Projektdokumentation – Geometrische Netzwerke

Sarah Lutteropp

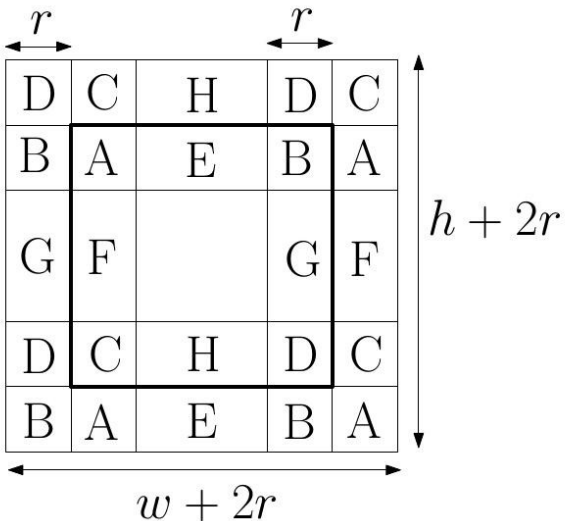
Programmierprojekt im Rahmen der Vorlesung „Graphenalgorithmen und Lineare Algebra Hand in Hand“ im Sommersemester 2014

Einleitung und Motivation

Der in NetworKit enthaltene PubWeb-Algorithmus zur geometrischen Generierung von Netzwerken platziert zufällige gleichverteilte Punkte auf dem Einheitstorus. In einem der Schritte des PubWeb-Algorithmus werden zu jedem dieser Punkte die k nächsten Nachbarn mit einer Distanz kleiner-gleich einem vorher festgelegten Radius bestimmt und mit dem Punkt verbunden. Dieser Radius ist für jeden Punkt gleich. Bisher ist die Bestimmung der innerhalb eines gegebenen Radius liegenden Nachbarn mit einem trivialen Algorithmus (alle Paare von Punkten miteinander testen) in NetworKit umgesetzt, was einen quadratischen Aufwand erfordert. Dies ist für eine große Anzahl von Punkten zu langsam. Ziel dieses Projektes ist es daher, die Bestimmung dieser Nachbarn mittels geeigneter Datenstrukturen zu beschleunigen.

Anpassung an den Torus

Da die im Folgenden verwendeten räumlichen Datenstrukturen (Reguläres Gitter, kD-Baum, Quadtree) ursprünglich für die euklidische Ebene definiert worden sind, müssen Anpassungen an den Torus vorgenommen werden. Während dies beim regulären Gitter sehr einfach über eine Sonderbehandlung der am Rand liegenden Gitterzellen geht, wird für die anderen räumlichen Datenstrukturen die Punktmenge angepasst. Hierfür sind zwei Ansätze denkbar. Ein möglicher Ansatz besteht darin, die Umkehrabbildung der stereographischen Projektion zu verwenden, um die Punkte auf eine dreidimensionale Kugel zu projizieren und mit den Kugelkoordinaten weiterzuarbeiten. Dieser Ansatz wurde nicht verfolgt. Stattdessen werden die Punkte mit Abstand kleiner-gleich dem festen Radius r zum Rand kopiert, um Nachbarschaftsbeziehungen auf einem Torus zu simulieren. Aus einer ursprünglichen Region der Größe $w \times h$ wird eine Region der Größe $(w+2r)\times(h+2r)$. Die folgende Abbildung zeigt, wohin die Punkte kopiert werden. Da die Punkte gleichverteilt vorliegen und der Radius eher klein ist, wird relativ wenig zusätzlicher Speicherplatz benötigt. Die Klasse `TorusAdjuster` stellt eine Methode zu Verfügung, welche die betroffenen Punkte passend kopiert.



Datenstrukturen und Implementierung

Alle neuerstellten Klassen befinden sich im Ordner `geometric`.

Reguläres Gitter

Das reguläre Gitter teilt eine begrenzte Fläche (hier: Quadrat oder Rechteck in $\mathbb{R}^2$) in Zellen gleicher Größe auf. Jede Zelle kann beliebig viele Punkte der vorgegebenen Punktmenge enthalten. Um die Punkte mit Abstand kleiner-gleich r von einem Punkt p zu bestimmen, wird folgendermaßen vorgegangen. Als erstes bestimmt man die Zelle, die den Punkt p enthält. Dann bestimmt man alle Nachbarzellen, die nahe genug an der gefundenen Zelle liegen. Für jeden in den gefundenen Zellen liegenden Punkt wird geprüft, ob der Abstand zu p maximal r beträgt. Da r im gegebenen Anwendungsfall immer gleich bleibt, bietet es sich an, Zellen der Größe $r \times r$ zu verwenden. Dies ist, bei gleichbleibendem r und gleichverteilten Punkten, zugleich eine theoretisch optimale Vorgehensweise. Bei k gefundenen Nachbarpunkten werden erwartet nur $O(k)$ Distanzberechnungen durchgeführt (Bentley 1977).

kD-Baum

Der kD-Baum ist ein binärer Suchbaum, der den Raum abwechselnd nach x- und y-Koordinate trennt. Hierbei sind die Trenngeraden achsenparallel. In den Blättern des kD-Baumes stehen die Punkte der Punktmenge. Verbreitete Vorgehensweisen zur Erzeugung eines kD-Baumes sind der Objekt-Median, der räumliche Median und die Surface Area Heuristik (welche den durchschnittlichen Suchaufwand senkt). Hier wurde der Objekt-Median implementiert, d.h. auf jeder Seite einer Trenngeraden befindet sich maximal die Hälfte aller verbleibenden Punkte. Der kD-Baum eignet sich besonders für achsenparallele rechteckige Anfrageregionen. Um alle Punkte mit Abstand kleiner-gleich r von einem Punkt p zu finden, sucht man die Knoten des kD-Baumes, die einen Bereich von $(2*r) \times (2*r)$ um diesen Punkt abdecken. Die darin enthaltenen Blattknoten werden dann auf eine Distanz kleiner-gleich r zu p getestet. Eine Bereichsanfrage mit einem achsenparallelen Rechteck auf einem kD-Tree benötigt $O(\sqrt{n}+k)$ Zeit, wobei n die Gesamtanzahl der Punkte und k die Anzahl der gefundenen Punkte ist.

Quadtree

Der Quadtree ist ein Suchbaum, der einen quadratischen Bereich rekursiv in jeweils vier Quadrate unterteilt, solange mehr als ein Punkt der Punktmenge in dem Bereich liegt. Die Punkte der Punktmenge befinden sich in Blättern des Quadtrees. Die Suche nach Punkten innerhalb eines gegebenen Radius um einen gegebenen Punkt funktioniert analog zu der Suche im kD-Baum. Im schlechtesten Fall kann eine Bereichsanfrage auf dem Quadtree linearen Zeitaufwand benötigen, in der Praxis tritt dieser Fall jedoch äußerst selten auf.

Anbindung an PubWeb

Die Anbindung an PubWeb erfolgt mittels der Klasse `PubWebNeighborWrapper`. Diese beinhaltet eine Methode `nodesInRadius`, die direkt die auch bislang verwendete Prioritätswarteschlange der Knoten im gegebenen Radius zurückgibt. Dem Konstruktor der Klasse wird der Graph, der Radius (wichtig für Gitter) und ein Enum `Spatial`, welches die Werte `REGULAR_GRID`, `KD_TREE` und `QUADTREE` annehmen kann und die zu verwendende räumliche Datenstruktur repräsentiert, übergeben.

Experimentelle Ergebnisse

Sind im Python-Notebook zu finden.

Fazit

Wie aufgrund der theoretischen Optimalität zu erwarten eignet sich das reguläre Gitter von den untersuchten Datenstrukturen am besten, um die Suche nach den Punkten innerhalb eines festen Radius um einen Punkt für PubWeb zu beschleunigen. Es lässt sich mit linearem Zeitaufwand aufbauen, ermöglicht es, die Suche zu parallelisieren, ist cache-effizient gespeichert und hat theoretisch optimale Anfragezeiten von $O(k)$. Der einzige Nachteil ist der immense Bedarf an Speicherplatz, der bei einem zu klein gewählten Radius zu hoch sein kann.

Will man möglichst wenig Speicher verbrauchen, eignet sich der kD-Baum mit $O(n)$ Speicherbedarf und $O(\sqrt{n}+k)$ Abfragezeit von den untersuchten Datenstrukturen am besten. Einen Quadtree kann man jedoch besser parallelisieren als einen kD-Tree.

Einen guten Mittelweg bildet der Bereichsbaum (Range-Tree) mit $O(n \log n)$ Speicherbedarf und $O(\log n + k)$ Abfragezeit (hier aus Zeitgründen nicht implementiert). Die Grundidee des Bereichsbaums ist es, einen binären Suchbaum für x-Koordinaten zu halten, in dem untergeordnete binäre Suchbäume für y-Koordinaten gespeichert sind.