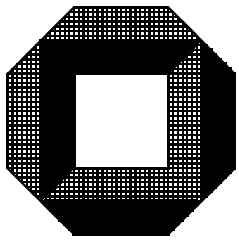
Schnelle Berechnung von kürzesten Wegen in Graphen unter Benutzung höherdimensionaler Layouts



Studienarbeit am Institut für Logik, Komplexität und Deduktionssysteme
Lehrstuhl Prof Dr. Dorothea Wagner
Universität Karlsruhe (TH)
Fakultät für Informatik

von
Sebastian Knopp

Betreuer:
Thomas Willhalm

Zusammenfassung

Der bekannteste Algorithmus zur Lösung des Kürzeste-Wege-Problems in Graphen mit positiven Kantengewichten ist der Algorithmus von Dijkstra.Verschiedene Beschleunigungsmethoden benutzen in einen Vorverarbeitungsschritt berechnete Daten, um den Suchraum zu verkleinern und damit die Laufzeit des Algorithmus zu verbessern. Häufig wird zu diesem Zweck ein zweidimensionales Layout des Graphen verwendet. In dieser Arbeit wird die Erweiterbarkeit dieser Techniken auf multidimensionale Layouts untersucht und es werden Methoden zur Erzeugung solcher hochdimensionaler Zeichnungen erklärt.

Die vorgestellten Verfahren wurden implementiert und die damit gemessenen Ergebnisse werden verglichen. Das Verfahren der Reichweite verbessert sich bei einer Erhöhung der Anzahl der Dimensionen nur kaum. Nutzt man als Kantenbeschriftungen Bounding-Boxen, so kann der Suchraum durch weitere Dimensionen merklich verkleinert werden. Für das Verfahren der Regionenpartitionierung hat sich die Integration des Layoutverfahrens der hochdimensionalen Einbettung zur Unterteilungs des Graphen in Regionen als gute Lösung gezeigt. Mit dieser Methode, die auch den geringsten Vorverarbitungsaufwand erfordert, konnten die besten Ergebnisse erzielt werden.

Diese Studienarbeit entstand an der Universität Karlsruhe (TH) am Institut von Prof. Dr. D. Wagner. Bedanken möchte ich mich bei meinem Betreuer Thomas Willhalm für das interessante Thema und seine Unterstützung.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt habe und nur die angegebenen Hilfsmittel und Quellen verwendet wurden.

Karlsruhe, im Februar 2005

Inhaltsverzeichnis

1 Einleitung

Das Kürzeste-Wege-Problem (*Single Source Shortest Path Problem*, SSSP) ist sicher eines der meiststudiertesten Probleme der algorithmischen Graphentheorie. In dieser Arbeit geht es um Techniken zur Beschleunigung der Suche nach einem kürzesten Weg von A nach B in sehr großen, dünnen Graphen. Eine Anwendung ist beispielsweise ein zentraler Server, der pro Sekunde sehr viele Anfragen nach kürzesten Routen bearbeiten muss. Der Kern dieser Verfahren ist der Algorithmus von Dijkstra, ein Greedy-Algorithmus, der in Graphen mit positiven Kantengewichten das Problem löst. Die hier betrachteten Graphen sind statisch. Das lässt eine umfangreiche Vorberechnung zu.

Da Dijkstras Algorithmus alle Knoten betrachtet, die näher am Startknoten liegen als der Zielknoten, ist der Suchraum dabei sehr groß. Verschiedene bekannte Verfahren erzielen eine Beschleunigung mit einer Verkleinerung des Suchraumes. *Bidirektionale Suche* führt dazu gleichzeitig zur normalen Suche eine Rückwärtssuche vom Zielknoten aus, die abgebrochen wird, sobald sich die Suchhorizonte treffen. *Zielgerichtete Suche* (A* Algortihmus) leitet durch temporär modifizierte Kantengewichte die Suche in Richtung des Zielknotens. Ein *Multilevel-Ansatz* nutzt verschiedene Vergröberungsstufen zur Verkleinerung des Suchraums. Die Techniken, die hier betrachtet werden sollen, nutzen ein Layout des Graphen. Ein solches ordnet den Knoten des Graphen Koordinaten zu.

Liegt dem Graphen eine Straßenkarte zugrunde, so können solche Zeichnungen beispielsweise durch die geographische Lage der Knoten gegeben sein. Nicht immer jedoch ist ein Layout im Vorhinein gegeben. In solchen Fällen können Methoden des Graphenzeichnens eingesetzt werden, um für die Beschleunigung ein Layout zur Verfügung zu stellen. In [WW05] wird gezeigt, dass solche automatisch generierten Layouts für diesen Zweck sogar besser sein können. Aktuelle Beschleunigungstechniken nutzen meist zwei-, höchstens aber dreidimensionale Layouts. Diese Arbeit soll die Erweiterbarkeit auf multidimensionale Layouts untersuchen. Da hier hauptsächlich solche Graphen betrachtet werden, bei denen Knoten einen Ort auf einer Landkarte repräsentieren, stehen mehrdimensionale Daten nicht auf natürliche Weise zur Verfügung. Daher ist der erste Schritt die Erzeugung einer höherdimensionalen Zeichnung. Dazu wird hier im Wesentlichen die in Kapitel 3 beschriebene hochdimensionale Einbettung verwendet.

Die Ausarbeitung erklärt die folgenden Beschleunigungstechniken und deren Erweiterung auf mehrere Dimensionen. Zum einen das von Gutmann in [Gut04] vorgestellte Verfahren, das einen formalen Parameter zur Klassifikation von Knoten bezüglich ihrer Wichtigkeit für Fernstrecken einführt, zum anderen zwei verschiedene Kantenbeschriftungsverfahren: Bounding-Box und Graphenpartitionierung. Die zweite Technik, bei der ein Graph in Regionen eingeteilt wird, kann unmittelbar mit der Layoutgenerierung verknüpft werden.

2 Definitionen

Sei ein Graph

$$G = (V, E, l)$$

gegeben, wobei V die Menge der Knoten, E die Menge der Kanten und l eine Funktion

$$l : E \to \mathbb{R}_0^+$$

bezeichnet (Kantengewichte). Weiter seien $n := \#V$ und $m := \#E$ die Anzahl der Knoten bzw. Kanten. Ein Weg P_{st} von s nach t in einem Graph ist eine Folge von Knoten $v_1, v_2 \ldots, v_p$ mit $(v_i, v_{i+1}) \in E$, $v_1 = s$ und $v_p = t$. Die Länge eines solchen Weges ist

$$l(P) = \sum_{i=1}^{p-1} l(v_i, v_{i+1})$$

Man definiert die *Länge eines kürzesten Weges* als

$$\delta(s,t) = \begin{cases} \min\{l(P_{st})\} & \text{falls ein Weg von } s \text{ nach } t \text{ existiert} \\ \infty & \text{sonst} \end{cases}$$

Ein kürzester Weg P_{st} von s nach t ist ein Weg Länge $l(P) = \delta(s,t)$.

Ist die Anzahl der Kanten $m \in O(n)$, so bezeichnet man den Graphen als *dünn*. Ein Graph heißt *groß*, wenn der verfügbare Speicherplatz nur einen Speicherverbrauch von $O(n)$ zulässt.
Ein d-dimensionales Layout ($d \in \mathbb{N}_{>0}$) ist eine Funktion

$$F : V \to \mathbb{R}^d$$

die den Knoten des Graphen Positionen in $\mathbb{R}^d$ zuordnet.

3 Layoutgenerierung

Die in Kapitel 4 vorgestellten Beschleunigungstechniken benötigen ein Layout des Graphen. Zu Straßenkarten sind oftmals zweidimensionale Layouts bereits vorgegeben, bei Graphen anderer Art kann es sein, dass kein Layout vorgliegt. Um trotzdem eine solche Zeichnung des Graphen zur Beschleunigung der kürzesten-Wege-Suche verwenden zu können, hat es sich als Nützlich erwiesen, dazu automatisch generierte Layouts zu verwenden. Dies kann in einigen Fällen, wie in [WWZ04] gezeigt wurde, sogar besser als die Verwendung von durch natürliche Daten gegebenen Layouts sein.

In dieser Arbeit soll die Erweiterung von Beschleunigungstechniken auf multidimensionale Layouts untersucht werden. Solche Zeichnungen stehen jedoch meist nicht zur Verfügung. Die Knoten einer Straßenkarte beispielsweise sind üblicherweise nur als zweidimensionale Punkte gegeben. Um also die erweiterten Beschleunigungsverfahren nutzen zu können, müssen zuvor höherdimensionale Layouts erzeugt werden.

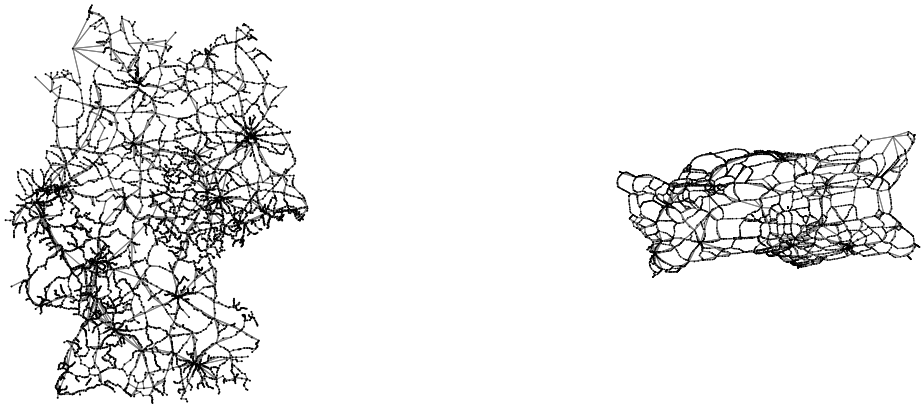


Abbildung 1: Originalkarte und eine aus dem gleichen Graphen durch Hochdimensionale Einbettung erzeugte Zeichnung

3.1 Hochdimensionale Einbettung

Die Idee dieses Verfahrens zur Erzeugung eines multidimensionalen Layouts besteht aus zwei Schritten. Zuerst wird ein hochdimensionales Layout erzeugt. Dieses wird dann durch eine Hauptkomponentenanalyse auf die gewünschte Dimensionsanzahl reduziert.

3.1.1 Hochdimensionale Zeichnung

In diesem Schritt soll ein hochdimensionales Layout des Graphen erzeugt werden. Um ein solches Layout mit h Dimensionen (z.B. $h = 50$) zu erzeugen, werden im hier vorgestellten Verfahren h sogenannte Pivot-Knoten $P_1, \ldots, P_h$ ausgewählt. Von jedem dieser Knoten aus wird ein vollständiger Kürzeste-Wege-Baum im Graphen aufgespannt. Dabei werden alle graphentheoretischen Entfernungen der Pivot-Knoten zu allen Knoten im Graphen bestimmt. Die i-te Koordinate im Layout eines Knotens wird durch ihren Abstand zum Pivotknoten P_i festgelegt. Damit liegt P_i in der i-ten Dimension bei 0, andere Knoten K liegen entsprechend ihres Abstandes zu P_i.

Mit einer solchen Betrachtung des Graphen von verschiedenen Punkten aus möchte man eine aussagekräftige Repräsentation seiner Struktur erhalten. Dazu muss eine geschickte Auswahl der Pivotknoten getroffen werden. Die Pivotknoten werden sukzessive bestimmt. Der erste P_1 wird zufällig ausgewählt. Für $j = 2, \ldots, h$ wird der Knoten hinzugefügt, dessen Summe der Abstände zu den bereits festgelegten Pivotknoten $\{P_1, \cdots, P_{j-1}\}$ maximal ist. Für die Auwahl der Pivotknoten wäre auch die Verwendung anderer Ansätze denkbar.

3.1.2 Projektion in ein Layout

Nun soll das vorhandene hochdimensionale Layout auf eines mit weniger Dimensionen reduziert werden. Im Originalpapier werden nur zwei- oder dreidimensionale Zeichnungen erzeugt, jedoch ist das Verfahren ohne weiteres auf mehr Dimensionen erweiterbar. Die für die Reduktion benutzte Methode ist die aus der multivarianten Statistik bekannte Hauptkomponentenanalyse. Diese transformiert eine Anzahl von (möglicherweise korrelierten)

	50	100	150	200	250	300
1	6616	6502	6559	6501	6904	6576
2	2942	2935	2988	2963	2898	2942
3	1700	1728	1664	1641	1602	1603
4	1584	1488	1418	1475	1416	1416
5	1458	1390	1322	1354	1273	1323
6	1362	1335	1301	1267	1173	1230
7	1276	1298	1185	1186	1100	1133
8	1234	1267	1185	1163	1087	1100
9	1208	1250	1168	1152	1034	1053
10	1170	1202	1122	1139	1031	1110

Tabelle 1: Straßengraph mit 44439 Knoten, 96994 Kanten, Beschleunigungsverfahren: Geometrische Container, Größe des Suchraums (in Anzahl besuchter Knoten), Ausgangsdimension / Zieldimension

Variablen in eine kleinere Anzahl von unkorrelierten Variablen, welche als Hauptkomponenten bezeichnet werden. Dabei besitzt die erste Komponente die größte Varianz. Die Varianz folgender Komponenten ist größer als die der Vorhergehenden. Für ein k-dimensionales Layout werden die ersten k Komponenten der Hauptkomponentenanalyse verwendet.

Den n Knoten des Graphen sind im ersten Schritt des Algorithmus h-dimensionale Koordinaten zugeordnet worden. Diese werden zunächst in jeder Dimension $i = 1, \ldots, h$ um Null zentriert, so dass $\sum_{j=1,\ldots,n} x_{ij} = 0$ ist. Anschliessend wird die durch $s_{ij} = \langle x_i, x_j \rangle$ definierte Kovarianzmatrix $S \in \mathbb{R}^h$ berechnet. Dann werden die Eigenvektoren E_i zu den h größten Eigenwerte e_i der Kovarianzmatrix bestimmt und so bezeichnet, dass $e_1 \geq e_2 \geq \cdots \geq e_h$. Die neuen Knotenkoordinaten sind nun durch Linearkombinationen der Form $\sum_{i=0,\ldots,h} E_{ji} x_i$ bestimmt.

Anzahl der Dimensionen In [HK02] stellen Harel und Koren fest, dass eine Ausgangsdimension von $h = 50$ für zwei- und dreidimensionale Layouts ausreichend ist. Es stellt sich die Frage, ob diese Zahl auch für höherdimensionale Layouts genügt. Die Qualität eines Layouts ist bei der im Kontext dieser Arbeit betrachteten Problemstellung durch die damit erreichbare Beschleunigung der kürzesten-Wege-Suche gekennzeichnet. Um dies zu testen, wurde hier das im späteren Kapitel 4.2.1 vorgestellte Beschleunigungsverfahren mit geometrischen Containern benutzt. Gemessen wurde dabei die Größe des Suchraumes, also die Anzahl der betrachteten Knoten. Angewendet wurde der Algorithmus auf verschiedene generierte Layouts mit 1-10 Dimensionen. Dafür wurden zugrundeliegende hochdimensionale Layouts mit 50 - 300 Dimensionen getestet. Die Ergebnisse finden sich in Tabelle 1 aufgelistet.

Es zeigt sich, dass, auch für Layouts mit mehr als zwei Dimensionen, die von Harel und Kohren benutzten 50 Dimensionen der hochdimensionalen Zeichnung ausreichend sind. Da

der quadratisch in die Laufzeit $O(h \cdot n + h^2 \cdot n)$ der Generierung des Layouts eingehende Faktor h nicht größer werden muss, bleibt auch im multidimensionalen Fall die Erzeugung des Layouts effizient.

Ausblick Im Kapitel 4.2.2 wird ein Beschleunigungsverfahren für die Kürzeste-Wege-Suche vorgestellt, das eine Einteilung der Knoten in Regionen vornimmt. Für die Erstellung dieser Unterteilung haben sich Methoden der Generierung eines Layouts durch hochdimensionale Einbettung als nützlich erwiesen und werden in diesem später folgenden Kapitel vorgestellt.

3.2 Springembedder

Eine zweite Methode zur Generierung von Layouts, die auf mehrere Dimensionen erweitert werden kann, ist das auf einer Idee von Eades [Ead84] basierende Springembedderverfahren. Dies nutzt eine Analogie aus der Physik bei der man sich die Knoten des Graphen als Stahlringe und die Kanten des Graphen als Federn vorstellt. Ausgehend von einer Startkonfiguration lässt man in einer Simulation verschiedene Kräfte wirken. Das Verfahren wird abgebrochen, wenn eine feste Anzahl von Iterationen überschritten wurde oder das System sich stabilisiert hat.

Es gibt verschiedene Ansätze für die auftretenden Kräfte. Beim hier betrachteten an Kamada und Kawai [KK89] angelehnten Ansatz entsprechen die verwendeten Federhärten graphentheoretischen Abständen. Es werden dabei aus Effizienzgründen nur die nähesten 30 Knoten betrachtet.

Zur Beschleunigung des Verfahren kann eine Multilevel-Technik eingesetzt werden. Der nächstgröbere Graph entsteht dabei durch eine Verschmelzung der Knoten eines maximalen Matchings. Die Anpassung eines zwei- bzw. dreidimensionalen Springembedders kann ohne Änderungen im eigentlichen Algorithmus erfolgen. Es müssen lediglich die Punkte und Vektoren mit ihren Operationen auf höhere Dimensionen angepasst werden.

4 Berechnung von kürzesten Wegen

Die in hier vorgestellten Kürzeste-Wege-Algorithmen basieren auf dem Algorithmus von Dijkstra. Dieser funktioniert wie im Folgenden beschrieben: Man definiert für jeden Knoten v seine tentative Distanz $dist(v)$, seinen Wegvorgänger $pred(v)$ und seinen Status $status(v)$, welcher die Werte {nicht-besucht, markiert, besucht} annehmen kann. Initial setzt man für alle Knoten $dist(v) = \infty$, $pred(v) = null$ und $status(v)$=nicht-besucht, bis auf den Startknoten s, welcher den Status markiert und die tentative Distanz Null bekommt.

Die Knoten mit dem Status markiert werden in einer Priority-Queue Q verwaltet. Die Schlüsselwerte für Q bilden die aktuellen tentativen Distanzen der Knoten. Solange die Priority-Queue noch Knoten enthält, entnimmt der Algorithmus einen Knoten mit minimaler tentativer Distanz, "scannt" ihn und setzt anschließend seinen Status auf besucht. "Scannen" eines Knotens v bedeutet, für alle Kanten $(v, u) \in E$ wird überprüft, ob der

Weg zu u über v und (v, u) kürzer ist, als der bisher gefundene Weg $dist(u)$. Falls ja, wird $dist(u)$ auf $dist(v) + l(v, u)$ aktualisiert, v als Vorgänger von u eingetragen und der Status von u auf markiert gesetzt.

Besitzt ein Knoten den Status besucht, kann durch "Scannen" seine tentative Distanz nicht weiter verringert werden und somit ist ein kürzester Weg zu diesem Knoten gefunden. Daher kann der Algorithmus nach der Entnahme des Zielknotens aus der Priority-Queue abgebrochen werden. Ein Knoten hat genau dann den Status markiert, wenn er sich noch in der Priority-Queue befindet. Daher kann der Status bei der Implementierung durch diesen Test ersetzt werden. Verwendet man einen Fibonacci Heap als Priority-Queue, so liegt die Worst-Case Laufzeit in $O(m + n \log n)$. Der folgende Pseudocode beschreibt Dijkstras Algorithmus:

```
1  procedure DIJKSTRA(G = (V, E, l), s)
2    begin
3      distance(s) := 0
4      pqueue.insert(s)
5      while not pqueue.empty do begin
6        v := pqueue.extractMin
7        forall_out_edges(v, u) do begin
8          if d(u) ≤ d(v) + l(v, u) then continue
9          d(u) := d(v) + l(v, u)
10         if u ∉ pqueue then
11           pqueue.insert(u)
12         else
13           pqueue.decreaseKey(u)
14       end
15     end
16   end
```

Der Suchraum besteht aus allen Knoten, die in die Priority-Queue eingefügt werden. Bricht man den Algorithmus ab sobald der Zielknoten den Status besucht hat, so sind das alle Knoten, die näher am Startknoten liegen als der Zielknoten. Der Suchraum des unveränderten Algorithmus von Dijkstra ist damit sehr groß. Es werde viele Knoten betrachtet, die offensichtlich nicht Teil des kürzesten Weges sein können, beispielsweise solche, die zum Zielknoten in entgegengesetzter Richtung liegen. Die im Folgenden beschriebenen Techniken beschleunigen den Algorithmus von Dijkstra durch eine Reduktion der Größe des Suchraums. Um dies zu erreichen werden Kanten und Knoten ignoriert, von denen man ausschliessen kann, dass sie auf dem kürzesten s-t Weg liegen. Während des "Scannens" eines Knotens werden die ausgehenden Kanten daraufhin untersucht. Im obigen Pseudocode erfolgt der Test in der Funktion *processEdge*, die an folgender Stelle eingefügt wird:

```
7        forall_out_edges(v, u) do begin
8  →       if processEdge(v, u) then continue
```

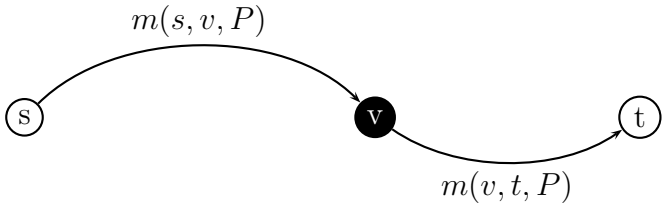


Abbildung 2: Die Reichweite eines Knotens v in einem Pfad ist $\min\{m(s, v, P), m(v, t, P)\}$. Die Reichweite eines Knotens v ist das Maximum der Reichweiten von u in allen kürzesten Wegen, auf denen u liegt.

Den folgenden Beschleunigungstechniken geht ein aufwendiger Vorverarbeitungsgsschritt voraus, in dem Vorberechnungen für den Ausschluss von Kanten angestellt werden. Die darin gewonnenen Informationen dürfen nicht mehr als O(m) Speicherplatz verbrauchen. Man kann die Einschränkung des Suchraums als Suche in einem kleineren Teilgraphen $G' \subseteq G$ betrachten, die garantiert den kürzesten s-t-Weg zurückliefert.

4.1 Reichweite

Eine häufige Anwendung für die Suche nach kürzesten Wegen ist die Suche nach Routen in Straßenkarten. Kommerzielle Anwendungen nutzen zu diesem Zweck die gegebene hierarchische Struktur des Verkehrsnetzwerks. In diesem sind verschiedene Straßen nach ihrer Bedeutung für Fernstrecken klassifiziert. Kreis-, Land-, Bundesstraßen und Autobahnen sind für Fernreisen von unterschiedlicher Wichtigkeit. Eine Verkleinerung des Suchraums kann erreicht werden, indem Straßen mit geringer Wichtigkeit nicht berücksichtigt werden, wenn sie weit von Start- und Zielknoten entfernt liegen. Verfahren dieser Art können allerdings nicht garantieren, dass immer ein kürzester Weg gefunden wird. Im schlechtesten Fall wird kein Weg gefunden, obwohl es einen gibt.

Das in [Gut04] beschriebene Verfahren benutzt diese Idee. Jedoch wird nicht eine vorgegebene Hierarchie des Graphen benutzt, sondern ein formelles Attribut für jeden Knoten definiert. Dieses wird als *Reichweite*[1] bezeichnet. Darin ist die Länge der kürzesten Wege, auf denen ein Knoten liegt, codiert. Für einen hohen Reach-Wert eines Knotens muss ein kürzester Weg durch diesen verlaufen, der sich von dort aus weit in beide Richtungen erstreckt.

Definitionen Vorausgesetzt wird, dass ein Layout für den Graphen zur Verfügung steht. Mit diesem können unmittelbar euklidische Distanzen

$$d : (V \times V) \to \mathbb{R}_0^+$$

bestimmt werden. Eine Metrik der Kanten

$$m : E \to \mathbb{R}_0^+$$

[1] Englisch: Reach

die durch die Bedingung

$$(u, v) \in E \Rightarrow m(u, v) \geq d(u, v)$$

konsistent zu den euklidischen Distanzen der Knoten ist, heißt *Reach*-Metrik. Für einen Weg P beschreibt $m(P)$ die Summe der $m(e)$ aller Kanten e des Weges. Für einen Weg P, $u, v \in E$ sei $m(u, v, P) := m(Q)$, wobei Q der von u zu v führende Teilpfad von P ist.

Die *Reichweite* eines Knotens v auf einem Weg P ist definiert als

$$r(v, P) := \min\{m(s, v, P), m(v, t, P)\}.$$

Die *Reichweite* eines Knotens v ist das Maximum der *Reichweiten* $r(v, Q)$ über alle kürzesten Wege Q des Graphen, die v enthalten. Es ist zu beachten, dass hier weiterhin kürzeste Wege bezüglich der gegebenen Kantengewichte des Graphen gemeint sind.

Es stellt sich nun die Frage, was für eine Reach-Metrik benutzt wird. Eine mögliche Metrik ist die euklidische Distanz selbst. Die hier verwendete Metrik benötigt zunächst die Definition der *Höchstgeschwindigkeit* $\mathrm{v}_{\max} := \max\{\frac{d(v,u)}{l(v,u)} : (v, u) \in E\}$. Mit dieser erfüllt $m(e) = \frac{l(e)}{\mathrm{v}_{\max}}$ $(e \in E)$ die Bedingungen einer Reach-Metrik.

Algorithmus Bevor konkrete Suchanfragen gestellt werden, wird in einem Vorverarbeitungsschritt für jeden Knoten des Graphen dessen Reichweite bestimmt und gespeichert. Bei der Suche nach einem kürzesten s-t Pfad kann ein Knoten $v \in V$ ignoriert werden, wenn $m(P)$ größer ist als die Reichweite $r(v)$ und die euklidische Distanz $d(v, t)$ größer ist als die Reichweite des Knotens v. Dabei ist P der berechnete Weg von s nach v vor dem Einfügen von v in die Priority-Queue. Die Funktion *processEdge*(u, v) kann also in folgender Weise definiert werden:

$$processEdge(v, u) := \begin{cases} wahr & r(v) \geq m(P) \text{ oder } r(v) \geq d(v, t) \\ falsch & \text{sonst} \end{cases}$$

In einem Vorverarbeitungsschritt müssen die Reichweiten der Knoten des Graphen berechnet werden. Die dazu notwendige Ermittlung der Reichweite $r(v, P)$ eines Knotens v auf einem Pfad P für jeden Knoten und jeden Pfad verursacht bei dünnen Graphen einen Aufwand von $\mathrm{O}(n^3)$. Wenn als Reach-Metrik die oben beschriebene Metrik $m(e) = \frac{l(e)}{\mathrm{v}_{\max}}$ benutzt wird, kann durch die Verwendung einer modifizierten Tiefensuche die Vorberechnungszeit auf $\mathrm{O}(n^2 \log\, n)$ reduziert werden. Von der Verwendung dieser Metrik wird im Folgenden ausgegangen.

Vorverarbeitung Um von der Menge $\{r(v, P) : P \text{ ist ein kürzester Weg}\}$ das Maximum zu bestimmen, setzt man zunächst die temporäre Reichweite $r_{\text{temp}}(v)$ auf Null und erhöht dann sukzessive für alle kürzesten Wege P den Wert r_{temp} auf $r(v, P)$, wenn $r(v, P) > r_{\text{temp}}(v)$. Um dies zu tun, erstellt man zu jedem Knoten einen Kürzeste-Wege-Suchbaum und aktualisiert die temporäre Reichweite für alle Knoten auf allen kürzesten Wegen in diesem Baum. Eine naive Implementierung würde $\mathrm{O}(n^2)$ Aktualisierungen benötigen. Beschleunigt werden kann dies mit dem folgenden

	#Knoten	Dijkstra	Paar	Summe	Reichweite
citymap 1	1429	726	677	684	316
citymap 2	2948	1502	1339	1338	468

Tabelle 2: Durchschnittliche Größe des Suchraums bei Varianten der Reichweite

Lemma (1). *Für zwei kürzeste Pfade P_{sx} und P_{sy} mit einem gemeinsamen Knoten v gilt:* $max\{r(v, P_{sx}), r(v, P_{sy}\} = min\{m(P_{sv}, max\{m(P_{vx}), m(P_{vy})\}\}$.

Die Distanzen $\delta(s, v)$ zum Startknoten s sind in einem Kürzeste Wege Baum bekannt. Mit einer Tiefensuche in diesem Baum sind auch die Maxima $\max_s(v) := \max\{m(v, x) : v \in P_{sx}, P_{sx}$ ist kürzester Weg$\}$ leicht zu bestimmen. Das Maximum aller pfadbezogenen Reichweiten über alle Pfade in diesem Baum des Knotens v ist dem obigen Lemma zufolge $\max\{\delta(s, v), \max_s(v)\}$, kann also noch im selben Durchlauf der Tiefensuche berechnet werden. Der Gesamtaufwand für die Berechnung der Reichweiten ist damit $O(n^2 \log n)$.

Bei sehr großen Graphen ist möglicherweise ein solcher Vorberechnungsaufwand nicht mehr akzeptabel. Da jedoch der Reach-Algorithmus für jede obere Schranke der Reichweiten korrekte kürzeste Wege findet, kann die Vorberechnung erheblich beschleunigt werden, wenn sie lediglich eine solche Schranke berechnet. Dabei wird allerdings ein größerer Suchraum und damit schlechtere Laufzeit in Kauf genommen werden. Wie eine solche obere Schranke gefunden werden kann wird in [Gut04] beschrieben.

Das Verfahren ist ohne weiteres durch Verwendung des euklidischen Abstandes für d-dimensionale Punkte auf höherdimensionale Layouts erweiterbar.

Varianten Es sind Varianten dieser Beschleunigungstechnik denkbar. Leicht veränderte Definitionen der pfadbezogenen Reichweiten $r(v, P)$ erfordern nur geringe Anpassung des Algorithmus. Die beschriebenen von der ursprünglichen abweichenden Versionen erwiesen sich in Experimenten als deutlich schlechter im Vergleich zur Originalversion. Die Ergebnisse dieser Tests, die dennoch hier kurz vorgestellt werden sollen, finden sich in Tabelle 4.1 wieder. Bei der ersten veränderten Definition von Reach wird statt dem Minimum die Summe der Strecken $r_{\text{summe}}(v, P_{st}) := m(s, v, P_{st}) + m(v, t, P_{st}) = m(s, t)$ betrachtet. Die zweite Variante speichert ein Wertepaar zu jedem Knoten, $r_{\text{Paar}}(v, P_{st}) := (m(s, v, P_{st}), m(v, t, P_{st}))$.

4.2 Kantenbeschriftungen

Diese Beschleunigungstechnik versieht jede Kante mit einer Zusatzinformation, die die Knoten beschreibt, zu denen ein kürzester Weg von dieser Kante aus beginnt. Dazu definiert man zunächst

$$S : E \to \mathcal{P}(V),\ (u, v) \mapsto \{w \in V : \text{Der kürzeste Weg von } u \text{ nach } w \text{ beginnt mit } (u, v)\}$$

Diese Abbildung ordnet jeder Kante genau die Menge der Knoten zu, zu denen ein kürzester Weg von dieser Kante aus beginnt. Benutzt man diese Mengen zur Verkleinerung des Suchraums,so ist das Ergebnis optimal, d.h. der Suchraum besteht nur noch aus dem kürzesten Weg selbst. Zwar kann man leicht mit einer All Pairs Shortest Path Suche S vollständig bestimmen. Um diese Vorberechnung festzuhalten, ist jedoch ein Speicheraufwand von $O(n \cdot m)$ notwendig. Da dies bei großen Graphen nicht möglich ist, speichert man für jede Kante $(u, v) \in E$ eine Obermenge $C(u, v) \subseteq S(u, v)$ die mit konstantem Speicherplatz pro Kante auskommt. Verwendet man C zur Beschneidung des Suchraums der kürzesten Wege Suche, d.h.

$$processEdge(v, u) := \begin{cases} wahr & t \in C(v, u) \\ falsch & \text{sonst} \end{cases}$$

so bleibt das Ergebnis korrekt, da C alle Knoten aus S umfasst. Diese Menge können auch für große Graphen gespeichert werden, da wegen des konstanten Speicheraufwandes pro Kante insgesamt linearer Speicheraufwand ausreichend ist.

4.2.1 Bounding-Box

Steht ein Layout des Graphen zur Verfügung, so kann mann diese Information zur Speicherung einer Menge von Knoten nutzen. Dabei speichert man die Menge $C(u, v)$ als *Bounding-Box*. Das ist im zweidimensionalen Fall ein achsenparalleles Rechteck, im Allgemeinen ein d-dimensionaler Quader, der so gewählt ist, dass er gerade alle Knoten aus $S(u, v)$ umfasst. In der durch den Quader definierten Menge $C(u, v)$ liegen in den wohl meisten Fällen auch Knoten $w \notin S(u, v)$. Im modifizierten Algorithmus von Dijkstra prüft die Funktion *processEdge*, ob der Zielknoten in der Bounding-Box der Kante liegt:

$$processEdge(v, u) := \begin{cases} wahr & t \in \text{BoundingBox}(v, u) \\ falsch & \text{sonst} \end{cases}$$

Nutzt man ein d-dimensionales Layout, so müssen pro Kante $2 \cdot d \cdot 32$ Bits zur Speicherung des Quaders aufgewendet werden. Das Verfahren benötigt eine All Pairs Shortest Path Vorberechnung, die einen Aufwand von $O(n^2\ log\ n)$ hat. Dabei führt man für jeden Knoten $v \in V$ eine Kürzeste-Wege-Suche aus, um die Bounding-Boxen für alle seine ausgehenden Kanten $(v, u) \in E$ zu bestimmen.

4.2.2 Graphenpartitionierung

Diesem Verfahren geht eine Einteilung des Graphen in Regionen voraus. Das ist eine Funktion $reg : V \to \{1, \cdots, z\}$, die jedem Knoten eine natürliche Zahl zuordnet. Die zu $n \in \{1, \cdots, z\}$ zugehörige Region besteht aus dem Urbild $reg^{-1}(n) = \{v \in V : reg(v) = n\}$. Mit dieser Partitionierung des Graphen kann man die Knotenmenge $C(u, v)$ zu einer Kante (u, v) als Bitvektor angeben. Dabei ist das i-te Bit gesetzt, wenn die Knoten der Region i in der Menge liegen, d.h. wenn es einen kürzesten Weg zu einem Knoten $v \in reg^{-1}(i)$ gibt,

der mit (u, v) beginnt. Im Algorithmus muss nur geprüft werden, ob das Bit der Region des Zielknotens t gesetzt ist:

$$processEdge(v, u) := \begin{cases} wahr & \text{Bitvektor}((v, u), reg(t)) = 1 \\ falsch & \text{sonst} \end{cases}$$

Das Verfahren arbeitet unabhängig von der Art und Weise der Regioneneinteilung korrekt. Diese hat jedoch erhebliche Auswirkungen auf die Größe des Suchraumes und damit auf die Effizienz des Verfahrens. In folgenden Abschnitten werden verschiedene Möglichkeiten zur Unterteilung des Graphen in Regionen vorgestellt. Benutzt man eine Einteilung mit z verschiedene Regionen, so werden z Bits pro Kante benötigt, für jeden Knoten wird in einem Integer[2] die Nummer seiner Region gespeichert.

Ein großer Vorteil dieses Verfahrens ist die Tatsache, dass im Vorverarbeitungsschritt auf eine aufwendige All Pairs Shortest Path Vorberechnung verzichtet werden kann. In [Sch05] wird gezeigt, dass es ausreichend ist, solche Kanten zu betrachten, die Knoten verschiedener Regionen verbinden. Für jeden sogenannten Grenzknoten[3], das sind die Zielknoten solcher überlappender Kanten, wird eine Rückwärtssuche ausgeführt. Eine Rückwärtssuche ist eine Kürzeste-Wege-Suche im inversen Graph $G' := (V, E')$ von $G = (V, E)$, der durch die Umkehrung der Kantenrichtungen aus G entsteht, d.h. E' ist hier definiert durch $(u, v) \in E' \Leftrightarrow (v, u) \in E$. Dies reicht aus, um die Bitvektoren aller Kanten korrekt zu setzen. Diese verbesserte Vorverarbeitung benötigt einen Aufwand von $O(k \cdot n \; log \; n)$, wobei k die Anzahl der Grenzknoten ist.

Unterteilung in Regionen Im ersten Schritt der Vorberechnung muss eine Einteilung des Graphen in Regionen vorgenommen werden. Steht ein Layout zur Verfügung, so kann dies im einfachsten Fall bespielsweise unter Zuhilfenahme eines Gitter passieren. Dabei wird allen Knoten im selben Gitterquadranten die gleiche Region zugeordnet. Eine weitere Möglichkeit bei zweidimensionalen Layouts ist die Verwendung eines sogenannten Quadtrees. Dieses in der algorithmischen Geometrie verwendete Verfahren teilt rekursiv achsenparallele Rechtecke in vier Rechtecke gleicher Größe, solange sich in diesen eine Mindestanzahl von Knoten befindet. Dieses Konzept ist, als Octtree im dreidimensionalen und als 2^d-Tree im d-dimensionalen Fall, auch auf höherdimensionale Layouts verallgemeinerbar.

kd-Tree Unterteilung Im Folgenden soll das Verfahren der kd-Tree Unterteilung mit Median Partitionierung genauer betrachtet werden. Für zweidimensionale Layouts konnten damit in Experimenten [Sch05] gute Ergebnisse erzielt werden. Wie bei einem Quadtree entstehen die Regionen durch schrittweise Teilung entlang der Achsen des Koordinatensystems. Durch eine Aufspaltung am Median der Knotenkoordinaten werden Partitionen mit im wesentlichen gleicher Knotenanzahl erzeugt.

[2] Eigentlich sind $log \; z$ Bit pro Knoten ausreichend
[3] Englisch: Boundary-Nodes

Zu Beginn der Regionenunterteilung befinden sich alle Knoten in einer Region. Darauf folgend werden solange Regionen mit maximaler Knotenanzahl halbiert, bis die gewünschte Gesamtanzahl der Regionen erreicht ist. Zur Teilung einer Partition wählt man zunächst eine Achse a des Koordinatensystems aus. Anschließend betrachtet man die Koordinaten dieser Achse und berechnet den Median dieser Werte. Alle Knoten, deren a-Koordinate größer ist als der Median, werden der ersten, alle übrigen der zweiten neuen Region zugeordnet.

Eine Möglichkeit zur Auswahl der Achsen ist das zyklische Durchlaufen der Dimensionen. Dies bedeutet beispielsweise im Zweidimensionalen, dass abwechselnd entlang der X- und der Y-Achse geteilt wird. Bei der Verwendung höherdimensionaler Layouts kann es sinnvoller sein, entlang der Achse mit der größten Varianz zu teilen.

Hochdimensionale Unterteilung Das folgende Partitonierungsverfahren basiert auf der oben beschriebenen kd-Tree Unterteilung und der in Kapitel 3.1 erklärten hochdimensionalen Einbettung. Die bei den kd-Bäumen verwendete schrittweise Halbierung der Regionen geschieht hier entlang einer Achse in einem durch hochdimensionale Einbettung erzeugten eindimensionalen Layout. Dies entspricht der Platzierung einer Achse im 50-dimensionalen Layout, so dass die Varianz der Knotenkoordinaten bezüglich dieser Achse maximal ist.

Zu Beginn wird einmal für den gesamten Graphen das in Kapitel 3.1.1 beschriebene hochdimensionale Layout berechnet. Dieses wird bei der Unterteilung einzelner Regionen zur Projektion der darin befindlichen Knoten in ein eindimensionales Layout verwendet. Die Projektion erfolgt durch Anwendung des in Kapitel 3.1.2 vorgestellten zweiten Schrittes der hochdimensionalen Einbettung. Das heißt man erstellt die Kovarianzmatrix S genau wie dort, jedoch beschränkt auf die Koordinaten der aktuell zu teilenden Region. Damit ist ein eindimensionales Layout bestimmt, d.h. alle Knoten liegen auf einer Achse. Diese können nun leicht durch eine Aufteilung am Median zwei neuen Regionen zugeordnet werden.

5 Experimentelle Ergebnisse

Die hier vorgestellten Verfahren wurden implementiert und auf einem Rechner mit einer Intel Xeon CPU, 2.80GHz und 2GB Arbeitsspeicher getestet. Dazu wurden Ausschnitte von Straßenkarten verschiedener Größe benutzt. Zur Bestimmung der durchschnittlichen Größe des Suchraums wurden wiederholt kürzeste Wege zwischen zwei zufällig ausgewählten Knoten des Graphen gesucht. Eine solche Messreihe wurde abgebrochen, wenn das 95% Koinfidenzintervall kleiner war, als 5% des durchschnittlichen Suchraums, spätestens aber nach 50000 Suchanfragen.

Die hier vorgestellten Messergebnisse verwenden durch hochdimensionale Einbettung (Kapitel 3.1) erzeugte Layouts. Untersucht werden soll zunächst das Verhalten der Beschleunigungsverfahren *Reichweite*, *Bounding-Box* und *Graphenpartitionierung* bei der Verwendung von ein- bis fünfdimensionalen Zeichnungen. Am deutlichsten sind Verbesserungen durch eine Erhöhung der Anzahl der Dimensionen bei Bounding-Box zu erkennen

#Knoten	Graph	Dijkstra	Orig. 2D	1D	2D	3D	4D	5D
citymap 1	1429	726	288	362	317	295	292	290
citymap 2	2948	1506	405	522	480	452	444	443
citymap 3	15868	7965	2056	2709	2435	2234	2209	2173
citymap 4	20036	10056	3817	4485	3930	3888	3879	3880
citymap 5	24106	12177	2285	3408	2533	2458	2395	2380
citymap 6	38823	19463	5594	6669	5960	5806	5784	5777
citymap 7	35802	17999	4196	9863	5325	4709	4492	4424
citymap 8	44878	22643	7485	9672	7841	7550	7509	7536
citymap 9	44439	22313	5782	7858	6604	6293	6238	6251

Tabelle 3: Durchschnittliche Größe des Suchraums, Reichweite, Originallayout, hochdimensionale Einbettung: 1-5 Dimensionen

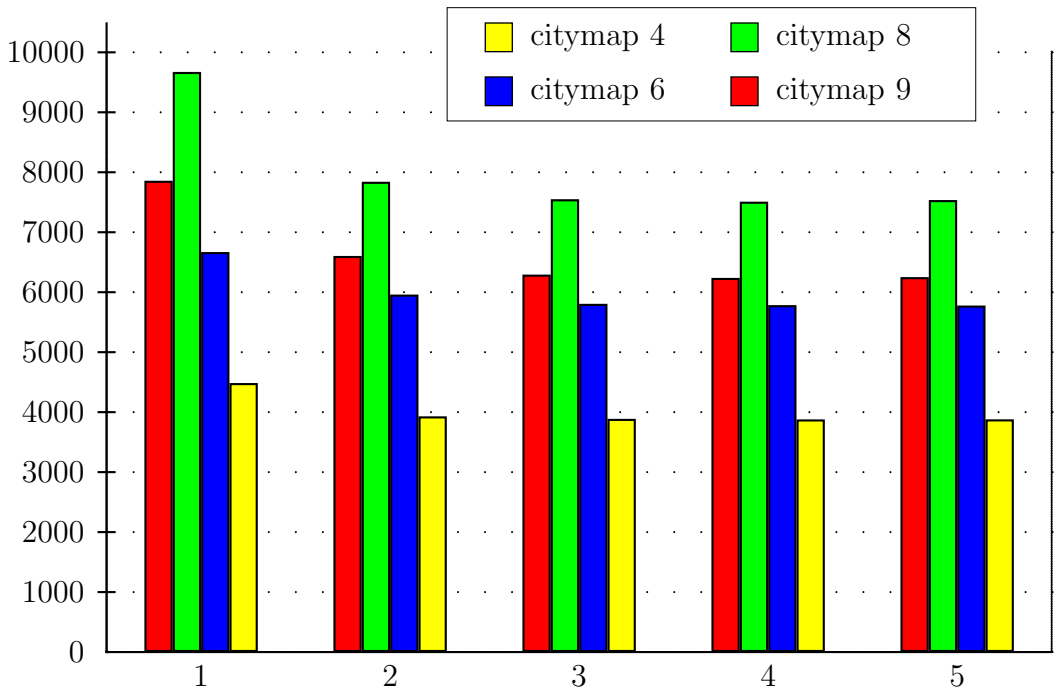


Abbildung 3: Durchschnittliche Größe des Suchraums, Reichweite, hochdimensionale Einbettung: 1-5 Dimensionen

#Knoten	Graph	Dijkstra	Orig. 2D	1D	2D	3D	4D	5D
citymap 1	1429	726	141	175	122	110	103	103
citymap 2	2948	1506	94	205	161	128	117	99
citymap 3	15868	7965	825	2403	1180	784	745	697
citymap 4	20036	10056	872	2434	905	758	719	685
citymap 5	24106	12177	586	2172	701	567	538	496
citymap 6	38823	19463	2254	4469	2425	1704	1659	1586
citymap 7	35802	17999	2626	4448	3279	1975	1698	1571
citymap 8	44878	22643	3111	7369	3146	2250	2182	2129
citymap 9	44439	22313	1896	5539	2890	2489	2191	1986

Tabelle 4: Durchschnittliche Größe des Suchraums, Bounding-Box, Originallayout, hochdimensionale Einbettung: 1-5 Dimensionen

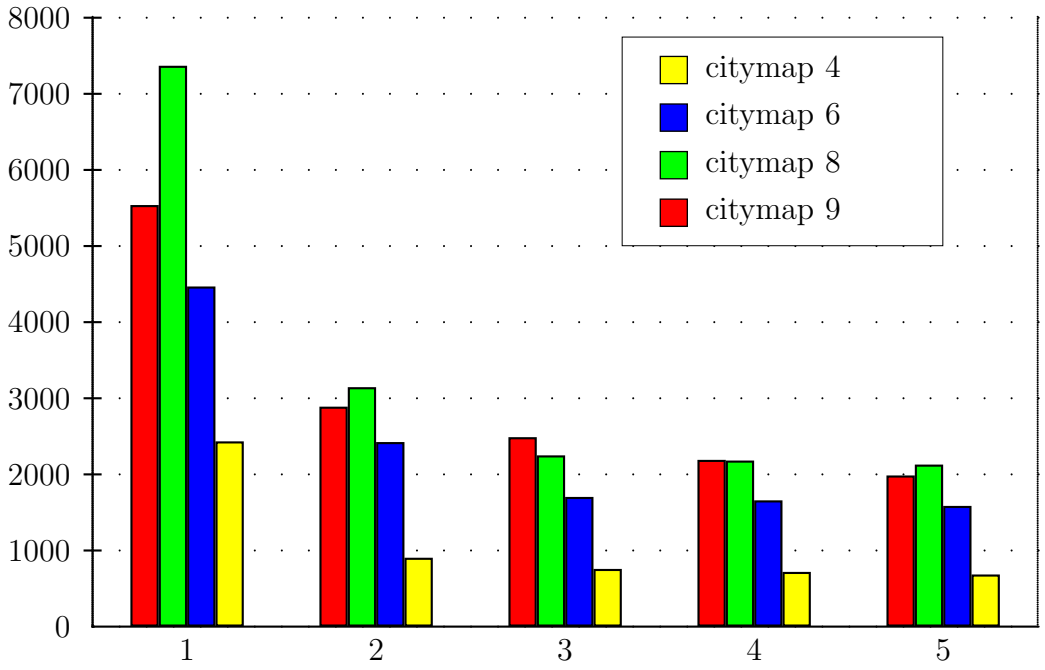


Abbildung 4: Durchschnittliche Größe des Suchraums, Bounding-Box, hochdimensionale Einbettung: 1-5 Dimensionen

#Knoten	Graph	Dijkstra	Orig. 2D	1D	2D	3D	4D	5D
citymap 1	1429	726	162	175	154	159	153	154
citymap 2	2948	1506	186	248	244	222	222	222
citymap 3	15868	7965	1021	1209	1206	1108	1116	1069
citymap 4	20036	10056	1431	2356	1366	1374	1379	1381
citymap 5	24106	12177	1269	2033	1233	1228	1240	1237
citymap 6	38823	19463	2748	4603	3181	2995	2990	2981
citymap 7	35802	17999	2484	5150	3930	3071	2968	2928
citymap 8	44878	22643	3508	6255	3350	3310	3303	3306
citymap 9	44439	22313	3089	5282	2884	2938	2954	2958

Tabelle 5: Durchschnittliche Größe des Suchraums, Graphnpartitionierung (kd-Tree, 32 Bit, Teilen nach Varianz), hochdimensionale Einbettung: 1-5 Dimensionen

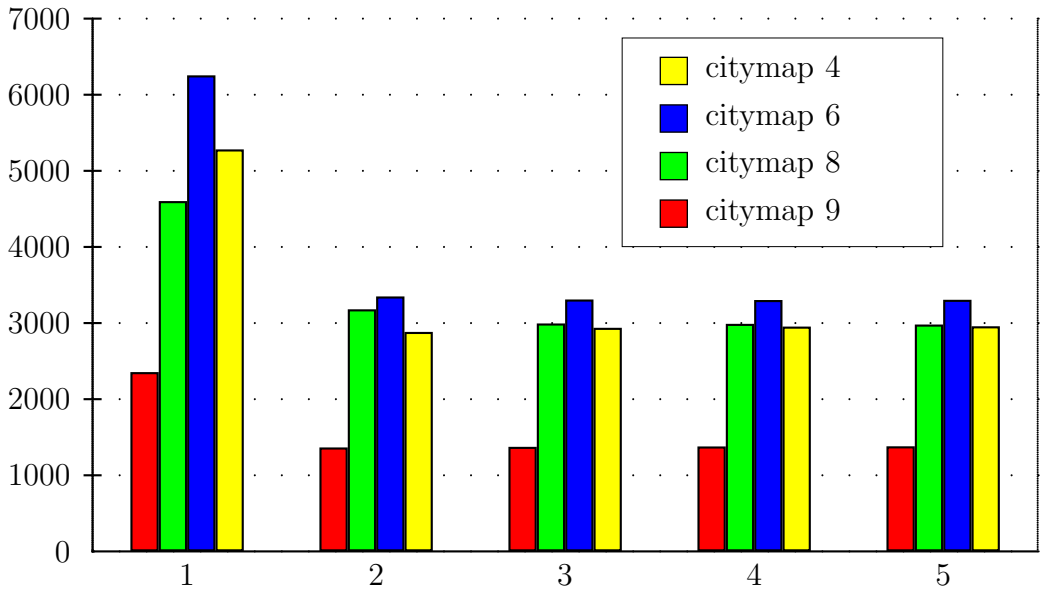
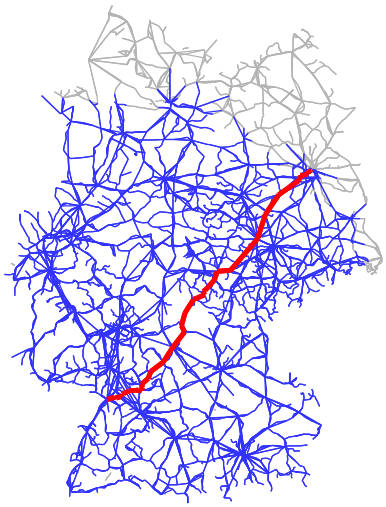
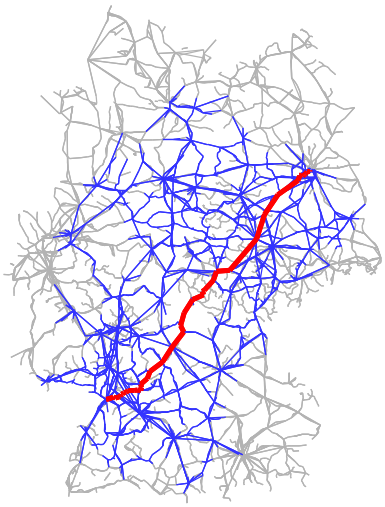


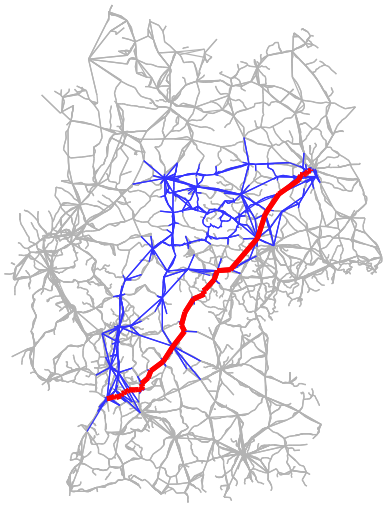
Abbildung 5: Durchschnittliche Größe des Suchraums, Graphnpartitionierung (kd-Tree, 32 Bit, Teilen nach Varianz), Originallayout , hochdimensionale Einbettung: 1-5 Dimensionen



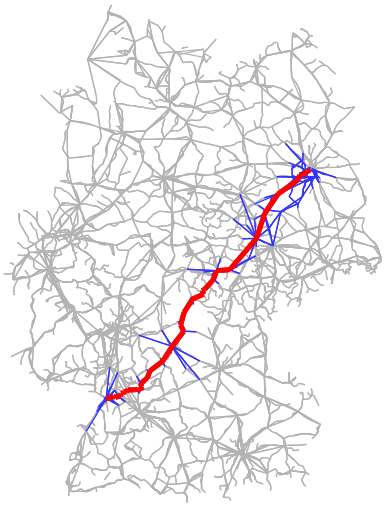
(a) Dijkstra



(b) Reichweite



(c) Bounding-Box



(d) Partitionierung

Abbildung 6: Suchräume der Beschleunigungstechniken, Anfrage: Karlsruhe - Berlin

	1. Dim.	2. Dim.	3 Dim.	4 Dim.	5 Dim.	6 Dim.	7 Dim.
citymap 1	8	3	16	2	1	1	0
citymap 2	13	8	7	1	0	1	1
citymap 3	14	7	7	2	1	0	0
citymap 4	14	14	1	1	1	0	0
citymap 5	13	13	3	2	0	0	0
citymap 6	15	11	5	0	0	0	0
citymap 7	5	13	8	4	1	0	0
citymap 8	9	12	10	0	0	0	0
citymap 9	15	8	8	0	0	0	0

Tabelle 6: Regionenunterteilung: Anzahl der Teilungen pro Dimension bei Auswahl der Achse nach Varianz, 32 Bit, 10D Layout

(Tabelle 4 und Abbildung 4). Der Suchraum verkleinert sich mit einer Erhöhung der Dimensionsanzahl bei einer Beschleunigung durch *Reichweite* nur kaum (Tabelle 3 und Abbildung 3). Bei einer Partitionierung des Graphen mittels kd-Trees, mit Auswahl der Teilungsachse durch die maximale Varianz, sind durch höherdimensionale Zeichnungen nur geringe Verbesserungen zu beobachten[4]. Bei manchen Graphen konnten mit den Verfahren Bounding-Box und Graphenpartitionierung unter Verwendung des hochdimensionalen Layouts Verbesserungen gegenüber dem zweidimensionalen geographischen Layout beobachtet werden. Für die Diagramme wurden zu Gunsten einer besseren Übersichtlichkeit vier Karten ausgewählt.

Weitere Messungen betreffen die Auswahl der Achsen bei der Regionenunterteilung bei Beschleunigung durch Graphenpartitionierung. Teilt man an der Achse, deren Daten die größte Varianz aufweisen, so zeigen die Tabelle 6 aufgeführten Messungen, dass dazu im Wesentlichen nur die ersten drei bis vier Dimensionen verwendet werden, in der achten oder höheren Dimensionen wurde nicht geteilt. Tabelle 7 zeigt, dass Teilen an der Achse mit der größten Varianz tatsächlich den Suchraum verkleinert.

In Abbildung 6 wurden die Suchräume der verschiedenen Verfahren sichtbar gemacht. Dazu sind die gescannten Kanten blau markiert, der kürzeste Weg ist rot eingezeichnet. Man sieht, dass die unveränderte Fassung von Dijkstras Algorithmus einen sich im wesentlichen kreisförmig ausbreitenden Suchraum aufweist, also alle Knoten erreicht werden, die näher am Startknoten liegen als der Zielknoten. Bei Reichweite werden dort, wo Start- und Zielknoten weit entfernt liegen, nur wichtige Verkehrswege betrachtet. Bei Boundig-Box und Graphenpartitionierung kann der Suchraum stark verkleinert werden und geht seltener in falsche Richtungen.

Die in Tabelle 8 aufgelisteten Messwerte zeigen, dass mit einer Aufteilung des Graphen in Regionen durch das im letzen Abschnitt des Kapitels 4.2.2 beschriebene Verfahren der hochdimensionalen Einbettung die Ergebnisse des Teilens per kd-Tree noch verbes-

[4]Den eigentlichen Rückschritt auf eine Dimension ausgenommen.

#Knoten	Graph	Varianz				Zyklisch			
		2D	3D	4D	5D	2D	3D	4D	5D
citymap 1	1429	154	159	153	154	148	148	150	158
citymap 2	2948	244	222	222	222	260	279	273	277
citymap 3	15868	1206	1108	1116	1069	1306	1351	1309	1307
citymap 4	20036	1366	1374	1379	1381	1390	1573	1718	1811
citymap 5	24106	1233	1228	1240	1237	1237	1286	1460	1527
citymap 6	38823	3181	2995	2990	2981	2979	2802	3069	3301
citymap 7	35802	3930	3071	2968	2928	3965	3044	2920	3164
citymap 8	44878	3350	3310	3303	3306	3557	3227	3929	4532
citymap 9	44439	2884	2938	2954	2958	2849	2959	3223	3384

Tabelle 7: Durchschnittliche Größe des Suchraums, Graphnpartitionierung (kd-Tree, 32 Bit), hochdimensionale Einbettung: 1-5 Dimensionen, Teilen nach maximaler Varianz / zyklisches Teilen

#Knoten	Graph	32 Bit		128 Bit	
		KD	PCA	KD	PCA
citymap 1	1429	150	135	110	104
citymap 2	2948	214	208	125	121
citymap 3	15868	1072	1118	580	590
citymap 4	20036	1379	1257	748	715
citymap 5	24106	1247	1170	632	615
citymap 6	38823	3008	2914	1540	1395
citymap 7	35802	2920	2686	1480	1402
citymap 8	44878	3306	3073	1708	1623
citymap 9	44439	2960	2918	1496	1422
citymap 10	78947	4042	3777	1885	1854
citymap 11	78352	4021	4179	1883	1935
citymap 12	85267	4258	4059	2050	1924
citymap 13	101656	4296	4205	1968	1913
citymap 14	111427	4638	4546	2190	2109
citymap 15	112990	6520	6363	3053	3074
citymap 16	133235	6228	5936	2920	2658

Tabelle 8: Durchschnittliche Größe des Suchraums, Graphenpartitionierung, Regionenunterteilung mittels 10D Layout, Teilen nach maximaler Varianz, kd-Tree (KD) oder hochdimensionaler Einbettung(PCA)

sert werden können. Bei Messungen mit einer Straßenkarte mit 362554 Knoten war die durchschnittliche Größe des Suchraums bei hochdimensionaler Einbettung mit 137 Knoten deutlich kleiner als kd-Tree mit 174 Knoten. Dabei wurden 128 Regionen und bidirektionale Suche verwendet.

6 Implementierung

Die Algorithmen wurden in der Sprache C++ unter Verwendung des Übersetzers GCC 3.3 auf einem Linux System der Version 2.6.4 implementiert. Die benutzten Datenstrukturen für Graphen entstammen der Bibliothek LEDA 4.5.

Die Implementierung baut auf einer bestehenen Bibliothek von Graphenalgorithmen auf, die intensiven Gebrauch von parametrisierter Vererbung macht. Diese Technik ermöglicht eine einfache Kombination und Erweiterbarkeit der Algorithmen. Die dabei entstehende tiefe Vererbungshierarchie führt jedoch trotz intensiver Optimierungen des Compilers zu schlechteren Laufzeiten als eine "flache" Ausprogrammierung. Daher wurde zur Bewertung der Beschleunigungstechniken in Kapitel 5 im Wesentlichen die Größe des Suchraumes herangezogen.

Die benutzten Mixin Klassen sind Mittel der generischen Programmierung und gehen von einer Superklasse aus, von der erst zu einem späteren Zeitpunkt abgeleitet wird. Durch diese Erweiterung einer unbekannte Basisklasse durch eine Vererbungsbeziehung ist das Hinzufügen zusätzlicher, als *Aspekte* bezeichneter, Eigenschaften möglich. In C++ kann dies durch Templates ausgedrückt werden. Diese Technik wurde hier verwendet, um die Beschleunigungstechniken auf den Algorithmus von Dijkstra anzuwenden. Diese sind dadurch leicht untereinander kombinierbar. Zusätzliche Funktionalitäten, wie etwa Statistikfunktionen oder die Visualisierung des Suchraums, können so leicht ergänzt werden. Zur Erweiterung auf mehrere Dimensionen wurden die Leda Typen `leda::point` bzw. `leda::point_3d` duch den Datentyp `std::valarray<double>` aus der Standardbibliothek von C++ ersetzt.

7 Ausblick

Es wurde gezeigt, dass durch die Verwendung eines generierten Layouts und die Erhöhung der Anzahl der Dimensionen eine Verbesserung von Beschleunigungsverfahren erreicht werden kann. Durch eine direkte Integration der Layoutgenerierung in die Regionenunterteilung der Graphenpartitionierung entsteht ein Beschleunigungsverfahren, dass nicht mehr auf ein gegebenes Layout angewiesen ist und das sich in Tests als das schnellste der hier verglichenen Verfahren herausgestellt hat. Bei diesem ist wegen der Verwendung der Regionentechnik auch der Vorberechnungsaufwand vergleichsweise gering.

Für die Qualität der Beschleunigungsverfahren wurde hier im Wesentlichen nur die Größe des Suchraums berücksichtigt. Eine "flache" Ausprogrammierung ohne die hier benutzte vielstufige Vererbungshierarchie würde aussagekräftigere Zeitmessungen zulassen.

Literatur

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms. *The MIT Press, Cambridge Massachusetts*, 2001.

[Ead84] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[Gol01] Andrew V. Goldberg. A simple shortest path algorithm with linear average time. *proceedings of the 9th European Symposium on Algorithms (ESA '01), Springer Lecture Notes in Computer Science LNCS 2161*, pages 230–241, 2001.

[Gut04] Ron Gutman. Reach-based routing: A new approach to shortest path algortihms optimized for road networks. In Lars Arge, Giuseppe F. Italiano, and Robert Sedgewick, editors, *Proc. Algorithm Engineering and Experiments (ALENEX'04)*, pages 100–111. SIAM, 2004.

[HK02] David Harel and Yehuda Koren. Graph drawing by high-dimensional embedding. In *Proceedings of the 10th International Symposium on Graph Drawing (GD '02)*, volume 2528 of *LNCS*, pages 207–219. Springer, 2002.

[HSW04] Martin Holzer, Frank Schulz, and Thomas Willhalm. Combining speed-up techniques for shortest-path computations. In Celso C. Ribeiro and Simone L. Martins, editors, *Experimental and Efficient Algorithms: Third International Workshop, (WEA 2004)*, volume 3059 of *LNCS*, pages 269–284. Springer, 2004.

[KK89] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[Sch05] Birk Schütz. Partition-based speed-up of dijkstra's algorithm. *Studienarbeit, Universität Karlsruhe (TH), Institut für Logik, Komplexität und Deduktionssysteme, Lehrstuhl Prof. Dr. D. Wagner*, 2005.

[WW05] Dorothea Wagner and Thomas Willhalm. Drawing graphs to speed up shortest-path computations. In *Proc. 7th Workshop Algorithm Engineering and Experiments (ALENEX'05)*, LNCS. Springer, 2005. To appear.

[WWZ04] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. Geometric shortest path containers. Technical Report 2004-5, Universität Karlsruhe, Fakultät für Informatik, 2004.