k-Ordnung Voronoi-Diagramm Erzeugung und Analyse der Komplexität

Xinyu Tu,Xinchuan Wang

Rheinische Friedrich-Wilhelms-Universität Bonn

Zusammenfassung Voronoi Diagramm k-ter Ordnung ist wichtig für die Untersuchung des Verkehrsnetzes moderner Städte. In diesem Bericht wird Voronoi Diagramm k-order auf einem rechteckigen Gitter mit Highways, wesen die Geschwindigkeit höher ist der normalen Straßen, und auf der Grundlage der Berechnung des kürzesten Weges mit dem Dijkstra-Algorithmus erstellt. Die Zeitkomplexität der Implementierung wird in Abhängigkeit von der Gittergröße, der Anzahl der Städte und der Diagrammordnung, bzw. k, sowohl in einer theoretischen Analyse als auch in numerischen Experimenten diskutiert.

1 Einleitung

Mit der Zunahme der Bevölkerung und der Zahl der Fahrzeuge, die die Menschen besitzen, steht der Verkehr in modernen Städten vor immer größeren Herausforderungen. Für die Stadtverwaltung wäre es sehr wichtig zu wissen, wo das größte Problem im derzeitigen Straßennetz liegt und ob der Bau oder die Verbesserung einer Straße das Problem lösen könnte. Da das Straßennetz in vielen modernen Städten einem rechteckigen Gitter ähnelt, bei dem die Straßen entweder horizontal oder vertikal verlaufen.[1] Die Kapazität und der Zustand der Straßen können als relative Geschwindigkeiten ausgedrückt werden, mit denen sich die Menschen auf ihnen bewegen können. Solche Gittersysteme können eine gute Modellierung für die Untersuchung des Verkehrsnetzes in Großstädten sein und ermöglichen es, die Skizze neuer Straßen auf die Verkehrsbedingungen abzuschätzen.

Voronoi-Diagram ist ein Modell, in der ein Gebiet in kleine Regionen zu unterteilen, so dass jede Region nur einen zentralen Stadt enthält. [2] In der Regel gewährleisten die unterteilten Regionen, dass für alle Punkte in dieser Region der nächstgelegene Standort auf dem Gitter der einzige Standort innerhalb der Region ist. Das Voronoi Diagramm findet breite Anwendung in der Geometrie, der Ökologie, dem Ingenieurwesen, der Informationstechnologie und so weiter. Die Verwendung von Voronoi Diagrammen lässt sich bis ins Jahr 1644 zurückverfolgen. Nachdem es von Georgy F. Voronoy [3] definiert und untersucht wurde, ist es in vielen Bereichen intensiv erforscht und genutzt worden.

Voronoi Diagramme k-ter Ordnung, wobei $k > 1$ ist, bei denen k nächstgelegener Städte berücksichtigt werden muss, sind jedoch aufgrund ihrer Komplexität weniger untersucht worden. Für ein Voronoi Diagramm k-ter Ordnung beträgt die strukturelle Komplexität $\theta(k(n-k))$. Die zeitliche Komplexität hängt

jedoch von den verschiedenen Ansätzen zu seiner Erstellung ab. Bei einer einfachen iterativen Konstruktion beträgt die zeitliche Komplexität $O(k^2nlogn)$ [4]. Berücksichtigt man die geometrische Dualität und die Anordnungen, so kann die Zeitkomplexität auf $O(n^2 + k(n-k)log^2n)$ [4] reduziert werden.

In diesem Bericht wird Voronoi Diagram der Ordnung k auf einem rechteckigen Gitter mit mehreren darauf verteilten Highways erstellt. Die Implementierung ist in Java geschrieben, und das Diagramm wird auf der Grundlage des Dijkstra-Algorithmus durch iterative Berechnung des Abstands zwischen den Städten und alle Punkten des Gitters erstellt. Da der Prozess der Berechnung der Abstände sehr zeitaufwändig ist, wurden bei der Implementierung Dijkstra mit Priority-Queue und HashMap als Datenstruktur für Städte verwendet, um den Berechnungsprozess zu beschleunigen. Die Zeitkomplexität dieser Implementierung wird ebenfalls diskutiert, zusammen mit einigen Möglichkeiten, sie zu verringern.

2 Preliminaries

In diesem Abschnitt werden hauptsächlich die Schritte und Grundsätze der Lösung erläutert.
Wir beginnen mit dem Aufbau des grundlegenden Gitters, indem wir die einzelnen Städte und Highway einrichten(Abbildung 1).Die blaue Punkte stellen die markierteStädte.Die blaue Linie im Gitter sind die Highway.

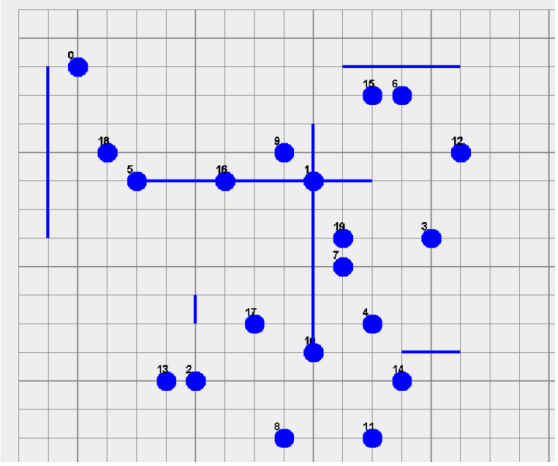


Abbildung 1: Gitter

Das Voronoi-Diagramm ist eine Möglichkeit, eine Region in kleinere Gebiete zu unterteilen. Als Nächstes müssen wir die Punkte in Gebiete unterteilen, die entsprechend der Entfernung zwischen ihnen unterteilt werden können.

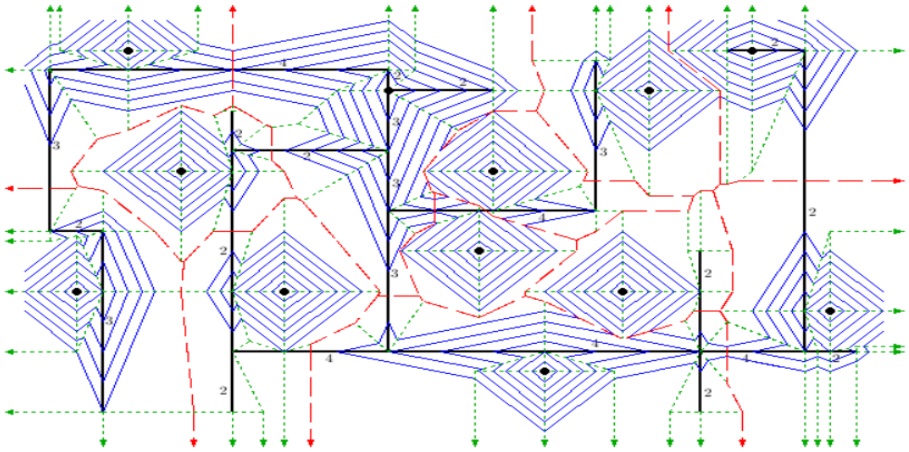


Abbildung 2: Wellen-Modell[6]

Wir wenden den Dijkstra-Algorithmus mit Priorityqueue zur Berechnung des kürzesten Weges an, um die nächstgelegene Punkt zu finden, und verwenden ein Wellen-Modell (Abb.2), um die nächstgelegene Punkt zur Stadt gleichzeitig visuell darzustellen. Wir wählen zunächst mehrere Städte auf dem Gitter aus, berechnen dann die Entfernungen von jeder der anderen Punkte und sortieren sie in verschiedenen Farben, d. h. teilen sie in Regionen ein. Wenn wir zum Beispiel vier Städte a,b,c,d auswählen, wird die Punkte, die der Stadt a am nächsten liegt, in eine Farbe eingeteilt, die Punkte, die der Stadt b am nächsten liegt, in eine andere Farbe, ebenso wie c und d. So wird das gesamte Diagramm in vier verschiedenfarbige Regionen unterteilt. Dies ist nur der Fall, wenn k=1 ist, und wird auf der höheren Ordern näher erläutert. Dann kann die Komplexität des rechnerischen Urteils beginnen. Zum leichteren Verständnis werden einige wichtige Begriffe eingeführt.

2.1 Gitter und L_1-Metric

Wir nennen ein Voronoi-Diagramm eines gegebenen städtischen Gitter G und einer gegebenen Menge S von Punkte in der Ebene ein City-Voronoi-Diagramm,d. h. $V_G(S)$.Die Größe von Gitter kann beliebig verändert werden.Die Gittergröße beträgt $i * j$,i für Zeile und j für Spalte. Für jeden einzelnen Start gibt das City-Voronoi-Diagramm die Fläche aller Punkte an, die bei der Fahrt von diesem Stadt aus zuerst erreicht werden.
Wir konstruieren äquidistante Gitter und können uns nur entlang von Straßen bewegen, d. h. auf horizontalen oder vertikalen Kanten.Wir verwenden also den L_1-Metric, um die Abstand zwischen zwei Punkte zu berechnen.Angenommen, es gibt zwei Punkte $a(a_x, a_y)$und $b(b_x, b_y)$ auf Gitter, dann ist der Abstand zwischen diesen beiden Punkten $L_1(a, b) = |a_x - b_x| + |a_y - b_y|$.Dann setzen wir die Highway ein. Die Bewegungsgeschwindigkeit auf einer normalen Straße ist 1

und auf einer Highway ist $v(v > 1)$.Die Geschwindigkeit auf der Highway kann beliebig eingestellt werden.

2.2 Wellen

Wellen hier sind die Wavefront Propagation[8].Die Wavefront Propagation ist ein perfektes Modell für die Definition von Voronoi-Diagrammen Wir können damit die Entstehung von $V_G(S)$ erklären und die Komplexität analysieren. Für einen Stadt $p \in S$ wenden wir das Konzept des Dijkstra-Algorithmus an, um die Nähepunkte $q \in S$ zu finden und sie zu verbinden, und finden dann die entsprechenden Nähepunkte $h \in S$ aus den Nähepunkte und verbinden sie wieder (wie in Abbildung 3). Wenn man auf eine Highway trifft, ändert sich die Geschwindigkeit und damit die Form (Abb. 3).

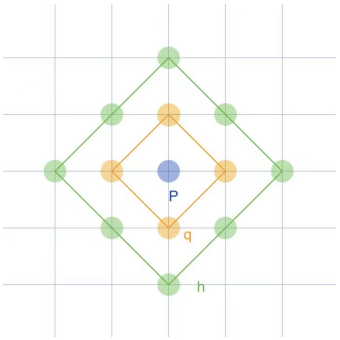


Abbildung 3

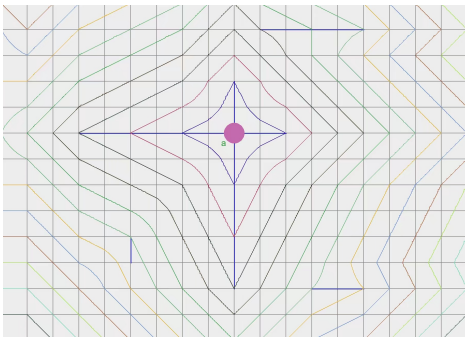


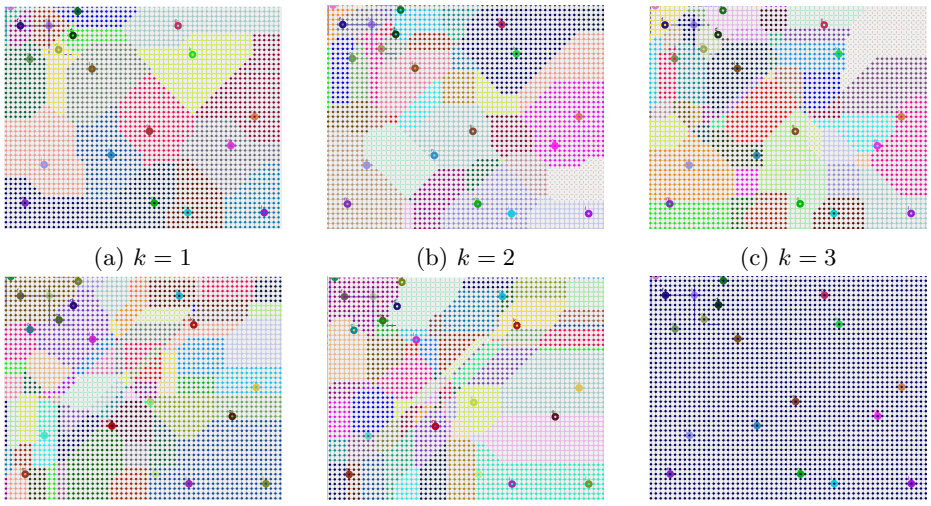
Abbildung 4

2.3 K-Order

Wir wählen eine beliebige Anzahl von Städten im Gitter aus und berechnen wir die Entfernung von anderen Punkten zu diesen Städten, geordnet nach der Entfernung von der nächstgelegenen zur am weitesten entfernten Stadt. Wir haben 20 Städte mit dem Code a,b,c,d,.... ausgewählt. Hier verwenden wir zur leichteren Unterscheidung Buchstaben für die Städtenamen, im Code verwenden wir Zahlen für die Städtenamen, damit man die Anzahl der Städte leichter erkennen kann. Es gibt eine Abstandsordnung der Punkte $x : b, d, s, a, \cdots$ Die Abstandsordnung zum Punkt y sind: $b, s, a, e, \cdots$ Die Abstandsreihenfolge für den Punkt z ist: $b, s, d, h, \cdots$. Wenn k=1 ist, bedeutet dies, dass die erste Stadt, die dem gleichen Punkt am nächsten liegt, in eine Region unterteilt ist, d.h. die nächstgelegene Stadt für x,y,z ist b. Wir können die Punkte x,y,z mit der gleichen Farbe beschriften. Ähnlich für k=2, y und z haben die gleiche Städte b und s,also y und z sind die gleiche Farbe. k=3, x und z haben die gleiche Farbe.Da die ersten k Städte ausgewählt werden sollen, um zu vergleichen, ob sie gleich sind, muss

hier nicht auf die Reihenfolge geachtet werden, so dass wir beschlossen haben, ein Hashset zum Speichern der Elemente zu verwenden.Wir werden später mehr über die Verwendung von Hashset erklären(Abbildung 5).Die Abbildung zeigt die Farbverteilung, nachdem wir k geändert haben.

- Punkt x: b,d,s,a,... k=1:x,y,z
- Punkt y: b,s,a,e,.... k=2:y,z
- Punkt z: b,s,d,h,.... k=3:x,z(Hashset)



(a) $k = 1$ (b) $k = 2$ (c) $k = 3$

(d) $k = 7$ (e) $k = 10$ (f) $k = 20$

Abbildung 5: Beispiel

3 Algorithmus

Der Prozess der Implementierung des Voronoi Diagramms umfasst zwei Prozesse. Der erste Prozess berechnet die Entfernung zwischen jeder Stadt und den Punkten auf dem Gitter, und der zweite Prozess gruppiert die Punkte auf dem Gitter nach den k-ten nächstgelegenen Städten und färbt die Punkt entsprechend Gruppen. Die beiden Prozesse werden in diesem Abschnitt im Detail besprochen. Darüber hinaus wird in diesem Abschnitt auch die Idee von Highways auf dem Gitter erörtert.

3.1 Berechnung des kürzesten Abstands

Der Prozess der Entfernungsberechnung basiert auf dem grundlegenden Dijkstra-Algorithmus. Es handelt sich um einen Algorithmus zur Ermittlung des kürzesten Weges zwischen zwei Punkten auf der Grundlage von Iterationen. In unserer Implementierung berechnen wir die Distanz zwischen normale Punkte und Städte, damit wir die k nächstgelegener Städte für diesen Punkt bestimmen kann. Hier wird keine Distanz zwischen zwei Städte oder zwischen zwei Punkte berechnet.

Zu Beginn wird der Abstand für alle Punkte (außer dem Startpunkt) auf „MAX VALUE “gesetzt, und Startpunkt wird auf 0 gesetzt. Dann werden alle Städte in PriotityQueue hinzufügt, um später schneller von Queue restliche Städte zu finden. Die Berechnung der Entfernung der Punkte auf dem Gitter für eine Stadt muss von dem Punkt selbst ausgehen und schrittweise erweitert werden, um das gesamte Gitter abzudecken.Dann wird in jeder Iteration der Stadt mit dem kleinsten Abstand aus PriorityQueue ausgewählt, und in for-Schleife werden alle adjazente Kante für diesen Punkt iteriert, der Abstand seiner Nachbarn wird bestimmt. Falls die Distanz der nächstgelegene Stadt u kleiner als die Summe von vorhandene minimale Distanz und die adjazente Kante, wird diese Stadt u von PriorityQueue gelöscht, und die kleinste Distanz der Stadt u1 wird aktualisiert. Jetzt wird die Wenn der PriorityQueue leer ist, d.h. kleinste Distanzen von alle Städte zu diesem Punkt schon berechnet ist, wird die Iteration beendet. Zu beachten ist, dass in diesem Schritt auch der Unterschied zwischen normalen Straßen und Highways berücksichtigt wird. Da das Programm die Entfernung eines Punktes (i,j),hier werden alle Punkte und Städte als 2-Dimensional Array,bzw. deren Position, gespeichert, sehe Abbildung 6 orange Punkt, aus der Entfernung seines Nachbarn, z.B. (i+1,j), bestimmt, wird die Gesamtentfernung von (i,j) als Gesamtentfernung von (i+1,j) plus der Entfernung von (i,j) zu (i+1,j) berechnet. Die Entfernung zwischen den beiden Punkten wird dann berechnet, indem berücksichtigt wird, ob die Kante zwischen ihnen auf einer Highway liegt.

3.2 Lagerung von Highways

Highways sind spezielle Kanten zwischen den Punkten auf dem Gitter. Im Gegensatz zu gewöhnlichen Kante, deren Geschwindigkeit konstant bei 1,0 liegt, ermöglichen Highways, sich schneller (mit einer Geschwindigkeit von $v > 1$) von einem Punkt zu der Stadt zu bewegen. Die Highways erhöhen die Komplexität unseres simulierten Systems und macht es dem Verkehrsnetz in realen Städten sehr viel ähnlicher. Allerdings sind Highways keine unabhängigen Einheiten wie Städte. Stattdessen sind sie spezielle Tyen der Kante, wie Abbildung 6 zeigt. In der Implementierung schreiben wir eine einzelne Klasse für Highways, nämlich „Road “. Sie werden als Punktpaare beschreibt, und als ArrayList gespeichert, nämlich die Position des Startpunkt und Endpunkt. Wir können auch die Geschwindigkeit durch deren Parameter „speed “verändert. Außerdem haben wir noch eine andere Klasse, Edge, durch deren parameter w, kennen wir während Berechnung der kleinsten Distanz, ob diese Kante Highway

ist, falls w=1, ist die normale Kante, sonst ist die Highway. Wenn zum Beispiel die Entfernung vom Punkt (i,j) zum Punkt (i+1,j+3), bzw. der orange Punkt zu der grüne Punkt in Abbildung 6, wird im Programm berechnet. Die Punktpaaren ((i,j),(i+1,j)), ((i+1,j),(i+1,j+3)) wird bevorzugt, stattdessen $(i, j+1) \rightarrow (i, j+2) \rightarrow (i, j+3) \rightarrow (i+1, j+3)$.Denn hier ist ((i+1,j),(i+1,j+3)) Highway, und die Geschwindigkeit wird als 1/v angegeben, schneller als die normalen Straßen (i,j)(i,j+3). Der Einfachheit halber wird nur der Abstand für Segmente auf Highways explizit gespeichert. Für andere Kanten wird ein Standardwert von 1,0 zurückgegeben, da das Punktpaar nicht in der HashMap gefunden wird.

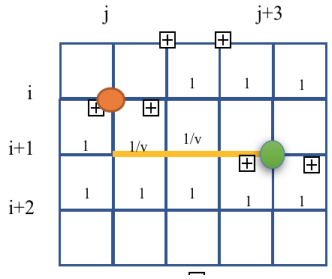


Abbildung 6

3.3 Gruppierung der Punkte

Nachdem die Entfernung zwischen einer beliebigen Kombination aus Stadt und Punkt aus dem Gitter berechnet wurde, können die Punkte in verschiedene Gruppen unterteilt werden, um das Voronoi Diagramm zu erstellen. Bei diesem Prozess wird jeder Punkt auf dem Gitter betrachtet, indem alle Entfernungen zwischen ihm und allen Städten ermittelt werden, und die k Städte mit den kürzesten Entfernungen werden ermittelt. Danach wird der Punkten in die entsprechende Gruppe eingeordnet, indem die k nächstgelegenen Städte berücksichtigt werden. Hier wird eine HashSet für die Speicherung der k-nächstgelegene Stäte benutzt, denn keine doppelten Elemente in HashSet existiert und Elemente, die in unterschiedlicher Reihenfolge angeordnet sind, als identisch behandelt. Zum Beispiel, wenn wir ein k=3 Voronoi Diagramm bekommen wollen, Punkte mit 3-nächstgelegene Städte 2,0,1 und 0,1,2 gehören in der gleiche Gruppe. Diese Reihenfolge der k-nächstgelegene Stäte wird als HashSet im Code implementiert, und als „key “von „topCities “gespeichert, deren „key “die Punkte sind, die identische k-nächstgelegene Stäte haben, damit es schneller gesucht werden kann, denn Operation auf HashMap beträgt nur O(1).

Da alle Punkte in entsprende Gruppen eingeteilt werden, werden sie entsprechend der Gruppen eingefärbt. Die Farben für jede Gruppe werden aus einer vordefinierten Farbliste [7] ausgewählt, so dass die Farben zwischen verschiedenen

Gruppen voneinander unterscheidbar sind.Um die Komplexität zu analysieren, vergleichen wir die Laufzeiten nach Änderung der Anzahl der Städte, der Größe des Gitters und der Größe von k. Auf diese Weise wird ein Voronoi Diagramm k-ter Ordnung erstellt und visualisiert.

4 Komplexität Analysis

Die Verwendung des Dijkstra-Algorithmus allein ist ineffizient und zeitaufwändig. Daher verwenden wir zur Optimierung eine PriorityQueue. Wie aus der Abbildung 7 ersichtlich ist, verwenden wir im Dijkstra-Algorithmus eine PriorityQueue, in der die Punkte in der Queue, beginnend mit der Punkt mit der geringsten Entfernung, eingeordnet werden, so dass die Komplexität hier $O(V)$ beträgt. Als Nächstes müssen wir nacheinander die Punkt mit der geringsten Entfernung aus der PriorityQueue herausnehmen, und der kleinste Wert, der jedes Mal aus der Queue herausgenommen wird, ist die Punkt, die beim nächsten Mal verwendet werden soll, ausgehend von der Startpunkt mit der Entfernung Null, so dass es $O(V)$ Punkte in der Queue gibt, dann erreichen wir unsere Nachbarn über die Kante, vergleichen die Entfernung und aktualisieren sie, und fahren mit der Schleife fort, bis alle Punkte aktualisiert sind, so dass die Komplexität $O(V \cdot (log(V) + E)$.Dabei ist E die Anzahl aller Kanten und V die Anzahl der Punkte($V = i * j$).
Der gesamte Code läuft hauptsächlich in alldistance() ab,wie Abbildung 8.Wir beginnen mit einem einzelnen Punkt, wir müssen bestimmen, ob der Punkt ein Punkt oder eine Stadt ist, wenn es ein Punkt ist, dann machen wir eine Entfernungssortierung, dann rufen wir Dijkstra() auf, um die kürzeste Entfernung zu berechnen, hier ist, wie oben erwähnt, die Laufzeit für jede Dijkstra() ist $O(V \cdot (log(V) + E)$. Dann werden die Städte mit Quicksort sortiert, wir setzen die Anzahl der Städte auf c, so dass die Laufzeit in der Methode sortcities() ist $O(c \cdot log(c))$. Anschließend werden die ersten k Städte auf der Grundlage der k Werte ausgewählt, und jeder ausgewählte Key markiert den entsprechenden Value mit einer Farbe. Wenn sie mit der vorhergehenden übereinstimmt, hat sie die gleiche Farbe, wenn sie sich von ihr unterscheidet, wird sie als eine andere Farbe markiert, wobei die Laufzeit O(1) beträgt. Es werden insgesamt k mal Iterationen durchgeführt, so dass die Laufzeit für den Färbungsteil O(k) beträgt. Die Laufzeit für einen einzelnen Punkt beträgt also $O(V \cdot (log(V) + E) + (c \cdot log(c)) + k)$. Und wir müssen alle Punkte farblich markieren, so dass es V mal Iterationen braucht. Unsere Gesamtlaufzeit beträgt also $O(V \cdot (V \cdot (log(V) + E) + (c \cdot log(c)) + k))$.

5 Diskussion

Die Zeitkomplexität des Algorithmus in dieser Implementierung ist definitiv nicht optimal. In der bisherigen Literatur wurden mehrere Möglichkeiten zur Verringerung der Komplexität vorgeschlagen. Die Verwendung von Fibonacci-Heaps

```java
public void dijkstra(City start){
    for(int i = 0;i<colNumber;i++){// i-mal iteration
        for(int j = 0;j<rowNumber;j++){// j
            cities[i][j].distance = Integer.MAX_VALUE;
            cities[i][j].parent = null;
        }
    }//insgesamt O(V) Anzahl der Punkte auf Gitter: V=i*j
    start.distance = 0;

    PriorityQueue<City> Q = new PriorityQueue<City>(City.com);
    for(int i = 0;i<colNumber;i++){// i
        for(int j = 0;j<rowNumber;j++){// j
            Q.add(cities[i][j]);
        }
    }//insgesamt O(V)  V=i*j

    while(Q.isEmpty() == false){// O(V)
        City u = Q.poll();//Operation auf Priority Queue: log(V)

        for (Edge ed : u.edges) {// O(E) Kante=E, alle Kante durchlaufen
            City v = ed.getOther(u);
            if(v.distance > u.distance + ed.w){
                Q.remove(v);
                v.distance = u.distance + ed.w;//min.dis. aktualisieren
                v.parent = u;
                Q.add(v);
            }
        }
    }//insgesamt O(V(log(V)+E))
```

Abbildung 7: Dijkstra

```java
private void allDistance(){
    topCities.clear();//hashmap clear
    int colorIndex = 0;
    for(int i = 0;i<colNumber;i++){
        for(int j = 0;j<rowNumber;j++){
            if(cities[i][j].state == 0){//city==1, point==0

                allSort.clear();
                dijkstra(cities[i][j]);//O(V(log(V)+E))

                sortCities();//Quicksort,Anzahl der Stadt:c->O(c*log(c))


                HashSet<City> top = new HashSet<City>();
                for(int p = 0;p<k;p++){//O(k),k-mal iteration
                    City c = allSort.get(p);
                    top.add(c);
                }

                if(topCities.containsKey(top)){
                    cities[i][j].color = topCities.get(top).get(0).color;
                    topCities.get(top).add(cities[i][j]);//O(1)
                }
                else{

                    cities[i][j].color = colorList.get(colorIndex);
                    colorIndex++;
                    ArrayList<City> points = new ArrayList<City>();
                    points.add(cities[i][j]);
                    topCities.put(top, points);//O(1)
                }//insgesamt O(V(V(log(V)+E))+(c*log(c))+k))
```

Abbildung 8: Farbunterteilung

oder binären Heaps könnte die Zeitkomplexität auf theoretischer Basis verringern [8]. In der Praxis könnte die Leistung jedoch schlechter sein als die theoretische Vorhersage [9](Skiena, 2008). Ein Uniform-Cost-Search-Algorithmus wurde ebenfalls als praktische, schnellere Alternative zum Dijkstra-Algorithmus vorgeschlagen [10](Felner, 2011).

6 Fazit

Abschließend wird in diesem Bericht Voronoi Diagramm k Ordnung auf rechteckigen Gitter berechnet, die Highways mit Geschwindigkeiten v enthalten. Bei der Implementierung wird der Dijkstra-Algorithmus zur Berechnung der Entfernung für den kürzesten Weg zwischen jedem Punkt auf dem Gitter und den Städten verwendet. Die Zeitkomplexität für die Implementierung beträgt ungefähr $O(V \cdot (V(log(V) + E) + k + clogc)$, wobei V die Anzahl der Punkte und E die Anzahl der Kante auf dem Gitter, c die Anzahl der Stadt ist.

Literatur

1. Gemsa, A., Lee, D., Liu, C.-H., & Wagner, D. (2012): *Higher order city Voronoi diagrams* SWAT'12: Proceedings of the 13th Scandinavian conference on Algorithm Theory, 59-70.
2. Cooperative authors. (2022, 3 11): Voronoi diagram. Retrieved from Wikipedia: $https://en.wikipedia.org/wiki/Voronoi_diagram$
3. Voronoi, G. (1908) : Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Premier mémoire. Sur quelques propriétés des formes quadratiques positives parfaites. Journal für die Reine und Angewandte Mathematik, 97-178.
4. Lee, D. T. (1982). On k-nearest neighbor Voronoi diagrams in the plane, IEEE Trans. Comput., 478-487.
5. https://i11www.iti.kit.edu/en/projects/geonet/cvd
6. Aichholzer, O., Aurenhammer, F. & Palop, B. Quickest Paths, Straight Skeletons, and the City Voronoi Diagram. Discrete Comput Geom 31, 17–35 (2004). $https://doi.org/10.1007/s00454-003-2947-0$
7. $http://godsnotwheregodsnot.blogspot.com/2012/09/color-distribution-methodology.htm$
8. Mehlhorn, Kurt & Sanders, Peter. (2008). Shortest Paths. $10.1007/978-3-540-77978-0_10$.
9. Steven S. Skiena. 2008. The Algorithm Design Manual (2nd. ed.). Springer Publishing Company, Incorporated.
10. Felner, Ariel. (2011). Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm..