten der Algorithmen dar. Der naive Algorithmus ist eine einfache lineare Suche durch eine Liste.



(a) Laufzeit [μs]

n	Naiv [μs]
9	218
10	389
11	789
12	1,548
13	3,085
14	6,106
15	12,264
16	24,509
17	48,785
18	97,811
19	$1.95 \cdot 10^5$
20	$3.9 \cdot 10^5$
21	$7.78 \cdot 10^5$
22	$1.56 \cdot 10^6$
23	$3.14 \cdot 10^6$
24	$6.33 \cdot 10^6$

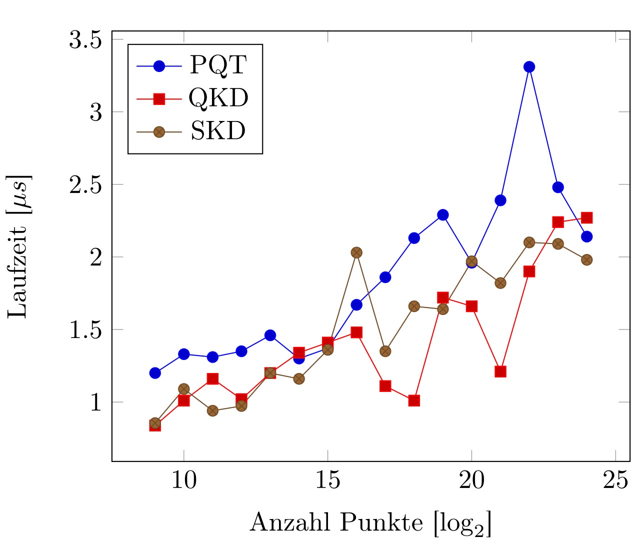
(b) Naiver Algorithmus

Abbildung 13: Laufzeit: Suche von 100 Punkten

Die Ergebnisse in Abschnitt 4 bestätigen unsere Erwartung, dass die Bäume bei den Suchen aufgrund der logarithmischen Zeitkomplexität deutlich besser abschneiden als der naive Algorithmus (Abb. 13b). Insgesamt sind Quadtrees hierbei schneller als KD-Trees, da bei einer homogenen Verteilung die Baumhöhe durch das Teilen der Punktmengen in vier statt zwei Teile geringer ist als bei KD-Trees. Da die Algorithmen in-situ arbeiten, wird kein Seicher für die Suchen allokiert.

k-Nearest-Neighbor-Search

Für die Tests aus Abb. 14 wurde die Anzahl an Nachbarn fest auf zehn gesetzt, während die Punktanzahl variiert wurde. Für die Tests aus Abb. 15 wurde eine Punktmenge der Kardinalität eine Million gewählt und die Anzahl der zu findenden Nachbarn k variiert. Der naive Algorithmus besteht darin, die Punktmenge aufsteigend nach Distanz zum Anfragepunkt zu sortieren und die ersten k Punkte zurückzugeben. Für alle Tests wurde derselbe Anfragepunkt gewählt.

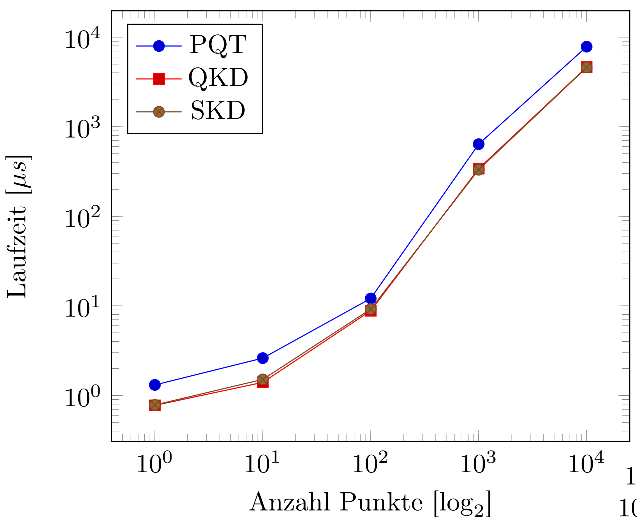


(a) Laufzeit [μs]

n	**Naiv [μs]**
9	3.76
10	8.34
11	18.4
12	40.5
13	87.3
14	187
15	400
16	852
17	1,802
18	3,811
19	8,055
20	17,268
21	37,580
22	79,346
23	$1.67 \cdot 10^5$
24	$3.54 \cdot 10^5$

(b) Naiver Algorithmus

Abbildung 14: k-NNS variables n, $k = 10$



(a) Laufzeit [μs]

k	**Naiv [μs]**
1	$2.06 \cdot 10^5$
10	$2.06 \cdot 10^5$
100	$2.05 \cdot 10^5$
1,000	$2.06 \cdot 10^5$
10,000	$2.05 \cdot 10^5$

(b) Naiver Algorithmus

Abbildung 15: k-NNS, variables k, $n = 10^6$

Für k-NNS mit festem k und variablem n zeigen die Ergebnisse in Abb. 15, dass KD-Trees schneller als Quadtrees sind. Die Unterschiede zwischen den beiden KD-Tree Varianten sind marginal. Dennoch sind QKD-Trees aufgrund der geringeren Speicherbelegung für die Bäume zu bevorzugen. Mit steigendem k sind KD-Trees deutlich schneller als Quadtrees. Da die Algorithmen in-situ arbeiten, wird kein Seicher für k-NNS allokiert.

5 Zusammenfassung und Ausblick

Zusammenfassung

Die Erkenntnisse, die aus den Experimenten in Abschnitt 4 gewonnen wurden, ergeben folgende Empfehlungen für die Wahl der Datenstrukturen in verschiedenen Szenarien, in denen sie eingesetzt werden können. Für kleine Punktmengen bietet sich die Verwendung von KD-Trees mit linearer Mediansuche an, während bei größeren Punktmengen PR-Quadtrees bevorzugt sind. In Anwendungen, bei denen eine geringe Speichernutzung essentiell ist, sind KD-Trees aus Abschnitt 2 aufgrund ihrer linearen Speicherkomplexität bevorzugt. Bei Bereichsanfragen sind Quadtrees, insbesondere PR-Quadtrees, schneller als KD-Trees. Die Speicherbelegung ist bei allen getesteten Datenstrukturen fast identisch. Die Suche nach einzelnen Punkten ist bei Quadtrees schneller als bei KD-Trees. Die Algorithmen allokieren keinen zusätzlichen Speicher. Bei k-NNS mit festem k und variablem n sind KD-Trees schneller als Quadtrees, wobei die Unterschiede zwischen den beiden KD-Tree-Varianten marginal sind. QKD-Trees sind aufgrund der geringeren Speicherbelegung und schnelleren Konstruktionszeit über SKD-Trees zu bevorzugen. Mit steigendem k sind KD-Trees deutlich schneller als Quadtrees.

Generell gibt es laut unseren Ergebnissen keinen Grund, Point-Quadtrees oder SKD-Trees zu nutzen, da die beiden Optimierungen aus Abschnitt 3.1 bzw. Abschnitt 2.1 stets bessere Laufzeiten und Speicherverbrauch verzeichnen. In allen getesteten Fällen sind die Laufzeiten von Quadtrees und KD-Trees den naiven Algorithmen deutlich überlegen.

Ausblick

Aufbauend auf dieser Einführung zu KD-Trees könnte man die Erweiterung vom zweidimensionalen auf den k-dimensionalen Raum vornehmen. Diese gestaltet sich als einfach, da man bei den Algorithmen lediglich anstatt zwischen x und y-Koordinaten nun zwischen $x_1, x_2, \ldots, x_k$ Koordinaten alternieren muss [Kle97, S.134]. Für Quadtrees ist die Betrachtung von Region-Quadtrees für Bildkomprimierung interessant. Bei dieser Variante repräsentiert die Wurzel des Baumes das gesamte Bild. Falls die Pixel in der Region eines Knotens nicht vollständig einfarbig sind, wird diese Region wieder in vier Quadranten unterteilt. Jeder Blattknoten repräsentiert einen einfarbigen Block von Pixeln [MR92]. Neben der hohen verlustfreien Komprimierung, sind Bearbeitungen des Bildes wie Rotation, Ausschneiden von Regionen und höhere verlustbehaftete Komprimierung leicht zu realisieren.

Literatur

[Dal18] Christoph Dalitz. *kdtree-cpp*. `https://lionel.kr.hsnr.de/~dalitz/data/kdtree/`. 2018.

[De 00] Mark De Berg. *Computational geometry: algorithms and applications*. Springer Science & Business Media, 2000, S. 95–105.

[FB74] Raphael A Finkel und Jon Louis Bentley. „Quad trees a data structure for retrieval on composite keys“. In: *Acta informatica* 4 (1974), S. 1–9.

[Kle97] Rolf Klein. *Algorithmische Geometrie*. Bd. 80. Springer, 1997.

[MR92] Tassos Markas und John Reif. „Quad Tree Structures for Image Compression Applications“. In: *Information Processing & Management* 28.6 (1992). Printed in Great Britain, ISSN: 0306-4573, S. 707–721. DOI: `IS.00+.@I`.

[OGR15] Ulises Olivares, Arturo García und Félix F Ramos. „Complete Quadtree Based Construction of Bounding Volume Hierarchies for Ray Tracing.“ In: *SIGRAD*. 2015, S. 120–004.

[Ore82] Jack A. Orenstein. „Multidimensional tries used for associative searching“. In: *Information Processing Letters* 14.4 (1982), S. 150–157. ISSN: 0020-0190. DOI: `https://doi.org/10.1016/0020-0190(82)90027-8`. URL: `https://www.sciencedirect.com/science/article/pii/0020019082900278`.

[Sam06] Hanan Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[Sam84] Hanan Samet. „The quadtree and related hierarchical data structures“. In: *ACM Computing Surveys (CSUR)* 16.2 (1984), S. 187–260.

[SN15] Johannes Schauer und Andreas Nüchter. „Collision detection between point clouds using an efficient kd tree implementation“. In: *Advanced Engineering Informatics* 29.3 (2015), S. 440–458.

Hier befinden sich die genauen Messwerte zu den Experimenten.

Kontruktion beider KD-Tree Varianten aus Abb. 2:

n	**SKD [ms]**	**QKD [ms]**	**SKD [kiB]**	**QKD [kiB]**
9	0.18	$2.6 \cdot 10^{-2}$	159	80
10	0.15	$5.4 \cdot 10^{-2}$	335	160
11	0.95	0.13	703	320
12	2.1	0.46	1,471	640
13	4.59	1.04	3,071	1,280
14	9.97	2.22	6,399	2,560
15	22.2	4.52	13,311	5,120
16	49.7	9.91	27,647	10,240
17	108	30.3	57,343	20,480
18	233	70.4	$1.19 \cdot 10^5$	40,960
19	503	145	$2.46 \cdot 10^5$	81,920
20	1,081	299	$5.08 \cdot 10^5$	$1.64 \cdot 10^5$
21	2,325	612	$1.05 \cdot 10^6$	$3.28 \cdot 10^5$
22	4,993	1,212	$2.16 \cdot 10^6$	$6.55 \cdot 10^5$
23	10,682	2,476	$4.46 \cdot 10^6$	$1.31 \cdot 10^6$
24	23,055	4,926	$9.18 \cdot 10^6$	$2.62 \cdot 10^6$
25	49,498	10,209	$1.89 \cdot 10^7$	$5.24 \cdot 10^6$

Abbildung 16: Kontruktionszeit [ms] und Speicherbelegung [kiB]

Kontruktion beider Quadtree Varianten aus Abb. 7:

n	**PQT [ms]**	**PRQT [ms]**	**PQT [kiB]**	**PRQT [kiB]**
9	0.16	$4.49 \cdot 10^{-2}$	173	76
10	0.27	0.1	357	158
11	0.57	0.21	744	332
12	1.13	2.38	1,541	692
13	2.32	3.92	3,115	1,473
14	4.57	2.83	6,350	3,050
15	9.39	3.81	12,923	6,368
16	20.8	12.4	26,402	13,218
17	43.3	16.5	53,755	25,744
18	88	33.7	$1.1 \cdot 10^{5}$	51,871
19	176	72.4	$2.23 \cdot 10^{5}$	$1.11 \cdot 10^{5}$
20	358	128	$4.55 \cdot 10^{5}$	$2.11 \cdot 10^{5}$
21	705	279	$9.28 \cdot 10^{5}$	$4.57 \cdot 10^{5}$
22	1,447	534	$1.89 \cdot 10^{6}$	$9.09 \cdot 10^{5}$
23	2,861	1,193	$3.84 \cdot 10^{6}$	$1.96 \cdot 10^{6}$
24	5,897	2,142	$7.81 \cdot 10^{6}$	$3.77 \cdot 10^{6}$
25	11,612	4,598	$1.59 \cdot 10^{7}$	$8.05 \cdot 10^{6}$

Abbildung 17: Kontruktionszeit [ms] und Speicherbelegung [kiB]

Vergleich der Höhen beider Quadtrees aus Abb. 8:

n	**SKDh**	**QKDh**
9	10	6
10	12	7
11	12	8
12	13	8
13	15	9
14	17	9
15	16	10
16	16	11
17	18	11
18	19	11
19	20	12
20	21	12
21	23	13
22	23	14
23	25	14
24	25	14
25	28	15

Abbildung 18: Höhen beider Quadtree Varianten

Konstruktionszeiten von Quadtrees und KD-Trees aus Abb. 9:

n	**SKD [ms]**	**QKD [ms]**	**PQT [ms]**	**PRQT [ms]**
9	0.18	$2.6 \cdot 10^{-2}$	0.16	$4.4 \cdot 10^{-2}$
10	0.15	$5.4 \cdot 10^{-2}$	0.27	0.1
11	0.95	0.13	0.57	0.21
12	2.1	0.46	1.13	2.38
13	4.59	1.04	2.32	3.92
14	9.97	2.22	4.57	2.83
15	22.2	4.52	9.39	3.81
16	49.7	9.91	20.8	12.4
17	108	30.3	43.3	16.5
18	233	70.4	88	33.7
19	503	145	176	72.4
20	1,081	299	358	128
21	2,325	612	705	279
22	4,993	1,212	1,447	534
23	10,682	2,476	2,861	1,193
24	23,055	4,926	5,897	2,142

Abbildung 19: Konstruktionszeiten beider Quadtree Varianten

Speicherbelegung von Quadtrees und KD-Trees aus Abb. 12:

n	**PQT [kiB]**	**PRQT [kiB]**	**SKD [kiB]**	**QKD [kiB]**
9	172	75	159	80
10	370	161	335	160
11	746	334	703	320
12	1,509	702	1,471	640
13	3,107	1,467	3,071	1,280
14	6,323	3,051	6,399	2,560
15	12,981	6,368	13,311	5,120
16	26,366	13,223	27,647	10,240
17	53,977	25,810	57,343	20,480
18	$1.1 \cdot 10^5$	51,949	$1.19 \cdot 10^5$	40,960
19	$2.23 \cdot 10^5$	$1.11 \cdot 10^5$	$2.46 \cdot 10^5$	81,920
20	$4.55 \cdot 10^5$	$2.11 \cdot 10^5$	$5.08 \cdot 10^5$	$1.64 \cdot 10^5$
21	$9.27 \cdot 10^5$	$4.57 \cdot 10^5$	$1.05 \cdot 10^6$	$3.28 \cdot 10^5$
22	$1.89 \cdot 10^6$	$9.09 \cdot 10^5$	$2.16 \cdot 10^6$	$6.55 \cdot 10^5$
23	$3.84 \cdot 10^6$	$1.96 \cdot 10^6$	$4.46 \cdot 10^6$	$1.31 \cdot 10^6$
24	$7.8 \cdot 10^6$	$3.77 \cdot 10^6$	$9.18 \cdot 10^6$	$2.62 \cdot 10^6$

Abbildung 20: Speicherbelegung beider Quadtree Varianten

Laufzeiten von Bereichsanfragen:

n	**SKD [ms]**	**PQT [ms]**	**QKD [ms]**	**PRQT [ms]**	**Naiv [μs]**
9	2.4	1.34	1.16	0.5	5.89
10	2.51	1.64	4.11	1.59	11.2
11	6.59	4.43	6.85	2.81	22.6
12	13.2	9.71	14.5	6.73	44.9
13	26.1	15.3	27	11.6	83.5
14	56.8	30.1	49.8	28.7	251
15	113	60.8	108	60.4	545
16	220	118	211	113	1,120
17	411	239	451	224	2,261
18	877	509	876	481	4,556
19	2,191	1,187	2,000	1,122	9,151
20	4,917	2,733	4,934	2,678	18,237
21	11,205	5,704	10,435	5,751	36,506
22	22,386	12,080	22,511	11,698	73,175
23	54,237	28,593	51,945	29,272	$1.47 \cdot 10^5$
24	$2.19 \cdot 10^5$	$1.2 \cdot 10^5$	$2.27 \cdot 10^5$	$1.19 \cdot 10^5$	$2.94 \cdot 10^5$

Abbildung 21: Höhen beider Quadtree Varianten

Speicherbelegung von Bereichsanfragen:

n	**SKD [kiB]**	**PQT [kiB]**	**QKD [kiB]**	**PRQT [kiB]**	NAIVEs
9	512	256	288	352	512
10	768	1,024	544	448	640
11	1,376	1,440	1,472	1,856	1,216
12	2,976	2,720	2,528	2,560	3,104
13	5,504	5,568	5,216	5,600	5,280
14	9,760	10,560	9,792	11,136	10,848
15	20,032	21,600	21,568	20,512	19,776
16	42,720	41,600	39,424	41,440	43,040
17	86,368	84,416	83,968	83,456	83,072
18	$1.65 \cdot 10^5$	$1.67 \cdot 10^5$	$1.68 \cdot 10^5$	$1.68 \cdot 10^5$	$1.67 \cdot 10^5$
19	$3.36 \cdot 10^5$	$3.41 \cdot 10^5$	$3.38 \cdot 10^5$	$3.4 \cdot 10^5$	$3.33 \cdot 10^5$
20	$6.68 \cdot 10^5$	$6.69 \cdot 10^5$	$6.72 \cdot 10^5$	$6.74 \cdot 10^5$	$6.71 \cdot 10^5$
21	$1.34 \cdot 10^6$	$1.33 \cdot 10^6$	$1.34 \cdot 10^6$	$1.33 \cdot 10^6$	$1.34 \cdot 10^6$
22	$2.69 \cdot 10^6$	$2.69 \cdot 10^6$	$2.68 \cdot 10^6$	$2.68 \cdot 10^6$	$2.69 \cdot 10^6$
23	$5.39 \cdot 10^6$	$5.34 \cdot 10^6$	$5.37 \cdot 10^6$	$5.4 \cdot 10^6$	$5.37 \cdot 10^6$
24	$1.07 \cdot 10^7$	$1.07 \cdot 10^7$	$1.08 \cdot 10^7$	$1.07 \cdot 10^7$	$1.07 \cdot 10^7$

Abbildung 22: Bereichsanfragen: Speicherbelegung [kiB]

Laufzeiten von Suchen nach 100 Punkten:

n	SKD [ms]	PQT [ms]	QKD [ms]	PRQT [ms]	Naiv [μs]
9	1.74	1.37	1.1	1.55	218
10	2.06	1.62	1.29	1.5	389
11	2.45	1.94	1.59	1.69	789
12	2.83	2.11	1.97	2.01	1,548
13	3.32	2.33	2.35	2.23	3,085
14	3.57	2.57	2.66	2.5	6,106
15	4.52	2.93	3.01	2.73	12,264
16	5.29	3.33	3.8	3.18	24,509
17	5.75	3.58	3.93	3.53	48,785
18	6.75	4.1	4.84	3.9	97,811
19	7.88	4.59	5.58	4.5	$1.95 \cdot 10^5$
20	8.71	4.92	6.2	4.73	$3.9 \cdot 10^5$
21	9.57	5.4	7.03	5.18	$7.78 \cdot 10^5$
22	11.4	5.99	8.63	5.62	$1.56 \cdot 10^6$
23	12.5	6.37	8.96	6.07	$3.14 \cdot 10^6$
24	14	6.91	10.3	6.74	$6.33 \cdot 10^6$

Abbildung 23: Suche: Laufzeit [μs]

Laufzeiten von k-NNS mit $k = 10$ aus Abb. 14:

n	PQT [ms]	QKD [ms]	SKD [ms]	Naiv [μs]
9	1.2	0.84	0.86	3.76
10	1.33	1.01	1.09	8.34
11	1.31	1.16	0.94	18.4
12	1.35	1.02	0.97	40.5
13	1.46	1.2	1.2	87.3
14	1.3	1.34	1.16	187
15	1.37	1.41	1.36	400
16	1.67	1.48	2.03	852
17	1.86	1.11	1.35	1,802
18	2.13	1.01	1.66	3,811
19	2.29	1.72	1.64	8,055
20	1.96	1.66	1.97	17,268
21	2.39	1.21	1.82	37,580
22	3.31	1.9	2.1	79,346
23	2.48	2.24	2.09	$1.67 \cdot 10^5$
24	2.14	2.27	1.98	$3.54 \cdot 10^5$

Abbildung 24: Suche: Laufzeit [μs]

Laufzeiten von k-NNS mit $n = 10^5$ aus Abb. 15:

k	**PQT [ms]**	**QKD [ms]**	**SKD [ms]**	**Naiv [μs]**
1	1.31	0.77	0.78	$2.06 \cdot 10^5$
10	2.61	1.4	1.51	$2.06 \cdot 10^5$
100	12.1	8.82	9.2	$2.05 \cdot 10^5$
1,000	639	341	331	$2.06 \cdot 10^5$
10,000	7,833	4,629	4,584	$2.05 \cdot 10^5$

Abbildung 25: Suche: Laufzeit [μs]

A Eigenständigkeitserklärung

Ich versichere an Eides statt, dass ich die vorliegende Arbeit in allen Teilen selbstständig und ohne unzulässige Hilfe Dritter absolviert sowie keine anderen als die genannten und explizit zugelassenen Hilfsmittel verwendet und mich im Allgemeinen prüfungskonform verhalten habe. Ich erkläre zudem, dass ich beim Einsatz von IT-/KI-gestützten Schreibwerkzeugen diese Werkzeuge in der Übersicht verwendeter Hilfsmittel mit ihrem Produktnamen, meiner Bezugsquelle und ... vollständig aufgeführt und/oder die betreffenden Textstellen in der Arbeit als mit KI generierter Unterstützung verfasst gekennzeichnet habe. Mir ist bewusst, dass Täuschungen bzw. Täuschungsversuche nach der für mich geltenden Prüfungsordnung geahndet werden.

Folgende Hilfsmittel habe ich für die Bearbeitung genutzt:

- ChatGPT 3.5 zum Erstellen von Latex-Code, zum Erstellen von variablen Query-Areas im Code.
- DeepL Writer zum Umformulieren einzelner Textpassagen