Transit Nodes und Dynamic Highway Node Routing

Christian Kaptur

17. Dezember 2008

Gliederung

1. Einführung
   - vorherige Arbeiten
2. Transit-Node Routing
   - Allgemein: Transit-Node Routing
   - Implentation
     - Eine Grid-basierte Implementation
     - Eine Highway Hierarchie-basierte Implemantation
3. Dynamic Highway-Node Routing
   - Server-Scenario
   - Mobile-Scenario
   - Experimente

Einführung

Einführung

- Betrachtung eines Graphen $G = (V, E)$ mit $n = |V|$, $m = |E|$
- ein Graph der US-Straßenkarte hat ca. $24mio$ Knoten und $58mio$ Kanten
- bei zufälligen Knoten hat Dijsktra eine Laufzeit von $O(n + \log(n + m))$
- aus diesem Grund Vorberechnungen zur Optimierung der Laufzeit

Einführung - vorherige Arbeiten

vorherige Arbeiten

Dijkstra

- Suche nach kürzestem Pfad von genau einer Quelle aus
- Aufspannung eines Baumes *shortest-path tree*
- Unterscheidung von *reached*, *unreached* und *settled* Knoten

vorherige Arbeiten

seperator based multi-level method

- Aufteilung des Graphen in mehrere Teilstücke durch Entfernung einer geringen Anzahl an Knoten (*separator nodes*)
- man erhält einen *next level*-Graph, wobei die Knoten als Transitknoten verwendet werden können
- allerdings ist so die Zahl der Transitknoten um das 8-fache höher als in den später vorgestellten Algorithmen

vorherige Arbeiten

Highway Hierarchies

- kommerzielle Systeme nutzen die Straßenkategorisierung, um Anfragen zu beschleunigen
- es existiert eine Straßenhierarchie (Autobahn, Landstraße etc.)
- liegen 2 Städte weit entfernt auseinander, so wird der meiste Weg auf der Autobahn zurückgelegt
- so kann die Suche in einem Graphen eingeschränkt werden - später mehr

Transit-Node Routing

Einführung

Transitknoten (transit nodes)

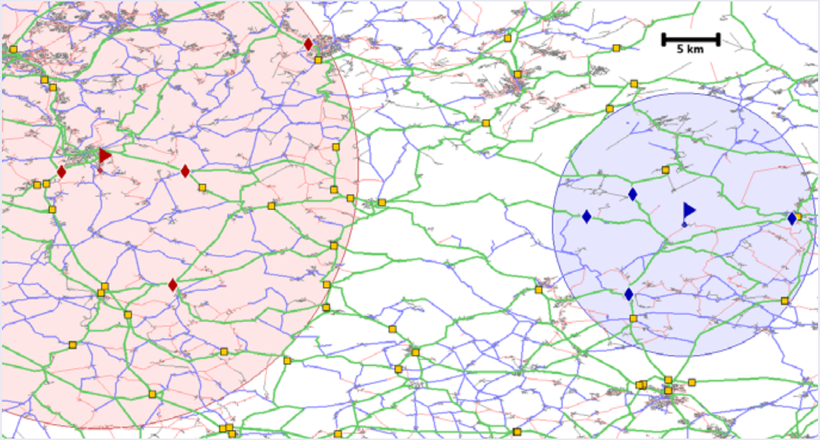
- liegen 2 beliebige Knoten weit genug auseinander, so verläuft ihr kürzester Pfad durch mindestens einen Transitknoten

Einführung

Transitknoten (transit nodes)

- Transit-Node Routing basiert auf 2 Beobachtungen
- für eine immens große Karte existiert eine vergleichbar kleine Menge an Transitknoten
- für jeden Knoten existiert nur eine geringe Anzahl an Transitknoten (*access*-Knoten), die erst ab einer bestimmten Entfernung auftreten

Implentation

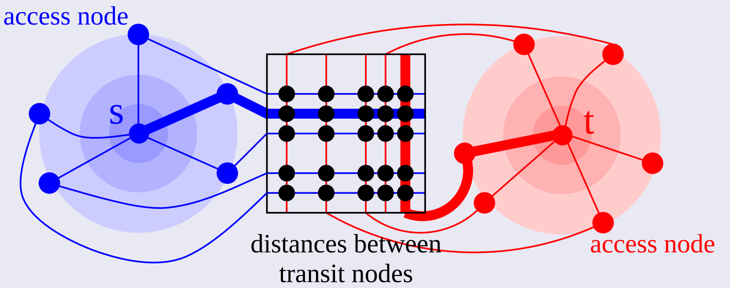


Einführung

Grundidee

- Vorberechnung der Abstände der Transitknoten untereinander und der Abstände jedes Knoten zu seinen access-Knoten
- der kürzeste Pfad zwischen 2 beliebigen Knoten, die weit genug auseinander liegen, kann so durch wenige Tabellen-*lookups* berechnet werden

Implentation



allgemein und formal

Transit-Node Routing

betrachte

- eine Menge Transitknoten $T \subset V$
- ein *access mapping* $A : V \rightarrow 2^T$, das jedem Knoten seine access-Knoten liefert
- ein *locality*-Filter $L : V \times V \rightarrow \{true, false\}$, der entscheidet, ob eine s-t-Anfrage lokal ist, d.h. der kürzeste Pfad keinen Transitknoten enthält

Transit-Node Routing

Distanzen

- wenn $\neg L(s,t)$, so gilt $d(s,t) = \min\{d(s,u)+d(u,v)+d(v,t) | u \in A(s), v \in A(t)\}$, andernfalls nutze einen anderen Suchalgorithmus
- um möglichst effizient zu sein, benötigt man eine gute Auswahl von (T, A, L)

Transit-Node Routing

zweiter Layer

- man legt einen zweiten *Layer* an, um die restlichen lokalen Anfragen schnell bearbeiten zu können
- mit $T \subset T_2$, $A_2 : V \to 2^{T_2}$ und L_2
- es soll gelten für $\neg L_2(s,t)$: $d(s,t) = \min\{d(s,u) + d(u,v) + d(v,t) | u \in A_2(s), v \in A_2(t)\}$
- benötigt wird nur zusätzlich $\{d(u,v) | u, v \in T_2 \wedge L(u,v)\}$

Transit-Node Routing

Computing Access Nodes: Backward Approach

- führe von jedem Transitknoten v solange den Rückwärts-Dijkstra-Algorithmus durch, bis der Baum von Knoten $w \in T$ überdeckt (*covered*) wird
- speichere v als access-Knoten für jeden Knoten u auf dem kürzesten Pfad von v, der nicht über einen anderen Transitknoten führt
- man bildet nun den Transit-Graph $G[T]$, in dem man Kante (v, w) mit Gewicht $d(v, w)$ zu E_T hinzufügt
- so erhält man für jeden Knoten seine access-Knoten und speichert sich mit Hilfe von $G[T]$ die Distanzen in einer Tabelle

Transit-Node Routing

Computing Access Nodes: Backward Approach - Anmerkung

- die Suche bei höheren Layer ist ähnlich
- die Suche kann schon gestoppt werden, wenn alle Pfade von Knoten eines niederen Layer überdeckt werden
- denn nur die lokalen Distanzen werden für die späteren Anfragen von Bedeutung sein

Transit-Node Routing

Computing Access Nodes: Forward Approach

- führe Dijstra-Algorithmus für alle Knoten $u \in V$ durch, bis alle kürzesten Pfade von Transitknoten überdeckt (*covered*) werden und nehme diese Transitknoten als access-Knoten von u
- diese Methode ist ohne weitere Anpassung viel ineffizienter als die vorherige
- aus diesem Grund ist es einfacher, zuerst $A(v)$ für Knoten $v \in T_2$ und $A_2(u)$ für beliebigen Knoten $u \in V$ zu bestimmen
- dann gilt: $A(u) = \bigcup_{v \in A_2(u)} A(v)$

Transit-Node Routing

Outputting Shortest Path

- gesucht ist aber nicht nur die Entfernung, sondern der kürzeste Pfad zwischen s, t
- Algorithmus:
  - suche nach einer Kante (s, u), so dass $d(s, u) + d(u, t) = d(s, t)$ und gebe (s, u) aus
  - wiederhole mit der Suche nach dem kürzesten Pfad von (u, t)

Transit-Node Routing

Outputting Shortest Path - Optimierung

- Annahme: der Pfad, der über die access-Knoten $x \in A(s)$ und $y \in A(t)$ verläuft mit $d(s,x) + d(x,y) + d(y,t)$ ist minimal
- man spart so die Suche nach den richtigen access-Knoten und die Pfadsuche braucht zwischen x, y nur $G[T]$ zu betrachten
- speichert man während den Vorberechnungen die kürzesten Pfade aus G zwischen Transitknoten ab, erfolgt die Wegsuche zwischen weit auseinander liegenden Knoten genauso schnell wie die bei nah beieinander liegenden

Transit-Node Routing - Implementation

Implentation

- es gibt 2 Möglichkeiten Transit-Node Routing zu implemtentieren
- man kann mit einem locality-Filter L starten, um dafür eine Menge T zu finden (Grid basierend)
- oder man startet mit einer Menge T und muss dafür einen effizienten lokalen Filter L berechnen (Highway Hierachies basiert)

Eine Grid-basierte Implementation

Grid basierend

das Grid

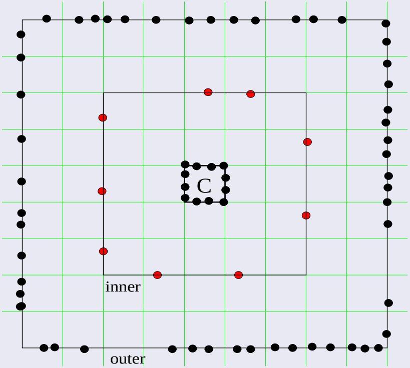
- betrachte das kleinste Quadrat, das alle Knoten aus V umfasst
- für eine Zahl g teilt man dieses Quadrat in $g \times g$ gleich große Quadrate

Grid basierend

Transitknoten

- sei S_{inner} ein Quadrat aus 5×5 Zellen und S_{outer} aus 9×9 Zellen um einen Zelle C herum
- sei E_C die Menge an Kanten, mit einem Knoten in und einem außerhalb von C und füge genau einen Knoten jeder Kante zu V_C hinzu
- mache gleiches, um V_{outer} und V_{inner} zu erhalten

Grid basierend



Grid basierend

Transitknoten

- die Menge der access-Knoten für C sind die Knoten $v \in V_{inner}$ mit der Eigenschaft, dass ein kürzester Pfad von einem Knoten aus V_C zu Knoten aus V_{outer} v enthält
- die Menge der Transitknoten ist die Vereinigung der access-Knoten Mengen über alle Zellen
- liegen 2 Knoten 4 Zellen auseinander, so verläuft ihr kürzester Pfad über solch einen Transitknoten
- ist ein Knoten für einige Zellen ein access-Knoten, so ist er auch das von vielen anderen Zellen

Grid basierend

Berechnung

- naiv könnte man für jede Zelle C den kürzesten Pfad von jeden Knoten aus V_C zu V_{outer} bestimmen
- dann jeden Knoten aus V_{inner} markieren, der wenigsten einmal auftauchen
- doch so eine Berechnung würde selbst bei einem 128×128 Grid mehrere Tage dauern
- aus diesem Grund verwendet man einen sog. sweep-line Algorithmus

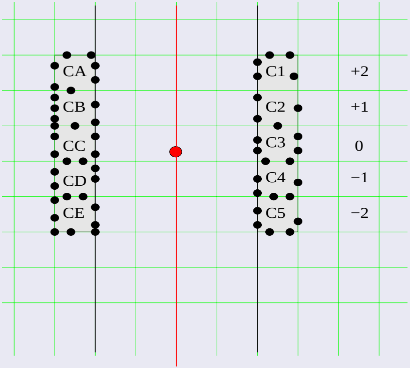
Grid basierend

sweep-line Algorithmus

- teile die Karte mit Hilfe von vertikalen Linien in mehrere Teile auf und betrachte eine vertikale Linie nach der anderen
- sei v ein Knoten, dessen Kante die vertikale Linie schneidet
- sei C_{left} die Menge der Knoten, die von v aus genau 2 Zellen horizontal und maximal 2 Zellen vertikal entfernt sind
- definiere gleichermaßen C_{right}

Grid basierend

sweep-line Algorithmus



Grid basierend

sweep-line Algorithmus

- führe den Dijkstra-Algorithmus für alle Knoten v der Linie so lange durch, bis alle Knoten am Rand von C_{right} und C_{left} gefunden wurden und speichere deren Entfernungen zu v
- betrachte alle Paare (v_L, v_R) mit einer maximalen vertikalen Distanz von 4 und wähle das v mit minimaler Distanz $d(v_L, v) + d(v, v_R)$
- es werden nun den jeweiligen Zellen von v_R und v_L, v als Transitknoten zugewiesen
- die acces-Knoten eines beliebigen Knoten u sind dann die Transitknoten der Zelle die u enthält

Grid basierend

Anfragen

- bei Anfrage prüft man nun, ob Start- und Zielknoten 4 Zellen entfernt sind
- falls ja, muss der kürzeste Pfad über mindestens einen Transitknoten verlaufen und der vorher beschriebene Algorithmus kann genutzt werden
- andernfalls wird eine lokale Suche, die aufgrund der Nähe der Knoten nun nicht mehr so lange Laufzeiten hat, verwendet

Grid basierend

Multi-Level Grid

- es existiert trade-off zwischen Größe des Grids und Anzahl der lokalen Anfragen
- bei einer Gridgröße von 64×64 ist z.B. 10% der Anfragen lokal
- für ein Grid mit 1024×1024 Zellen sind gerade mal $0,1\%$ der Anfragen lokal
- ABER: Anzahl der Transitknoten und die Berechnungen sind so immens, dass es nicht mehr schnell von einer einzelnen Maschine berechenbar ist

Grid basierend

Multi-Level Grid

	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{T}\rvert \times \lvert\mathcal{T}\rvert$/node	avg. $\lvert A\rvert$	% global queries	preprocessing
64×64	2 042	0.1	11.4	91.7%	498 min
128×128	7 426	1.1	11.4	97.4%	525 min
256×256	24 899	12.8	10.6	99.2%	638 min
512×512	89 382	164.6	9.7	99.8%	859 min
$1\,024 \times 1\,024$	351 484	2 545.5	9.1	99.9%	964 min

Grid basierend

Multi-Level Grid

- um eine kleine Anzahl an lokalen Anfragen und eine kleine Anzahl an Transitknoten zu erhalten, Einführung einer Hierarchie des Grids
- das erste Level ist ein 128×128 Grid und man berechnet die Transitknoten für dieses Grid wie beschrieben
- das zweite Level ist ein 256×256 Grid, man berechnet auch die Transitknoten wie beschrieben, speichert aber nur Distanzen von solchen Knotenpaaren, die in Bezug auf das 128×128 Grid lokal sind
- bei Anfrage betrachtet man erst das grobe Grid; ist die Anfrage lokal, so betrachtet man das feine Grid; ist die Anfrage immer noch lokal, benutzt man ein lokales Suchverfahren

Grid basierend

Experimente

- getestet wurde mit einem US-Straßenkarten-Graph auf einer Dual-Opteron-Maschine mit 8 Gbyte RAM

non-local (99%)	local (1%)	all queries	preprocessing	space per node
12 μs	5112 μs	63 μs	20 h	**21 bytes**

Eine Highway Hierarchie-basierte Implemantation

Highway Hierarchie basierend

Highway Hierarchie

- eine Highway Hierarchie enthält mehrere Level G_0, G_1, ..., G_L
- G_0 entspricht dem Original-Graph, G_1 erhält man aus dem Highway-Netz von Level 0, G_2 berechnte sich aus dem *core* G'_1 aus Level 1 usw.
- wenn man festlegt, welcher Knoten, beim Dijkstra-Algorithmus von s aus, bei 2 gleichbewerteten als Erstes entnommen wird, erhält man eine feste Reihenfolge
- damit erhält man den sogenannten Dijkstra-Rang $rk_s(v)$

Highway Hierarchie basierend

Highway Hierarchie

- für jeden Knoten v definiert man eine Nachbarschaft $N(v)$
- ein Highway-Netz eines Grpahen $G = (V, E)$ wird derfiniert über seine Kantenmenge
- eine Kante $(u, v) \in E$ gehört zum Highway-Netz, wenn es Knoten $s, t \in V$ existieren, so dass die Kante (u, v) im kürzesten Pfad $< s, .., u, v, .., t >$ mit der Eigenschaft $v \notin N(s)$ und $u \notin N(t)$

Highway Hierarchie basierend

Transitknoten

- Knoten der hohen Level der Highway Hierarchie haben die Eigenschaft, dass sie im kürzesten Pfad vieler weit genug entfernten Knoten enthalten sind
- für ein Level K verwendet man die Knoten des Highway Netzwerkes als Transitknoten
- bisherige Versuchen verwendeten maximal Level 4 und 5

Highway Hierarchie basierend

Transitknoten

- man kann auch verschiedene Transitknoten-Layer einführen
- z.B. als 1. Layer $K_1 := K$, als 2. Layer $K_2 = [K/2]$ und Layer 3 (soweit vorhanden) $K_3 = [K/4]$
- Achtung: Layer $\neq$ Level

Highway Hierarchie basierend

Idee des locality-Filter

- wollen feststellen, für welche Paare s, t gilt, $d(s, t)$ kann nicht in niedrigerem Layer berechnet werden
- für jedes Paar wählen wir bestimmten Knoten p (*witness*) aus dem kürzesten Pfad (s, t) aus
- witness-Knoten kann vererbt werden
- wollen wissen, ob $L(s, t) = true$, so prüfen wir, ob ein gemeinsamer witness-Knoten existiert

Highway Hierarchie basierend

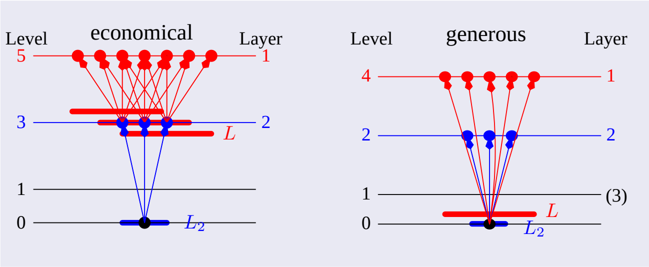
Idee des locality-Filter

- sei $p(s,t)$ ein bestimmter Knoten zwischen dem kürzesten Pfad
- sei $l(u) := \min\{l | u \in T_l\}$
- sei $K_l(s)$ die Funktion, die jedem Knoten seine witness-Knoten liefert, folgendermaßen definiert:
    - für $l > l(s) + 1$: $K_l(s) := \emptyset$
    - für $l = l(s) + 1$: $K_l(s) := \{p(s,t) | t \in V \wedge l(s) = l(t) \wedge d(s,t) < d_{<l}(s,t)\}$
    - für $l < l(s) + 1$: $K_l(s) := \bigcup_{u \in A_{l(s)}(s)} K_l(u)$
- $L_l(s,t) := \bigvee_{k>l} (K_k(s) \cap K_k(t) \neq \emptyset)$

Highway Hierarchie basierend

access-Knoten

- um access-Knoten zu bestimmen, wird die Forward-Methode mit dem Unterschied der Highway-Suche, verwendet
- zwei Möglichkeiten, eine ökonomische und eine großzügige Variante



Highway Hierarchie basierend

many-to-many routing

- wird verwendet, um die Distanztabelle zu bestimmen
- über alle Layer wird zuerst eine Rückwärtssuche von allen Transitknoten v aus gestartet und es werden Informationen in Form von $(u, v, d(u, v))$ gespeichert
- dann wird ein Vorwärtsscan von allen Transitknoten v' ausgeführt und mit Hilfe der Einträge $d(v', u) + d(u, v)$ berechnet

Highway Hierarchie basierend

Anfragen

- für ein Paar (s, t) wird zuerst $A(s)$ und $A(t)$ ermittelt
- dann wird im obersten Level in die Distanztabelle geschaut und $d(s, t)$ ermittelt
- wenn $\neg L(s, t)$, dann sind wir fertig; andernfalls machen wir das Selbe für den zweiten Layer
- wenn $L_2(s, t)$ auch wahr, dann führe bidirektionale Highway-Suche durch, die gestoppt wird wenn die äußeren Grenzen Layer-3 verlassen

Highway Hierarchie basierend

kürzester Pfad

- um den kürzesten Pfad zwischen Knoten s, t zu bestimmen, wird zuerst der Layer-i bestimmt, der verwendet wird, um die Distanz zu ermitteln
- dann wird der Pfad von s zu seinem access-Knoten u, den access-Knoten v von t und der Pfad zwischen v und u im Layer-i bestimmt
- wenn s und u gegeben sind, wird mit Hilfe der Tabellen nach Kante (s, s') mit $d(s, s') + d(s', u) = d(s, u)$ gesucht und ausgegeben
- für den Fall, u ist nicht access-Knoten von s', muss man alle acces-Knoten von s' betrachten und den Knoten ermitteln mit $d(s, s') + d(s', u') + d(u', u) = d(s, u)$

Highway Hierarchie basierend - Experimente

Highway Hierarchie basierend

Experimente

	metric	variant	layer 1 $\|\mathcal{T}\|$	layer 1 $\|$table$\|$ [$\times 10^6$]	layer 1 $\|A\|$	layer 2 $\|\mathcal{T}_2\|$	layer 2 $\|$table$_2\|$ [$\times 10^6$]	layer 2 $\|A_2\|$	space [B/node]	time [h]
USA	time	eco	12 111	147	6.1	184 379	30	4.9	111	0:59
		gen	10 674	114	5.7	485 410	204	4.2	244	3:25
	dist	eco	15 399	237	17.0	102 352	41	10.9	171	8:58
EUR	time	eco	8 964	80	10.1	118 356	20	5.5	110	0:46
		gen	11 293	128	9.9	323 356	130	4.1	251	2:44
	dist	eco	11 610	135	20.3	69 775	31	13.1	193	7:05

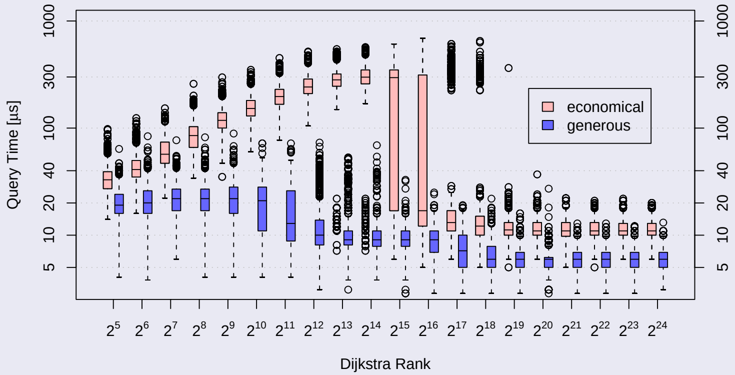
Highway Hierarchie basierend

Experimente

	metric	variant	query time
USA	time	eco	11.5 μs
		gen	4.9 μs
	dist	eco	87.5 μs
EUR	time	eco	13.4 μs
		gen	5.6 μs
	dist	eco	107.4 μs

Highway Hierarchie basierend

Experimente



Dynamic Highway-Node Routing

Dynamic Highway-Node Routing

Einleitung

- Vorstellung von dynamischen Methoden der Routenplanungen für änderbare Kantengewichte, basierend auf Highway Hierarchies
- im Server-Scenario wird durch Aktualisierung der Datenstruktur auf Änderungen reagiert
- im Mobilen-Scenario soll keine Zeit für das Updaten des Graphen an irrelevanten Zweigen verschwendet werden

Dynamic Highway-Node Routing - Server-Scenario

Server-Scenario

Einleitung

- bei Änderungen der Gewichte der Kanten behalten wir die Highway-Knoten-Menge bei
- es soll nicht die gesamte Konstruktion von vorn beginnen, sondern nur an Knoten, die wirklich betroffen sind
- denn ein Pfad von nicht betroffenen Knoten bleibt auch durch Veränderung der Kostenfunktion unverändert

Server-Scenario

Algorihtmus

- sei A^l_u die Menge an Knoten, die durch eine Veränderung einer Kante von u im Level-l mit betroffen sind
- mit folgendem Algorithmus kann dann auf Veränderung reagiert werden:

input: set of edges E^m with modified weight;
define set of modified nodes: $V^m_0 := \{u \mid (u, v) \in E^m\}$;
foreach level $\ell, 1 \leq \ell \leq L$, **do**
  $V^m_\ell := \emptyset$; $R_\ell := \bigcup_{u \in V^m_{\ell-1}} A^\ell_u$;
  foreach node $v \in R_\ell$ **do**
    repeat construction step from v;
    if something changes, put v to V^m_ℓ;

Dynamic Highway-Node Routing - Mobile-Scenario

Mobile-Scenario

Erweiterung

- Betrachtung einer schnellen Variante des vorherigen Algorithmus: in den letzten beiden Zeilen bleibt „put v to V_l^m"
- Definition eines *reliable level* $r(u) := \min\{i - 1 | u \in R_i\}$ mit $\min \emptyset := \infty$

Mobile-Scenario

Erweiterung

- um bei der Suche nach dem kürzesten Pfad korrekte Resultate zu erhalten, muss der Suchalgorithmus angepasst werden
- 1. bei der Suche wird keine Kante (u, v) ausgelassen, wenn Level $l > r(u)$
- 2. wenn die Suche einen Knoten u erreicht, wobei $l > r(u)$ ist, dann wird die Suche an diesem Knoten zu Level $r(u)$ *downgraded*
- das heißt: wird ein Knoten erreicht, an dem wir die Konstruktion der Hierarchie von vorn beginnen müssten, führt man die Suche auf einem kleineren, aber unberührten Level weiter

Experimente

Experimente

Server-Scenario

|change set|	any road type				motorway				national		regional		urban	
	+	−	∞	×	+	−	∞	×	+	∞	+	∞	+	∞
1	2.7	2.5	2.8	2.6	40.0	40.0	40.1	37.3	19.9	20.3	8.4	8.6	2.1	2.1
1000	2.4	2.3	2.4	2.4	8.4	8.1	8.3	8.1	7.1	7.1	5.3	5.3	2.0	2.0

Experimente

Mobile-Scenario

|change set|	affected queries	#settled nodes absolute	#settled nodes relative	query time [ms] init	query time [ms] search	query time [ms] total
1	0.6 %	2 347	(1.7)	0.3	2.0	2.3
10	6.3 %	8 294	(5.9)	1.9	7.2	9.1
100	41.3 %	43 042	(30.4)	10.6	36.9	47.5
1 000	82.6 %	200 465	(141.8)	62.0	181.9	243.9
10 000	97.5 %	645 579	(456.6)	309.9	627.1	937.0