technische universität
dortmund

Bachelorarbeit

Analyse der orientierten Dilation auf Triangulierungen

Nicolas Wünderich
Oktober 2023

Gutachter:
Prof. Dr. Kevin Buchin
M. Sc. Antonia Kalb

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Algorithm Engineering (LS 11)
http://ls11-www.cs.tu-dortmund.de

Kapitel 1

Einleitung

1.1 Motivation und Hintergrund

Die Bachelorarbeit beschäftigt sich mit *gerichteten geometrischen t-Spanngraphen*. Diese sind gerichtete Teilgraphen $\vec{G} = (P, \vec{E})$ des vollständigen Graphen $\vec{K}_n$ über die Punktmenge P [8]. Die Länge der Kante (p_1, p_2) zwischen zwei beliebigen Punkten $p_1, p_2 \in P$ ist in $\vec{K}_n$ der euklidische Abstand $|p_1 - p_2|$. Wenn $\vec{E}$ weniger Kanten umfasst, als in $\vec{K}_n$ vorhanden sind, kann es Punktpaare $p_1, p_2 \in P$ geben, deren kürzester Pfad $d_{\vec{G}}(p_1, p_2)$ ein Umweg gegenüber der Kante (p_1, p_2) ist. Die Länge des enstehenden Umwegs entspricht dem, um einen Faktor t, der als *Dilation* bezeichnet wird, gestreckten euklidischen Abstand. Der Parameter t eines t-Spanngraphen ist genau der maximale Umweg, welcher auf diese Weise [8].

Die Kanten der betrachteten Spanngraphen sind orientiert, wenn aus $(p_1, p_2) \in \vec{E}$ folgt $(p_2, p_1) \notin \vec{E}$. Spanngraphen dieser Art werden als *orientierte geometrische t-Spanngraphen* bezeichnet. Die Dilation kann damit entgegen der Kantenrichtung niemals 1 sein und eignet sich nur bedingt als Qualitätsmaß. Für orientierte t-Spanngraphen wird *orientierte Dilation* betrachtet, welche nicht den direkten Weg zwischen p_1 und p_2 als Maß nimmt, sondern den orientierten Kreis, der p_1 und p_2 enthält. Der *optimale orientierte Kreis* $\Delta(p_1, p_2)$ aus $\vec{K}_n$ für zwei Punkte $p_1, p_2 \in P$ ist das Dreieck $\Delta_{p_1 p_2 p_3}$, wobei $p_3 \in P \setminus \{p_1, p_2\}$ so gewählt ist, dass $\Delta_{p_1 p_2 p_3}$ minimalen Umfang hat. In einem orientierten Spanngraph $\vec{G}$ ist der kürzeste orientierte Kreis mit $C_{\vec{G}}(p_1, p_2)$ gegeben. Die orientierte Dilation t von $\vec{G}$ entspricht dem maximalen Faktor, den der Weg von $C_{\vec{G}}(p_1, p_2)$ länger ist als $\Delta(p_1, p_2)$ für alle distinkten Punkte $p_1, p_2 \in P$ [8].

Von großem Interesse ist es oft, einen t-Spanngraphen zu einer gegebenen Punktmenge zu finden, der zum einen möglichst wenig Kanten hat und zum anderen einen möglichst kleinen Faktor t aufweist. Dieses Problem ist im Allgemeinen NP-schwer [9].

Orientierte geometrische t-Spanngraphen finden auf verschiedenen Gebieten praktische Anwendung. So können diese für die Navigation von Robotern verwendet werden [14], zum

Beispiel um kürzeste Wege zu bestimmen und um Kollisionen untereinander zu vermeiden. Im Bereich der Geoinformationssysteme lassen sie sich nutzen, um Infrastrukturen, wie Straßen- und Stromnetze, darzustellen [1]. Auch können sie zur Konstruktion drahtloser Sensornetzwerke verwendet werden sowie dabei die Latenzzeit von Nachrichten zwischen Sensoren und Empfänger minimieren.

1.2 Aufbau der Arbeit

In Kapitel 2 werden grundlegende Begriffe und Konzepte vorgestellt und definiert, die für das Verständnis der Arbeit essentiell sind. Zuerst werden allgemeine Begriffe bezüglich Graphen erläutert. Darauf aufbauend wird dann die Graphenklasse der Spanngraphen definiert und zum Schluss auf Triangulationsverfahren eingegangen.

In Kapitel 3 wird dann ein Algorithmus präsentiert, der aus zweidimensionalen Punktmengen variabler Größe n einen orientierten t-Spanngraph mit seiner orientierten Dilation t berechnet. Diese Berechnung findet in 4 Teilschritten statt. Zuerst wird eine konvexe Punktmenge P erzeugt. Diese Einschränkung ist gewählt, da zu konvexen Punktmengen immer eine wohlorientierte Triangulierung existiert [8]. Dies ist nicht für jede beliebige Punktmenge der Fall. Im zweiten Teilschritt wird die konvexe Punktmenge mit zwei unterschiedlichen Triangulationsverfahren, der gierigen und der Delaunay-Triangulation, trianguliert. Eine *Triangulation* ist die Aufteilung von einer Menge Punkte in Dreiecke, sodass alle Punkte ein zusammenhängendes Netz bilden. Die resultierende Triangulation T wird im nächsten Schritt konsistent orientiert, wobei die Kanten jedes Dreiecks der Triangulierung anschließend so gerichtet werden, dass diese einen orientierten Kreis für seine Eckpunkte bilden. Die daraus resultierende Menge gerichteter Kanten ist, zusammen mit der eingegebenen Punktmenge, ein orientierter $O(1)$-Spanngraph [8]. Zuletzt wird noch die maximale orientierte Dilation t bestimmt und ausgegeben.

Zu jedem dieser Teilschritte werden mehrere Algorithmen präsentiert, die für zufällig generierte, unterschiedlich große zweidimensionale Punktmengen auf Laufzeit und Speicherplatzverbrauch getestet werden. Zuletzt werden die Algorithmen mit den effizientesten Teilschritten, für die gierige und die Delaunay-Triangulation, auf ihre Laufzeit und ihren Speicherplatzverbrauch, sowie ihre größte und kleinste maximale Dilation analysiert und miteinander verglichen.

Algorithmus	$n = 2^5$	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Orientieren2	$0,003504$	$0,014036$	$0,039953$	$0,145452$	$0,60227$	$8,57855$	$34,3312$	$145,875$
Änderungsrate		$4,01$	$2,85$	$3,64$	$4,14$	$14,24$	4	$4,25$

Tabelle 3.16: Laufzeiten von KonsistentOrientieren-Nachbarschaftsmatrix in Sekunden und Änderungsraten für $\frac{2n}{n}$

Algorithmus	$n = 2^5$	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Orientieren3	$0,01051$	$0,03803$	$0,1076$	$0,4091$	$4,090$	$16,96$	$67,27$	$273,1$
Änderungsrate		$3,62$	$2,83$	$3,8$	10	$4,15$	$3,97$	$4,06$

Tabelle 3.17: Laufzeiten von KonsistentOrientieren-Adjazenzlisten in Sekunden und Änderungsraten für $\frac{2n}{n}$

Der Algorithmus KonsistentOrientieren-Nachbarschaftsmatrix benötigt nach Theorem 3.4.6 $\mathcal{O}(n^2)$ Speicher, was einer Vervierfachung des belegten Speichers bei Verdopplung von n entspricht. Experimentell belegt KonsistentOrientieren-Nachbarschaftsmatrix im Schnitt $3,71$-fachen Speicher bei Verdopplung von n auf $2n$. Dies liegt nah am theoretischen Wert und bestätigt diesen experimentell.
Nach Theorem 3.4.8 hat KonsistentOrientieren-Adjazenzlisten wie KonsistentOrientieren-Naiv lineare Speicherkomplexität. Der belegte Speicher erhöht sich hier im Schnitt um das $2,15$-fache bei Verdopplung von n. Die experimentellen Ergebnisse stimmen damit näherungsweise mit der Theorie überein.

3.5 Berechnung der maximalen Dilation

Im vierten und letzten Teilschritt wird die maximale orientierte Dilation t der konsistent orientierten Triangulation $\vec{T} = (P, \vec{E})$ bestimmt. Dafür muss der kürzeste Zyklus von einem Paar distinkter Punkte $p, p' \in P$ bestimmt werden. Hierfür wird der *Dijkstra-Algorithmus* angewandt. Dieser berechnet in einem gerichteten Graphen den kürzesten Weg von einem Punkt zu allen anderen Punkten [20]. Der Zyklus $C_{\vec{G}}(p, p')$ ist die Summe der kürzesten Wege von p nach p' und von p' nach p, womit durch zweimaliges Anwenden von Dijkstra der Zyklus berechnet werden kann. Die optimale Laufzeit für Dijkstra beträgt bei nicht-negativen Kantengewichten und unter Verwendung eines Fibonacci-Heaps $\mathcal{O}(|P| \log |P| + |\vec{E}|) = \mathcal{O}(n \log n)$ [11]. In den folgend vorgestellten Algorithmen wird der Dijkstra-Algorithmus als Funktion `dijkstra` verwendet.
Zuerst wird eine Variante MaximaleDilation-Naiv vorgestellt, die `dijkstra` verwendet, um kürzeste Zyklen zu bestimmen. Folgend wird eine verbesserte Variante MaximaleDilation-DijkstraMatrix beschrieben, die ebenfalls `dijkstra` verwendet, dabei aber durch Vorspei-

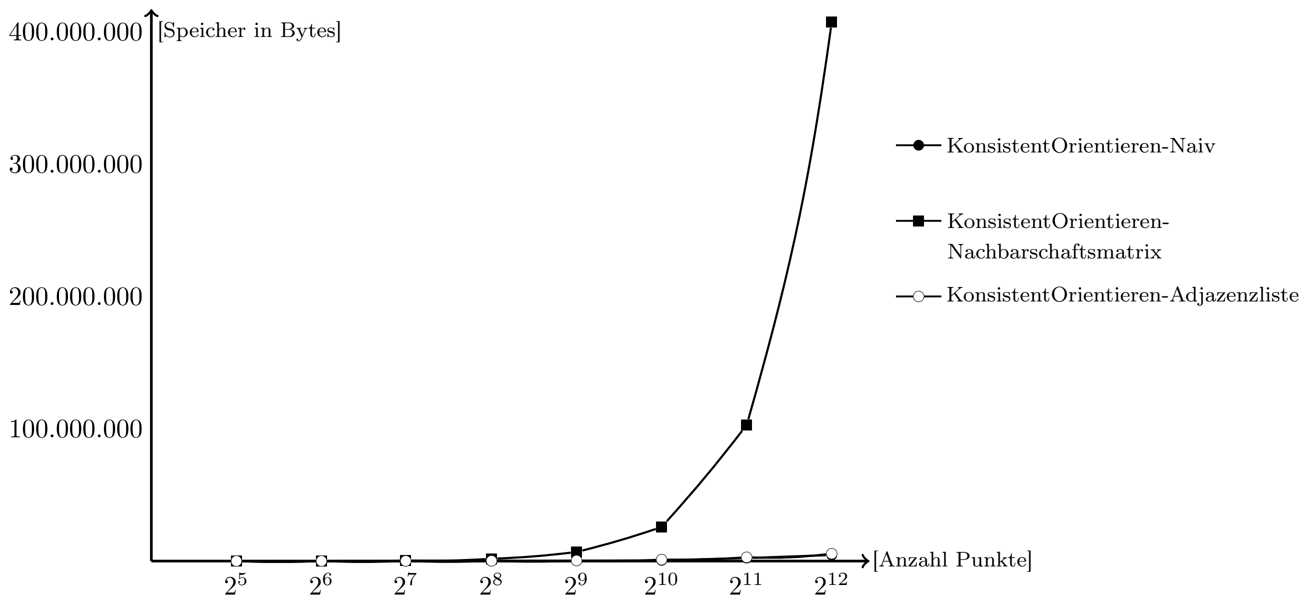

Abbildung 3.8: Speicherbedarf von KonsistentOrientieren-Naiv, KonsistentOrientieren-Nachbarschaftsmatrix & KonsistentOrientieren-Adjazenzlisten. Die x-Achse entspricht der Anzahl Punkte n aus P und die y-Achse dem maximal benötigten Speicher in Bytes in Abhängigkeit von n, um die ungerichtete Triangulation $T = (P, E)$ konsistent zu orientieren.

Algorithmus	$n = 2^5$	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Orientieren1	19904	37344	73744	145344	289744	576944	2380144	4770224
Änderungsrate		$1,88$	$1,97$	$1,97$	$1,99$	$1,99$	$4,13$	2

Tabelle 3.18: Speicherbedarf von KonsistentOrientieren-Naiv in Bytes und Änderungsraten für $\frac{2n}{n}$

cherung der kürzesten Wege zwischen allen möglichen Paaren distinkter Punkte eine effizientere Laufzeit erreicht.

3.5.1 Maximale Dilation mit Dijkstra bestimmen

MaximaleDilation-Naiv bestimmt die maximale orientierte Dilation t einer konsistent orientierten Triangulation $\vec{T} = (P, \vec{E})$. Hierfür wird für jedes Paar distinkter Punkte $p_i, p_j \in P$ zuerst die Länge des optimalen Dreiecks $\Delta(p_i, p_j)$ berechnet. Danach wird mit dem Dijkstra-Algorithmus die Länge des kürzesten Weges w_1 von p_i nach p_j in $\vec{T}$ und die Länge des kürzesten Weges w_2 von p_j nach p_i in $\vec{T}$ bestimmt. Die Länge des kürzesten Zyklus $C_{\vec{G}}(p_i, p_j)$ entspricht damit der Summe von w_1 und w_2 und die orientierte Dilation $odil(p_i, p_j)$ wird mit $\frac{|C_{\vec{T}}(p_i,p_j)|}{|\Delta(p_i,p_j)|}\}$ berechnet. Indem eine Variable t mit 1 initialisiert wird und im Fall, dass $odil(p_i, p_j) > t$ auf $odil(p_i, p_j)$ gesetzt wird, enthält t nach Berechnung der orientierten Dilation für alle Punktpaare die gesuchte maximale orientierte Dilation und wird ausgegeben.

Eingabe: konsistent orientierte Triangulation $\vec{T} = (P, \vec{E})$
Ausgabe: maximale orientierte Dilation t

$t \leftarrow 1$
for $i = 1$ **to** $|P| - 1$ **do**
  $p_i \leftarrow P[i]$
  for $j = i + 1$ **to** $j < |P|$ **do**
    $p_j \leftarrow P[j]$
    $|\Delta(p_i, p_j)| \leftarrow$ `optimalesDreieck`
    $w_1 \leftarrow$ `dijkstra` *// Anwendung des Dijkstra-Algorithmus um den kürzesten Weg von p_i nach p_j zu bestimmen*
    $w_2 \leftarrow$ `dijkstra`
    $|C_{\vec{T}}(p_i, p_j)| \leftarrow w_1 + w_2$
    $t \leftarrow \max\{t, \frac{|C_{\vec{T}}(p_i, p_j)|}{|\Delta(p_i, p_j)|}\}$
  end for
end for
return t

Algorithmus 3.15: MaximaleDilation-Naiv

Eingabe: Punktmenge P von $\vec{T}$ und zwei distinkte Punkte $p_i, p_j \in P$
Ausgabe: $|\Delta(p_i, p_j)|$

$opt \leftarrow \infty$
for all $p \in P$ **do**
  if $p \neq p_i$ **and** $p \neq p_j$ **then**
    $opt \leftarrow \min\{opt, |(p_i, p_j)| + |(p_i, p)| + |(p_j, p)|\}$
  end if
end for
return opt

Algorithmus 3.16: optimalesDreieck

Laufzeit

Initialisieren von t mit dem Wert 1 hat eine Zeitkomplexität von $\mathcal{O}(1)$. Die zwei verschachtelten for-Schleifen weisen eine Laufzeit von $\mathcal{O}(n^2)$ auf. Während p_i dabei $n-1$ mal initialisiert wird ($\mathcal{O}(n)$) benötigt die Initialisierung von p_j Laufzeit von $\mathcal{O}(n^2)$. Die Berechnung von $|\Delta(p_i, p_j)|$ wird in der Funktion `optimalesDreieck` durchgeführt. In dieser wird über alle Punkte aus P iteriert, womit die Funktion $\mathcal{O}(n)$ Zeit beansprucht. Der Dijkstra-Algorithmus ist in $\mathcal{O}(n \log n)$ ausführbar. Die Anweisungen aller weiteren Zeilen benötigen konstante Zeit. Durch die Anzahl an Schleifeniterationen erhalten wir Zeiten von $\mathcal{O}(n^2), \mathcal{O}(n^3)$ und $\mathcal{O}(n^3 \log n)$. Damit ist die Laufzeit von MaximaleDilation-Naiv abhängig von der Anzahl Punktpaare und deren jeweiligem Aufruf von `dijkstra`.

3.5.1 Theorem. *MaximaleDilation-Naiv berechnet in $\mathcal{O}(n^3 \log n)$ Zeit die maximale Dilation einer gerichteten Triangulierung.*

Speicher

Die Variablen in MaximaleDilation-Naiv und `optimalesDreieck` belegen konstanten Speicher. In `dijkstra` wird eine Prioritätswarteschlange und eine Liste der besuchten Knoten gespeichert, womit diese Funktion $\mathcal{O}(|P| + |E|) = \mathcal{O}(n)$ Speicher benötigt.

3.5.2 Theorem. *MaximaleDilation-Naiv berechnet mit $\mathcal{O}(n)$ Speicherplatz die maximale Dilation einer gerichteten Triangulierung.*

Implementierung

Die Implementierung verwendet die bereits beschriebenen Datenstrukturen *Punkt* für P und *Kante_gerichtet* für $\vec{E}$. Die Variable zur Speicherung der maximalen Dilation t ist vom Typ double.
Für die Berechnung der Länge des optimalen Zyklus $\Delta(p_i, p_j)$ wird eine benutzerdefinierte Methode `optimales_dreieck(Punkt` $p1$`, Punkt` $p2$`, std::vector<Punkt>` P`)` implementiert. Um *opt* $\leftarrow \infty$ zu realisieren, wird *opt* mit dem maximalen Wert seines Datentyps initialisiert. Da *opt* vom Typ double ist, wird *opt* auf `std::numeric_limits<double>::max()` gesetzt, wobei `std::numeric_limits` eine Vorlagenspezialisierung ist, die Informationen über die Eigenschaften der arithmetischen Typen bereitstellt. Die statische Funktion `max()`

Algorithmus	$n = 2^5$	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Orientieren2	43248	134832	466496	1718528	6746952	25748296	103140344	407624736
Änderungsrate		$3,12$	$3,46$	$3,68$	$3,92$	$3,82$	$4,01$	$3,95$

Tabelle 3.19: Speicherbedarf von KonsistentOrientieren-Nachbarschaftsmatrix in Bytes und Änderungsraten für $\frac{2n}{n}$

	$n = 2^5$	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Orientieren2	29144	51744	95864	187184	377344	1072104	2803224	5608904
Änderungsrate		$1,78$	$1,85$	$1,95$	$2,02$	$2,84$	$2,61$	2

Tabelle 3.20: Speicherbedarf von KonsistentOrientieren-Adjazenzlisten in Bytes und Änderungsraten für $\frac{2n}{n}$

gibt den größten positiven endlichen Wert zurück, den der Typ darstellen kann. Um sicherzustellen, dass $p \neq p_i$ **and** $p \neq p_j$ wird die Methode `punkte_gleich(Punkt` $p1$`, Punkt` $p2$`)` verwendet, die **true** zurückgibt, wenn $p1$ und $p2$ dieselben x- und y-Koordinaten haben. Für die Berechnung der Distanz zweier Punkte wird `distanz_punkte(Punkt` p_1`, Punkt` $p2$ genutzt, welche die euklidische Distanz zwischen $p1$ und $p2$ zurückgibt. Damit *opt* auf das Minimum von $\{opt, |(p_i, p_j)| + |(p_i, p)| + |(p_j, p)|\}$ gesetzt wird, werden dann die drei euklidischen Abstände mit `distanz_punkte` berechnet und mit einer if-Bedingung überprüft, ob ihre Summe kleiner ist als der Wert von *opt*. Wenn ja, dann wird *opt* auf $|(p_i, p_j)| + |(p_i, p)| + |(p_j, p)|$ gesetzt.
Für `dijkstra` wurde die $\mathcal{O}(n^2)$ Implementierung von www.geeksforgeeks.org verwendet. Da diese als Eingabe eine $n \times n$-Matrix mit allen Distanzen zwischen direkten Nachbarn als Eingabe erhält ergibt sich in dieser Implementierung eine Speicherkomplexität von $\mathcal{O}(n^2)$. Ähnlich wird verfahren um t auf das Maximum von $\{t, \frac{|C_{\vec{T}}(p_i,p_j)|}{|\Delta(p_i,p_j)|}\}$ zu setzen. Hier wird in einer if-Bedingung überprüft ob $t < \frac{|C_{\vec{T}}(p_i,p_j)|}{|\Delta(p_i,p_j)|}$. Ist diese Bedingung **true**, dann wird t auf $\frac{|C_{\vec{T}}(p_i,p_j)|}{|\Delta(p_i,p_j)|}$ gesetzt.

3.5.2 Maximale Dilation mit Dijkstra-Matrix bestimmen

Bei MaximaleDilation-Naiv wird die Laufzeit davon bestimmt, dass für jedes Paar distinkter Punkte $p_i, p_j \in P$ `dijkstra` aufgerufen wird, um den kürzesten Weg zwischen p_i und p_j zu berechnen. Ein anderer Ansatz besteht darin, alle kürzesten Pfade im Voraus zu bestimmen und diese in einer $n \times n$-Matrix vorzuspeichern. So genügt es später, die benötigten Werte aus der Matrix auszulesen. Ansonsten ist die Vorgehensweise dieselbe wie bei MaximaleDilation-Naiv.

Laufzeit

Wie im vorgerigen Abschnitt beschrieben benötigt MaximaleDilation-Naiv $\mathcal{O}(n^3 \log n)$ Zeit. Dies hing von der Anzahl Punktpaare und deren jeweiligem Aufruf von `dijkstra` ab. In MaximaleDilation-DijkstraMatrix wird `dijkstra` nicht mehr für jedes Punktpaar $p_i, p_j \in P$ ausgeführt. Dafür einmal für jeden Punkt, wobei die Distanzen von jedem der n Punkte zu jedem anderen Punkt in einer zweidimensionalen Liste gspeichert wer-

Eingabe: konsistent orientierte Triangulation $\vec{T} = (P, \vec{E})$
Ausgabe: maximale orientierte Dilation t

$t \leftarrow 1$
$Dist \leftarrow$ leere $n \times n$-Matrix // $Dist[1,2] =$ *kürzester Weg von* p_1 *nach* p_2 *in* $\vec{T}$
for all $p \in P$ **do**
  $Dist \leftarrow Dist \cup \{$`dijkstra(p)`$\}$ // ***dijkstra(p)*** *gibt eine Liste mit n Elementen, den Distanzen von p zu allen anderen Punkten, zurück*
end for
for $i = 1$ **to** $i < |P| - 1$ **do**
  $p_i \leftarrow P[i]$
  for $j = i + 1$ **to** $j < |P|$ **do**
    $p_j \leftarrow P[j]$
    $|\Delta(p_i, p_j)| \leftarrow$ `optimalesDreieck`
    $|C_{\vec{T}}(p_i, p_j)| \leftarrow distanzen[i][j] + distanzen[j][i]$
    $t \leftarrow \max\{t, \frac{|C_{\vec{T}}(p_i,p_j)|}{|\Delta(p_i,p_j)|}\}$
  end for
end for
return t

Algorithmus 3.17: MaximaleDilation-DijkstraMatrix

den. Bei n Iterationen ergibt sich eine Laufzeit von $\mathcal{O}(n^2 \log n)$. Bei n^2 Aufrufen von `optimalesDreieck` mit Zeitaufwand $\mathcal{O}(n)$ ergibt sich folgende Zeitkomplexität:

3.5.3 Theorem. *MaximaleDilation-DijkstraMatrix berechnet in* $\mathcal{O}(n^3)$ *Zeit die maximale Dilation einer gerichteten Triangulierung.*

Speicher

Die $n \times n$-Matrix $Dist$ benötigt $\mathcal{O}(n^2)$ Speicher. Ansonsten entspricht die Speicherkomplexität der von MaximaleDilation-Naiv.

3.5.4 Theorem. *MaximaleDilation-DijkstraMatrix berechnet mit* $\mathcal{O}(n^2)$ *Speicherplatz die maximale Dilation einer gerichteten Triangulierung.*

Implementierung

Der hier vorgestellte MaximaleDilation-DijkstraMatrix unterscheidet sich von Algorithmus 3.15 MaximaleDilation-Naiv hauptsächlich in der Vorspeicherung der Distanzen aller Punktpaare $p_i, p_j \in P$ in einer $x \times n$-Matrix $Dist$. Diese wird als std::vector<std::vector< double>> implementiert. Dann wird eine for-Schleife für jeden Punkt $p \in P$ durchlaufen, wobei für jedes p der Dijkstra-Algorithmus ausgeführt wird, welcher hier jeweils eine Liste std::vector<double> mit den kürzesten Wegen von p zu jedem Punkt $p' \in P \setminus \{p\}$ zurückgibt. Die zurückgegebenen Listen werden mit `push_back` in $Dist$ gespeichert. Um den kürzesten Zyklus $C_{\vec{T}}(p_i, p_j)$ zu erhalten, muss $Dist[i][j] + Dist[j][i]$ berechnet werden.

3.5.3 Auswertung

Im Folgenden werden die experimentellen Ergebnisse bezüglich Laufzeit und Speicherverbrauch der Algorithmen MaximaleDilation-Naiv aus Abschnitt 3.5.1 und MaximaleDilation-DijkstraMatrix aus Abschnitt 3.5.2 vorgestellt. Hierfür wurden die Algorithmen mit Punktmengen P von wachsender Kardinalität $|P| = n$ ausgeführt. Für jedes n wurden dabei 100 zufällige, konvexe Punktmengen durch ErzeugeKonvex aus Abschnitt 3.1.1 erzeugt, mit Gierig-Naiv aus Abschnitt 3.2.1 trianguliert und mit KonsistentOrientieren-Adjazenzlisten aus Abschnitt 3.4.3 konsistent orientiert. Aufgrund von Zeitbeschränkungen wurden die Benchmark-Durchläufe jeweils vor Überschreitung von 1000 Sekunden abgebrochen. Dadurch konnten für MaximaleDilation-Naiv bis zu 2^8 Punkte und für MaximaleDilation-DijkstraMatrix bis zu 2^{10} Punkte betrachtet werden.

Laufzeiten

Die experimentellen Ergebnisse für die Zeitkomplexität von MaximaleDilation-Naiv und MaximaleDilation-DijkstraMatrix sind in Abb. 3.9 graphisch dargestellt. Die exakten Werte sind in Tabelle 3.21 und Tabelle 3.22 aufgelistet. Abzulesen ist, dass die Laufzeit von MaximaleDilation-Naiv bei gleicher Eingabe n um einen Faktor von $2,1$ schneller wächst als die Laufzeit von MaximaleDilation-DijkstraMatrix.
Algorithmus MaximaleDilation-Naiv hat nach Theorem 3.5.1 eine Zeitkomplexität von $\mathcal{O}(n^3 \log n)$. Damit erhöht sich die Laufzeit für $2n$ Punkte relativ zu n Punkten um mehr als das Achtfache und weniger als das 16-fache. Die Änderungsraten der Laufzeiten sind in Tabelle 3.21 angegeben. Hieraus folgt für die Verdopplungen von n auf $2n$ eine im Schnitt $12,38$-mal so hohe Laufzeit. Die Zeit liegt damit im Bereich der theoretisch bestimmten Laufzeit.
Nach Theorem 3.5.3 hat MaximaleDilation-DijkstraMatrix eine Laufzeit von $\mathcal{O}(n^3)$, wonach sich die benötigte Zeit bei einer Verdopplung von n verachtfacht. Nach Tabelle 3.22 braucht MaximaleDilation-DijkstraMatrix für $2n$ im Schnitt $5,85$ mal so lang wie für n Punkte. Diese Ergebnisse sind unter dem erwarteten Wert, nähern sich diesem aber an und entsprechen $\mathcal{O}(n^3)$ näherungsweise für $n \geq 2^7$.

Algorithmus	$n = 2^3$	2^4	2^5	2^6	2^7	2^8
MaximaleDilation-Naiv	$0,001$	$0,016167$	$0,11749$	$1,28097$	$17,0893$	$243,225$
Änderungsrate		$16,17$	$7,27$	$10,9$	$13,34$	$14,23$

Tabelle 3.21: Laufzeiten von MaximaleDilation-Naiv in Sekunden und Änderungsraten für $\frac{2n}{n}$

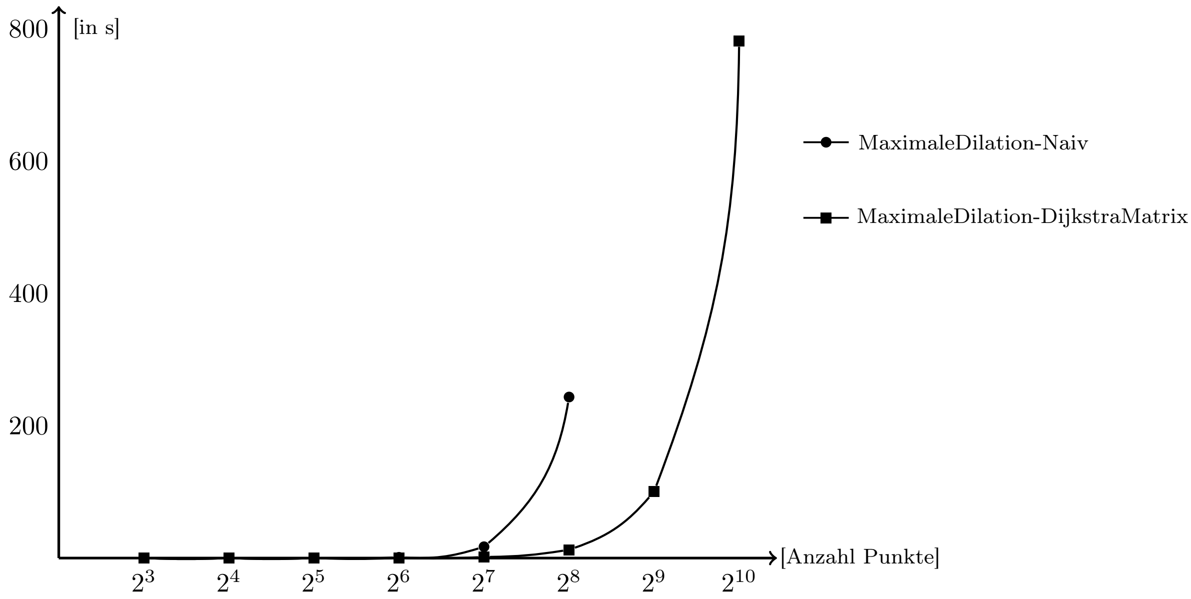

Abbildung 3.9: Laufzeiten von MaximaleDilation-Naiv & MaximaleDilation-DijkstraMatrix. Die x-Achse entspricht der Anzahl Punkte n aus P und die y-Achse der maximal benötigten Zeit in Sekunden in Abhängigkeit von n, um die maximale Dilation zu bestimmen.

$n = 2^3$	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
$0,011948$	$0,016647$	$0,033422$	$0,211778$	$1,61224$	$12,7128$	$100,053$	$781,6$
	$1,39$	$2,01$	$6,34$	$7,61$	$7,89$	$7,87$	$7,82$

Tabelle 3.22: Laufzeiten von MaximaleDilation-DijkstraMatrix in Sekunden und Änderungsraten für $\frac{2n}{n}$

Speicher

Der Speicherbedarf ist analog zur Laufzeit graphisch in Abb. 3.10 und die exakten Werte für MaximaleDilation-Naiv und MaximaleDilation-DijkstraMatrix in den Tabellen 3.23 und 3.24 dargestellt. Abzulesen ist, dass MaximaleDilation-DijkstraMatrix einen um einen Faktor von $1,09$ höheren Speicherbedarf aufweist als MaximaleDilation-Naiv. Dieses Resultat entspricht den theoretischen Ergebnissen der Theoreme 3.5.1 und 3.5.4.
Hiernach hat MaximaleDilation-Naiv eine Speicherkomplexität von $\mathcal{O}(n)$. Nach den experimentellen Werten aus Tabelle 3.23 erhöhen sich die Werte bei einer Verdopplung von n auf $2n$ Punkten im Schnitt um das $2,99$-fache und näheren sich dabei mit steigendem n einer Vervierfachung an. Dies liegt über der erwarteten Verdopplung der Laufzeit bei Verdopplung von n, was an der verwendeten Implementierung von `dijkstra` liegt. Diese erhält als Eingabe eine $n \times n$-Matrix mit den Distanzen zwischen allen direkten Nachbarn und führt zu dem erwarteten quadratischen Speicheraufwand, der hier experimentell bestätigt wird.

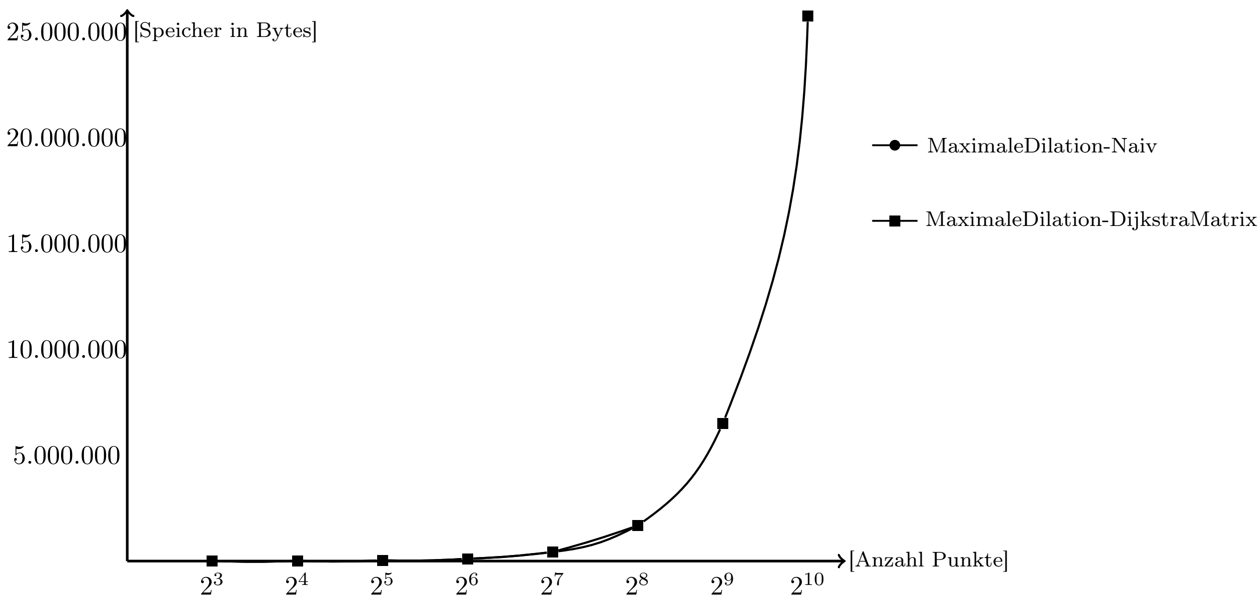

Abbildung 3.10: Speicherbedarf von MaximaleDilation-Naiv & MaximaleDilation-DijkstraMatrix. Die x-Achse entspricht der Anzahl Punkte n aus P. Die y-Achse beschreibt dazu den maximal benötigten Speicher in Bytes in Abhängigkeit von n.

Der Algorithmus MaximaleDilation-DijkstraMatrix benötigt nach Theorem 3.5.4 $\mathcal{O}(n^2)$ Speicher, was einer Vervierfachung des belegten Speichers bei Verdopplung von n entspricht. Experimentell belegt MaximaleDilation-DijkstraMatrix im Schnitt $3,25$-fachen Speicher bei Verdopplung von n auf $2n$. Die Wachstumsrate nähert sich dabei dem theoretischen Wert immer näher an und bestätigt diesen experimentell.

Algorithmus	$n = 2^3$	2^4	2^5	2^6	2^7	2^8
MaximaleDilation-Naiv	9244	13008	38128	124752	124752	1677568
Änderungsrate		$1,41$	$2,93$	$3,27$	$3,58$	$3,76$

Tabelle 3.23: Speicherbedarf von MaximaleDilation-Naiv in Bytes und Änderungsraten für $\frac{2n}{n}$

$n = 2^3$	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
9324	13168	38128	124592	446016	1678048	6500928	25748984
	$1,41$	$2,9$	$3,27$	$3,58$	$3,76$	$3,87$	$3,96$

Tabelle 3.24: Speicherbedarf von MaximaleDilation-DijkstraMatrix in Bytes und Änderungsraten für $\frac{2n}{n}$