

**Seminararbeit :**  
**Routenplanung mithilfe von Hierarchien**

Simon Koops  
Matrikelnummer : 6336  
Fachbereich : Technische Informatik  
Fachsemester : 5

13.11.2009

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>3</b>
1.1	Nachteile herkömmlicher kommerzieller Systeme . . . . .	3
1.2	Prinzipien des Preprocessing . . . . .	3
1.3	Serversysteme und mobile Systeme . . . . .	4
1.4	Darstellung von Straßennetzen als Graph . . . . .	4
1.5	Algorithmus von Dijkstra . . . . .	4
1.5.1	Zweiseitige Routing-Algorithmen . . . . .	5
1.6	Anleihen bei bekannten Algorithmen . . . . .	5
<b>2</b>	<b>Hierarchisches Routing</b>	<b>6</b>
2.1	Problemstellung . . . . .	6
2.2	Hierarchische Strukturen . . . . .	6
2.3	Der Nachbarschaftsradius eines Knoten . . . . .	6
2.4	Das Finden kürzester Pfade . . . . .	7
2.5	Berechnung des tatsächlichen Weges . . . . .	8
2.6	Der Grund-Algorithmus als Pseudocode im Vergleich zum Dijkstra . . . . .	9
<b>3</b>	<b>Abwandlungen des Grund-Algorithmus</b>	<b>10</b>
3.1	Highway-Node Routing . . . . .	10
3.1.1	Statisches Highway-Node-Routing . . . . .	10
3.1.2	Unterschiede und Gemeinsamkeiten zu Highway-Hierarchies . . . . .	11
3.1.3	Dynamisches Highway-Node-Routing . . . . .	11
3.2	Many-to-May Shortest Paths . . . . .	12
3.2.1	Zentrale Idee . . . . .	12
3.2.2	Optimierung für große Distanztabellen . . . . .	13
3.3	Transit-Node Routing . . . . .	13
3.3.1	Zentrale Idee . . . . .	13
3.3.2	Berechnung der Zugangsknoten . . . . .	14
<b>4</b>	<b>Messbare Erfolge des Algorithmus</b>	<b>15</b>
4.1	Verwendete Hard- und Software . . . . .	17
<b>5</b>	<b>Quellen</b>	<b>18</b>

# 1 Einleitung

Dieser Seminarvortrag befasst sich mit dem Algorithmus *Highway Hierarchies* der in der Dissertation von Dominik Schultes beschrieben wird. Er dient zur effizienten Bestimmung von optimalen Routen in Straßennetzen. Als Dominik Schultes seine Dissertation schrieb, arbeitete er als wissenschaftlicher Mitarbeiter in der Arbeitsgruppe von Prof. Sanders am Institut für Theoretische Informatik der Universität Karlsruhe (TH) und forschte zum Thema „Routenplanung in Straßennetzen“. Die mündliche Prüfung zur Dissertation war Anfang Februar 2008.

## 1.1 Nachteile herkömmlicher kommerzieller Systeme

Viele Navigationsgeräte und PC-Programme zur Routenplanung verwenden heutzutage heuristische Algorithmen, die in akzeptabler Zeit Ergebnisse bringen, die (hoffentlich) gut sind, aber nicht garantieren können, die beste Lösung finden. Es muss also ein Kompromiss zwischen Exaktheit und Geschwindigkeit der Berechnung gefunden werden.

Um zu seinem Ziel zu gelangen, wird man i.A. versuchen, schnell auf eine größere Straße zu fahren, auf der man hofft, schneller an sein Ziel zu gelangen. Auch viele Routing-Algorithmen arbeiten auf diese Weise. Oft wird man so einen guten Weg finden, jedoch nicht immer, da die großen Straßen einen Umweg machen könnten oder nicht in die gewünschte Richtung führen, und man durch eine Abkürzung schneller an sein Ziel gelangen würde. Einfache Algorithmen verlassen eine große Straße aber nicht, da sie davon ausgehen, dass man auf ihnen schneller ans Ziel kommt. Mögliche Abkürzungen werden also nicht gefunden. Da auch solche Algorithmen zweiseitig sind, (s.u.), ist es nicht nötig, die große Straße zu verlassen, um das Ziel in einer kleinen Straße zu finden.

Ein weiterer Nachteil ist, dass die Priorität, bzw. Größe einer Straße, manuell angegeben werden muss.

## 1.2 Prinzipien des Preprocessing

Um einen Algorithmus, der Anfragen auf einer Datenstruktur ausführt, zu beschleunigen, kann es hilfreich sein, im voraus Berechnungen durchzuführen, die eine einzelne Abfrage später beschleunigen.

Dabei werden i.A. zusätzliche Informationen gespeichert, die die spätere Suche vereinfachen, allerdings den Speicherverbrauch erhöhen.

Einfache Beispiele sind das Sortieren einer Datenstruktur oder das Anlegen einer Baumstruktur.

Je nach verwendetem System ist dies von Vorteil (Handy, Auto-Navi, PC, Großrech-

ner...). Ein Algorithmus, bei dem man die Zeit des Preprocessing und den Notwendigen Speicher je nach Bedarf einfach anpassen kann, muss also nicht mehrfach implementiert werden, um kürzere Preprocessing-Zeiten zu erhalten oder, auf Kosten von Speicher, die Geschwindigkeit zu erhöhen.

### 1.3 Serversysteme und mobile Systeme

Um eine Anfrage zu bearbeiten, gibt es zwei Möglichkeiten. Entweder das Benutzergerät beantwortet die Anfrage selber, dazu muss es alle notwendigen Daten lokal gespeichert haben. Die Berechnung erfolgt dann auf dem Gerät selber. Die Geschwindigkeit hängt dann von der verwendeten Hardware ab. Das Preprocessing kann schon bei der Herstellung geschehen, dann fällt das lokale Preprocessing weg, es ist aber auch keine Dynamik möglich. Die Karten müssen dann von Zeit zu Zeit aktualisiert werden. Dieser Fall wird im weiteren nicht beachtet.

Eine andere Möglichkeit besteht darin, dass die Anfrage an einen Server weitergeleitet wird. Dazu wird eine Verbindung, z.B. das Internet, zu diesem benötigt. Der Server hat eine wesentlich größere Rechenkapazität und mehr Speicherplatz, sodass ein besseres Preprocessing möglich ist und eine Anfrage schneller bearbeitet werden kann. Auch ist es einfacher, Staunachrichten u.ä. auf dem Server zu verarbeiten, als diese an alle einzelnen mobilen Geräte zu verteilen.

Dies ist u.a. über das Rundfunknetz mittels TMC (Traffic Message Channel) möglich. Dazu wird eine Radioempfangseinheit am Navi benötigt, oder ein Radio, das diese Information an das Navi übermittelt. Bei schlechtem UKW-Empfang oder schlecht geführten Daten kann dieses System zu Problemen führen.

### 1.4 Darstellung von Straßennetzen als Graph

Straßennetze lassen sich einfach als Graph darstellen. Jede Kreuzung entspricht dabei einem Knoten, jede Straße zwischen zwei Kreuzungen entspricht einer Kante. Diese Kanten werden bewertet. Als Bewertungsmaß können dabei die Strecke, die erwartete benötigte Zeit o.ä. dienen. Die Lage eines Knoten und die Länge einer Kante in der Darstellung sind dabei irrelevant, sie dienen nur zur Darstellung (außer dies wird extra angegeben).

### 1.5 Algorithmus von Dijkstra

Da der Algorithmus von Dijkstra die Grundidee des im folgenden beschriebenen Algorithmus ist, soll er hier kurz erklärt werden :

Die Grundidee des Algorithmus von Dijkstra ist es, ab einem Startknoten die kürzesten Wege zu (vorerst) allen Knoten zu finden und längere Wege dabei auszuschließen. Er besteht aus folgenden Schritten:

1. Weise allen Knoten die beiden Eigenschaften *Distanz* und *Vorgänger* zu. Initialisiere die Distanz im Startknoten mit 0 und in allen anderen Knoten mit  $\infty$ .
2. Solange es noch unbesuchte Knoten gibt, wähle darunter denjenigen mit minimaler Distanz aus und
  - a) Speichere, dass dieser Knoten schon besucht wurde
  - b) Berechne für alle noch unbesuchten Nachbarknoten die Summe des jeweiligen Kantengewichtes und der Distanz im aktuellen Knoten
  - c) Ist dieser Wert für einen Knoten kleiner als die dort gespeicherte Distanz, aktualisiere sie und setze den aktuellen Knoten als Vorgänger.

Dieser Schritt wird auch als Update bezeichnet und ist die zentrale Idee von Dijkstra.

In dieser Form berechnet der Algorithmus ausgehend von einem Startknoten alle kürzesten Wege. Ist man dagegen nur an dem Weg zu einem ganz bestimmten Knoten interessiert, so kann man in Schritt (b) schon abbrechen, wenn der gesuchte Knoten der aktuelle ist.

### 1.5.1 Zweiseitige Routing-Algorithmen

Zweiseitige Algorithmen beginnen die Suche gleichzeitig vom Start- und Zielpunkt aus. Sobald ein Knoten von beiden Suchvorgängen (Vorwärts und Rückwärts) erreicht wurde (d.h., der kürzeste Weg dorthin wurde gefunden), ist der Suchvorgang beendet, da von diesem Punkt aus der kürzeste Weg zu Start- und Zielpunkt bekannt ist. Auf diese Weise kann der Algorithmus deutlich beschleunigt werden.

## 1.6 Anleihen bei bekannten Algorithmen

Einige der hier verwendeten Verfahren sind bereits länger bekannt, und wurden in diesem Algorithmus zusammengeführt und verfeinert. Andere Verfahren wurden nicht verwendet, da sie Nachteile aufweisen, die zu keiner oder nur sehr geringer Verbesserung des Algorithmus führen. Dies soll am A\*-Algorithmus verdeutlicht werden.

Der A\*-Algorithmus kommt aus dem Gebiet der künstlichen Intelligenz und versucht, die Knoten zu besuchen, die *voraussichtlich* am schnellsten zum Ziel führen. Dazu wird eine Schätzfunktion verwendet, mit der die minimale Entfernung zum Ziel geschätzt wird. Die minimale Entfernung ist die Luftlinie. Da die Kantengewichtung bei Suchalgorithmen allerdings nicht die zurückgelegte Strecke, sondern die (erwartete) benötigte Zeit ist, muss die Luftlinienentfernung durch die Geschwindigkeit geteilt werden. Da aber über die mögliche Geschwindigkeit auf der noch zu fahrenden Strecke nichts bekannt ist, muss von der niedrigsten Geschwindigkeit ausgegangen werden, was zu sehr pessimistischen Schätzungen führt. Deshalb hat es sich in den hier beschriebenen Verfahren nicht bewährt, den A\*-Algorithmus zu verwenden.

## 2 Hierarchisches Routing

### 2.1 Problemstellung

Da der Algorithmus von Dijkstra ggf. alle Knoten besuchen muss, hat er eine hohe Laufzeit. Um diesem Problem entgegenzuwirken, ist es wünschenswert, die Anzahl der Knoten, die besucht werden, zu reduzieren. Da man keine Knoten ersatzlos löschen kann, da diese ein Start- oder Zielpunkt darstellen oder auf einer idealen Route liegen könnten, werden hierarchische Strukturen verwendet, wobei in den höheren Ebenen die Knoten, die zum Erreichen weit entfernter Ziele nicht nötig sind, entfernt werden.

### 2.2 Hierarchische Strukturen

Wenn diese Hierarchien vorhanden sind, kann ein Suchalgorithmus von seinem Startpunkt aus, auf der untersten Hierarchieebene (im folgenden *Ebene*) routen, bis er einen Knoten einer höheren Ebene trifft, und nun auf dieser weiterrount. Wenn die obere Ebene beispielsweise nur noch 1/4 der Knoten der unteren Ebene besitzt, muss der Algorithmus nur noch gut 1/4 der gesamten Knoten besuchen. Wenn nun mehrere Ebenen vorhanden sind, ist der Anteil der besuchten Knoten noch viel kleiner. Da ein solcher Algorithmus beidseitig arbeitet, werden die beiden Suchvorgänge sich irgendwann (voraussichtlich auf einer höheren Ebene) treffen.

Man könnte als Hierarchien die bereits vorhandenen Hierarchien der Straßennetze verwenden. Man kann Straßen zwischen Neben- und Wohnstraßen, Hauptstraßen, Schnellstraßen, Autobahnen usw. unterscheiden. Wenn man optimale Routen, also immer den kürzesten Weg, finden möchte, ist es notwendig, diese Hierarchien selber zu berechnen, da es manchmal schneller ist, eine größere Straße zu verlassen, um auf einer kleineren eine Abkürzung zu finden.

Der Vorgang des Bildens einer überlagerten Ebene lässt sich mehrfach wiederholen, so dass die Anzahl der zu besuchenden Knoten weiter abnimmt. Dieses Verfahren wird im *Highway-Hierarchies*-Algorithmus verwendet.

### 2.3 Der Nachbarschaftsradius eines Knoten

Es wird für jeden Knoten ein sog. Nachbarschaftsradius eingeführt, der eine festgelegte Anzahl an Knoten enthält, die sich „in der Nähe“ des Ausgangsknoten befinden.

## 2.4 Das Finden kürzester Pfade

Als wichtigster Teil des Algorithmus muss festgelegt werden, wann Knoten und Kanten in den Graphen der nächsthöheren Ebene aufgenommen werden.

Im folgenden wird für Knoten, Kanten und Pfade folgende Notation verwendet :

Schreibweise	Bedeutung
$u$	Ein Knoten
$(u, v)$	Eine Kante zwischen den Knoten $u$ und $v$
$\langle s, t, \dots, v \rangle$	Ein Pfad führt von $s$ direkt nach $t$ und endet bei $v$

Eine Kante  $(u, v)$  gehört in einem solchen Berechnungsschritt zur nächsten Ebene, wenn es zwei Knoten  $s$  und  $t$  gibt, für die folgendes gilt:

Auf einem Pfad  $\langle u, \dots, s, t, \dots, v \rangle$  liegt  $s$  in der Nachbarschaft von  $u$  und  $t$  liegt in der Nachbarschaft von  $v$ :

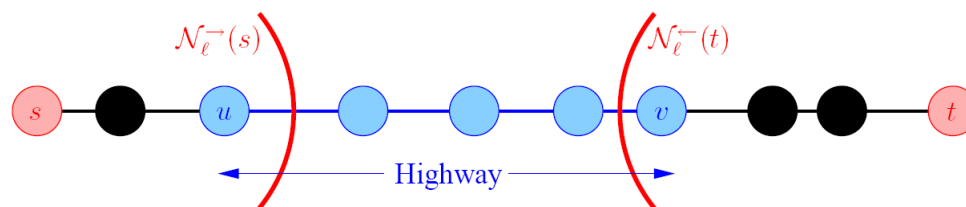


Abbildung 2.1: Ein kürzester Pfad von einem Knoten  $s$  zu einem Knoten  $t$ . Kanten, die nicht vollständig in der Nachbarschaft von  $s$  oder  $t$  liegen, sich aber auf dem kürzesten Pfad zwischen diesen befinden, sind Kanten des Fernstraßennetzwerks.

Man könnte nun vermuten, dass von jedem Punkt aus eine Suche zu jedem anderen Punkt gestartet werden muss, um alle kürzesten Pfade zu finden. Allerdings liegt jede solche Kante auch in einem Baum kürzester Pfade, der als Wurzel den Knoten  $s$  hat und dessen Blätter alle „weit genug“ von  $s$  entfernt sind.

Für jeden Knoten des Graphen wird nun ein Suchbaum aufgebaut, der „groß genug“ ist, um Highwaykanten zu finden. Anschließend werden aus diesem Baum die Kanten herausgesucht, die Highwaykanten sind.

Der Aufbau dieses Suchbaumes erfolgt mit Hilfe des Algorithmus' von Dijkstra. Da der Suchbaum nicht allzugroß wird, ist die Laufzeit nicht zu groß. Das Erweitern des Baumes hört jeweils an einem Blatt auf, wenn folgende Bedingung erfüllt ist :

Knoten	Bedeutung
$s_0$	Wurzel des Baumes
$s_1$	direkter Nachbarknoten von $s_0$
$p$	das betrachtete Blatt des Baumes

Es wird folgender Pfad betrachtet :  $\langle s_0, s_1, \dots, p \rangle$   
Abbruchbedingung :

$s_1$  ist ein Vorgänger von  $p$  (auf dem betrachteten Pfad)  $\wedge$   
 $p$  liegt nicht in dem Nachbarschaftsbereich von  $s_1$   $\wedge$   
 $s_0$  liegt nicht im Nachbarschaftsbereich von  $p$   $\wedge$   
{Menge der Knoten des Pfades}  $\cap$   
{Menge der Knoten des Nachbarschaftsbereiches von  $s_1$ }  $\cap$   
{Menge der Knoten des Nachbarschaftsbereiches von  $p$ } hat maximal ein  
Element.

(d.h., „zwischen“ den Nachbarschaftsbereichen von  $s_1$  und  $p$  liegt höchstens ein Knoten).

Danach wird der Baum von den Blättern aus Rückwärts durchlaufen und für jede Kante geprüft, ob sie sich nicht komplett innerhalb des Nachbarschaftsradius der Wurzel oder des Blattes befindet. Erfüllen sie diese Bedingung, gehören sie zum Graphen der nächsten Ebene. Auf diese Weise entsteht ein *Highway Network*.

In einigen Fällen liegen zwei oder mehr Highwaykanten hintereinander, deren Knoten jeweils nur mit 2 Kanten verbunden sind. Diese Kanten können abgekürzt und zu einer Kante zusammengefasst werden. Der resultierende Graph heißt *contracted Highway Network* oder *Core*. Diese beiden Schritte (Berechnung des Highway Netwrk und des Cores) werden mehrfach wiederholt, so dass die o.g. Ebenen entstehen.

Weiterhin wird die oberste Ebene durch eine Distanztabelle ersetzt, um die Suche dort zu beschleunigen. Dies erhöht den Speicherbedarf, verringert aber die Dauer einer Suche.

## 2.5 Berechnung des tatsächlichen Weges

Bis jetzt wurde nur die kürzeste Entfernung zwischen Start und Ziel berechnet, der Weg ist noch nicht bekannt. Dieser muss nun berechnet werden. Da es viel Speicher verbrauchen würde, für jede Abkürzung den ganzen Pfad zu speichern, wird nur jeweils der Index der verwendeten Kante gespeichert. Da ein Knoten meistens nur wenige Ausgangskanten hat, sind für diese Speicherung nur wenige Bits notwendig.

## 2.6 Der Grund-Algorithmus als Pseudocode im Vergleich zum Dijkstra

*input:* source node  $s$  and target node  $t$

*output:* distance  $d(s, t)$

```

1   $d' := \infty$ ;
2   $\text{insert}(\overleftarrow{Q}, s, (0, 0, r_0^-(s)))$ ;  $\text{insert}(\overrightarrow{Q}, t, (0, 0, r_0^+(t)))$ ;
3  while  $(\overleftarrow{Q} \cup \overrightarrow{Q} \neq \emptyset)$  do {
4      select  $\text{direction} \in \{\rightarrow, \leftarrow\}$  such that  $\overline{\overline{Q}} \neq \emptyset$ ;
5       $u := \text{deleteMin}(\overline{\overline{Q}})$ ;
6      if  $u$  has been settled from both directions then
           $d' := \min(d', \overrightarrow{\delta}(u) + \overleftarrow{\delta}(u))$ ;
7      if  $\text{gap}(u) \neq \infty$  then  $\text{gap}' := \text{gap}(u)$  else  $\text{gap}' := r_{\ell(u)}^{\overline{\overline{Q}}}(u)$ ;
8      foreach  $e = (u, v) \in \overline{\overline{E}}$  do {
9          for  $(\ell := \ell(u), \text{gap} := \text{gap}'; w(e) > \text{gap};$ 
               $\ell++, \text{gap} := r_{\ell}^{\overline{\overline{Q}}}(u))$ ; // go "upwards"
10         if  $\ell(e) < \ell$  then continue; // Restriction 1
11         if  $u \in V_{\ell}' \wedge v \in B_{\ell}$  then continue; // Restriction 2
12          $k := (\delta(u) + w(e), \ell, \text{gap} - w(e))$ ;
13         if  $v$  has been reached then  $\text{decreaseKey}(\overline{\overline{Q}}, v, k)$ ;
           else  $\text{insert}(\overline{\overline{Q}}, v, k)$ ;
14     }
15 }
16 return  $d'$ ;

```

Abbildung 2.2: Der Suchalgorithmus als Pseudocode. Unterschiede zum zweiseitigen Dijkstra sind markiert:

Die Zeilennummern hinzugefügter oder veränderter Zeilen sind durch eine Umrandung markiert, Änderungen im Code sind unterstrichen.

Man kann erkennen, dass sich der Algorithmus zum suchen des kürzesten Weges mit dem highway-Hierarchies-Algorithmus zum Algorithmus von Dijkstra nur wenig unterscheidet.

## 3 Abwandlungen des Grund-Algorithmus

Es existieren einige Abwandlungen des Algorithmus, die für spezielle Fälle optimiert sind. Sie werden im folgenden beschrieben.

### 3.1 Highway-Node Routing

Um die Berechnungen, die nur auf dem Fernstraßengraph arbeiten, zu vereinfachen, gibt es mehrere Möglichkeiten. Die erste wäre, einfach auf dem Originalgraph zu rechnen. Eine andere Möglichkeit wäre, eine Distanztabelle anzulegen, in der die Verbindungen zwischen allen Knoten enthalten sind. Diese würde aber extrem groß werden. Es wird ein überlagerter Graph berechnet. Dieser enthält alle Knoten des Fernstraßengraphen und einige Kanten, die sich zwischen zwei Highwayknoten befinden. Es werden aber nicht alle Kanten verwendet, die möglich sind, sondern nur eine möglichst geringe Auswahl. Ein überlagerter Graph auf dem Fernstraßennetzwerk, der Kanten zwischen allen Knoten enthält, entspricht einer Distanztabelle.

Der überlagerte Graph dient aber nicht nur zum Routen auf dem Fernstraßengraph selbst, sondern auch zum allgemeinen Routing. Das Routing beginnt auf dem Originalgraph, bis jeder jeder Zweig des Suchbaumes mindestens einen Knoten des Fernstraßengraphen enthält. Ab diesem Punkt muss nur noch auf dem Fernstraßengraphen weitergeroutet werden.

#### 3.1.1 Statisches Highway-Node-Routing

Auch das Überlagern der Graphen lässt sich verschachteln, d.h. auf einen überlagerten Graph, wird wieder ein überlagerter Graph berechnet, so dass wieder eine Hierarchie entsteht.

Ein Fernstraßengraph entspricht hierbei den Knoten einer höheren Ebene des Highway-Hierarchies-Algorithmus. Es wird, von jedem Punkt aus, die Verbindung zu den Nachbar-knoten im Highway-Network im Originalgraphen gesucht. Wenn ein Highway-Nachbarknoten gefunden wird, wird eine Kante zu diesem mit der Bewertung der Strecke im Originalgraph erzeugt.

#### Bearbeitung einer Anfrage

##### Synchrone Variante :

Es wird ein beidseitiger Algorithmus verwendet, der dem Dijkstra-Algorithmus ent-

spricht, mit dem Unterschied, dass in die nächsthöhere Ebene gewechselt wird, sobald alle Zweige des Suchbaums einen Knoten des nächsthöheren Levels erreicht haben. Außerdem darf ein Suchvorgang nur in die nächsthöhere Ebene wechseln, sobald auch der aus der anderen Richtung dies tut. Hierin besteht die Synchronisierung. Wenn nun die Berechnung der einen Richtung innerhalb einer Ebene sehr lange dauert, kann die andere Richtung nicht weiterrechnen, was zu Performanceverlusten führt.

Diese Überlegung führt zur Einführung der **Asynchronen Variante** :

Der Suchvorgang darf nun in einer höheren Ebene fortgesetzt werden, auch wenn der Suchvorgang in der unteren Ebene noch nicht beendet ist. Der Suchvorgang ist nun nicht beendet, sobald sich die beiden Suchvorgänge treffen, da es sein kann, dass eine Suche, die sich noch in einem tieferen Level befindet, noch einen kürzeren Weg findet.

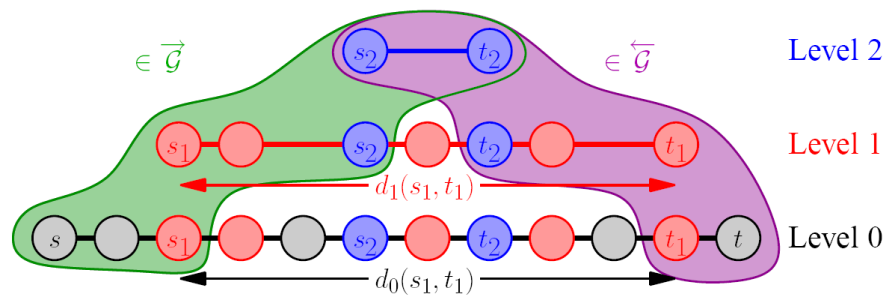


Abbildung 3.1: Darstellung des Suchalgorithmus.

### 3.1.2 Unterschiede und Gemeinsamkeiten zu Highway-Hierarchies

Wie bereits erwähnt, verwendet der Highway-Node-Algorithmus die gleichen Vorberechnungen wie der Highway-Hierarchies-Algorithmus. Dabei wird der Nachbarschaftsradius bei der Berechnung auf 0 gesetzt. Anstelle dessen, wird die Stall-On-Demand-Technik verwendet. Es können viele Teile des Highway-Hierarchie-Codes wiederverwendet werden. Der Teil, der sich mit dem Nachbarschaftsradius beschäftigt, entfällt jedoch, da er immer 0 ist.

### 3.1.3 Dynamisches Highway-Node-Routing

Ein dynamischer Routing Algorithmus kann auf unvorhergesehene Ereignisse wie Staus, o.ä. reagieren und die Route anpassen. Außerdem besteht die Möglichkeit, die Kostenfunktion zu ändern, d.h. die Bewertung der Kanten zu modifizieren.

In ein Server-Client-System, auf dem alle Berechnungen auf dem Server laufen, muss die Datenstruktur geändert werden, wenn sich die Bewertungen temporär ändern. Ein unabhängiges, mobiles System muss nur eine neue Suche starten, und dabei die Änderungen berücksichtigen, die Datenstruktur bleibt unverändert, um keine unnötigen neuen Berechnungen durchzuführen, wenn die Route nicht durch die Störung führt.

### Änderung der Kostenfunktion

Wird die Kostenfunktion nicht allzusehr verändert, wenn z.B. die Route für einen LKW anstatt die eines PKW berechnet werden soll, sind die notwendigen Änderungen gering, da auch ein LKW auf einer Autobahn schneller fahren kann als auf einer Landstraße oder in der Stadt. Es müssen also nur die überlagerter Graphen neu berechnet werden, nicht aber die einzelnen Ebenen aus dem Highway-Hierarchies-Algorithmus.

### Änderung der Bewertung weniger Kanten

Ähnlich wie beim ändern der Kostenfunktion muss beim Ändern einzelner Kanten nicht die komplette berechnung neu durchgeführt werden, sondern nur die Teile, der überlagern Graphen, die eine Änderung in der jeweils unteren Schicht haben, müssen neu berechnet werden.

Im Falle eines mobilen Gerätes, das nur die eigene Route berechnet, muss die Route nur geändert werden, falls die Änderung die Route betrifft.

## 3.2 Many-to-May Shortest Paths

Dieses Verfahren hat ein anderes Ziel als die weiter oben erklärten Algorithmen. Es soll hier eine Distanztabelle erstellt werden. Eine Distanztabelle hat als Zeilen und Spalten jeweils eine Menge von Knoten: die Start- und die Zielmenge. Diese Mengen müssen nicht gleich sein. In der Distanztabelle sollen nun die Entfernungen von allen Startknoten zu allen Zielknoten stehen.

### 3.2.1 Zentrale Idee

Am einfachsten zu implementieren wäre es, von jedem Knoten der Startmenge zu jedem Knoten der Zielmenge eine normale Suche durchzuführen, beispielsweise mit einem der oben beschriebenen Algorithmen. Die gesamte Anzahl an Suchvorgängen wäre also  $|Startmenge| * |Zielmenge|$ . Da dies sehr ineffizient ist wird es in diesem Algorithmus verbessert.

Bei diesem Verfahren wird  $|Startmenge|$  mal eine Rückwärtssuche und  $|Zielmenge|$  mal eine Vorwärtssuche durchgeführt, die alle identisch sind, sofern man nicht direkt nach dem Treffen der beiden Suchen abbricht. Man wird also den gesamten Vorgang jeweils nur einmal durchführen. Es sind also nur noch  $|Startmenge| + |Zielmenge|$  Suchvorgänge notwendig. Es wird also jede Rückwärtssuche nur einmal durchgeführt und das Ergebnis in geeigneter Weise gespeichert.

Zu Beginn wird eine  $|Startmenge| \times |Zielmenge|$  große Tabelle erstellt, in der alle Einträge mit  $\infty$  initialisiert werden. Während der Rückwärtssuche, mit der begonnen wird, wird bei jedem gefundenen Knoten ein Eintrag erstellt, der die Distanz vom jeweiligen Startpunkt der Suche zu diesem Punkt enthält. Bei der Vorwärtssuche wird nun bei jedem Erreichen eines solchen Punktes der gespeicherte Wert und die Entfernung vom aktuellen Startpunkt addiert und geprüft, ob sie kleiner ist, als der Wert, der für das

jeweilige Knotenpaar in der Distanztabelle steht. Ist sie kleiner, wird der Wert in der Tabelle aktualisiert. Auf diese Weise entsteht die gesuchte Distanztabelle.

### 3.2.2 Optimierung für große Distanztabellen

Der Algorithmus kann verbessert werden, indem die Rückwärtssuche nur solange sucht, bis die höchste Suchebene erreicht wird. Die Korrektheit wird dadurch nicht gefährdet, da die Vorwärtssuchen auch diese Punkte erreichen werden.

Dies hat einerseits den Vorteil, dass die Vorwärtssuchen schneller werden, und den weiteren, viel größeren Vorteil, dass viel weniger Einträge erstellt werden müssen, in denen die Entfernung zum jeweiligen Startpunkt steht. So wird die Vorwärtssuche schneller, da weniger Einträge verglichen werden müssen, was bei der Berechnung großer Distanztabellen die meiste Zeit benötigt. Bei großen Distanztabellen wird der Algorithmus noch schneller, wenn die Rückwärtssuche nicht endet, sobald das höchste Level erreicht wird, sondern bereits auf einem niedrigeren Level. Auf diese Weise trifft die Vorwärtssuche zwar auf mehr Knoten, es müssen aber weniger Einträge geprüft werden. Der Einschätzung des Autors nach, endet die Vorwärtssuche, sobald alle Punkte der jeweils höchsten Ebene geprüft wurden.

## 3.3 Transit-Node Routing

Das Transit-Node-Routing befasst sich mit der effizienteren Suche nach Routen zu Zielen, die weit entfernt liegen.

### 3.3.1 Zentrale Idee

Wenn ein Autofahrer ein weit entferntes Ziel erreichen will, wird er seinen Startpunkt i.A. immer über die gleichen Straßen verlassen, die voraussichtlich auf eine Autobahn, oder, in Abhängigkeit der Entfernung des Ziels, auf eine andere große Straße führen.

Die Zielknoten dieser Wege werden im folgenden Zugangsknoten genannt. Die Anzahl der Zugangsknoten ist dabei relativ klein. Es kann also von jedem Startpunkt aus die Entfernung zum nächsten Zugangsknoten gespeichert und auf diese Weise schnell abgerufen werden. Es muss bei diesem Verfahren darauf geachtet werden, dass Start- und Zielpunkt nicht zu nah zusammen liegen, da sonst ein Umweg über Zugangsknoten gemacht wird. Die Entfernung wird vor der Durchführung geprüft, um dies zu verhindern. Dieses Verfahren kann mehrfach angewendet werden, wodurch Zugangsknoten für verschiedene Ebenen entstehen. Sobald eine höhere Ebene verwendet wird, kann dann wieder der passende Zugangsknoten verwendet werden. Sobald keine keine Zugangsknoten in höhere Ebenen mehr existiert oder Start- und Zielpunkt zu nah zusammenliegen, wird ein normaler Routing-Algorithmus verwendet.

### **3.3.2 Berechnung der Zugangsknoten**

Um die Zugangsknoten zu finden, können die gleichen Algorithmen verwendet werden, die beim Highway-Node-Routing verwendet werden, um die Zugänge zur nächsten Ebene zu finden. Ein Zugangsknoten entspricht also dem Punkt, an dem beim Highway-Node-Routing in die nächste Ebene gewechselt wird.

## 4 Messbare Erfolge des Algorithmus

Es können Graphen, die das komplette westeuropäische oder US-Amerikanische Straßennetz enthalten, in akzeptabler Geschwindigkeit verarbeitet werden. Die verwendeten Graphen hatten folgende Größen :

Mit *Memory Overhead* ist der zusätzliche Speicher gemeint, der zusätzlich zum Ori-

Straßennetz	Knoten	Kanten
Europa	18 mio	42,2 mio
USA/CAN	18,7 mio	47,2 mio
USA (Tiger)	24,3 mio	58, mio

ginalgraphen, mit dem Dijkstra (bei speichereffizienter Implementierung) auskommt, benötigt wird. Der Overhead wird dabei in „bytes per Node“ angegeben. Wenn der Gesamt Speicherbedarf angegeben wird, ist der Platz gemeint, der zur Speicherung der Daten der Graphen benötigt wird. Der dynamische Speicherbedarf für Priority-Queues, Listen usw. wird dabei nicht einbezogen. Bei der Berechnung der Zeit für eine Anfrage wird der Durchschnitt aus 10 000 zufälligen Anfragen gebildet, sofern nicht anders angegeben.

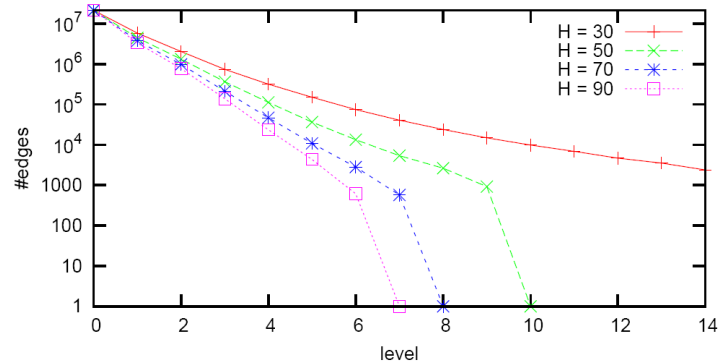


Abbildung 4.1: Dieses Diagramm zeigt die Anzahl der sich ergebenden Ebenen und die Kanten, die diese noch haben, wenn der Nachbarschaftsradius H variiert wird.

Abbildung 4.1 zeigt, wieviele Ebenen maximal entstehen, und wieviele Kanten diese haben, wenn der Nachbarschaftsradius variiert wird. Bei niedrigen Nachbarschaftsradius sind die unterschiede zwischen den Ebenen wesentlich geringer, als bei höheren Nachbarschaftsradii.

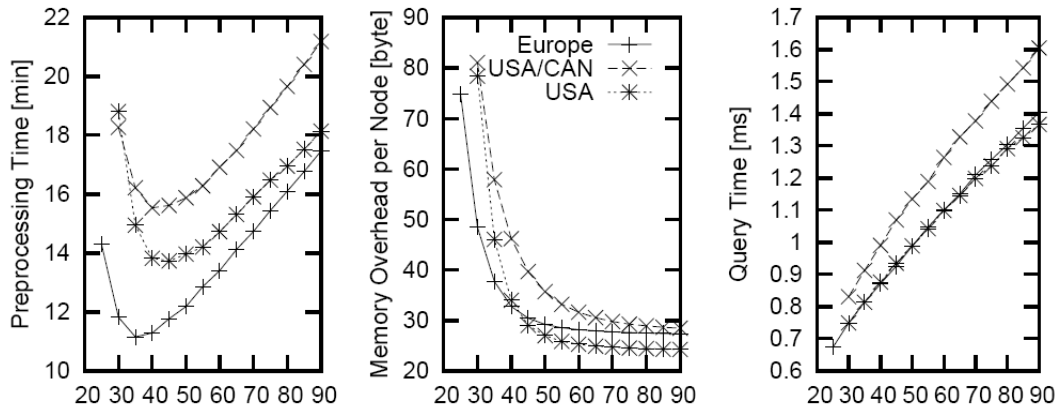


Abbildung 4.2: Diese Diagramme zeigen die Preprocessingzeit, den Speicher-Overhead und die Dauer einer Suchanfrage unter Veränderung des Nachbarschaftsradius. Es wurden verschiedene Graphen verwendet, die jeweils in die Diagramme eingezeichnet sind.

Aus Abbildung 4.2 wird ersichtlich, wie sich die Preprocessingzeit, der Speicher-Overhead und die Dauer einer Suchanfrage verhalten, wenn der Nachbarschaftsradius während des Preprocessing variiert wird.

# levels	PREPROC.		QUERY	
	time [min]	over- head	time [ms]	#settled nodes
6	12	48	0.75	709
7	10	34	0.93	852
8	10	30	1.14	991
9	10	30	1.35	1 123
10	10	29	1.54	1 241
11	10	29	1.67	1 326

Abbildung 4.3: Performance des Preprocessing und der Anfragen, bei variiert Anzahl von Ebenen.

Die Tabelle in Abbildung 4.3 zeigt, dass sich die Preprocessingzeit unwesentlich ändert, wenn mehr Ebenen erzeugt werden, da der Zeitbedarf zur Berechnung der oberen Ebenen sehr gering ist. Die Dauer einer Suchanfrage steigt bei einer höheren Anzahl an Ebenen, da auf der höchsten Ebene eine Distanztabelle verwendet wird und auf den höheren Ebenen nicht mehr geroutet werden muss, die größere Distanztabelle ist auch der Grund für den steigenden Memory-Overhead bei weniger Ebenen.

level	#nodes	shrink factor	#edges	shrink factor	average degree
0	18 029 721		42 199 587		2.3
1	2 739 732	6.6	11 884 352	3.6	4.3
2	423 635	6.5	2 226 290	5.3	5.3
3	118 844	3.6	780 147	2.9	6.6
4	35 617	3.3	292 630	2.7	8.2
5	11 944	3.0	117 123	2.5	9.8
6	4 364	2.7	49 290	2.4	11.3
7	1 817	2.4	23 108	2.1	12.7
8	864	2.1	12 434	1.9	14.4
9	454	1.9	6 579	1.9	14.5
10	249	1.8	4 029	1.6	16.2
11	146	1.7	2 459	1.6	16.8

Abbildung 4.4: die Tabelle zeigt die Größe der Graphen in den verschiedenen Ebenen.

Die Tabelle in Abbildung 4.4 zeigt, dass das Erzeugen neuer Ebenen in den ersten Schritten am effektivsten ist (hoher shrink factor), da dort die meisten Knoten und Kanten eingespart werden können. In den höheren Ebenen sinkt der Effekt erheblich.

## 4.1 Verwendete Hard- und Software

Zur Berechnung wurde ein AMD Opteron-Prozessor 270 mit 2 GHz, 8 GB Arbeitsspeicher und  $2 \times 1$  MB L2 Cache verwendet. Das Betriebssystem war SuSE Linux 10.0 (Kernel 2.6.13), als Compiler wurde GNU C++ compiler 4.0.2 mit Optimierungslevel 3 verwendet.

## 5 Quellen

D. Schultes - Route Planning in Road Networks (2008)  
de.wikipedia.org - Dijkstra-Algorithmus (11/09)  
de.wikipedia.org - Traffic\_Message\_Channel (11/09)