Algorithmen für Routenplanung

4. Übung

Thomas Pajor

22. Juni 2009

Contraction Hierarchies

Sei $G = (V, E, \mathrm{len})$ ein gerichteter Graph.

Idee. Erzeuge totale Ordnung $<$ auf den Knoten V. Sei jetzt o.B.d.A. $V = \{1, 2, \ldots, n\}$ bezüglich $<$. Führe nun für jeden Knoten $v \in V$ eine Knoten- und Kantenreduktion durch.

Knotenreduktion. Für einen Knoten $v \in V$ führe einen Kontraktionsschritt durch: Füge für jedes paar $(u, v) \in E$ mit $u > v$ und $(v, w) \in E$ mit $w > v$ eine Kante (Shortcut) (u, w) mit Gewicht

$$\mathrm{len}(u, w) := \mathrm{len}(u, v) + \mathrm{len}(v, w)$$

ein. Existiert bereits eine Kante (u, w) in G, so setze

$$\mathrm{len}(u, w) := \min\big(\mathrm{len}(u, w), \mathrm{len}(u, v) + \mathrm{len}(v, w)\big).$$

Frage: Ist der Shortcut (u, w) wichtig? "Unnötige" Shortcuts führen zu großem $|E|$ was zu (unnötig) schlechter Query-Performanz führen kann.

Beobachtung: Zur Beibehaltung der korrekten Distanzen zwischen u und v ist ein Shortcut (u, w) nicht notwendig wenn es einen Pfad $P = [u, \ldots, w]$ gibt, der v nicht enthält und

$$\mathrm{len}\, P \leq \underbrace{\mathrm{len}(u, v) + \mathrm{len}(v, w)}_{\mathrm{len}(u,w)}$$

gilt. P heißt auch *Zeuge* für das Tripel (u, v, w).

Zeugensuche. Es gibt u.A. folgende Möglichkeiten zur Zeugensuche:

- **Lokale Dijkstra-Suche.**
  Führe von u eine Dijkstra-Suche in $G \setminus \{v\}$ durch. Ist $\operatorname{dist}(u, w) \leq \operatorname{len}(u, v) + \operatorname{len}(v, w)$, füge Shortcut nicht ein. Nachteil: Aufwändig und Zeitintensiv.

- **Limitierte Suchraumtiefe.**
  Führe lokale Suche durch aber limitiere Anzahl `deleteMin` Operationen (abgearbeitete) Knoten, oder prune Pfade mit mehr Hops als ein Parameter h. (h kann während des Preprocessing erhöht werden).

- **Schnelle 1-Hop-Zeugensuche.**
  Übungsaufgabe.

- **Schnelle 2-Hop-Zeugensuche.**
  Übungsaufgabe.

Kantenreduktion. Führe von v eine Dijkstra-Suche aus die bei Knoten w anhält für die $(v, w) \in E$ gilt. Es können Kanten (v, w) gelöscht werden, wenn $\operatorname{len}(v, w) \leq \operatorname{dist}(v, w)$. Nachteil: Dijkstra-Suche zu teuer.

Aber: implizit bei Zeugensuche von Knoten u können Kanten $(u, x) \in E$ gelöscht werden, für die ein alternativer Pfad $P = [u, \ldots, x]$ gefunden wird mit $\operatorname{len} P \leq \operatorname{len}(u, x)$.

Kriterien zur Knotenordnung. Contraction Hierarchies steht und fällt mit der "richtigen" Knotenordnung $<$. Die Bewertung der Priorität kann als Linearkombination der folgenden Kriterien gewählt werden.

- **Kantendifferenz.**
  Differenz aus der Anzahl hinzukommender Kanten (Shortcuts) und wegfallender Kanten durch die Kontraktion.

- **Kosten der Kontraktion.**
  Suchraumgröße der Dijkstra-Suchen für die Zeugensuchen der Knotenreduktion.

Uniformität ist wünschenswert. Wie kann man erreichen dass Knoten möglichst gleichverteilt über den Graph kontrahiert werden?

- **Kontrahierte Nachbarn.**
  Bevorzuge Knoten der weniger bereits kontrahierte Nachbarn hat.

- **Anzahl Hops.**
  Zähle die Anzahl Hops die neu eingefügte Shortcuts repräsentieren.

- **Voronoi-Regionen.**
  Sei $u \in V$ der höchste kontrahierte Knoten (Alle Knoten $x \leq u$ sind bereits kontrahiert, und alle Knoten $x > u$ sind nicht kontrahiert). Die Voronoi Region $R(v)$ eines *nicht* kontrahierten Knotens $v \in V$ ist die Menge aller Knoten $x \in V$ mit $x \leq u$ die näher an v liegen als zu jedem anderen nicht kontrahierten Knoten $v' > u$, also

  $$R(v) := \{x \in V \mid x \leq u \text{ und } \forall v' \in V \text{ mit } v' > u : \operatorname{dist}(v, x) \leq \operatorname{dist}(v', x)\}.$$

  Wähle $\sqrt{|R(v)|}$ für den Bewertungsterm. $|R(v)|$ ist "Kreisfläche" dann ist $\sqrt{|R(v)|}$ der "Durchmesser". Bevorzuge große Voronoi-Regionen.

  Nach Kontraktion von v: Update der Zugehörigkeit der Knoten aus $R(v)$ über modifizierte Dijkstra-Suche ausgehend von den Voronoi-Nachbarn von v.

Sonstige Kriterien:

- Betweeness-Zentralität,
- Reach-Werte (teuer!)

Query-Algorithmus. Führe bidirektionale Suche auf eingeschränkten Graphen durch. Definiere dazu

- Upward-Graph $G_{\uparrow} = (V, E_{\uparrow})$ für Vorwärtssuche durch

  $$E_{\uparrow} := \{(u, v) \in E \mid u < v\}.$$

- Downward-Graph $G_{\downarrow} = (V, E_{\downarrow})$ für Rückwärtssuche durch

  $$E_{\downarrow} := \{(u, v) \in E \mid u > v\}.$$

Abwechslungsstrategie: Führe die Suchen abwechselnd durch (Verzahnung).

Abbruchkriterium: $\operatorname{minKey}(\overrightarrow{Q}) \geq \mu$ **und** $\operatorname{minKey}(\overleftarrow{Q}) \geq \mu$ (oder die Prioritätswarteschlangen sind leer).

Transit-Node Routing

Beobachtung. Viele weite Queries gehen über *die gleichen* wichtigen Transit-Knoten (z.B. Autobahnauffahrten). Weiterhin ist die Menge von Transit-Knoten sehr klein. Außerdem gibt es nur wenige relevante Transit-Knoten die für einen Knoten v wichtig sind (Access-Nodes).

Idee. Berechne Distanztabelle zwischen wichtigen *Transit-Knoten* vor, und berechne Distanzen von jedem Knoten zu seinen Access-Node vor. Verallgemeinerung dieses Konzepts auf $L+1$ Level von Transit-Knoten T_l wobei $T_0 := V$.

Zutaten. Seien nun $L+1 \in \mathbb{N}$ Mengen von Transit-Knoten

$$T_L \subseteq T_{L-1} \subseteq \cdots \subseteq T_1 \subseteq T_0 = V$$

gegeben. Transit-Node Routing hat folgende Zutaten.

- **Forward- und Backward-Access-Mappings** $\overrightarrow{A}_l : V \to 2^{T_l}$ und $\overleftarrow{A}_l : V \to 2^{T_l}$: Für jeden Knoten die relevanten Transit-Knoten auf Level l.

- **Locality-Filter**: $\mathcal{L}_l : V \times V \to \{\texttt{true}, \texttt{false}\}$ wobei $\mathcal{L}_l(s,t) = \texttt{false}$ genau dann wenn der kürzeste s-t-Weg durch Table-Lookups auf Leveln $> l$ berechnet werden kann.

- **Distanztabellen** zwischen Transit-Knoten $D_l : T_l \times T_l \to \mathbb{R} \cup \{\infty\}$.

Weiterhin bezeichne $\mathrm{dist}_l(u,v)$ die Distanz von u nach v bezüglich Level l von Transit-Node Routing und $\mathrm{dist}_{\geq l}$ die minimale (nicht notwendigerweise korrekte) Distanz die unter Benutzung von Leveln $\geq l$ berechnet werden kann.

Die Distanztabelle kann platzeffizient gespeichert werden, indem redundante Einträge (bezüglich unterschiedlichen Leveln l) vermieden werden. Speichere

$$D_l(s,t) := \begin{cases} \mathrm{dist}(s,t) & \text{falls } \mathrm{dist}(s,t) < \mathrm{dist}_{>l}(s,t) \\ \infty & \text{sonst} \end{cases}$$

Dabei verwendet man eine geschickte Hash-Datenstruktur um die ∞-Einträge nicht physikalisch abzuspeichern.

Query. Die Query von Transit-Node Routing reduziert sich auf folgenden Algo:

Algorithmus 1 : Transit-Node Routing Query

$d \leftarrow \infty$
für $l := L \ldots 0$ **tue**
  $d \leftarrow \min(d, \mathrm{dist}_l(s,t))$
  wenn $\mathcal{L}_l(s,t) = \mathit{false}$ **dann**
    abbruch
return d

Dabei entspricht $\mathrm{dist}_l(s,t)$ für $l > 0$ Table-Lookups von s, t zu allen Access-Nodes aus $\overrightarrow{A}_l(s) \subseteq T_l$ bzw. $\overleftarrow{A}_l(t) \subseteq T_l$ und Table-Lookups zwischen den Access-Nodes in D_l, also

$$\mathrm{dist}_l(s,t) := \min_{\substack{u \in \overrightarrow{A}_l(s) \\ v \in \overleftarrow{A}_l(t)}} \big(\mathrm{dist}(s,u) + \underbrace{\mathrm{dist}_l(u,v)}_{=D_l(u,v)} + \mathrm{dist}(v,t)\big).$$

Der Abstand $\mathrm{dist}_0(s,t)$ muss mit einem "konventionellen" Dijkstra (oder einer anderen Speed-Up Technik) berechnet werden.

Vorberechnung.

- **Access-Mappings.**
  Übungsaufgabe.

- **Distanztabellen.**
  Übungsaufgabe.

- **Locality-Filter.**
  Speichere für jeden Knoten v bezüglich Level l den Abstand zum weitest entferntesten Knoten w mit $\varrho_l := \mathrm{dist}_l(v,w) < \mathrm{dist}_{>l}(v,w)$ implizit bei der Vorberechnung der Access-Mappings. Dann definiere

  $$\mathcal{L}_l(v,w) = \texttt{true} :\Leftrightarrow \bigvee_{k \le l} \big(\mathrm{dist}_l(s,t) \le \overrightarrow{\varrho}_k(s) + \overleftarrow{\varrho}_k(t)\big).$$

  Die Anzahl Auswertungen kann reduziert werden, indem man während der Vorberechnung ϱ auf

  $$\overrightarrow{\varrho}'_l(s) := \max_{k \le l} \overrightarrow{\varrho}_k(s) \qquad \text{und} \qquad \overleftarrow{\varrho}'_l(t) := \max_{k \le l} \overleftarrow{\varrho}_k(t)$$

  setzt, dann ist

  $$\mathcal{L}_l(v,w) = \texttt{true} :\Leftrightarrow \big(\mathrm{dist}_l(s,t) \le \overrightarrow{\varrho}_l(s) + \overleftarrow{\varrho}_l(t)\big).$$

  Die Anzahl falsch-positiver Ergebnisse von $\mathcal{L}_l$ steigt natürlich an.