

Algorithmen für Routenplanung

4. Sitzung, Sommersemester 2013

Thomas Pajor | 29. April 2012

INSTITUT FÜR THEORETISCHE INFORMATIK · ALGORITHMIK · PROF. DR. DOROTHEA WAGNER



A*-Algorithmus

- Benutze Potentialfunktion $\pi : V \rightarrow \mathbb{R}$ mit

$$\text{len}(u, v) - \pi(u) + \pi(v) \geq 0 \quad \forall (u, v) \in E$$

- Keys in der Priority-Queue sind $d[u] + \pi(u)$ für alle $u \in V$
- Wenn $\pi(t) = 0$, dann $\pi(u) \leq d(u, t)$
- Je besser untere Schranke für $d(u, t)$ desto besser die Zielrichtung

A* \leftrightarrow Dijkstra

- Definiere $G_\pi = (V, E, \text{len}_\pi)$ mit

$$\text{len}_\pi(u, v) := \text{len}(u, v) - \pi(u) + \pi(v)$$

- Dann: Dijkstra auf G_π entspricht A* auf G

Frage: Was ist eine gute Potentialfunktion für A*?

Idee: Benutze Landmarken und die Dreiecksungleichung

Vorbereitung:

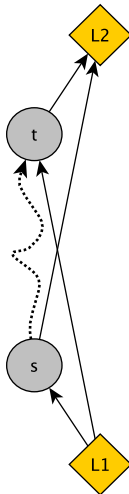
- Wähle $L \subseteq V$ ($L \approx 16$) Landmarken
- Berechne Distanz von und zu allen Landmarken

Anfrage:

- Benutze

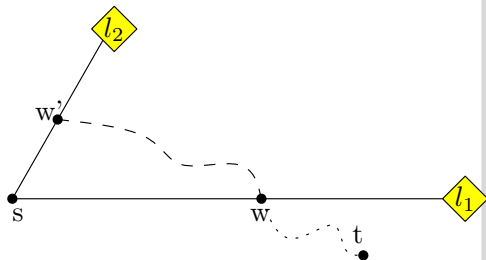
$$\pi(u) := \max_{\ell \in L} \{ \max\{d(\ell, t) - d(\ell, u), d(u, \ell) - d(t, \ell)\} \}$$

als gültiges Potential



Problem:

- Landmarken können Suche in die falsche Richtung ziehen
- Auswertung von vielen Landmarken erzeugt großen overhead

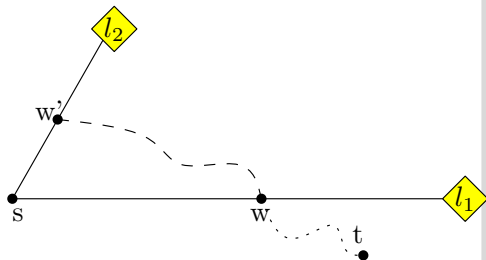


Lösung:

- wähle während Initialisierung eine Teilmenge von L als aktiv
- diese, für die $\pi_t^l(s)$ maximal sind
- solche Landmarken liefern die besten Schranken und sind (hoffentlich) die besten
- Führe die Suche nur mit aktiven Landmarken aus

Problem:

- Landmarken können Suche in die falsche Richtung ziehen
- Auswertung von vielen Landmarken erzeugt großen overhead



Verbesserung:

- Benutze Heuristik um während der Suche die Wahl der aktiven Landmarken anzupassen.
- Starte mit nur zwei Landmarken und füge während der Suche weitere hinzu..
- Schlechte Landmarken können auch wieder inaktiv gesetzt werden.

Erster Ansatz:

- $2 \cdot |L|$ 32-bit Vektoren der Größe n
- Problem: viele Cache-Misses

Besser:

- 1 64-bit Vektor der Größe $|L| \cdot n$
- speicher Distanz von und zu Landmarke in einem 64-bit integer
- Zugriff auf Knoten mit id k und Landmarken-nummer l in Segment $|L| \cdot k + l$
- dadurch deutlich erhöhte Lokalität
- beschleunigt die Anfragezeit um ca. einen Faktor 4

Der Suchraum von Dijkstra's Algorithmus

$$V_d(s, t) = \{v \in V \mid \text{dist}(s, v) \leq \text{dist}(s, t)\}$$

ist ein graphentheoretischer Kreis.

Eine Intuition für den ALT-Suchraum

Der Suchraum von ALT ist

$$V^L(s, t) \subseteq \{v \in V \mid \text{dist}(s, v) + \pi^L(v) \leq \text{dist}(s, t)\}$$

mit Potentialen

$$\pi_t^{l+}(u) = d(l, t) - d(l, u)$$

$$\pi_t^{l-}(u) = d(u, l) - d(t, l)$$

Der Suchraum von ALT ist

$$V^L(s, t) \subseteq \{v \in V \mid \text{dist}(s, v) + \pi^L(v) \leq \text{dist}(s, t)\}$$

mit Potentialen

$$\pi_t^{l+}(u) = d(l, t) - d(l, u)$$

$$\pi_t^{l-}(u) = d(u, l) - d(t, l)$$

Wir betrachten jedes $l \in L$ einzeln

$$V_{l-}(s, t) = \{v \in V \mid \text{dist}(s, v) + \text{dist}(v, l) \leq \text{dist}(s, t) + \text{dist}(t, l)\}$$

$$V_{l+}(s, t) = \{v \in V \mid \text{dist}(s, v) - \text{dist}(l, v) \leq \text{dist}(s, t) - \text{dist}(l, t)\}$$

$$V_I(s, t) = \{v \in V \mid \text{dist}(s, v) + \text{dist}(v, t) \leq \text{dist}(s, t) + \text{dist}(t, I)\}$$

ist eine graphentheoretische Ellipse.

$$V_{I-}(s, t) = \{v \in V \mid \text{dist}(s, v) + \text{dist}(v, I) \leq \text{dist}(s, t) + \text{dist}(t, I)\}$$

ist eine graphentheoretische Ellipse.

$$V_{I+}(s, t) = \{v \in V \mid \text{dist}(s, v) - \text{dist}(I, v) \leq \text{dist}(s, t) - \text{dist}(I, t)\}$$

ist eine graphentheoretische Hyperbel.

$$V_{I-}(s, t) = \{v \in V \mid \text{dist}(s, v) + \text{dist}(v, I) \leq \text{dist}(s, t) + \text{dist}(t, I)\}$$

ist eine graphentheoretische Ellipse.

$$V_{I+}(s, t) = \{v \in V \mid \text{dist}(s, v) - \text{dist}(I, v) \leq \text{dist}(s, t) - \text{dist}(I, t)\}$$

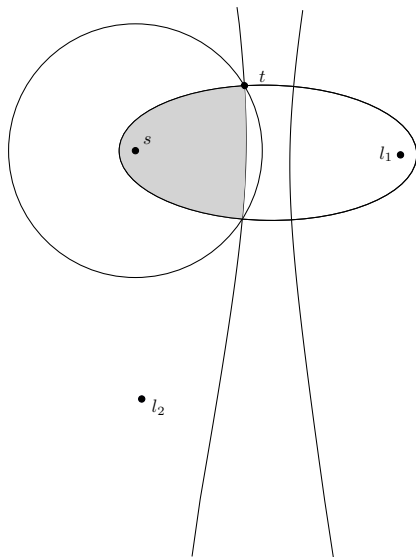
ist eine graphentheoretische Hyperbel.

Der Suchraum

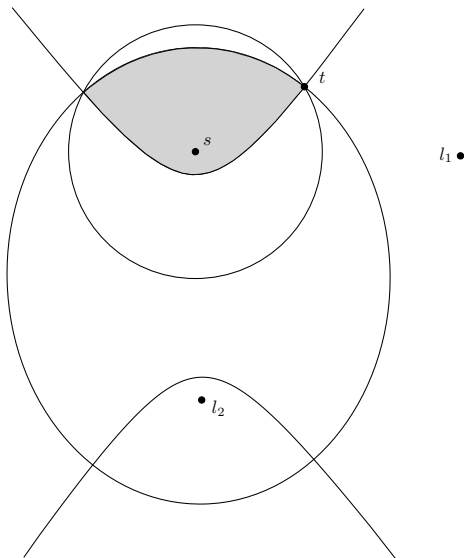
$$V^L(s, t) \subseteq \{v \in V \mid \text{dist}(s, v) + \pi^L(v) \leq \text{dist}(s, t)\}$$

ist die Schnittmenge aus den Ellipsen und Hyperbeln über alle Landmarken in L .

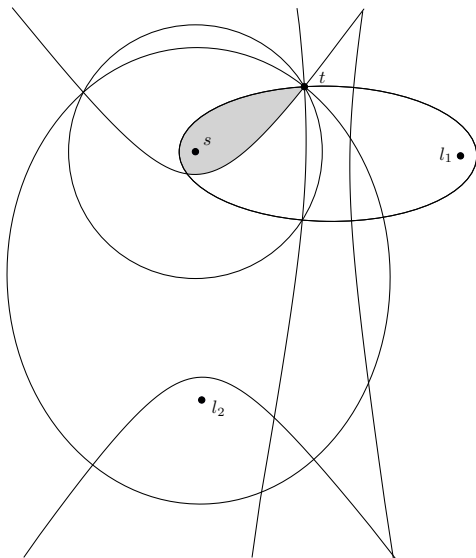
Eine Intuition für den ALT-Suchraum



Eine Intuition für den ALT-Suchraum



Eine Intuition für den ALT-Suchraum



Bidirektionaler A^*



Bidirektionaler A*



Erster Ansatz:

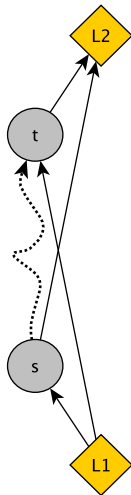
- benutze Vorwärtspot π_f und Rückwärtspot π_b

$$\pi_f(u) = \max_{\ell \in L} \{ \max \{ d(\ell, t) - d(\ell, u), d(u, \ell) - d(t, \ell) \} \}$$

$$\pi_b(u) = \max_{\ell \in L} \{ \max \{ d(\ell, u) - d(\ell, s), d(s, \ell) - d(u, \ell) \} \}$$

Problem:

- Suchen operieren auf unterschiedlichen Längenfunktionen
- konservatives Abbruchkriterium:
 - stoppe erst, wenn $\minKey(\vec{Q}) > \mu$ oder $\minKey(\overleftarrow{Q}) > \mu$



Zweiter Ansatz:

- Wann operieren Suchen auf dem gleichen Graphen?
- wenn

$$\begin{aligned}\text{len}_{\pi_r}(v, u) &= \text{len}_{\pi_f}(u, v) \\ \text{len}(u, v) - \pi_r(v) + \pi_r(u) &= \text{len}(u, v) - \pi_f(u) + \pi_f(v) \\ -\pi_r(v) + \pi_r(u) &= -\pi_f(u) + \pi_f(v)\end{aligned}$$

- $\pi_r + \pi_f \equiv \text{const.}$

Idee:

- Kombiniere π_f und π_r zu $p_f + p_r \equiv 0$:

$$p_f = \frac{\pi_f - \pi_r}{2} \quad p_r = \frac{\pi_r - \pi_f}{2} = -p_f$$

Somit

- wie bidirektionaler Dijkstra
- aber mit $d(s, u) + p_f(u)$ und $d(v, t) + p_r(v)$ als keys
- stoppe wenn $\minKey(\vec{Q}) + \minKey(\overleftarrow{Q}) > \mu_{p_f} = \mu + p_f(s) - p_f(t)$
- dadurch bidirektional zielgerichtet

- **A***, **Landmarken**, **Triangle inequality** (Dreiecksungleich)
- bidirektionaler Landmarken-A* (2. Ansatz)
- aktive Landmarken
- meist 16 Landmarken
- meist avoid oder maxCover Landmarken

Eingaben:

- Straßennetzwerke
 - Europa: 18 Mio. Knoten, 42 Mio. Kanten
 - USA: 22 Mio. Knoten, 56 Mio. Kanten

Evaluation:

- Vorberechnung in Minuten und zusätzliche Bytes pro Knoten
- durchschnittlicher Suchraum (# abgearbeitete Knoten) und Suchzeiten (in *ms*) von 10 000 Zufallsanfragen

Zufallsanfragen ALT

Vorberechnung: Bis 24 Landmarken maxCover, sonst avoid.

	algorithm	PREPRO		QUERY UNIDIR.			QUERY BIDIR.		
		time [min]	space [B/n]	# settled nodes	time [ms]	spd up	# settled nodes	time [ms]	spd up
EUR	DIJKSTRA	0	0	9 114 385	5 591.6	1.0	4 764 110	2 713.2	2.1
	ALT-4	12.1	32	1 289 070	469.1	11.9	355 442	254.1	22.0
	ALT-8	26.1	64	1 019 843	391.6	14.3	163 776	127.8	43.8
	ALT-16	85.1	128	815 639	327.6	17.1	74 669	53.6	104.3
	ALT-24	145.2	192	742 958	303.7	18.4	56 338	44.2	126.5
	ALT-32	27.1	256	683 566	301.4	18.6	40 945	29.4	190.2
	ALT-64	68.2	512	604 968	288.5	19.4	25 324	19.6	285.3
USA	DIJKSTRA	0	0	11 847 523	6 780.7	1.0	7 345 846	3 751.4	1.8
	ALT-8	44.5	64	922 897	329.8	20.6	328 140	219.6	30.9
	ALT-16	103.2	128	762 390	308.6	22.0	180 804	129.3	52.4
	ALT-32	35.8	256	628 841	291.6	23.3	109 727	79.5	85.3
	ALT-64	92.9	512	520 710	268.8	25.2	68 861	48.9	138.7

- bidirektionale Suche: Beschleunigung von 2
- unidirektionaler ALT: mehr als 16 Landmarken nicht sinnvoll
- bidirektionaler ALT: verdoppelung der Landmarken halbiert den Suchraum (ungefähr)
- 16 Landmarken: Beschleunigung ≈ 100 für Europa
- 64 Landmarken: Beschleunigung ≈ 300 für Europa
- hoher Speicherverbrauch (Graph-DS: 424 MB, pro Landmarke: 144 MB, Europa)
- USA schlechter als Europa (Vermutung: schlechtere Hierarchie)

Problem:

- Zufallsanfragen geben wenig Informationen
- Wie ist die Varianz?
- Werden nahe oder ferne Anfragen beschleunigt?

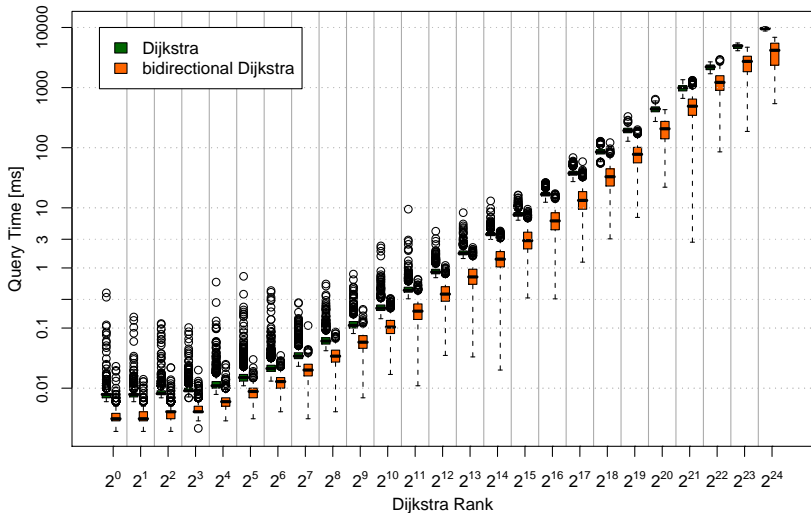
Problem:

- Zufallsanfragen geben wenig Informationen
- Wie ist die Varianz?
- Werden nahe oder ferne Anfragen beschleunigt?

Idee:

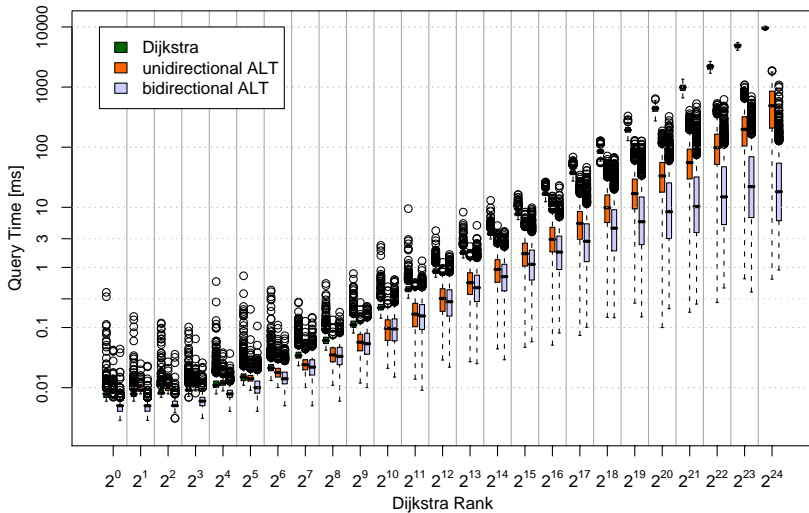
- DIJKSTRA definiert für gegebenen Startknoten Ordnung für auf den Knoten
- DIJKSTRA-Rang $r_s(u)$ eines Knoten u für gegebenes s
- wähle 1000 Startknoten und analysiere jeweils die Suchzeiten um die Knoten mit Rang $2^1, \dots, 2^{\log n}$ zu finden
- zeichne Plot

Lokale Anfragen Bidirektionale Suche



- Ausreißer bei nahen Anfragen (vor allem unidirektional)
- Beschleunigung unabhängig vom Rang (immer ca. Faktor 2)
- Varianz etwas höher als bei unidirektionaler Suche
- manche Anfragen sehr schnell

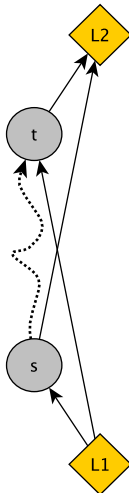
Lokale Anfragen ALT



- Beschleunigung steigt mit Rang
- kaum Beschleunigung für nahe Anfragen
- hohe Varianz für ALT
- Ausreißer bis zu Faktor 100 langsamer als Median

Zusammenfassung ALT

- Zielgerichtet durch geänderte Reihenfolge, wie Knoten abgearbeitet werden
- wird erreicht durch hinzufügen eines Knotenpotentials π
- korrekt wenn $len(u, v) - \pi(u) + \pi(v) \geq 0$ für alle Kanten
- Potentiale durch Landmarken
- kann bidirektional gemacht werden
- Beschleunigung von bis zu 300 gegenüber Dijkstra
- aber hoher Platzverbrauch (Reduktion später)
- hohe Varianz (manche Anfragen haben keine guten Landmarken)



Literatur ALT:

- Andrew Goldberg, Georg Harrelson, SODA 2005
- Andrew Goldberg, Renato Werneck, ALENEX 2005

Wie Suche zielgerichtet machen?

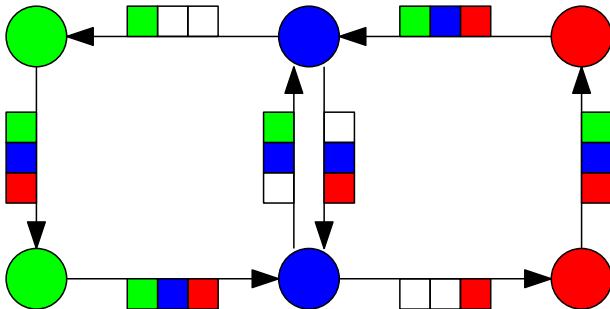
Wie Suche zielgerichtet machen?

- Nichtbeachten von Kanten oder Knoten die in die “falsche” Richtung führen
- Reihenfolge in der Knoten besucht werden ändern

Jetzt: ersteres

Idee:

- partitioniere Graphen in k Zellen
- hänge Label mit k Bits an jede Kante
- gibt an, ob e für Ziele in Zielzelle T benötigt wird



ARC-FLAG-DIJKSTRA($G = (V, E)$, s , t , Arc-Flags AF.(·))

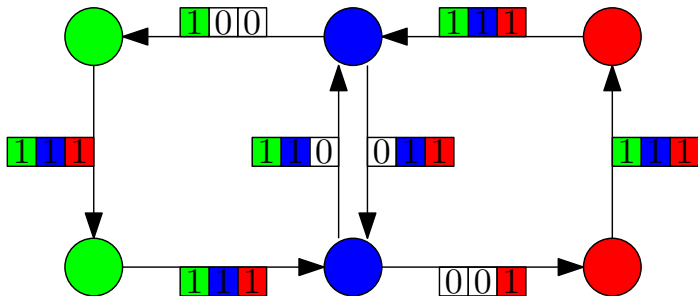
```
1  $d[s] = 0$ 
2  $Q.clear()$ ,  $Q.add(s, 0)$ 
3 while ! $Q.empty()$  do
4    $u \leftarrow Q.deleteMin()$ 
5   forall edges  $e = (u, v) \in E$  do
6     if  $d[u] + \text{len}(e) < d[v]$  and  $AF_T(e) = \text{true}$  then
7        $d[v] \leftarrow d[u] + \text{len}(e)$ 
8       if  $v \in Q$  then  $Q.decreaseKey(v, d[v])$ 
9       else  $Q.insert(v, d[v])$ 
```

Zwei Schritte:

- Partitionierung
- setze korrekte Flaggen

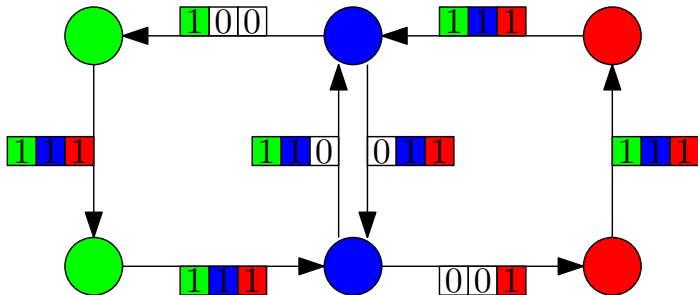
Flaggenberechnung: Erster Versuch

- von jedem Knoten
- konstruiere Rückwärts KW-Baum
- setze Flagge der Region der Wurzel für jede Baumkante
- APSP



Beobachtung: Man muss durch Randknoten in die Zelle

- setze Intra-Zellen Kanten auf `true`
- einen Rückwärts Dijkstra-Baum pro Randknoten b
- setze Flagge $AF_{region(b)}(e) = \text{true}$ wenn e Baumkante des Baums von b ist



verallgemeinere Dijkstra:

- für jede Region r (mit Randknotenmenge B_r)
- hänge Label $l(u)$ der Größe $|B_r|$ an jeden Knoten u
- speichert den Abstand $d(u, b)$ für alle $b \in B_r$
- triviale Initialisierung der Randknoten, füge alle in Queue ein
- nehme Knoten u aus der Queue (key: $\min\{l(u)\}$)
- relaxiere Kanten (u, v) :
 - erzeuge Label l durch $l(u) + \text{len}(u, v)$
 - checke ob l das Label $l(v)$ verbessert
- breche ab, wenn keine Verbesserungen mehr möglich sind
- setze Flagge auf `true` wenn $d(u, b) + \text{len}(u, v) = d(v, b)$ gilt

Bemerkungen:

- Knoten können mehrfach besucht werden (label-correcting)
- Prioritäten der Knoten beliebig wählbar
 - z.B. Minimum über alle Einträge von u oder auch nur der vorläufigen
 - hängt von Eingabe ab
- hoher Speicherverbrauch durch n Label der Größe $|B_r|$
 - 18 Mio. Knoten und 1000 Randknoten \Rightarrow 72 GB
 - teile Arbeit in mehrere Schritte mit maximal 100 Randknoten
 - meist Beschleunigung um Faktor 4 gegenüber Randknotenansatz

Montag, 06.05.2013