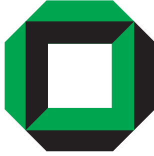

Universität Karlsruhe (TH)
Institut für Theoretische Informatik
Lehrstuhl Prof. Dr. Peter Sanders

Nächster-Nachbar-Suche mittels Knotenhierarchie in der Delaunay-Triangulierung

Studienarbeit

vorgelegt von

Marcel Birn

Betreuer

Dipl.-Inf. Johannes Singler
Prof. Dr. Peter Sanders

Karlsruhe, 15. Februar 2009

Zusammenfassung

Das Suchen des nächsten Nachbarn eines Anfragepunkts in einer gegebenen Punktmenge ist ein klassisches Problem der algorithmischen Geometrie. Wir präsentieren ein neues Verfahren, das durch Ergebnisse aus der Routenplanung (insbesondere Contraction Hierarchies [10]) inspiriert ist. In eine Delaunay-Triangulierung der Punktmenge werden zusätzliche Kanten eingefügt, die als Abkürzung auf der Suche nach dem nächsten Nachbarn genutzt werden können. Unsere experimentellen Ergebnisse zeigen, dass unser einfaches Verfahren dank kompakter Datenstruktur etablierte Verfahren in den meisten Fällen schlägt.

Inhaltsverzeichnis

1 Einleitung **4**

2 Delaunay-Triangulierung **5**
2.1 Eigenschaften der Delaunay-Triangulierung 5
2.2 Beweise zum Schnitt von Kreisen 6
2.3 Delaunay-Triangulierung und nächster Nachbar 8

3 Bisherige Arbeiten **10**
3.1 Nächster-Nachbar-Suche mit Hilfe der Delaunay-Triangulierung 10
3.2 Delaunay-Hierarchie 12
3.3 Nächster-Nachbar-Suche in R-Bäumen 14
3.4 Nächster-Nachbar-Suche in einem k-d-Baum 16

4 Nächster-Nachbar-Suche mittels Knotenhierarchie **18**
4.1 Algorithmus NNK 19
4.2 Laufzeit und Speicherverbrauch 23

5 Implementierung **26**
5.1 Delaunay_triangulation_vertex_hierarchy_2 26
5.2 Triangulation_vertex_hierarchy_base_2 29
5.3 Exact_distance_2 30

6 Experimente **34**
6.1 Laufzeitvergleiche 34
6.2 Benötigter Speicher 37
6.3 Wie gut sind die inexakten Varianten? 38
6.4 Eigenschaften vom NNK-Graphen 39

7 Zusammenfassung **42**

Abbildungsverzeichnis **43**

Literatur **44**

1 Einleitung

Im Rahmen dieser Studienarbeit wird ein neuer Algorithmus zur Nächster-Nachbar-Suche im 2-dimensionalen Raum vorgestellt. Bei einer Nächster-Nachbar-Suche ist im Allgemeinen eine Datenbasis P von Elementen aus einem d-dimensionalen Raum gegeben, d.h. $P \subset R^d$. Weiter gibt es eine Distanzfunktion $dist(a, b)$, die die Distanz zwischen zwei Elementen $a, b \in P$ berechnet. Die Distanzfunktion kann den euklidischen Abstand berechnen aber auch einen beliebigen anderen Abstand. Unter einer Nächster-Nachbar-Suche versteht man nun, dass für eine Anfrage $q \in R^d$ das Elemente $p \in P$ zurückgegeben wird, das den Abstand zu q minimiert. Die einfachste Möglichkeit den nächsten Nachbarn von q zu finden ist, q mit jedem Element aus P zu vergleichen. Durch eine Vorverarbeitung von P, lässt sich die Suche deutlich beschleunigen.

Für die Nächster-Nachbar-Suche gibt es eine Vielzahl von Anwendungsmöglichkeiten. Die erste Anwendung, an die man dabei wahrscheinlich denkt, ist, wie weit bin ich von einem bestimmten Ort entfernt und wo befindet sich dieser. Zum Beispiel könnte die nächste Tankstelle oder das nächste Postamt von Interesse sein (das Problem wurde von Donald Knuth in *The Art of Computer Programming* danach auch *post-office problem* genannt). In diesem Fall wäre P eine Teilmenge von $\mathbb{R}^2$ und die Elemente stellen Koordinaten von Postämtern auf einer Landkarte (oder Tankstellen, ...) dar. Weitere Anwendungsmöglichkeit finden sich in der Mustererkennung von Daten. Dies kann zum Beispiel in der Spracherkennung der Fall sein [2]. Dabei können die Elemente von P Merkmalsvektoren einzelner Wörter entsprechen. Diese Merkmalsvektoren können aus den Spektrogrammen der einzelnen Wörter erzeugt werden. Ein weiterer Einsatzzweck ist die Bilderkennung, d.h. es wird zu einem gegebenen Bild das Bild in einer Datenbank gesucht, das ihm am stärksten ähnelt. Ein einfacher Ansatz für den zugehörigen Merkmalsvektor ist, dass jedes Pixel einem Eintrag in diesem entspricht und dann anhand der Unterschiede in den Farbwerten der Pixel eine Distanzfunktion aufgestellt wird. Dies hat aber zur Folge, dass die Vektoren sehr schnell sehr groß werden können und die Ergebnisse auch nicht sonderlich gut sein müssen, da nicht auf Strukturen oder Farbverschiebungen geachtet wird. Der Merkmalsvektor könnte auch aus den Farben, Strukturen, Formen und Bewegungen, die in den Bildern vorkommen, bestehen [6]. Die Nächster-Nachbar-Suche lässt sich auch zur Analyse von DNA-Sequenzen verwenden. Zum Beispiel ist es von Bedeutung zu wissen, ob eine gegebene DNA-Sequenz ein so genannter Promoter ist [4].

Der hier vorgestellte Algorithmus zu Nächster-Nachbar-Suche verwendet Delaunay-Triangulierungen einer gegebenen Punktemenge zur Suche. In der Arbeit wird zunächst die Delaunay-Triangulierung einer Punktemenge erklärt und einige ihrer wichtigen Eigenschaften. Anschließend werden bisherige Arbeiten zur Nächster-Nachbar-Suche vorgestellt. Im Anschluss wird dann der neue Algorithmus hergeleitet und seine Implementierung beschrieben. Zur Implementierung wurde die Programmbibliothek CGAL verwendet. Anschließend wird die Implementierung mit schon bestehenden Implementierungen verglichen.

2 Delaunay-Triangulierung

Unter einer Triangulierung einer Punktemenge P, in einem 2-dimensionalen Raum versteht man eine planare Aufteilung der konvexen Hülle von P in Dreiecke, wobei die Ecken der Dreiecke die Punkte aus P sind [3]. Eine Triangulation mit besonderen Eigenschaften ist die Delaunay-Triangulierung.

Definition 1 (Delaunay-Triangulierung [3])**:** *Für eine Delaunay-Triangulierung DT einer Punktemenge P gilt für jedes Dreieck $uvw \in DT(u, v, w \in P)$, dass der Umkreis U von uvw keinen weiteren Punkt aus P in seinem Innern enthält.*

Delaunay-Triangulierungen werden unter anderem zur Modellierung von Oberflächenstrukturen verwendet. Eine ihrer Eigenschaft ist, dass der minimale vorkommende Winkel der Triangulation maximiert wird [3]. Dadurch entstehen Dreiecke, die keine sehr kleinen oder sehr großen Winkel besitzen und dadurch nicht so „langgezogen" wirken.

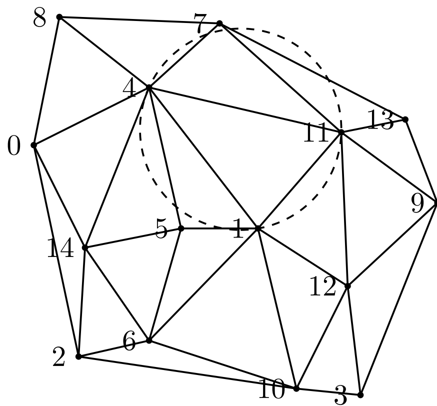

Abb. 1.1: Delaunay-Triangulierung mit einem Umkreis

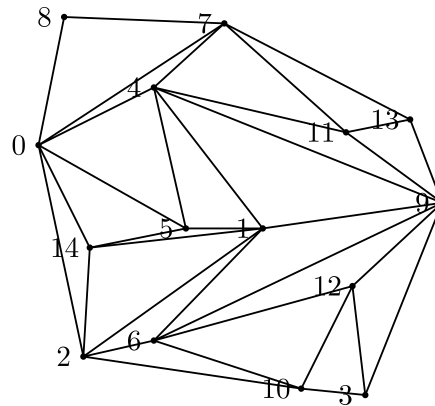

Abb. 1.2: Weitere Triangulation, der gleichen Punktemenge

Abbildung 1: Vergleich zwischen Delaunay-Triangulierung und einer einfachen Triangulierung

In dieser Arbeit werden drei Arbeiten vorgestellt, die Delaunay-Triangulierungen verwenden, um den nächsten Nachbarn einer Anfrage zu finden. Es werden auch nur Delaunay-Triangulierungen im euklidischen Raum betrachtet.

2.1 Eigenschaften der Delaunay-Triangulierung

Eine Delaunay-Triangulierung hat einige interessante Eigenschaften, die im weiteren Verlauf diese Arbeit verwendet werden. So sind bei einer Triangulation, d.h. nicht nur einer Delaunay-Triangulierung, die Anzahl der Kanten und Dreiecke fest durch die Anzahl der

Punkte auf der konvexen Hülle der Punktemenge vorgegeben.

Lemma 1 ([3])**:** *Für jede Triangulation einer Punktemenge P, bestehend aus n Punkten von denen k auf der konvexen Hülle von P liegen gilt, dass die Anzahl der Kanten*

$$m = 3n - 3 - k$$

und die Anzahl der Dreiecke

$$n_D = 2n - 2 - k$$

beträgt.

Ähnlich der Definition der Delaunay-Triangulierung, dass drei Punkte nur dann ein gemeinsames Dreieck bilden, wenn ihr Umkreis keinen weiteren Punkt enthält, lässt sich eine Aussage darüber treffen, wann zwei Punkte eine gemeinsame Kante in der Delaunay-Triangulierungbilden können.

Lemma 2 ([3])**:** *Zwei Punkte $p_i, p_j \in P$ bilden eine Kante der Delaunay-Triangulierung genau dann, wenn es einen Kreis gibt, mit p_i und p_j auf seinem Rand, der keinen weiteren Punkt $\in P$ in seinem Innern enthält.*

2.2 Beweise zum Schnitt von Kreisen

In dieser Arbeit werden bei Beweisen zur Delaunay-Triangulierung häufig die beiden folgenden Lemmas verwendet, die Aussagen darüber machen, wie zwei Kreise zueinander liegen. Das erste Lemma beschreibt, wie viele gemeinsame Punkte zwei Kreise auf ihrem Rand besitzen können.

Lemma 3: *Die Ränder zweier Kreise C_1 und C_2 haben entweder keinen, einen, zwei oder unendlich viele gemeinsame Punkte.*

Das zweite Lemma besagt, wie sich zwei Kreise schneiden, die beide von der gleichen Strecke geschnitten werden.

Lemma 4: *Gegeben sei ein Kreis C_1 und zwei Punkte t_1 und t_2, die nicht innerhalb von C_1 liegen (auf dem Rand ist erlaubt) und für die gilt, dass die Strecke $\overline{t_1 t_2}$ C_1 schneidet. Die Gerade durch $\overline{t_1 t_2}$ sei g. Die Gerade g teilt den Kreis C_1 in zwei Seiten auf, die mit A bzw. B bezeichnet werden. Für einen weiteren Kreis C_2 durch t_1, t_2 und einen Punkt s, der auf Seite A auf dem Rand von C_1 liegt, gilt: Das Kreissegment von C_1 auf Seite B liegt innerhalb von C_2, d.h.*

$$C_1' := \{p \in C_1 | p \text{ liegt auf Seite } B \text{ von } g\} \subset C_2.$$

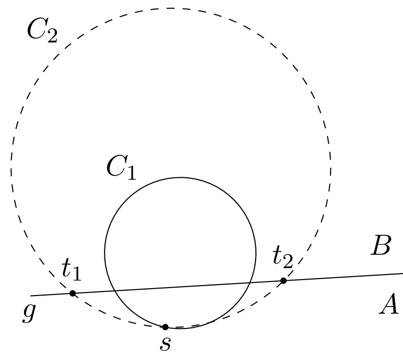

Abbildung 2: Schnitt von zwei Kreisen mit einer Geraden.

Beweis: Nach Lemma 3 können C_1 und C_2 nur einen, zwei oder unendlich viele gemeinsame Punkte besitzen (s ist bereits ein gemeinsamer Punkt).

Fall 1 („unendlich viele gemeinsame Punkte“): Die Kreise C_1 und C_2 sind identisch und somit gilt $C_1 \subset C_2$.

Fall 2 („zwei gemeinsame Punkte“): Hier gibt es zwei mögliche weitere Fälle: Entweder liegt der zweite Schnittpunkt s_2 auf Seite A oder B.
Falls s_2 auf Seite A liegt können C_1 und C_2 keinen gemeinsamen Punkt mehr auf Seite B besitzen. Die beiden Kreise hätten ansonsten drei gemeinsame Punkte (s, s_2 und einen weiteren auf Seite B) und wären somit identisch (Lemma 3) und es gilt Fall 1. Weiter gilt, dass t_1 und t_2 nicht innerhalb von C_1 liegen (wenn überhaupt auf dem Rand) und C_1 zwischen ihnen liegt. Daraus folgt, dass $C_1^{'} \subset C_2$, ansonsten müssten sich C_1 und C_2 nochmals auf Seite B schneiden.
Liegt t_1 oder t_2 auf dem Rand von C_1 so müssen C_1 und C_2 identisch sein (die beiden Kreise haben 3 gemeinsame Punkte, s, s_2 und t_1 oder t_2), es gilt also der vorherige Fall. Liegen t_1 und t_2 nun beide nicht auf C_1, so können C_1 und C_2 keinen weiteren Schnittpunkt auf der Seite B besitzen. Betrachtet man als erstes die Schnittpunkte p_1 und p_2 von g mit C_1, so bilden diese zusammen mit s und s_2 ein Sehnenviereck. Für dieses gilt, dass die Summe gegenüberliegender Winkel 180° beträgt [8]. Betrachtet man nun das Viereck $t_1 s t_2 s_2$, so müssen die beiden Winkel bei s bzw. s_2 größer sein als beim Viereck $p_1 s p_2 s_2$. Folglich muss deren Summe $> 180°$ sein und $t_1 s t_2 s_2$ können kein Sehnenviereck bilden, also auch nicht auf einem gemeinsamen Kreis liegen. D.h. der Schnittpunkt s_2 kann nicht existieren und es gilt $C_1^{'} \subset C_2$.

Fall 3 („ein gemeinsamer Punkt“): s ist schon der gemeinsame Punkt es kann also kein weiterer Schnittpunkt auf Seite B existieren, folglich gilt ebenfalls $C_1^{'} \subset C_2$. □

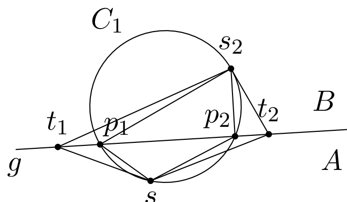

Abbildung 3: Sehnenviereck.

2.3 Delaunay-Triangulierung und nächster Nachbar

Eine Delaunay-Triangulierung hat auch einige interessante Eigenschaften, die einen Anfragepunkt q und seinen nächsten Nachbarn v in der Delaunay-Triangulierung betreffen. Im Gegensatz zu einem Voronoi-Diagramm reicht eine einfach Punktlokalisierung von q, d.h. in welchem Dreieck sich q in der Delaunay-Triangulierung befindet, nicht aus, um den nächsten Nachbarn angeben zu können. Ein Voronoi-Diagramm einer Punktemenge ist der duale Graph, der zugehörigen Delaunay-Triangulierung [3]. Hat man die Zelle des Voronoi-Diagramms gefunden, in der q enthalten ist, so müssen nur noch die Ecken dieser Zelle überprüft werden, um den nächsten Nachbarn von q zu finden.

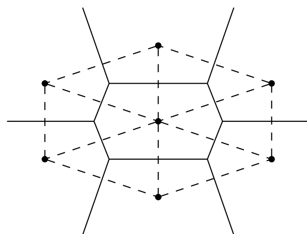

Abbildung 4: Voronoi-Diagramm und Delaunay-Triangulierung (gestrichelt)

In einer Delaunay-Triangulierung muss der nächste Nachbar von q nicht einer der Knoten des umschließenden Dreiecks sein (Abb. 5). Es lässt sich aber trotzdem eine Aussage darüber treffen, wie q zu seinem nächsten Nachbarn liegt.

Lemma 5: *Sei v der nächste Nachbar von q in der Punktemenge P. Falls q innerhalb der konvexen Hülle von P liegt oder sich eine Kante zwischen q und v befindet, gilt für den Punkt q, dass er innerhalb des Umkreises eines zu v gehörenden Dreiecks liegt.*

Beweis: Liegt q innerhalb eines an v angrenzenden Dreiecks $\triangle$, so muss q auch folglich innerhalb des Umkreises von $\triangle$ liegen. Es wird nun der Fall betrachtet, dass q nicht innerhalb eines an v angrenzenden Dreiecks liegt, sich dafür aber eine Kante $\overline{mn}$ zwischen q und v befindet (ist immer der Fall, falls q innerhalb der konvexen Hüllen von P liegt).

Für einen Kreis C mit q als Mittelpunkt und v auf seinem Rand gilt, dass C leer ist, da v der nächste Nachbar von q ist. Daraus folgt, dass C von der Strecke $\overline{mn}$ geschnitten wird. Da v auf dem Rand von C liegt muss nach Lemma 4 q im Innern des Umkreises

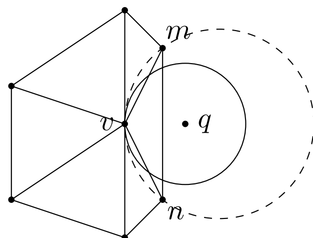

Abbildung 5: Anfragepunkt q liegt innerhalb des Umkreises eines Dreiecks des nächsten Nachbarn von q.

vom Dreieck vmn liegen.
Somit liegt q immer innerhalb eines Umkreises von einem an v angrenzenden Dreieck.

□

Der Anfragepunkt kann durch mehr als einer Strecke von seinem nächsten Nachbarn getrennt sein (Abb. 6).

Lemma 6: *Für jedes Dreieck △ das zwischen dem Anfragepunkt q und seinem nächsten Nachbarn v liegt, gilt: Der Umkreis von △ enthält q.*

Beweis: Falls ein Dreieck zwischen q und v liegt, so liegen auch mindestens zwei Strecken S zwischen q und v, d.h. die beiden Strecken schneiden $\overline{qv}$. Für den Umkreis U, des an v angrenzenden Dreiecks, der q enthält (Lemma 5) gilt, dass er diese Strecken S schneidet aber keinen der Endpunkte im Innern enthält (nach Voraussetzung ist U leer). Für jeden Umkreis der Dreiecke, die eine Strecke aus S besitzen, muss gelten, dass diese U nicht auf der Seite von q schneiden, da sonst nach Lemma 4, v in ihm enthalten ist und somit kann das zugehörige Dreieck keines der Delaunay-Triangulierung sein (Ausnahme: v ist eine Ecke des Dreiecks, dann enthält der Umkreis aber q).

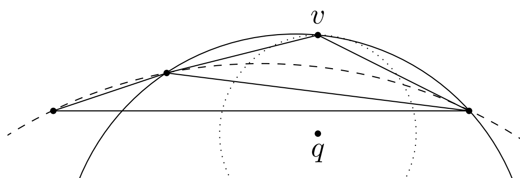

Abbildung 6: Dreieck zwischen Anfragepunkt und nächstem Nachbarn.

Da U von den Umkreisen auf der Seite von q nicht geschnitten wird, müssen diese q enthalten. □

3 Bisherige Arbeiten

Die einfachste Möglichkeit den nächsten Nachbarn eines Punkts q in einer Punktemenge P zu finden ist den Abstand von q zu jedem Punkt aus P zu berechnen und sich dabei den Punkt mit dem geringsten Abstand zu merken. Die Menge P in der der nächste Nachbar gesucht wird, wird im Rest dieser Arbeit auch mit Stellen bezeichnet.

```
einfacher_nächster_nachbar(q,P):
    naechster_nachbar = beliebiger Punkt aus P

    forall x ∈ P do
      if (|x q| < |naechster_nachbar q|) do
        naechster_nachbar = p
      done
    done

    return naechster_nachbar
```

Es ist nicht sonderlich effizient bei jeder Anfrage nach dem nächsten Nachbarn von q die komplette Punktemenge zu durchsuchen. Durch Vorberechnungen ist es möglich Datenstrukturen zu erzeugen, bei denen es nicht mehr notwendig ist jeden einzelnen Punkt zu betrachten, sondern nur noch einen Bruchteil. Mit Hilfe von Bäumen (k-d-Baum, r-Baum) werden die Punkte aus P zu einzelnen Teilmengen zusammengefasst, die jeweils ein zusammenhängendes Gebiet umfassen. Bei r-Bäumen dürfen sich diese Gebiete auch überschneiden, dafür ist man aber nicht auf Punkte eingeschränkt, sondern kann beliebige geometrische Objekte verwenden. Durch Zusammenfassen der Punkte zu regionalen Teilmengen ist es möglich, während der Suche Mengen von Punkten auszuschließen und so schneller den nächsten Nachbarn zu finden.

Ein weiterer Ansatz den nächsten Nachbarn zu finden ist die Verwendung von Delaunay-Triangulierungen. Hierbei wird ein Graph aufgebaut, der es erlaubt sich zielgerichtet sich dem nächsten Nachbarn von q zu nähern, indem man von einem Startknoten aus Kanten entlang geht, die einen näher zu q führen.

3.1 Nächster-Nachbar-Suche mit Hilfe der Delaunay-Triangulierung

Zum Finden des Nächsten Nachbarn, einer Anfrage q, in einer Delaunay-Triangulierung DT geht man wie folgt vor: Die Suche beginnt bei einem beliebigen Knoten von DT. Vom aktuellen Knoten aus betrachtet man alle benachbarten Knoten, d.h. die Knoten, die über eine Kante erreicht werden können, dann überprüft man, ob diese näher an q liegen. Die Kante, die zu dem Knoten führt, der am nächsten an q liegt, wird traversiert. Diesen Schritt wiederholt man so lange, bis man keinen weiteren Knoten findet, der näher an q liegt. Der letzte Knoten ist dann der nächste Nachbar.

delaunay_nächster_nachbar(q,DT):

```
naechster_nachbar = beliebiger Knoten der DT

do
  kandidat = naechster_nachbar
  forall v Nachbar von naechster_nachbar do
    if (|v q| < |kandidat q|) do
      kandidat = v
    done
  done
  if (kandidat ≠ naechster_nachbar) do
    naechster_nachbar = kandidat
  done
until (keinen naeheren Knoten gefunden)

return naechster_nachbar
```

Algorithmus 1: Nächster Nachbar mit Delaunay-Triangulierung

Im folgenden wird nun die Korrektheit dieses Algorithmus bewiesen: Es ist nicht möglich, dass man zu einem Knoten v gelangt dessen Nachbarn alle weiter von q entfernt sind als er selbst, v aber nicht der nächste Nachbar von q ist. D.h. es existiert keine Kante von v zum nächsten Nachbarn von q.

Beweis: Annahme: Wir erreichen einen Knoten v der Delaunay-Triangulierung, dessen Nachbarn $t_1, \ldots, t_k$ alle weiter vom Anfragepunkt q entfernt sind, v aber trotzdem nicht der nächste Nachbar von q ist. D.h. es muss ein Knoten $p \in P$ existieren, der innerhalb des Kreises C ist, wobei C q als Mittelpunkt hat und v auf seinem Rand liegt. Weiter gilt, dass $t_1, \ldots, t_k$ nicht im Innern von C sind (die Punkte wären sonst näher an q als v). Nach Voraussetzung verbindet keine Kante die beiden Punkte v und p. Es muss also eine Kante $\overline{mn}$ ($m, n \in P$) existieren, die die Kante $\overline{vp}$ schneidet, wobei weder m noch n innerhalb von C liegen.

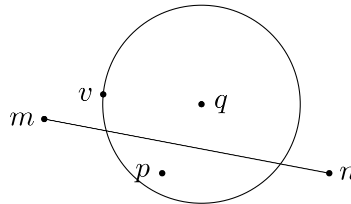

Abbildung 7: Knoten v wird durch Strecke $\overline{mn}$ vom nächsten Nachbarn p von q getrennt.

Für jeden Kreis U durch m und n, der C auf Seite A schneidet, gilt, dass der Teil von C auf Seite B innerhalb von U liegt (Lemma 4). Somit muss auch v innerhalb von U liegen. Der Fall, dass v auf dem Rand von U liegt ist nur möglich, wenn C und U identisch sind. Dann liegt aber p innerhalb von U (nicht auf dem Rand!). Falls C von U nicht auf

der Seite A geschnitten wird liegt der Teil von C auf Seite A innerhalb von U und somit auch p.

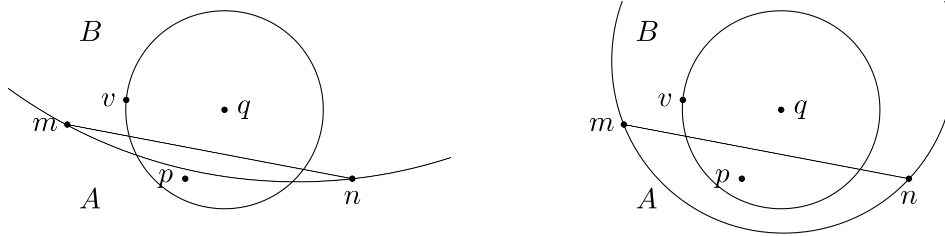

Abbildung 8: Die Umkreise von m und n können nicht leer sein.

Es existiert also kein Kreis mit m und n auf seinem Rand, der leer ist. D.h. er enthält keinen Punkt aus P in seinem Innern und $\overline{mn}$ kann also keine Kante der Delaunay-Triangulierung sein (Lemma 2). Es muss also entweder eine Kante $\overline{vp}$ der Delaunay-Triangulierungexistieren, oder es existiert ein weiterer Punkt $z \in P$ innerhalb von C, der eine Kante $\overline{zy}$ $(y \in P)$ besitzt die $\overline{vp}$ schneidet aber der auch eine Kante zusammen mit v besitzt (folgt rekursiv aus obiger Folgerung, dass keine Kante der Delaunay-Triangulierung existieren kann, die v und z trennt und deren Endpunkte nicht innerhalb von C sind). Es existiert also eine Kante von v zu einem Knoten, der näher an p liegt. Dies ist aber ein Widerspruch zu der Annahme, dass man in einer Sackgasse landen kann.

In jedem Schritt kommt man dem nächsten Nachbarn somit näher und da die Delaunay-Triangulierung nur endlich viele Knoten besitzt, muss der Algorithmus nach endlich vielen Schritten terminieren. Ein Knoten der einmal besucht wurde, kann kein zweites mal besucht werden, da er weiter entfernt sein muss. □

3.2 Delaunay-Hierarchie

In [5] wird die Delaunay-Hierarchie vorgestellt. Dies ist ein Verfahren zur Nächster-Nachbar-Suche das mehrere Ebenen von Delaunay-Triangulierungen verwendet (Abb. 9). Dazu werden zufällige Teilmengen P_i von P verwendet, die wie folgt aussehen:

$$P = P_0 \supset P_1 \supset \ldots \supset P_k$$

Ein Punkt p aus P_i ist mit der Wahrscheinlichkeit $1/\alpha \in (0,1)$ auch in P_{i+1} enthalten. Daraus ergibt sich eine erwartete Größe von $O(\log(n))$ $(n = |P|)$ für k. Für jede Punktemenge P_i gibt es die zugehörige Delaunay-Triangulierung DT_i. Ein Punkt p ist Teil von Ebene i, falls er $\in P_i$ und $\notin P_{i+1}$ ist. Der zugehörige Knoten v ist ein Knoten von $DT_0, \ldots, DT_i$ und hat für jede Delaunay-Triangulierung einen Verweis zu einem benachbarten Dreieck. Ein Dreieck hat Verweise zu seinen drei benachbarten Dreiecken und Knoten. Durch diese Darstellung ist es nicht notwendig die Kanten direkt zu speichern, diese lassen sich durch die gegebenen Dreiecke und zugehörigen Knoten bestimmen.

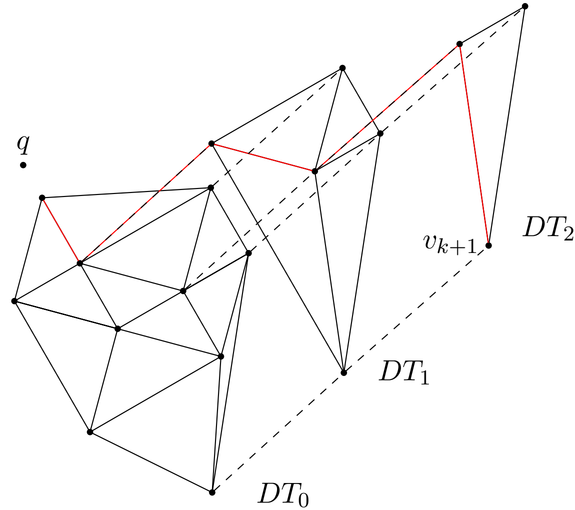

Abbildung 9: Delaunay-Hierarchie.

Die Suche nach dem nächsten Nachbarn eines Punkts q beginnt bei einem Knoten v_{k+1} der höchsten Delaunay-Triangulierung DT_k. Von v_{k+1} aus wird nach dem nächsten Nachbarn v_k von q in DT_k gesucht. Sobald v_k gefunden wurde, geht die Suche in der nächst niedrigeren Ebene DT_{k-1} weiter. Als Startknoten für die Suche nach dem nächsten Nachbarn in DT_{k-1} wird v_k verwendet (v_k ist auch ein Knoten von DT_{k-1}, da $P_k \subset P_{k-1}$). Sobald der neue nächste Nachbar v_{k-1} gefunden wurde, geht die Suche in der nächst niedrigeren Ebene weiter. Dies wird so lange wiederholt bis man in DT_0 angekommen ist und dort den nächsten Nachbarn v_0 von q gefunden hat. Da $P = P_0$ ist v_0 der gesuchte nächste Nachbar von q in P.

Die Suche nach dem nächsten Nachbarn von q in DT_i erfolgt in drei Schritten:

Im ersten Schritt wird ausgehend vom Knoten v_{i+1} nach dem nächste Nachbarn gesucht. Dazu wird zunächst mit Hilfe des gespeicherten, angrenzende Dreiecks von v_{i+1}, in DT_i, nach dem angrenzenden Dreieck von v_{i+1} gesucht, das von der Strecke $\overline{v_{i+1}q}$ geschnitten wird.

Im zweiten Schritt wird von diesem Dreieck aus entlang $\overline{v_{i+1}q}$ gegangen, bis das Dreieck t_i gefunden wird, das q enthält.

Im dritten und letzten Schritt wird nun der nächste Nachbar v_i von q in DT_i gesucht. Dazu werden alle Dreiecke betrachtet, die möglicherweise $\overline{qv_i}$ schneiden. D.h. beginnend mit Dreieck t_i werden rekursiv die relevanten benachbarten Dreiecke (d.h. die Dreiecke mit einer gemeinsamen Kante) betrachtet, wobei ein benachbartes Dreieck nicht relevant ist, falls eine der folgenden Bedingungen zutrifft. Nach Lemma 6, gilt dass der Umkreis C' jedes von $\overline{qv_i}$ geschnittenen Dreiecks q enthält. Für das zugehörige Dreieck $ww'w''$ gelte o.B.d.A $|qw| \leq |qw'|, |qw''|$. Weiter sei C der Kreis mit Mittelpunkt q und Radius

$|qw|$.

- Ein benachbartes Dreieck ist nicht relevant, falls es zuvor schon besucht wurde.
- Falls C' nicht von C zwischen w' und w'' geschnitten wird, muss der Nachbar mit der gemeinsamen Kante $\overline{w'w''}$ nicht betrachtet werden. Das „äußere" Kreissegment von C' der Kante $\overline{w'w''}$ ist leer und somit in diesem Bereich auch C. Folglich kann die Kante $\overline{w'w''}$ nicht von $\overline{qv_i}$ geschnitten werden.

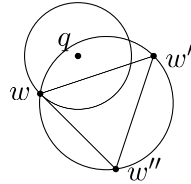

Abbildung 10: Ausschluss von Dreiecken in der Delaunay-Hierarchie.

- Falls der Winkel zwischen qww' größer als 90° ist,schneidet C nicht $\overline{ww'}$ und somit wird auch nicht $\overline{ww'}$ von $\overline{qv_i}$ geschnitten. Das benachbarte Dreieck mit der gemeinsamen Kante $\overline{ww'}$ ist also nicht relevant. Entsprechend gilt dies auch für die Kante $\overline{ww''}$.

3.3 Nächster-Nachbar-Suche in R-Bäumen

Im folgenden wird eine Verfahren zur Suche des nächsten Nachbarn mit Hilfe von R-Bäumen, aus [11], vorgestellt. R-Bäume ermöglichen es, beliebige geometrische Objekte zu speichern und auf diese gezielt zuzugreifen. Dabei wird eine Menge von Objekten in einem minimal umschließenden Rechteck (MBR, minimum bounding rectangle) zusammengefasst. Das Prinzip der MBRs lässt sich gut auf beliebige Dimensionen verallgemeinern. Ein Rechteck im euklidischen Raum der Dimension n wird von zwei Endpunkten $S = (s_1, \ldots, s_n)$ und $T = (t_1, \ldots, t_n)$ aufgespannt, für die gilt:

$$\forall i \in \{1, \ldots, n\} : s_i \leq t_i.$$

Das MBR einer Objektmenge $O = \{o_1, \ldots, o_m\}$ ist wie folgt definiert:

$$s_i = \min\{x_i | X = (x_1, \ldots, x_n) \in O\}, \; i \in \{1, \ldots, n\}$$
$$t_i = \max\{x_i | X = (x_1, \ldots, x_n) \in O\}, \; i \in \{1, \ldots, n\}$$

Darauf aufbauend werden in [11] zwei Distanzen von einem Punkt P zu einem Rechteck R definiert (wobei nur die quadrierten Distanzen betrachtet werden, diese sind leichter zu berechnen). Zum einen $\mathit{MINDIST}$ und zum anderen $\mathit{MINMAXDIST}$.

$\mathit{MINDIST}(P, R)$ ist die minimale Distanz zwischen P und R. Liegt P auf dem Rand

oder innerhalb von R so ist $MINDIST(P, R) = 0$ und ansonsten gilt

$$MINDIST(P, R) = \sum_{i=1}^{n} (p_i - r_i)^2$$

wobei

$$r_i = \begin{cases} s_i & \text{falls } p_i < s_i \\ t_i & \text{falls } p_i > t_i \\ p_i & \text{sonst} \end{cases} .$$

$MINMAXDIST(P, R)$ ist die minimale Distanz aller maximaler Distanzen von P zu einer Seite von R. Die Abb. 11 zeigt ein Beispiel für den 2-Dimensionalen Fall.

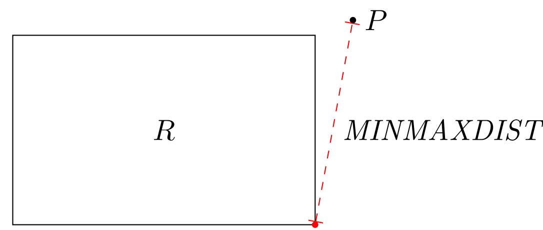

Abbildung 11: $MINMAXDIST$ von einem Punkt P zu einem Rechteck R.

$MINMAXDIST$ lässt sich wie folgt berechnen.

$$MINMAXDIST(P, R) = \min_{1 \leq k \leq n} \left((p_k - rm_k)^2 + \sum_{\substack{i \neq k \\ 1 \leq i \leq n}} (p_i - rM_i)^2 \right)$$

wobei rm_k die k-te Koordinate der Seite, die in Dimension k am nächsten an P liegt ist, d.h.

$$rm_k = \begin{cases} s_k & \text{falls } p_k \leq \frac{s_k + t_k}{2} \\ t_k & \text{sonst} \end{cases}$$

und rM_i die i-te Koordinate der Seite, die in Dimension i, am weitesten von P entfernt ist, d.h.

$$rM_i = \begin{cases} s_i & \text{falls } p_k \geq \frac{s_k + t_k}{2} \\ t_i & \text{sonst} \end{cases} .$$

Bei der Suche nach dem nächsten Nachbarn eines Punkts P in einem R-Baum lassen sich anhand der folgenden drei Regeln einzelne Knoten bzw. Objekte ausschließen.

1. Falls für das MBR R eines Knotens N, ein weiteres MBR R' existiert mit $MINDIST(P, R) > MINMAXDIST(P, R')$ so muss der Knoten N nicht weiter beachtet werden. Es kann den nächsten Nachbarn von P nicht enthalten.

2. Ein Objekt o, für das die Entfernung zu P größer ist, als die Entfernung $MINMAXDIST(R, P)$ eines MBR R zu P, muss ebenfalls nicht weiter beachtet werden.

3. In einem MBR R, für das $MINDIST(P, R)$ größer ist als die Entfernung von P zu einem Objekt o, kann der nächste Nachbar von P nicht enthalten sein und der zugehörige Knoten braucht somit nicht weiter verfolgt zu werden.

Der Algorithmus selber durchläuft den Baum, bei der Wurzel beginnend, mit Hilfe einer Tiefensuche. Die anfängliche Distanz von P zu seinem nächsten Nachbarn beträgt ∞. Für jeden Knoten, der kein Blatt ist, wird wie folgt vorgegangen: Zuerst wird *MINDIST* für die nachfolgenden Knoten berechnet und anschließend werden die Knoten anhand dieser Distanz aufsteigend sortiert, in einer Liste L eingefügt. Die Knoten werden in aufsteigender Reihenfolge eingefügt, da *MINDIST* schon eine Abschätzung der Entfernung zum nächsten Nachbarn ist. Daraus ergibt sich die Hoffnung, dass sich der nächste Nachbar in einem Knoten mit geringerer $MINDIST$ befindet und so die restlichen Knoten frühzeitig ausgeschlossen werden können. Anhand der Regeln 1 und 2 werden nun alle Knoten aus L entfernt, die nicht den nächsten Nachbarn enthalten können. Danach wird die Suche rekursiv auf den restlichen Knoten weiter verfolgt. Gelangt die Suche bei einem Blattknoten an, so wird in diesem nach dem vorläufigen nächsten Nachbarn gesucht und anschließend, bevor der nächste Knoten rekursiv aufgerufen wird, die jeweilige Liste anhand Regel 3 überprüft, ob weitere Knoten entfernt werden können. Dies wird so lange verfolgt, bis es keine Liste mehr gibt, die noch Elemente enthält.

3.4 Nächster-Nachbar-Suche in einem k-d-Baum

In [2] werden k-d-Bäume als eine Erweiterung von binären Bäumen vorgestellt. Diese dienen zur strukturierten Speicherung von k-dimensionalen Elementen. Unter anderem wird ein Algorithmus zur Nächster-Nachbar-Suche beschrieben, der in [7] weiter verfeinert wird, durch eine Optimierung des k-d-Baums. Jeder Knoten des Baums beschreibt einen Teil des k-d-Raums und hat zwei Verweise auf Nachfolgeknoten, die den Raum noch weiter partitionieren. Die Blattknoten enthalten die zu speichernden Elemente, wobei die Anzahl der Elemente, die in einem Blattknoten gespeichert werden können, begrenzt ist. Beginnend mit dem Wurzelknoten, der den gesamten k-d-Raum repräsentiert, wird dieser wie folgt unterteilt. Jedes Blatt verwendet eine Dimension des k-d-Raums, um die Elemente aufzuteilen, d.h. die Elemente werden anhand einer Koordinate aufgeteilt. Der Wert, der entscheidet, ob ein Element zum linken oder rechten Nachfolgeknoten gehört, entspricht dem Median der Elemente in der j-ten Koordinate. Der Median wird gewählt, damit die beiden Nachfolgeknoten immer ungefähr gleiche viele Elemente enthalten. Welche Koordinate pro Knoten ausgewählt wird, um die Elemente aufzuteilen, hängt davon ab, in welcher Dimension die Elemente den größten Wertebereich aufspannen.

Die Suche nach dem nächsten Nachbarn eines Punkts p beginnt bei der Wurzel, zusätz-

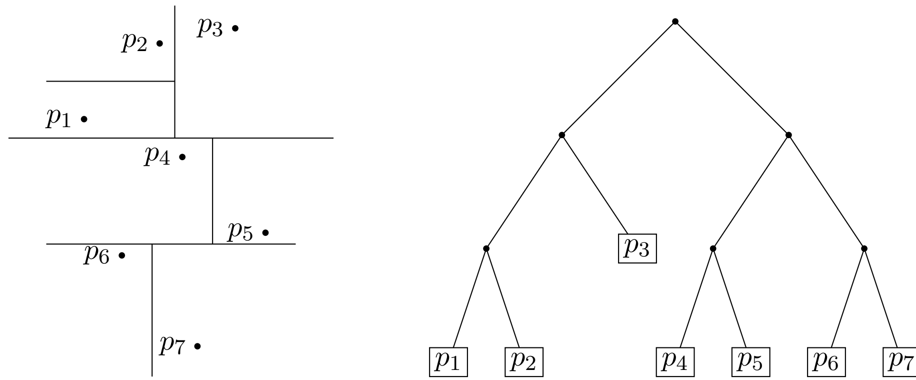

Abbildung 12: Beispiel für einen k-d-Baum.

lich wird immer noch der vorübergehende nächste Nachbar gespeichert. Anfangs liegt dieser im „Unendlichen". Für jeden Knoten wird zunächst unterschieden, ob er ein Blattknoten ist oder nicht. Ist ein Knoten kein Blattknoten, so wird überprüft in welchem Nachfolgeknoten p enthalten ist, d.h. es wird ermittelt, ob die j-te Koordinate kleiner oder größer ist, als der Wert der im aktuellen Knoten zum Aufteilen verwendet wurde. Ist der Wert kleiner, so erfolgt die Suche rekursiv im linken Nachfolgeknoten und sonst im rechten.Falls die Suche einen Blattknoten überprüft, so wird in diesem nach dem nächsten Nachbarn gesucht, d.h. es wird überprüft, ob ein Element näher an p liegt als der nächste bisher gefundene.

Sobald ein Rekursionsschritt beendet wurde, ist es noch notwendig zu überprüfen, ob der nächste Nachbar nicht doch im anderen Nachfolgeknoten liegen kann. Dazu wird eine k-dimensionale Kugel, mit Radius Entfernung von p zum aktuell nächsten Nachbarn und Mittelpunkt p, betrachtet. Schneidet diese Kugel die Region des anderen Nachfolgeknotens nicht, so kann dieser kein Element enthalten das näher an p liegt und muss folglich nicht weiter betrachtet werden. Ansonsten erfolgt die Suche rekursiv in diesem. Um die Wahrscheinlichkeit möglichst gering zu halten, dass die Kugel die Region des anderen Nachfolgeknotens schneidet, wird beim Aufbau des k-d-Baumes immer die Koordinate gewählt, bei der die Elemente den größten Wertebereich aufspannen.

4 Nächster-Nachbar-Suche mittels Knotenhierarchie

In den vorherigen Kapiteln wurden schon zwei Verfahren vorgestellt, die die Delaunay-Triangulierung einer Punktemenge verwenden, um den nächsten Nachbarn eines Punkts q zu finden. Hier wird nun ein neuer Algorithmus dieser Art vorgestellt.

Viele Algorithmen zur Erstellung einer Delaunay-Triangulierung gehen inkrementell vor, d.h. die Punkte der Punktemenge P werden Schritt für Schritt in die Delaunay-Triangulierung eingefügt. Während jeder Einfügeoperation wird dabei die Delaunay-Triangulierung erstellt, die sich durch das Hinzufügen des weiteren Punkts ergibt. Um den neuen Punkt p hinzuzufügen, wird zunächst in der schon bestehenden Delaunay-Triangulierung nach dem Dreieck $\triangle$ gesucht, das p enthält [3]. Nun wird p durch Kanten mit den Ecken dieses Dreiecks verbunden, wodurch drei neue Dreiecke entstehen (vier falls p auf einer schon existierenden Kante liegt). Die so entstandenen Dreiecke sind aber möglicherweise keine Dreiecke der Delaunay-Triangulierung. Für die neu hinzugefügten Kanten $\overline{t_i p}$ (t_i Ecke von $\triangle$) gilt aber, dass sie Kanten der Delaunay-Triangulierung sind. Schrumpft man den Umkreis von $\triangle$, indem man den Mittelpunkt entlang der Strecke von t_i zu seinem ursprünglichen Mittelpunkt verschiebt, entsteht irgendwann ein Kreis der p auf seinem Rand liegen hat. Dieser Kreis enthält keinen weiteren Punkt und folglich ist $\overline{t_i p}$ eine Kante der Delaunay-Triangulierung (Lemma 2).

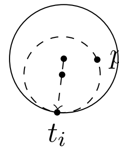

Abbildung 13: Schrumpfen eines Kreises.

Es müssen also nur noch die alten Kanten von $\triangle$ überprüft werden, ob sie weiterhin Kanten der neuen Delaunay-Triangulierung sind. Gilt für eine Kante, dass sie keine Kante der Delaunay-Triangulierung mehr ist, so findet ein so genannter *edge flip* statt. Hierbei wird die alte Kante durch die Kante von t_i zu dem Knoten der gegenüber der alten Kante liegt ersetzt. Diese Prozedur wird so lange rekursiv fortgesetzt, bis bei den so neu entstandenen Dreiecken alle Kanten auch Kanten der Delaunay-Triangulierung sind [3].

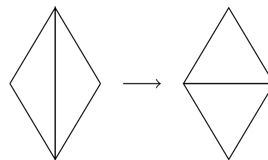

Abbildung 14: Edge flip

Anstatt die Kanten wie oben beschrieben zu löschen, werden diese gespeichert. Zu einer gegebenen Stellenmenge $P = \{p_0, \ldots, p_{n-1}\}$ bezeichnet DT_i die Delaunay-Tri-

angulierung, die nach dem Einfügen von Punkt p_i entsteht, während die Delaunay-Triangulierung von P inkrementell erstellt wird. Der Graph DT, der zusätzlich die gelöschten Kanten speichert, sieht wie folgt aus:

$$DT = \bigcup_{i=0}^{n-1} DT_i \ .$$

Beim Aufbau der Delaunay-Triangulierung muss nun darauf geachtet werden, nur Kanten der eigentlichen Delaunay-Triangulierung zu betrachten. An der Korrektheit des bereits vorgestellten Algorithmus 1 ändert sich dabei nichts.

Lemma 7: *Zu einer Delaunay-Triangulierung können beliebige Kanten hinzugefügt werden, ohne dass die Korrektheit von Algorithmus 1 verloren geht.*

Beweis: Durch die zusätzlichen Kanten kann es vorkommen, dass nicht mehr eine Kante der endgültigen Delaunay-Triangulierung traversiert wird, sondern eine der zusätzlichen. Wird eine solche Kante traversiert, kommt man dem Anfragepunkt q näher, kann aber nicht in einer Sackgasse landen. Zu Algorithmus 1 wurde bereits gezeigt, dass dies in einer Delaunay-Triangulierung nicht passiert. Der neu entstandene Graph ist aber nur eine Erweiterung der Delaunay-Triangulierung, d.h. die Kanten der eigentlichen Delaunay-Triangulierung existieren weiterhin. Wurde also nicht der nächste Nachbar von q erreicht, so führen zumindest die Kanten der Delaunay-Triangulierung näher zu q. Da die Knotenmenge der Triangulierung endlich ist und sich der Algorithmus in jedem Schritt q nähert, muss er auch terminieren. □

Aus der Speicherung der zusätzlichen Kanten ergibt sich eine erste Version unseres neuen Algorithmus zur Suche nach dem nächsten Nachbarn eines Punkts q. Die Idee hinter der Speicherung der zusätzlichen Kanten ist, dass diese Kanten Abkürzungen auf dem Weg zum nächsten Nachbarn sein können. D.h. es werden einzelne Knoten ausgelassen, die in der Delaunay-Triangulierung besucht würden, es können dafür aber auch andere Knoten dazukommen. Die Kanten gehören zu einer Delaunay-Triangulierung, deren Punkte eine Teilmenge P_i von P sind und führen zumeist zielgerichtet zum nächsten Nachbarn von q in P_i. Ein Vergleich zwischen Algorithmus 1 und dem gerade vorgestellten, ist in Abb. 15 dargestellt.

4.1 Algorithmus NNK

Die neuen Kanten können zwar eine Abkürzung auf dem Weg zum nächsten Nachbarn von q sein, es müssen dafür aber zusätzliche Kanten betrachtet werden.

Eine Möglichkeit dies zu vermeiden ist, die erste Kante zu nehmen, die näher ans Ziel führt. Dies ändert an der Korrektheit nichts, da solange man den nächsten Nachbarn noch nicht erreicht hat, mindestens eine Kante existiert, die einen näher zu q führt (siehe Beweis zu Algorithmus 1). Das Problem dabei ist, dass dieser Algorithmus unter

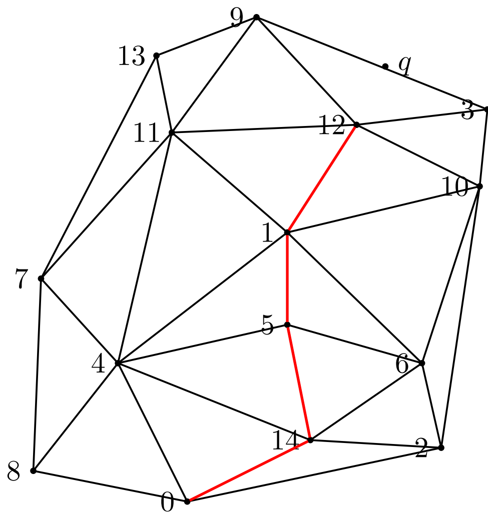

Abb. 15.1: Weg durch eine einfache Delaunay-Triangulierung.

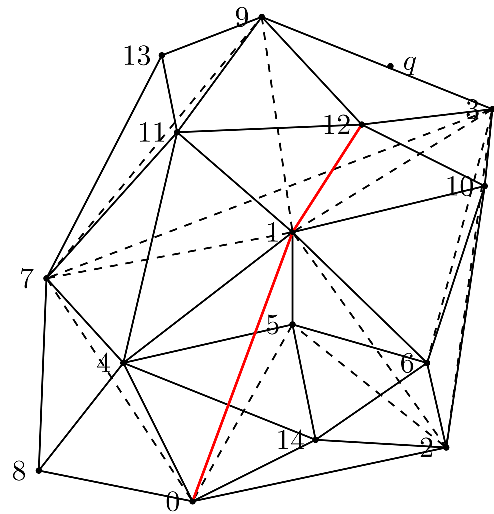

Abb. 15.2: Weg durch die erweiterte Delaunay-Triangulierung DT.

Abbildung 15: Weg zum nächsten Nachbarn von q, beginnend bei Knoten 0.

Umständen Kanten nimmt, die nur zu leichten Verbesserungen führen. Dadurch kann es zu großen Umwegen kommen, je nachdem, in welcher Reihenfolge die ausgehenden Kanten betrachtet werden. Ein Beispiel dazu wird in Abb. 16 gezeigt.

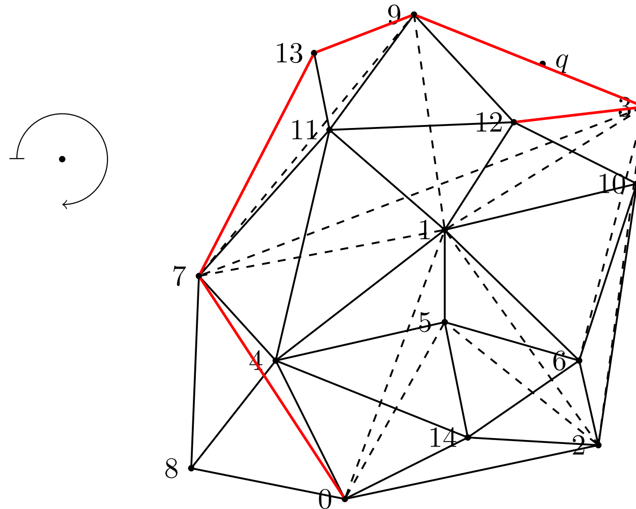

Abbildung 16: Beginn bei Knoten 0. Kanten werden von links nach rechts überprüft.

Führt man nun eine Hierarchie auf den Knoten ein, bei der jeder Knoten eine eindeutige ID zugeordnet bekommt, die der Einfügereihenfolge der Knoten entspricht, kann man das obige Problem entschärfen. Es werden nun nur noch Kanten genommen die zu einem Knoten mit einer höheren ID führen. Die Kanten müssen dabei aber in aufsteigen-

der Reihenfolge der IDs der Zielknoten angeordnet sein, da der Algorithmus ansonsten in einer Sackgasse landen kann (Abb. 17). D.h. der Algorithmus terminiert, hat aber nicht den nächsten Nachbarn erreicht. Hieraus ergibt sich auch direkt, dass der zuerst eingefügte Knoten der Anfangsknoten für die Suche sein muss. Würde man mit einem anderen Knoten beginnen, so wäre es unmöglich, Knoten mit einer niedrigeren ID zu erreichen. Der Algorithmus lässt sich dahingehend weiter optimieren, dass nur Kanten zu Knoten mit einer höheren ID gespeichert werden, die anderen Kanten werden nicht genommen.

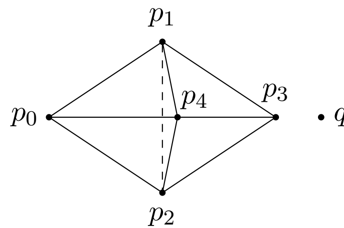

Abbildung 17: Gegenbeispiel Knotenhierarchie [12]. Wird beginnend bei p_0 zuerst die Kante zu p_4 genommen, landet man in einer Sackgasse, da p_3 der nächste Nachbar ist und die Kante von p_4 nach p_3 nicht genommen werden darf.

Dadurch wird das in Abb. 16 dargestellte Problem dahingehend entschärft, dass ein neues Kriterium hinzugekommen ist, welches Kanten ausschließt (In dem Beispiel existieren zwei Kanten zu Knoten mit niedrigeren IDs). In Abb. 18 ist dargestellt, wie die Nächster-Nachbar-Suche mittels der soeben vorgestellten Knotenhierarchie aus dem Beispiel von Abb. 16 abläuft.

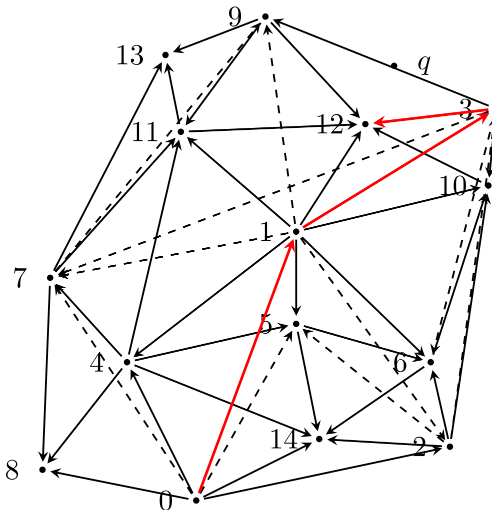

Abbildung 18: Beispiel: Nächster-Nachbar-Suche mittels Knotenhierarchie

Dies ergibt folgenden Algorithmus 2 zur Suche nach dem nächsten Nachbarn eines An-

fragepunkts q. Im Rest dieser Arbeit wird der Algorithmus auch mit NNK (Nächster Nachbar Knotenhierarchie) abgekürzt. Mit *Kantenliste(p)* werden die Kanten des Knotens p bezeichnet, die aufsteigend nach der ID des Zielknotens sortiert sind.

```
knotenhierarchie_nächster_nachbar(q, DT):
    naechster_nachbar = p0
    kandidat = erster Knoten aus Kantenliste(naechster_nachbar)

    while (ende von Kantenliste(naechster_nachbar) nicht erreicht) do
      if (|q kandidat| < |q naechster_nachbar|) do
        naechster_nachbar = kandidat // Kante traversieren
        kandidat = erster Knoten aus Kantenliste(naechster_nachbar)
      else
        kandidat = naechster Knoten aus Kantenliste(naechster_nachbar)
      done
    done

    return naechster_nachbar
```

Algorithmus 2: Nächster-Nachbar-Suche mittel Knotenhierarchie.

Die Korrektheit von NNK wird nun im folgenden bewiesen.

Beweis ([12]): Der Beweis erfolgt durch Induktion über die Anzahl der Knoten. Dabei wird gezeigt, dass der Algorithmus 2 in jedem Schritt i den nächsten Nachbarn v_i von q in DT_i findet.

Induktionsanfang ($i = 0$): In DT_0 existiert nur der Knoten p_0. Folglich ist dieser auch der nächste Nachbar von q in DT_0. Als erstes wird v_0 (*naechster_nachbar*) auf p_0 gesetzt. ✓

Induktionsvoraussetzung: v_i ist der nächste Nachbar von q in DT_i.

Induktionsschritt $i \to i+1$: p_{i+1} ist der neu hinzugekommene Knoten von DT_{i+1}. Liegt p_{i+1} weiter von q entfernt als v_i, so ist v_i weiterhin der nächste Nachbar von q (Induktionsvoraussetzung), somit gilt $v_{i+1} = v_i$ und es wird keine Kante traversiert.
Ist p_{i+1} näher an q als v_i, so existiert eine Kante von v_i nach p_{i+1}, denn der Kreis C mit q als Zentrum und v_i auf seinem Rand muss p_{i+1} in seinem Innern haben. Er enthält aber keinen weiteren Punkt aus DT_i nach Induktionsvoraussetzung, also auch keinen weiteren Knoten aus DT_{i+1}. Der Kreis C wird nun wie folgt verändert (geschrumpft): v_i liegt weiter auf seinem Rand aber das Zentrum wird auf der Achse $\overline{qv_i}$ in Richtung v_i solange verschoben bis ebenfalls p_{i+1} auf seinem Rand liegt (Abb. 19). Der neue Kreis C liegt vollständig innerhalb von dem alten Kreis C und ist somit leer. D.h. die Kante $\overline{v_i p_{i+1}}$ ist eine Kante von DT_{i+1} (Lemma 2). Da der Knoten p_{i+1} eine höhere ID als v_i hat (er wurde später eingefügt) nimmt der Algorithmus 2 die Kante und erreicht so den neuen nächsten Nachbarn von q, es gilt also $v_{i+1} = p_{i+1}$. ✓

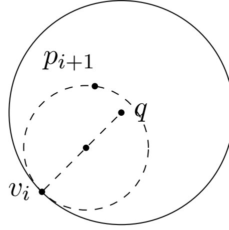

Abbildung 19: Schrumpfen eines Kreises.

Da $DT = \cup_{i=0}^{n-1} DT_i$, existiert in DT auch die Kante v_i nach p_{i+1} (falls p_{i+1} der nächste Nachbar von q in DT_{i+1} ist). Der Algorithmus erreicht also auch den nächsten Nachbarn von q in DT_{n-1} und dies ist der gesuchte Knoten. Wie bei den vorherigen Beweisen schon erklärt, bewegt sich der Algorithmus nicht rückwärts, d.h. es werden keine schon besuchten Knoten nochmals besucht. Da P endlich ist, terminiert NNK.

□

Besteht die Delaunay-Hierarchie aus insgesamt n Ebenen, wobei in Ebene i die Punkte $0, \ldots, n-i$ enhalten sind (die Nummerierung der Ebenen ist andersherum definiert), ergibt sich dar gleiche Weg wie in NNK. Der Algorithmus sucht jeweils den nächsten Nachbarn von q in den einzelnen Delaunay-Triangulierungen. Die Delaunay-Hierarchie durchsucht aber jede Ebene, wodurch sich im besten Fall eine Laufzeit von $\Omega(n)$ ergibt. In NNK entspricht jeder Besuch eines Knotens einer solchen Ebene. Da NNK Abkürzungen nehmen kann, werden einzelne Knoten (Ebenen) ausgelassen oder müssen erst gar nicht betrachtet werden, falls keine Kante zu einem näheren Knoten führt. Auch ist der Speicherverbrauch erheblich höher, da für jede Ebene eine Delaunay-Triangulierung gespeichert wird.

Früher in der Arbeit wurde bereits erwähnt, dass die Knotenhierarchie von Contraction Hierarchies [10] inspiriert wurde. Dort werden ebenfalls n Ebenen auf eine projiziert. In diesem Fall wird der neu entstandene Graph zur Suche nach dem kürzesten Pfad verwendet wird.

4.2 Laufzeit und Speicherverbrauch

Im Korrektheitsbeweis zu NNK wurde gezeigt, dass NNK durch die einzelnen Delaunay-Triangulierungen $DT_0, \ldots, DT_{n-1}$ läuft und dabei nacheinander den nächsten Nachbarn von q in DT_i findet. Diese Eigenschaft wird im folgenden Satz 1 verwendet, um die erwartete Laufzeit von NNK zu beweisen.

Satz 1: *Der Algorithmus 2 hat im schlimmsten Fall eine Laufzeit von* $\mathrm{O}(n)$*, wobei die erwartete Laufzeit* $\mathrm{O}(\log^2(n))$ *beträgt, falls beim Aufbau des Graphen die Knoten zufällig gleichverteilt, über einen Raum, eingefügt werden. Dabei ist* n *die Anzahl der Knoten von*

DT.

Beweis: Wird in NNK eine Kante nicht genommen, so entspricht dies einem Knoten, der „übersprungen" wird, d.h. der Knoten wird im späteren Verlauf sicher nicht mehr betrachtet. Daraus ergibt sich, dass maximal $n-1$ Kanten überprüft werden: Entweder wird ein Knoten übersprungen oder es wird die Kante zu einem näheren Knoten traversiert. Die Delaunay-Triangulierung hat aber insgesamt nur n Knoten, somit wird nach spätestens $n-1$ betrachteten Kanten (-1 wegen des Startknotens) der Knoten mit der höchsten ID überprüft. Die Laufzeit liegt also im schlechtesten Fall in $\mathrm{O}(n)$.

Im Korrektheitsbeweis zu NNK wurde bereits gezeigt, dass nur die direkte Kante vom nächsten Nachbarn v_i von q in DT_i zum nächsten Nachbarn v_{i+1} von q in DT_{i+1} ($i \in \{0, \ldots, n-1\}$) genommen wird. Diese Kante existiert aber nur, falls der neu eingefügte Knoten p_{i+1} der neue nächste Nachbar von q ist. Da die Knoten zufällig gleichverteilt zu DT hinzugefügt wurden, beträgt die Wahrscheinlichkeit $1/i+2$, dass p_{i+1} der neue nächste Nachbar von q in DT_{i+1} ist ($i+2$ ist die Anzahl der Knoten in DT_{i+1}). Daraus erhalten wir für die erwartete Anzahl an besuchten Knoten:

$$\sum_{i=0}^{n-1} \frac{1}{(i+1)} = \sum_{i=1}^{n} \frac{1}{i}.$$

Im Beweis zu Satz 2 wird gezeigt, dass die erwartete Anzahl an Kanten, die bei einer Einfügeoperation neu entstehen, maximal 6 beträgt. Für einen beliebigen Knoten p und bei der Einfügeoperation von Knoten p_i (i-te Einfügeoperation) gilt, dass mit einer Wahrscheinlichkeit von $6 \cdot 1/i-1$, die beiden Knoten p und p_i durch eine Kante verbunden werden. Es existieren $i-1$ Knoten auf die 6 Kanten verteilt werden. Daraus ergibt sich folgende erwartete Anzahl an Kanten für den Knoten p_i

$$\sum_{j=i+1}^{n} \frac{6}{j-1} = \sum_{j=i}^{n-1} \frac{6}{j}.$$

Somit erhalten wir als eine obere Abschätzung für die erwartete Anzahl an betrachteten Kanten

$$\sum_{i=1}^{n} \left(\frac{1}{i} \cdot \sum_{j=i}^{n-1} \frac{6}{j} \right).$$

Dies lässt sich nun noch weiter Abschätzen:

$$\begin{aligned}
\sum_{i=1}^{n} \left(\frac{1}{i} \cdot \sum_{j=i}^{n-1} \frac{6}{j} \right) &\leq 6 \cdot \sum_{i=1}^{n} \left(\frac{1}{i} \cdot \sum_{j=1}^{n} \frac{1}{j} \right) \\
&= 6 \cdot \sum_{i=1}^{n} \left(\frac{1}{i} \right) \cdot \sum_{j=1}^{n} \left(\frac{1}{j} \right) \\
&\overset{[1]}{\leq} 6 \cdot (\log(n) + 1) \cdot (\log(n) + 1)
\end{aligned}$$

Daraus folgt die obere Abschätzung für die erwartete Laufzeit: $\mathrm{O}(\log^2(n))$ □

Für die obere Abschätzung der erwarteten Laufzeit ist zu beachten, dass dabei vom schlechtesten Fall bei der Betrachtung der Kanten eines Knotens ausgegangen wird. D.h. es wird angenommen, dass alle Kanten eines Knoten überprüft werden. NNK nimmt aber die erste Kante, die näher ans Ziel führt. Die obere Abschätzung der erwarteten Laufzeit lässt sich also möglicherweise noch weiter verringern (siehe Kapitel 6.4).

Satz 2: *Die erwartete Anzahl an Kanten die während des gesamten Aufbaus von DT, erstellt werden liegt in* $\mathrm{O}(n)$. *D.h. auch die Anzahl der Kanten von DT liegt in* $\mathrm{O}(n)$, *da jede einmal erstellte Kante gespeichert wird.*

Beweis (nach [3], Seite 197)**:** Sei $P_r = \{p_1, \ldots, p_r\}$ die Knotenmenge der ersten r Knoten, die in die Triangulierung eingefügt werden. n sei die Anzahl von Knoten in P. Es gilt also $P_r \subset P$. Jede Kante, die im r-ten Schritt erstellt wird, ist eine Kante von p_r und ist ebenfalls eine Kante der Delaunay-Triangulierung von P_r [3]. D.h. der Grad von Knoten p_r entspricht der Anzahl der neu hinzugekommenen Kanten. Weiter gilt, dass die Anzahl an Kanten der Delaunay-Triangulierung von P_r $3r - 3 - k = n_K$ beträgt (Lemma 1), wobei k die Anzahl der Kanten der konvexen Hülle ist. Je kleiner k ist desto größer wird n_k, da eine konvexe Hülle aber aus mindestens 3 Knoten bestehen muss (P_r spannt einen 2D-Raum auf), ist die kleinste mögliche Zahl für k 3 (die konvexe Hülle hat 3 Kanten). Somit gilt für die Anzahl der Kanten

$$n_K \leq 3r - 6.$$

Außerdem verbindet jede Kante 2 Knoten der Delaunay-Triangulierung, somit gilt für den gesamten Grad deg des Graphen

$$deg \leq 2 * (3r - 6) = 6r - 12$$

und somit ist der erwartete Grad pro Knoten i in Schritt r

$$deg_i \leq \frac{deg}{r} = 6 - \frac{12}{r} \leq 6.$$

Es werden erwartet maximal 6 Kanten pro Schritt, zum Graphen, hinzugefügt. In n Schritten werden also erwartet maximal $6 \cdot n$ Kanten erstellt, d.h. die erwartete Anzahl an Kanten liegt in $\mathrm{O}(n)$. □

5 Implementierung

In der Einleitung wurde bereits erwähnt, dass zur Implementierung von NNK die Programmbibliothek CGAL verwendet wurde. Dabei sind die hier vorgestellten Klassen Erweiterungen bereits existierender Klassen von CGAL bzw. lassen sich mit solchen parametrisieren.

Für die experimentellen Ergebnisse aus Kapitel 6 wurde außerdem die Implementierung der Delaunay-Hierarchie in CGAL angepasst. In der bereits existierenden Implementierung wurde anstatt der neuen Nächster-Nachbar-Suche eine einfache Suche in der Delaunay-Triangulierung verwendet. Die Suche in der Delaunay-Hierarchie wurde auch leicht abgeändert. Anstatt in jeder Ebene nach dem nächsten Nachbarn zu suchen, wird in den Ebenen > 0 nur nach dem Dreieck gesucht, das den Anfragepunkt enthält. Von diesem Dreieck wird der nächste Eckknoten zum Anfragepunkt verwendet, um eine Ebene herabzusteigen. Diese leichte Abänderung weist Geschwindigkeitsvorteile gegenüber dem originalen Algorithmus auf. Vom nächsten Nachbarn aus sucht der originale Algorithmus ebenfalls nach dem Dreieck, das den Anfragepunkt enthält. Ist der gefundene nächste Nachbar kein Knoten des alten Dreiecks, das den Anfragepunkt enthält, so läuft der Algorithmus „rückwärts“, um das neue Dreieck zu finden. Dieses ist entweder komplett oder teilweise im alten Dreieck enthalten.

5.1 Delaunay_triangulation_vertex_hierarchy_2

Die Klasse `Delaunay_triangulation_vertex_hierarchy_2` ist für den Aufbau der Delaunay-Triangulierung und der darauf basierenden Datenstruktur für die nächste Nachbarsuche zuständig. `Delaunay_triangulation_vertex_hierarchy_2` ist eine Unterklasse von `Delaunay_triangulation_2`. Diese Klasse wird weiterhin zum Aufbau und zur Speicherung der Delaunay-Triangulierung verwendet. Für den Aufbau der Delaunay-Triangulierung wurde die Funktionen `insert(...)` neu implementiert. Die einzelnen `insert`-Funktionen sind:

- `insert(const Point &p, Face_handle start)`: Dieser Funktion wird der einzufügende Punkt `p` und ein Face_handle `start` übergeben. Das Face_handle wird als Startposition zur Suche der Einfügestelle von `p` verwendet. Das Suchen und Einfügen erfolgt wie bisher. Der neu entstandene Knoten bekommt noch zusätzlich eine eindeutige ID, die der Einfügereihenfolge entspricht. Jede Kante, die während dieser Einfügeoperation entsteht, ist eine Kante von `p`. Da CGAL diese aber nur implizit speichert, werden diese noch extra gespeichert. Für die neue Suche sind nur Kanten zu Knoten mit höheren IDs wichtig, daher werden die Kanten nicht in `p` gespeichert, sondern in den Knoten, von denen diese ausgehen. Da `p` gerade erst eingefügt wurde, ist er der Knoten mit der höchsten ID und folglich werden nur die Kanten betrachtet die zu ihm hinführen.

- `insert(const Point &p)`: Diese Funktion sucht zunächst, mittels der neu implementierten Nächsten-Nachbar-Suche, den nächsten Nachbar von `p`. Der nächste Nachbar wird verwendet, um einen Vertex_handle zu einem zugehörigen Dreieck zu bekommen. Der Punkt `p` und der Vertex_handle werden anschließend `insert(const Point &p, Face_handle start)` übergeben, um p zur Delaunay-Triangulierung hinzuzufügen. Der nächste Nachbar von `p` ist relativ in der Nähe von der eigentlichen Einfügeposition von p. Daher ist ein benachbartes Dreieck vom nächsten Nachbarn eine gute Ausgangsposition zur eigentlichen Suche nach der Einfügeposition in `insert(const Point &p, Face_handle start)`.

- `insert(InputIterator first, InputIterator last)`: Dieser Funktion werden der Anfang und das Ende eines Iterators, der einzufügenden Punktemenge, übergeben. Die Punktemenge wird zunächst teilweise nach einer Hilbertkurve sortiert, um anschließend die Punkte, nach und nach, mittels `insert(const Point &p, Face_handle start)` einzufügen. Für `start` wird immer das Vertex_handle zu einem benachbarten Dreieck vom zuvor eingefügten Punkt verwendet.

Jede `insert()`-Funktion liefert einen Vertex_handle auf den neu eingefügten Knoten zurück. Für den Fall, dass ein Punkt eingefügt wird, der schon einmal zuvor eingefügt wurde, wird der Vertex_handle auf den zuvor eingefügten Knoten zurückgegeben und die Delaunay-Triangulierung bleibt unverändert.

Die Funktion `nearest_vertex(const Point &query_point)` wurde ebenfalls neu implementiert. Zur Suche des nächsten Nachbarn wird jetzt der Algorithmus NNK verwendet.

Die Funktion `nearest_vertex` verwendet zur Suche die ausgehenden Kanten der Knoten, diese entsprechen dem in Kapitel 2 vorgestellten Graphen. Die Kanten selbst sind aber nur Vertex_handle für die entsprechenden Zielknoten und die Knoten selbst sind in keinem zusammenhängenden Speicherbereich gespeichert. Dies hat zur Folge, dass die Nächster-Nachbar-Suche nicht sehr cacheeffizient arbeitet. Um die Cacheeffizienz zu erhöhen, wurde die Klasse `Edge_Array` in `Delaunay_triangulation_vertex_hierarchy_2` implementiert. Diese besteht aus einem großen Array, in dem die einzelnen Kantenlisten gespeichert sind. Jeder Eintrag speichert die Koordinaten des Zielknotens, einen Verweis auf den zughörigen Knoten und die Position (Index), ab der die Kantenliste des Zielknotens im Sucharray anfängt. Die Kantenlisten werden in aufsteigender Reihenfolge der zughörigen Knoten-IDs gespeichert.

Koordinate	Knoten	Index	...

Das höchstwertige Bit des Indexes zeigt an, ob der aktuelle Eintrag das letzte Element der gerade betrachteten Kantenliste ist. Falls der Zielindex = 0 ist, zeigt dies an, dass der Zielknoten eine leere Kantenliste hat. Da der Index = 0 für die Startposition des ersten eingefügten Knotens steht, wir aber immer nur Kanten zu Knoten mit einer höheren ID verwenden, ist Index = 0 nie eine zulässige Position. Daher kann Index = 0 verwendet werden, um eine leere Kantenliste zu signalisieren.

Die Funktion `nearest_vertex()(Point &p)` von `Edge_Array` implementiert die Nächster-Nachbar-Suche.

Für die Suche nach dem nächsten Nachbarn bieten beide Möglichkeiten auch jeweils eine inexakte Suche an, d.h. bei der Rechnung mit `doubles` kann es zu Rundungsfehlern kommen und somit auch zu falschen Ergebnissen (in Kapitel 5.3 wird auf weiter auf die exakte Suche eingegangen). In Abb. 20 ist ein Laufzeitvergleich der vier gerade vorgestellten Suchmöglichkeiten gegenüber der Nächster-Nachbar-Suche mittels CGALs Delaunay-Hierarchie dargestellt.

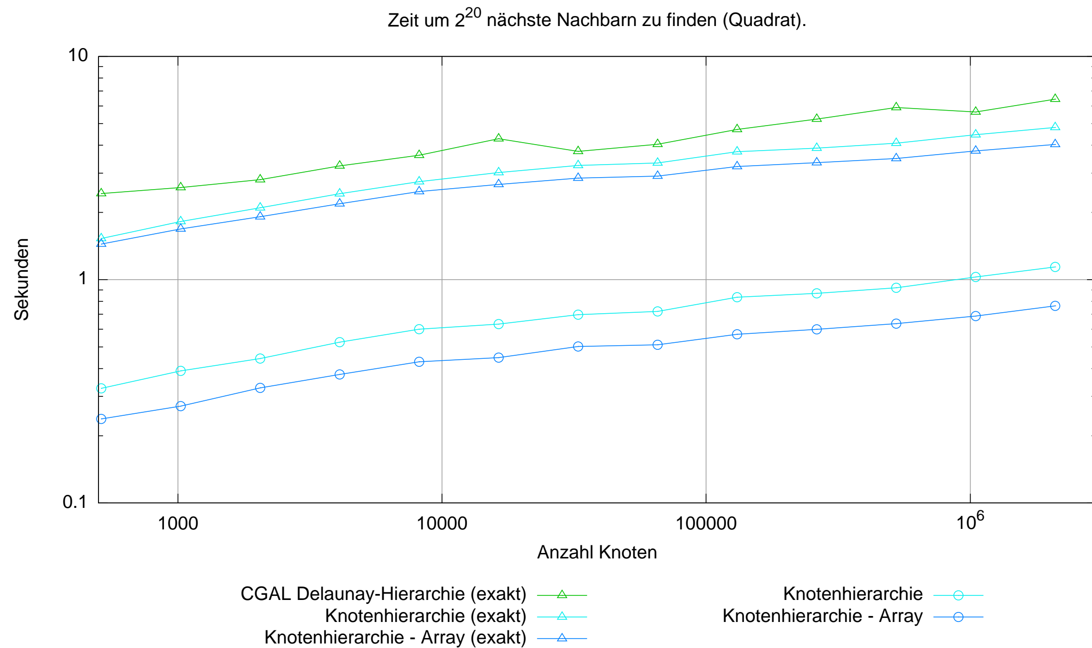

Abbildung 20: Laufzeitvergleich: NNK gegen Delaunay-Hierarchie

`Delaunay_triangulation_vertex_hierarchy_2` ist nicht dazu geeignet, die Delaunay-Triangulierung, außer durch die Einfügeoperationen, zu ändern, d.h. Knoten können aktuell nicht nachträglich gelöscht werden, ohne die Korrektheit der Nächsten-Nachbar-Suche zu zerstören. In Abb. 21 ist ein Beispiel dargestellt, bei dem der Algorithmus nicht den nächsten Nachbarn findet. Eine Möglichkeit die Korrektheit wieder herzustellen, wäre die Delaunay-Triangulierung neu aufzubauen. Dies hätte aber eine sehr hohe Laufzeit. Möglicherweise könnte es ausreichen alle eingehenden Kanten eines gelöschten Knotens auf die ausgehenden Kantne umzuleiten. Dies würde aber zu einem starken Anstieg der Kantenanzahl führen (eingehende Kanten · ausgehende Kanten) und es ist auch nicht ohne weiteres möglich auf die eingehenden Kanten zuzugreifen. Eingehende Kanten werden nicht gespeichert, es wäre also notwendig diese erst aufwändig zu suchen.

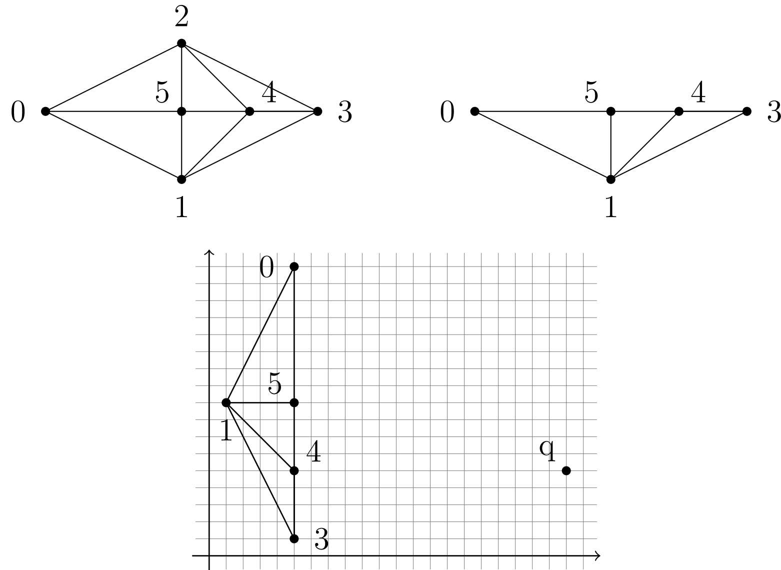

Abbildung 21: Zuerst wird der Knoten 2 gelöscht und anschließend ist eine Beispiel Anfrage dargestellt. Dabei würde der Algorithmus zuerst zu Knoten 5 gehen, da dieser näher an q liegt. Von 5 aus geht der Algorithmus aber nicht weiter zu Knoten 4, dem eigentlichen nächsten Nachbarn.

5.2 Triangulation_vertex_hierarchy_base_2

`Triangulation_vertex_hierarchy_base_2` dient als Basisklasse für Knoten von `Delaunay_triangulation_vertex_hierarchy_2`. `Triangulation_vertex_hierarchy_base_2` erweitert die standard-CGAL-Knoten um eine ID und eine Liste/Array von ausgehenden Kanten. Zum Einfügen von neuen Kanten stehen zwei Funktionen zur Verfügung:

- `insert_to_edge_list(Vertex_handle &v)`: Diese Funktion fügt den Vertex_handle `v` sortiert in die Kantenliste ein, d.h. die Vertex_handle werden in aufsteigender Reihenfolge, entsprechend ihren IDs, in die Kantenliste eingefügt. Dies hat den Aufwand $O(n)$, wobei n die Größe der Kantenliste ist.

- `append_to_edge_list(Vertex_handle &v)`: Diese Funktion fügt `v` hinten zur Kantenliste hinzu. Dies hat den amortisierten Aufwand $O(1)$.

Weiter bietet `Triangulation_vertex_hierarchy_base_2` noch den Iterator `edge_iterator` an, um einfach durch die Kantenliste zu wandern. Die Funktion `begin_edge_iterator()` wird verwendet, um auf das erste Element zuzugreifen und `end_edge_iterator()` zeigt an ob, man das Ende erreicht hat.

Zum Speichern der Kanten wird ein `vector` verwendet. Dieser hat im Vergleich zu einer einfach verketteten Listen einen geringen Speicherverbrauch. Jedes Element der Liste

speichert einen Verweis auf den Nachfolger und den Vertex_handle. Ein Vertex_handle ist intern ein Pointer, somit müssen insgesamt $2n$ Pointer gespeichert werden (n ist die Größe der Liste). Bei einem `vector` muss der Nachfolger nicht extra gespeichert werden, jedoch kann es sein, dass der `vector` doppelt so viel Speicherplatz reserviert wie er für seine Elemente bräuchte. Somit ergibt sich, dass der Speicherverbrauch $\leq 2n$ ist. Weiter bietet der `vector` auch noch leichte Geschwindigkeitsvorteile (Abb. 22).

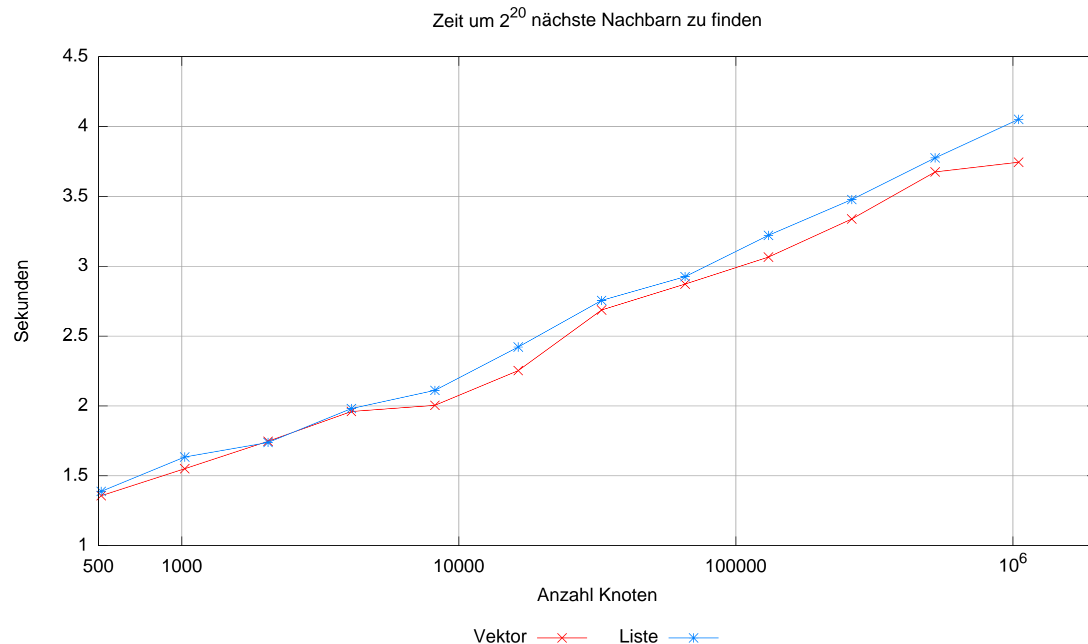

Abbildung 22: Laufzeitvergleich: Vektor gegen Liste.

5.3 Exact_distance_2

Die Klasse `Exact_distance_2` dient zum exakten Vergleich von Distanzen zwischen Punkten und stellt ein neues Prädikat von CGAL dar. Ein Prädikat in CGAL liefert dabei Eigenschaften von einem oder mehreren übergebenen Objekten zurück. Die Eigenschaften können Wahrheitswerte sein (z.B. bilden zwei Knoten eine Kante?), aber auch Aufzählungen, die mehr Eigenschaften unterscheiden können. `Delaunay_triangulation_vertex_hierarchy_2` verwendet in der Nächsten-Nachbar-Suche `Exact_distance_2` und nicht die von CGAL zur Verfügung gestellte Funktion Compare_distance_2. Der Grund darin liegt, dass `Exact_distance_2` deutlich schneller ist. `Exact_distance_2` stellt nicht direkt eine Distanz dar, sondern speichert intern zwei Distanzen, die miteinander verglichen werden. Die beiden Distanzen müssen einen Punkt gemeinsam haben, d.h. es werden immer Distanzen der Form $|ab|$ und $|ac|$ miteinander verglichen, wobei a, b und c drei Punkte sind. Dies hat die folgenden beiden Gründe:

1. Unsere Distanzvergleiche in `Delaunay_triangulation_vertex_hierarchy_2` sehen immer so aus, dass wir einen Basispunkt a haben von dem aus wir zwei Distanzen zu den Punkten b und c bestimmen, und diese dann miteinander vergleichen.

2. Diese Schreibweise orientiert sich auch mehr an der CGAL-Funktion Compare_distance_2, die ebenfalls drei Punkte erwartet um anschließend die Distanzen zu vergleichen.

In der Klasse werden die quadrierten Distanzen gespeichert (so muss nicht extra noch die Wurzel berechnet werden) und noch die 3 Punkte. Die Punkte müssen extra gespeichert werden für den Fall, dass festgestellt wird, dass der Vergleich der quadrierten Distanzen zu ungenau ist. In diesem Fall wird ein Verfahren von CGAL zum exakten Vergleich der Distanzen verwendet.

Um feststellen zu können, dass ein Vergleich nicht exakt genug ist, muss für die Bestimmung der quadrierten Distanzen immer Protect_FPU_rounding „aktiviert“ sein. Dies ist ein spezielles CGAL-Objekt. Anstatt dies immer für jede Berechnung der quadrierten Distanzen zu „aktivieren“, ist es sinnvoller bei der Erzeugung des `Exact_distance_2`-Objekts Protect_FPU_rounding zu aktivieren und erst wieder bei der Freigabe des Objekts Protect_FPU_rounding zurückzusetzen. In den Abbildungen 23 und 24 ist dieser Geschwindigkeitsvorteil dargestellt.

Abbildung 23: Vergleich der Konstruktionszeiten zwischen global und lokal gesetztem Protect_FPU_rounding.

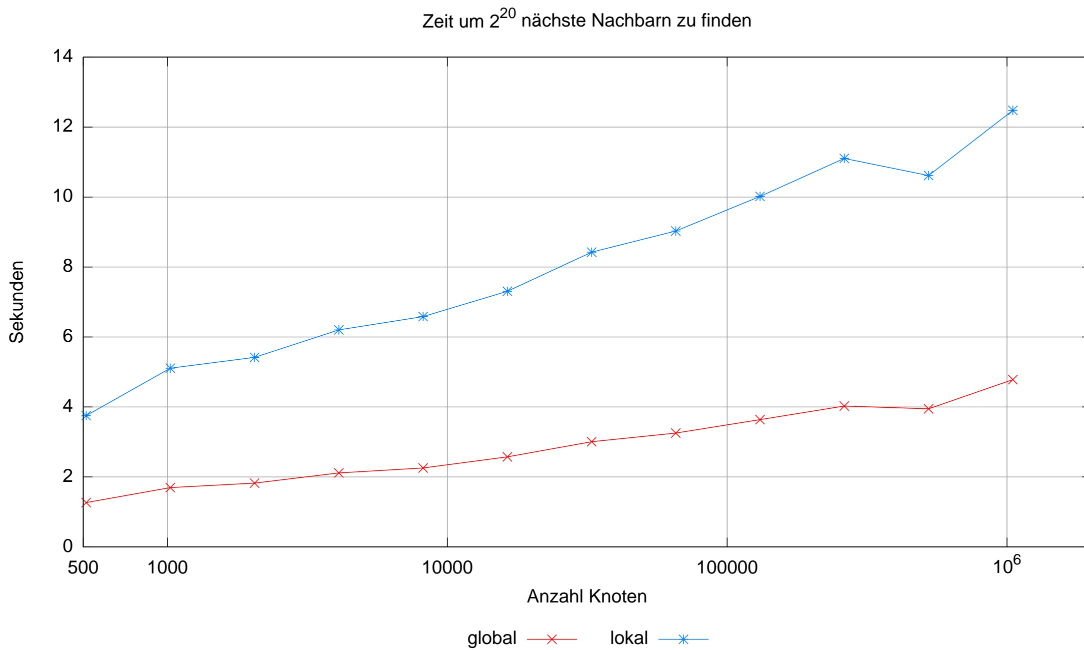

Abbildung 24: Vergleich der Laufzeiten von NNK zwischen global und lokal gesetztem Protect_FPU_rounding.

Die Geschwindigkeitsvorteile kommen daher, dass der Prozessor so nicht jedes mal in einen anderen Modus geschaltet werden muss.

Mit Hilfe der folgenden Funktionen kann man die einzelnen Punkte und somit auch die Distanzen setzen:

- `operator()(const I_Point &p1, const I_Point &p2, const I_Point &p3)`: Die drei Punkte `p1`, `p2` und `p3` sowie die beiden zugehörigen Distanzen werden gesetzt.

- `operator()(const I_Point &p1, const I_Point &p2)`: In diesem Fall werden nur die Punkte `p1` und `p2` und die zugehörige Distanz gesetzt. Diese Funktion wird von der Nächsten-Nachbar-Suche verwendet, da so der dritte Punkt nicht so oft gesetzt werden muss.

- `set_third_point(const I_Point &p3)`: Diese Funktion setzt den dritten Punkt `p3` und die zugehörige zweite Distanz.

- `third_to_second_point()`: Diese Funktion speichert den dritten Punkt an der Stelle vom zweiten und ebenso die zweite Distanz an der Stelle von der Ersten. Dies intern zu machen hat den Vorteil, dass die Distanz nicht nochmals extra berechnet werden muss. Aus diesem Grund verwendet die Nächster-Nachbar-Suche diese Funktion.

Die Funktion `compare()` dient, wie der Name schon sagt, zum eigentlichen Vergleich der beiden Distanzen. Dabei sieht das Ergebnis so aus, dass `LARGER` zurückgegeben wird, falls die Distanz zwischen dem ersten und zweiten Punkt größer ist, als die zwischen dem ersten und dritten Punkt. Entsprechend wird `EQUAL` und `SMALLER` zurückgegeben.

6 Experimente

Die im vorherigen Kapitel beschriebene Implementierung von NNK wird in diesem Kapitel unter den Gesichtspunkten der Laufzeit, des Speicherverbrauchs und der Exaktheit betrachtet. Mit Exaktheit ist dabei gemeint, wie gut die Ergebnisse der einfachen Rechnung mit `doubles` im Vergleich zur exakten (Ausschließen von Rundungsfehlern) sind.

Die folgenden Tests wurden alle auf einem Intel Core 2 Quad mit 2.4 GHz und 2GB Arbeitsspeicher ausgeführt. Als Betriebssystem wurde openSuse-Linux eingesetzt. Die Programme wurden alle mit g++ Version 4.3 und der Optimierungsstufe O3 kompiliert.

6.1 Laufzeitvergleiche

Die Laufzeit von Algorithmus 2 wird mit zwei Implementierungen von CGAL sowie der Implementierung von ANN (approximate nearest neighbor) [9] verglichen. Die erste Implementierung von CGAL verwendet die in Kapitel 3.2 vorgestellte Datenstruktur Delaunay-Hierarchie, die zweite Implementierung von CGAL verwendet einen k-d-Baum. Die Nächster-Nachbar-Suche von ANN verwendet ebenfalls einen k-d-Baum. Zur Laufzeitmessung werden vier verschiedene Arten von Stellen zur Suche verwendet. Die Anordnungen der Stellen orientieren sich dabei an denen aus [5]. Die Stellen sehen wie folgt aus:

- *Quadrat*: Die Punkte sind gleichverteilt in einem Quadrat.
- *Kreis*: Die Punkte sind gleichverteilt auf dem Rand eines Kreises angeordnet.
- *Parabel*: Die Punkte sind gleichverteilt auf einer Parabel angeordnet ($f(x) = x^2$).
- *Gemischt*: Die Punkte befinden sich zu 95% gleichverteilt auf dem Rand eines Kreises und zu 5% gleichverteilt innerhalb eines Quadrats, das den Kreis einschließt.

Von der Implementierung von Algorithmus 2 gibt es vier verschiedene Benchmarks. Zwei von diesen verwenden direkt `Delaunay_triangulation_vertex_hierarchy_2` zur Bestimmung des nächsten Nachbarn, wobei einmal exakt gerechnet wird, d.h. unter Verwendung von `Exact_distance_2` und beim anderen Rundungsfehler auftreten können. Die anderen beiden Benchmarks verwenden die kompaktere Speicherung in einem Array, wobei eine exakte und eine inexakte Version verwendet wird. Zur Laufzeitmessung von Delaunay-Hierarchie wurden ebenfalls exakte Distanzvergleiche verwendet.

Bei den folgenden Benchmarks wurde das Laufzeitverhalten gemessen, das sich durch unterschiedlich große Punktemengen, in denen zu suchen ist, ergibt.

Im ersten Benchmark (Abb. 25) sind die Stellen innerhalb eines Quadrats angeordnet und bestehen aus $2^8, \ldots, 2^{21}$ Punkten. Die zur Suche nach dem nächsten Nachbarn verwendete Anfragemenge Q besteht aus 2^{20} Elementen. Die Punkte aus Q sind dabei

ebenfalls gleichverteilt in einem Quadrat angeordnet, wobei das Quadrat von Q in jeder Richtung um den Faktor 1,05 größer ist als die Quadrate der Stellen.

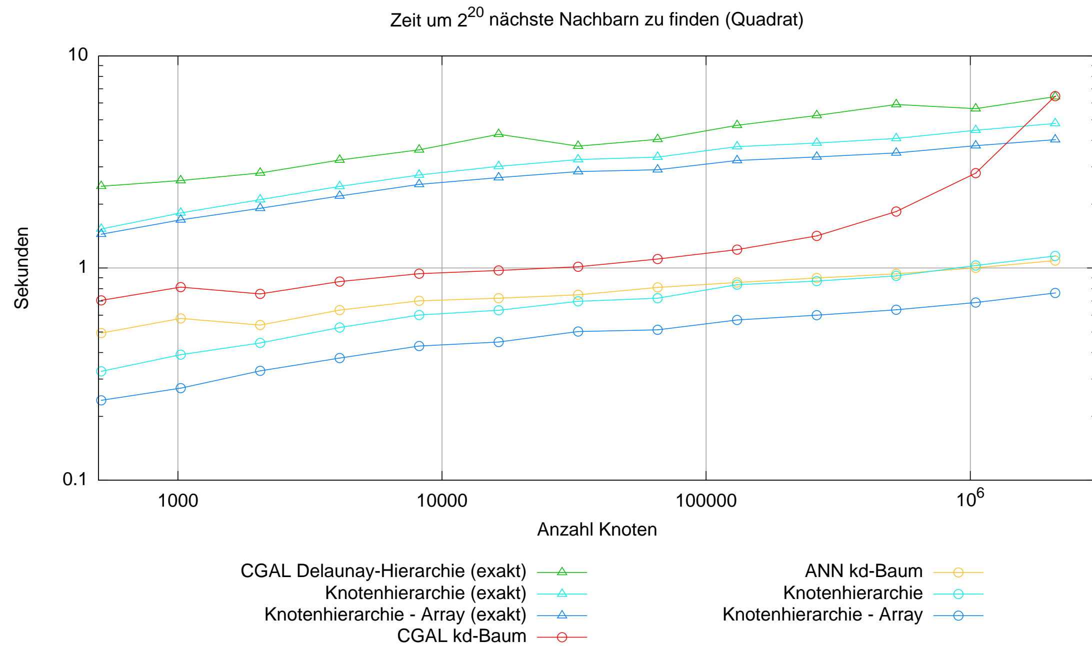

Abbildung 25: Laufzeitvergleich bei Anordnung innerhalb eines Quadrats.

Beim nächsten Benchmark (Abb. 26) sind die Punkte der Stellen auf dem Rand eines Kreises angeordnet und die Anfragepunkte sind zufällig gleichverteilt innerhalb eines Quadrats, das die Stellen einschließt. Die Anfragemenge hat dabei eine Größe von 2^{16} Elementen.

Für den dritten Benchmark (Abb. 27) sind die Stellen auf einer Parabel, auf dem Intervall -10^6 bis 10^6, angeordnet, wobei die Punkte aus Q gleichverteilt innerhalb eines Rechtecks sind, das die Parabel umschließt.

Im letzten Benchmark (Abb. 28) haben wir nun den Fall, dass die Punkte aus P zu 95% gleichverteilt auf einem Kreis angeordnet sind und zu 5% gleichverteilt innerhalb eines Quadrats, das den Kreis umschließt. Für Q gilt in diesem Fall, dass es eine Größe von 2^{19} Elementen hat und die Punkt wieder innerhalb eines Quadrats angeordnet sind, das die Mengen P umschließt.

Für alle Benchmarks gilt, dass die exakten Implementierungen von Algorithmus 2 gegenüber der Delaunay-Hierarchie (ebenfalls exakt) Laufzeitvorteile haben, bis auf einen „Ausreißer“ bei der gemischten Anordnung (Abb. 28). Sind die Punkte nur auf einer „Linie“ (d.h. hier Kreis oder Parabel) angeordnet, sind die Laufzeitvorteile sehr viel deutlicher. Die Implementierungen der Nächster-Nachbar-Suche mit Hilfe von k-d-Bäumen sind für den ersten Fall (Abb. 25) zwar schneller als die exakten Implementierungen,

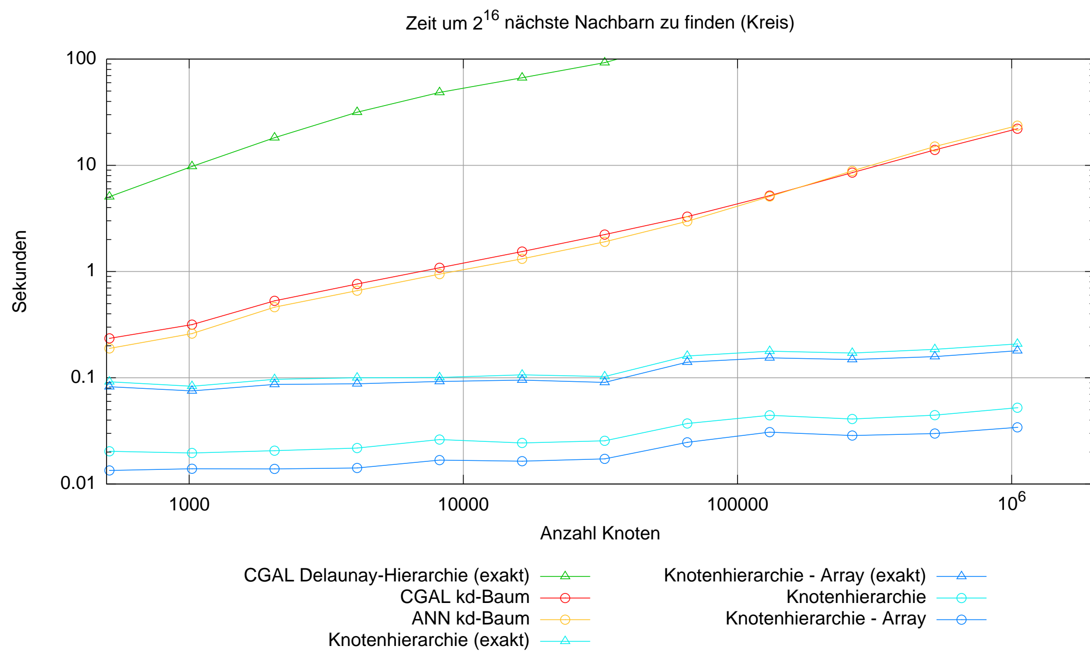

Abbildung 26: Laufzeitvergleich bei Anordnung auf einem Kreis.

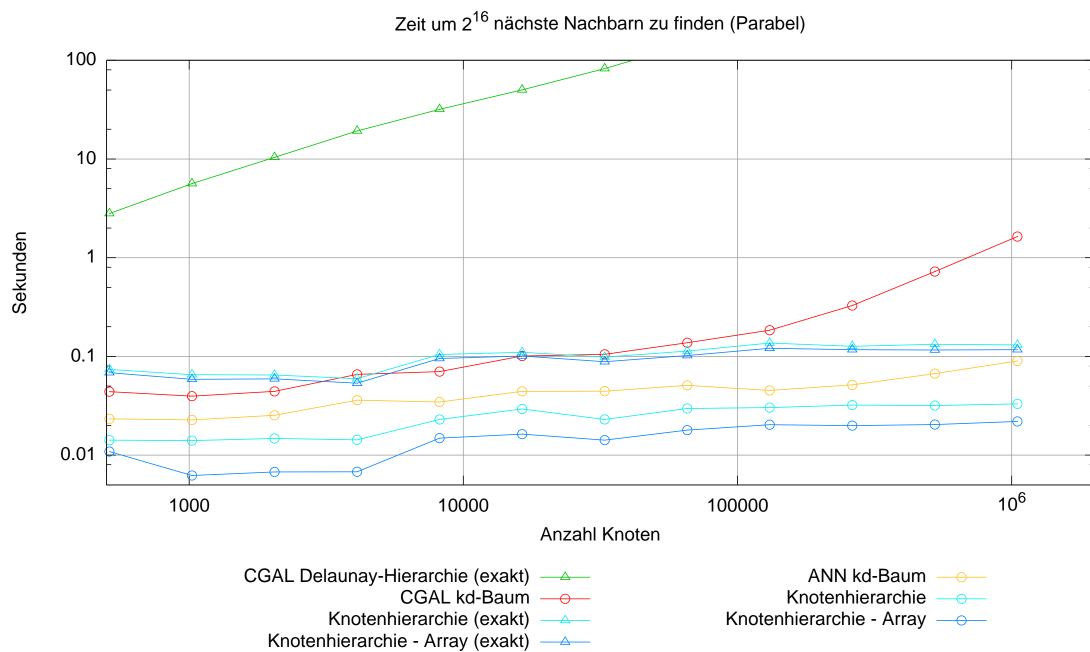

Abbildung 27: Laufzeitvergleich bei Anordnung auf einer Parabel.

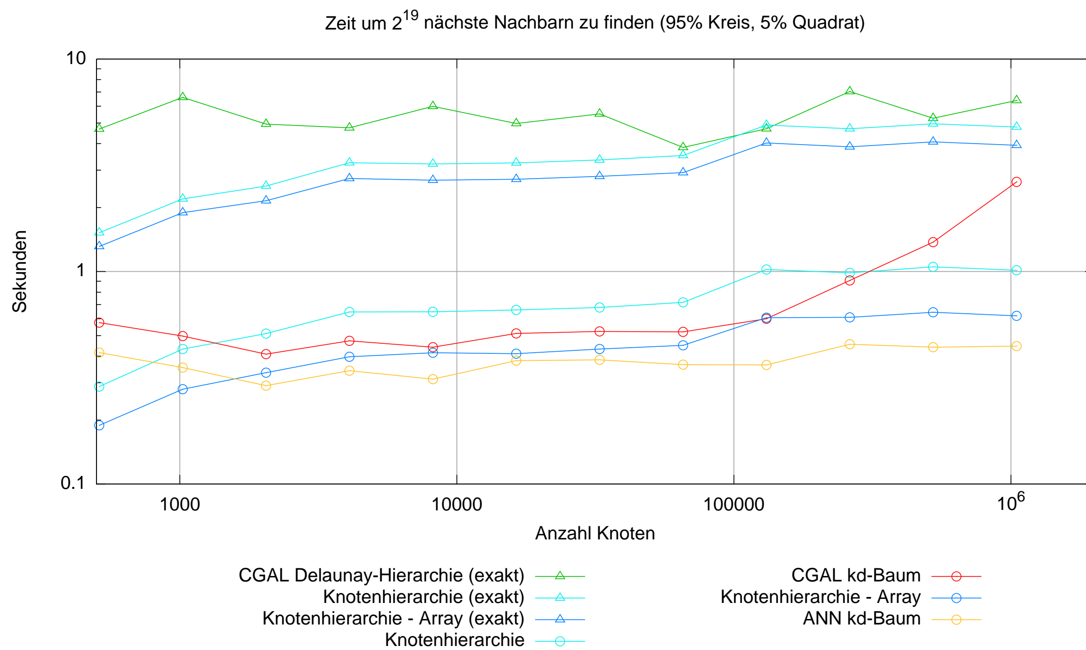

Abbildung 28: Laufzeitvergleich bei gemischter Anordnung auf einem Kreis und innerhalb eines Quadrats

doch die ebenfalls inexakten Versionen von Algorithmus 2 sind meistens schneller oder nur minimal langsamer, für einen Graph der Größe 2^{20} und 2^{21}. Zu beachten ist auch, dass die k-d-Baum-Implementierung von CGAL in diesem Fall, für Graphen mit mehr als 2^{18} Knoten, eine sehr starke Laufzeitzunahme aufweist im Gegensatz zu der Implementierung von ANN. Dieses Verhalten ist auch bei den Benchmarks zur Anordnung entlang einer Parabel und der gemischten Anordnung (Kreis und Quadrat) zu beobachten. Für den letzten Fall, bei dem die Punktemenge gemischt ist, liefert die Implementierung von ANN die besten Laufzeiten.

6.2 Benötigter Speicher

Die Implementierung von Algorithmus 2 in CGAL speichert zusätzlich zur Delaunay-Triangulierung der jeweiligen Punktemenge noch jede Kante (die Kanten der Triangulierung plus die zuvor gelöschten). Hieraus ergibt sich ein höherer Speicherverbrauch gegenüber der einfachen Delaunay-Triangulierung von CGAL.

Den größten zusätzlichen Speicherverbrauch hat die Knotenhierarchie, wenn die Punkte innerhalb eines Quadrats angeordnet sind. Die gilt ebenfalls für den größten zusätzlichen Speicherverbrauch (Tabelle 1).

Der geringste zusätzliche Speicherverbrauch ergibt sich für eine gemischte Verteilung der

Anzahl Knoten	32768	65536	131072	262144	524288	1048576
Delaunay (B/K)	178,56	157,6	154,88	153,92	153,06	152,62
Knoten-Hierarchie (B/K)	290,88	286,24	283,12	282,2	281,18	280,84
Differenz (B/K)	112,32	128,8	128,32	128,28	128,12	128,22
Zuwachs (%)	62,96	81,71	82,83	83,35	83,7	84,01

Tabelle 1: Speicherverbrauch Quadrat (Angaben in Byte/Knoten)

Punkte (Tabelle 2).

Anzahl Knoten	32768	65536	131072	262144	524288	1048576
Delaunay (B/K)	180,48	174,88	169,36	165,64	160,74	152,83
Knoten-Hierarchie (B/K)	276,16	269,76	263,84	257,72	249,36	234,04
Differenz (B/K)	95,68	94,88	94,48	92,08	88,62	81,22
Zuwachs (%)	53,05	54,23	55,77	55,59	55,13	53,14

Tabelle 2: Speicherverbrauch Kreis (95%), Quadrat (5%) gemischt (Angaben in Byte/Knoten)

Der zusätzliche Speicherverbrauch für die beiden anderen Verteilungen (Kreis, Parabel) liegt zwischen den beiden eben genannten (näher an dem Verbrauch von einem Quadrat).

Die Implementierung von Algorithmus 2 benötigt die Delaunay-Triangulierung nur zum Einfügen von weiteren Knoten, für die eigentliche Suche ist diese aber nicht notwendig. Ist eine Punktemenge fest, d.h. der Graph wird einmal erstellt und später nicht mehr verändert, ist es möglich den Speicher, der für die Delaunay-Triangulierung benötigt wird, freizugeben. Hierdurch lässt sich der Speicherverbrauch deutlich senken.

6.3 Wie gut sind die inexakten Varianten?

Anhand obiger Benchmarks lässt sich sehr gut erkennen, dass die inexakten Versionen von Algorithmus 2 deutlich schneller sind als die exakten. Inexakt bedeutet in diesem Fall, dass es bei der Berechnung der quadrierten Distanzen zu Rundungsfehlern kommen kann, weil als Datentyp `doubles` verwendet wurden. Durch diese Rundungsfehler kann es passieren, dass entweder ein falscher Weg genommen wird (der Zielknoten ist scheinbar näher) oder nicht erkannt wird, dass ein Zielknoten näher am Anfragepunkt liegt.

Zur Überprüfung wie genau die Ergebnisse sind, wurden Testläufe durchgeführt auf den bereits vorgestellten Punktanordnungen. Dabei kam noch eine weitere Anordnung hinzu, bei der die Punkte auf dem Rand einer Ellipse angeordnet sind. Die Anfragemenge bestand jeweils aus 2^{20} Elementen und die Punktemengen in denen gesucht wurde bestanden aus $2^9, \ldots, 2^{20}$ Elementen. Die Punkte der Anfragemengen waren dabei innerhalb eines Quadrats, das die Punktemengen umschließt, gleichmäßig verteilt. Falsche Ergebnisse wurden dabei nur 2 mal festgestellt. In beiden Fällen war dies der Fall für

die Anordnung auf dem Rand einer Ellipse. Wobei je ein Fehler für einen Graph mit 2^{18} Knoten und einen Graph mit 2^{19} festgestellt wurde.
Die Überprüfung wurde auch für Anordnungen der Anfragemenge die einem Kreis, Ellipse oder Parabel entsprechen durchgeführt. Der Kreis, die Ellipse und die Parabel haben dabei denen aus den Punktemengen entsprochen. Hierbei konnten ebenfalls keine weiteren Fehler festgestellt werden.

Fehler treten für die oben angegebenen Anordnungen nur sehr selten auf und insbesondere ließ sich der prozentuale Fehler nicht messen. Für diesen wurde 0 gemessen. Dies dürfte damit zusammenhängen, dass sich die verglichenen Distanzen nur minimal unterschieden haben und es deshalb auch zu den Rundungsfehlern kam. Werden exakte Ergebnisse nicht zwingend benötigt oder ist eine geringe Fehlerrate zulässig, so empfiehlt es sich, die inexakte Variante zu nehmen, da diese eine deutlich kürzere Laufzeit aufweist.

6.4 Eigenschaften vom NNK-Graphen

In Kapitel 4.2 wurde bewiesen, dass die erwartete Anzahl an Kanten in einem NNK-Graphen nur linear mit der Anzahl der Knoten wächst. Dies lässt sich auch experimentell bestätigen, wie in Abb. 29 zu sehen ist.

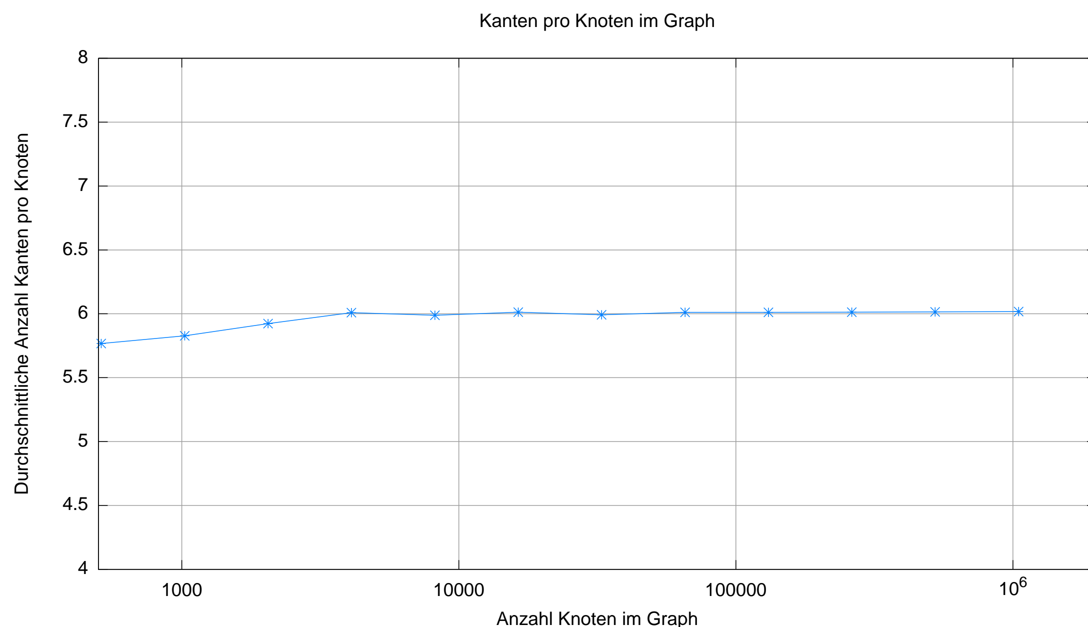

Abbildung 29: Anzahl Kanten pro Knoten in einem NNK-Graphen.

Ebenfalls in Kapitel 4.2 wurde gezeigt, dass die erwartete Anzahl an Kanten pro Knoten

wie folgt aussieht:

$$\sum_{j=i+1}^{n} \frac{6}{j-1} = \sum_{j=i}^{n-1} \frac{6}{j}.$$

Dies lässt sich nun in folgenden schöneren Ausdruck umformen:

$$\begin{aligned}\sum_{j=i+1}^{n} \frac{6}{j-1} &= \sum_{j=i}^{n-1} \frac{6}{j} \\ &= \sum_{j=1}^{n-1} \frac{6}{j} - \sum_{j=1}^{i-1} \frac{6}{j} \\ &\overset{[1]}{\approx} 6 \cdot (\log(n) - \log(i)) = 6 \cdot \log(n/i)\end{aligned}$$

Für einen Graphen mit einer festen Anzahl n an Knoten, ergibt sich somit für die erwartete Anzahl an Kanten pro Knoten i, eine negative logarithmische Kurve. Dieses erwartete Verhalten lässt sich in Abb. 30 gut sehen.

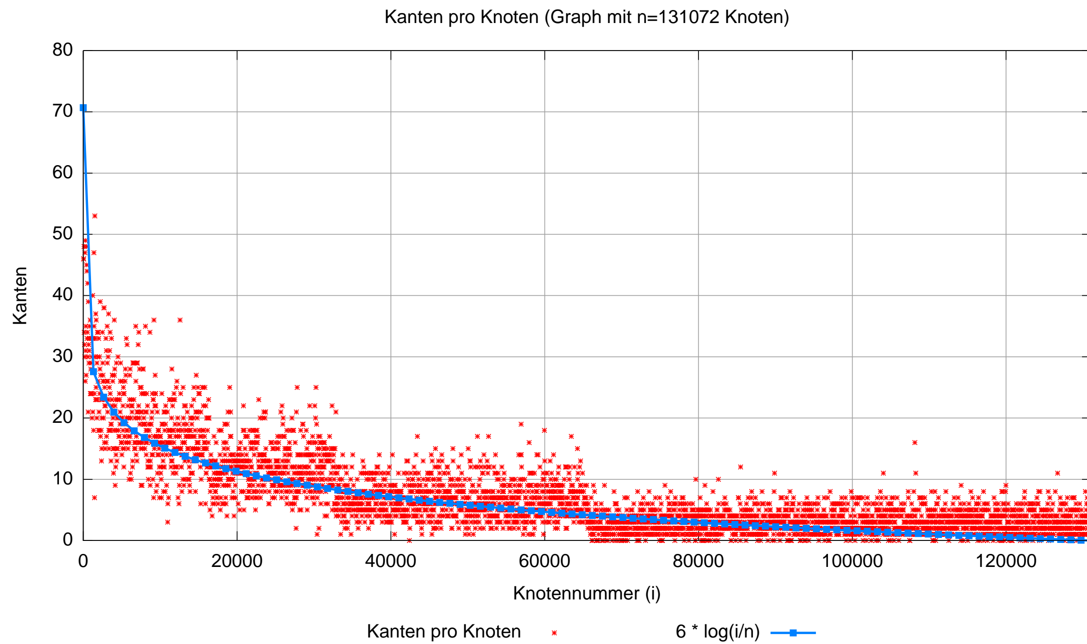

Abbildung 30: Kanten pro Knoten in NNK-Graphen, der Größe 2^{13}.

Für die Laufzeit von NNK ist maßgeblich die Anzahl der betrachteten Kanten pro Anfrage verantwortlich. Für diese wurde in Kapitel 4.2 eine erwartete obere Grenze von $O(\log^2(n))$ angegeben. Dies lässt sich auch experimentell (Abb. 31) bestätigen. Die experimentellen Ergebnisse lassen auch auf eine noch bessere erwartete Laufzeit schließen.

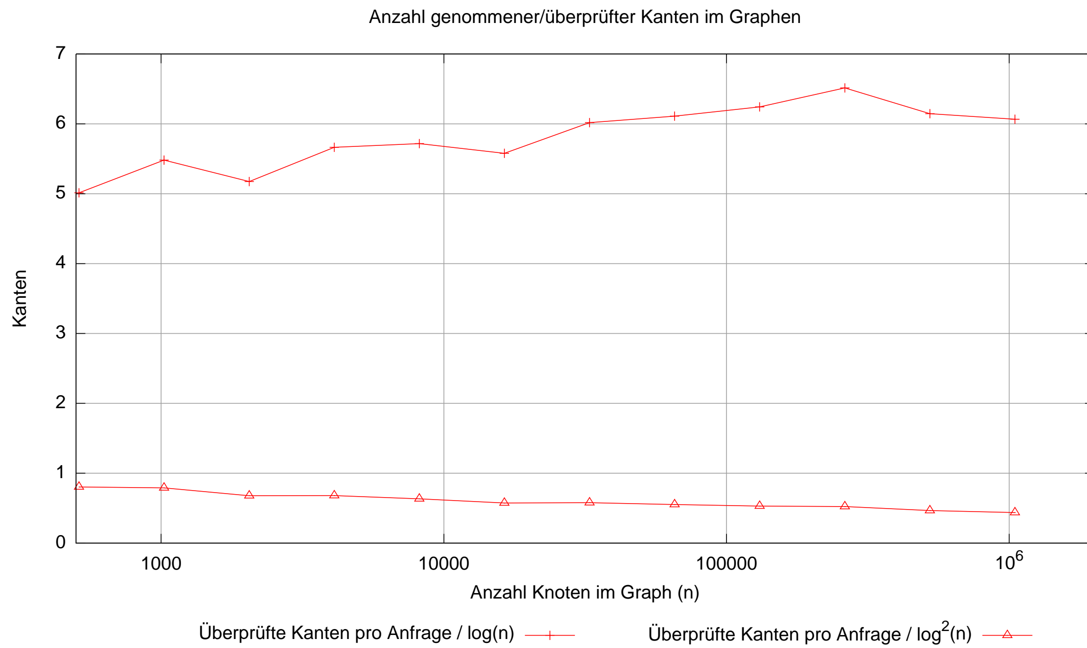

Abbildung 31: Anzahl durchschnittlich betrachteter und traversierter Kanten pro Anfrage dividiert durch $\log(n)$ und $\log^2(n)$

Für alle in diesem Kapitel betrachteten Graphen, wurden die Knoten gleichverteilt in einem Quadrat angeordnet.

7 Zusammenfassung

Der Algorithmus NNK weist gegenüber den Implementierungen von CGAL in allen, hier vorgestellten Fällen Laufzeitvorteile auf. Besonders die inexakten Versionen haben deutlich geringere Laufzeiten. Auch sind die Laufzeiten gegenüber unterschiedlichen Anordnungen der Punktemengen sehr robust. D.h. die Laufzeit variiert nicht so stark.

Die k-d-Bäume weisen bei Anordnungen auf einem Kreis sehr schlechte Laufzeiten auf und die Delaunay Hierarchie weist nur bei einer gleichmäßig verteilten Anordnung über einer Fläche gute Laufzeiten auf. Nur für eine gemischte Anordnung (auf Kreis und innerhalb eines Quadrats) konnte die Implementierung von ANN eine bessere Laufzeit aufweisen.

Nachteile der Implementierung sind sicherlich der höhere Speicherverbrauch durch die zusätzlichen Kanten. Auch bringt eine Erweiterung auf höhere Dimensionen einen starken Anstieg der Kantenanzahl mit sich. Hier haben k-d-Bäume Vorteile. Die Größe des Baumes hängt dort nicht von der Dimension des Raums ab sondern, von der Anzahl der Punkte. Ein weiterer Nachteil ist die bisher fehlende Implementierung einer Löschfunktion, insbesondere eine, die eine geringe Laufzeit aufweist (am besten konstante).

Sind die Punkte in einem 2-dimensionalen Raum angeordnet und Löschoperationen nicht notwendig, wie es zum Beispiel für eine Erweiterung der Funktionalität von Routenplanern der Fall wäre (die bestehenden Daten werden nur relativ selten erneuert), ist der hier vorgestellte Algorithmus NNK sehr gut geeignet, die Anfragen zu beantworten.

Abbildungsverzeichnis

1 Vergleich zwischen Delaunay-Triangulierung und einer einfachen Triangulierung . . . 5
2 Schnitt von zwei Kreisen mit einer Geraden. . . . 7
3 Sehnenviereck. . . . 8
4 Voronoi-Diagramm . . . 8
5 Anfragepunkt q in Umkreis. . . . 9
6 Dreieck zwischen Anfragepunkt und nächstem Nachbarn. . . . 9
7 Knoten v wird durch Strecke $\overline{mn}$ vom nächsten Nachbarn p von q getrennt. 11
8 Die Umkreise von m und n können nicht leer sein. . . . 12
9 Delaunay-Hierarchie. . . . 13
10 Ausschluss von Dreiecken in der Delaunay-Hierarchie. . . . 14
11 $MINMAXDIST$ von einem Punkt P zu einem Rechteck R. . . . 15
12 Beispiel für einen k-d-Baum. . . . 17
13 Schrumpfen eines Kreises. . . . 18
14 Edge flip . . . 18
15 Nächster Nachbar mittels Delaunay-Triangulierung und DT. . . . 20
16 Schlechtes Beispiel: Nächster Nachbar in Delaunay-Triangulierung . . . 20
17 Gegenbeispiel zu Knotenhierarchie . . . 21
18 Beispiel: Nächster-Nachbar-Suche mittels Knotenhierarchie . . . 21
19 Schrumpfen eines Kreises. . . . 23
20 Laufzeitvergleich: NNK gegen Delaunay-Hierarchie . . . 28
21 Gegenbeispiel: Knoten löschen . . . 29
22 Laufzeitvergleich: Vektor gegen Liste. . . . 30
23 Vergleich der Konstruktionszeiten zwischen global und lokal gesetztem Protect_FPU_rounding. . . . 31
24 Vergleich der Laufzeiten von NNK zwischen global und lokal gesetztem Protect_FPU_rounding. . . . 32
25 Laufzeitvergleich bei Anordnung innerhalb eines Quadrats. . . . 35
26 Laufzeitvergleich bei Anordnung auf einem Kreis. . . . 36
27 Laufzeitvergleich bei Anordnung auf einer Parabel. . . . 36
28 Laufzeitvergleich bei gemischter Anordnung auf einem Kreis und innerhalb eines Quadrats . . . 37
29 Anzahl Kanten pro Knoten in einem NNK-Graphen. . . . 39
30 Kanten pro Knoten in NNK. . . . 40
31 Betrachtete und traversierte Kanten. . . . 41

Literatur

[1] Aigner, Martin und Günter M. Ziegler: *Das Buch der Beweise.* Springer, 2. Auflage, 2004.

[2] Bentley, Jon Louis: *Multidimensional binary search trees used for associative searching.* Commun. ACM, 18(9):509–517, 1975.

[3] Berg, Mark de, Otfried Cheong, Marc van Kreveld und Mark Overmars: *Computational Geometry: Algorithms and Applications.* Springer, 2., rev. ed. Auflage, 2000.

[4] Cost, Scott und Steven Salzberg: *A Weighted Nearest Neighbor Algorithm for Learning with Symbolic Features.* Mach. Learn., 10(1):57–78, 1993.

[5] Devillers, Olivier: *The Delaunay Hierarchy.* International Journal of Foundations of Computer Science, 13:163–180, 2002.

[6] Flickner, M., H. Sawhney, W. Niblack, J. Ashley, Qian Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele und P. Yanker: *Query by image and video content: the QBIC system.* Computer, 28(9):23–32, Sep 1995.

[7] Freidman, Jerome H., Jon Louis Bentley und Raphael Ari Finkel: *An Algorithm for Finding Best Matches in Logarithmic Expected Time.* ACM Trans. Math. Softw., 3(3):209–226, 1977.

[8] Kemnitz, Arnfried: *Mathematik zum Studienbeginn.* Vieweg, 7. Auflage, 2005.

[9] Mount, D. M. und S. Arya: *ANN: A library for approximate nearest neighbor searching.* CGC 2nd Annual Fall Workship on Computational Geometry, 1997.

[10] Robert Geisberger, Peter Sanders, Dominik Schultes und Daniel Delling: *Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks.* In: *LNCS 5038*, Seiten 319–333, Provincetown, Massachusetts, May/June 2008. WEA 2008.

[11] Roussopoulos, Nick, Stephen Kelley und Frédéric Vincent: *Nearest neighbor queries.* In: *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, Seiten 71–79, New York, NY, USA, 1995. ACM.

[12] Vitaly Osipov, Ludmila Scharf, Johannes Singler und Peter Sanders: *Engineering Nearest Neighbor Queries.* Persönliche Kommunikation, 10 2008.