Universität Karlsruhe
Fakultät für Informatik
ITI Wagner

Algorithmen für Routenplanung
Sommersemester 2009
Daniel Delling, Thomas Pajor

Viertes Übungsblatt - Musterlösung

Aufgabe 1: SHARC-Routing

Gegeben sei ein Graph $G = (V, E, \text{len})$ und eine k-level Partition $\mathcal{P} = \{\mathcal{P}_0, \ldots, \mathcal{P}_k\}$ als Eingabe. Zur Erinnerung sei angemerkt, dass sich Arc-Flags auf Level i auf Zellen aus Level $i-1$ der multi-level Partition beziehen.

(a) Es werden iterativ für jedes $i = 0 \ldots k-1$ ausschließlich Knoten $v \in V$ mit $v \in C$, $C \in \mathcal{P}_i$ kontrahiert, für die gilt dass für alle Paare von Kanten $(u, v) \in E$ und $(v, w) \in E$ sowohl $u \in C$ alsauch $v \in C$ gilt. Es werden also keine Shortcuts eingefügt, die über Zellengrenzen von Level-i-Zellen hinweg gehen.

Bei der Kontraktion werden die Flaggen für kontrahierte Kanten wie folgt gesetzt.

"Core → Component". Kanten $e = (u, v)$, wobei u ein Knoten ist, der noch nicht kontrahiert wurde, gilt:

- Flags von Leveln $j < i$ werden aus vorhergehenden Iterationen des Algorithmus übernommen.
- Für Level i werden alle Flags auf `false` gesetzt, außer das Own-Cell-Flag bezüglich des jeweiligen Levels. Damit wird erreicht, dass bei der Query nur in den Leveln abgestiegen werden muss, wenn sich der Zielknoten bereits in der aktuellen Zelle befindet. Für höhere Level ist das Own-Cell-Flag nicht relevant, es können alle Flags auf `false` gesetzt werden.

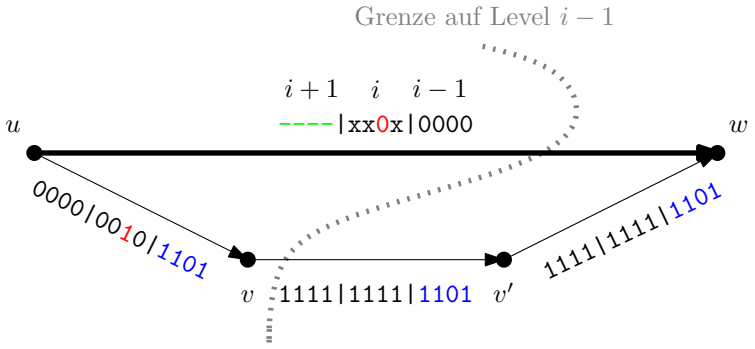
"Component → Component oder Core". Bei Kanten $e' = (v, x)$ für die v ein kontrahierter Knoten ist, werden die Flags wie folgt gesetzt.

- Flags von Leveln $j < i$ werden ebenfalls aus vorangegangenen Iterationen übernommen.
- Flags von Leveln $j \geq i$ werden zunächst komplett auf `true` gesetzt, da zu diesem Zeitpunkt keinerlei Information verfügbar ist zu welchen Zellen auf den Leveln j kürzeste Wege von v über die Kante e' beginnen.

Nachdem die Kontraktion durchgeführt wurde, werden Arc-Flags bezüglich Level i regulär auf dem kontrahierten Graphen berechnet. Dabei werden kontrahierte Kanten nicht relaxiert.

Die Abbildung auf der nächsten Seite veranschaulicht die verschiedenen Arten von Flaggen die gesetzt werden.

Die grünen Flags werden in späteren Iterationen des Algorithmus gesetzt, die blauen Flags werden von vorhergehenden Iterationen übernommen. Schwarze Flags werden in der aktuellen Iteration gesetzt. Während der Kontraktion werden alle Flags bezüglich Leveln $\geq i$ auf `true` gesetzt. Lediglich bei Kanten (u, v), zu denen es einen Shortcut (u, w) gibt, werden bis auf das Own-Cell-Flag (rot) alle Flags auf Level i auf `false` gesetzt. Flags auf höheren Leveln werden auf `false` gesetzt.



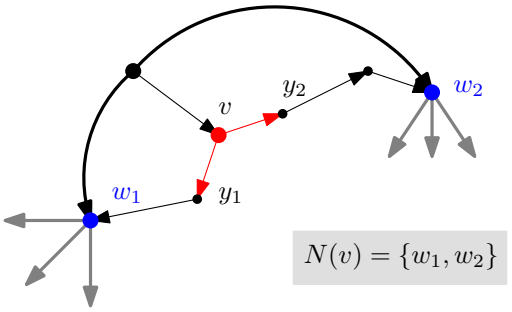
Nach der Kontraktion werden bezüglich Level i entlang nicht kontrahierter Kanten (dick) alle Flags bis auf das Own-Cell-Flag gemäß den kürzeste-Wege-Bäumen aus der Vorberechnung gesetzt. Zusätzlich werden auf Shortcut-Kanten das Own-Cell-Flag (rot) sowie alle Flags bezüglich niedrigerer Level auf `false` gesetzt.

(b) In einer zweiten Phase der SHARC-Vorberechnung werden die Arc-Flags auf den kontrahierten Kanten, die prinzipiell auf `true` gesetzt wurden, ausgedünnt.

Dazu wird levelweise von oben nach unten vorgegangen. Für jedes $i = k-1 \dots 0$ sei $v \in V$ ein beliebiger Knoten der in Level i kontrahiert wurde, dann wurden während der Kontraktion für alle ausgehenden Kanten dieses Knotens die Arc-Flags bezüglich allen Leveln $\geq i$ auf `true` gesetzt.

Es bezeichne nun $l(v)$ für einen Knoten $v \in V$ denjenigen Level, bei dem der Knoten v kontrahiert wurde. Sei jetzt i ein fester Level für eine Iteration der Verfeinerungsphase.

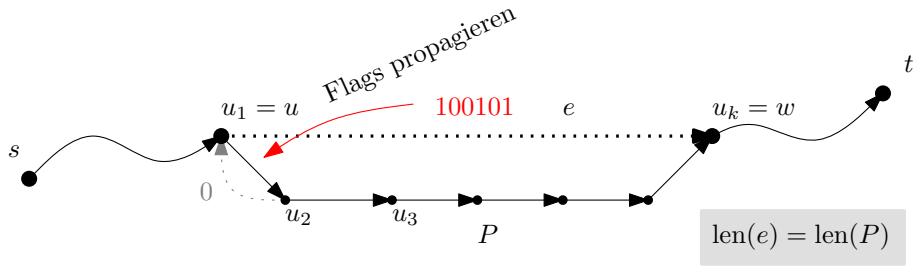
Zur Ausdünnung der Flags von allen ausgehenden Kanten von v kann nun wie folgt vorgegangen werden. Zunächst werden alle Flags bezüglich Level i, bis auf das Own-Cell-Flag, auf allen ausgehenden Kanten von v auf `false` gesetzt. Als nächstes führen wir eine lokale Dijkstra-Suche beginnend von v aus, wobei nur Kanten $e = (x, y)$ relaxiert werden für die $l(y) \geq i$ gilt – das heißt, wir steigen nicht ab. Die Suche wird solange durchgeführt, bis alle Pfade im Suchraum durch einen Knoten w mit $l(w) > i$ abgedeckt sind. Die Menge dieser Knoten w bezeichnen wir als Exit-Nodes $N(v)$.



Für jeden Knoten $w \in N(v)$ können wir nun entlang des kürzeste-Wege-Baums Richtung v traversieren um diejenige ausgehende Kante $e_w = (v, x)$ von v zu bestimmen, die auf dem kürzesten Weg zu w liegt. Die ausgehenden Flags von w werden schließlich wie folgt an e_w propagiert: Für jede ausgehende Kante (w, y) mit $l(y) \geq i$ und $y \notin N(v)$ werden die gesetzten Flags für alle Level $\geq i$ an die Kante e_w übertragen.

(c) Sei $e = (u, w)$ ein Shortcut-Kandidat zur Entfernung. Weiterhin haben wir nur 1-Level-Arc-Flags. Sei nun $P_{u,w} := [u_1, \dots, u_k]$ mit $u_1 = u$ und $u_k = w$ der Pfad, der durch den Shortcut überbrückt wird. Dann gilt $\operatorname{len} P = \operatorname{len} e$.

Es reicht nun aus die Arc-Flags von e an die erste Kante $e_1 = (u_1, u_2)$ des Pfades zu propagieren. Es bleibt damit zu zeigen, dass ein kürzester s-t-Weg $P_{s,t} = [s, \ldots, u, w, \ldots, t]$, der den Shortcut e benutzt hat, weiterhin gefunden wird.



Sei $c = \text{cell}(t)$. Wir zeigen nun, dass entlang des Weges $P'_{s,t} = [s, \ldots, u_1, \ldots, u_k, \ldots t]$ die Arc-Flags AF_c geöffnet sind. Dazu teilen wir den Weg in zwei Teile P_{s,u_2} und $P_{u_2,t}$, wobei $P_{u_2,t}$ nicht notwendigerweise dem Suffix von $P'_{s,t}$ entsprechen muss.

Für Kanten aus $P_{s,u_2} \cap P'_{s,t}$ ist dies trivialerweise erfüllt. Weiterhin ist das Flag AF_c auf der Kante (u_1, u_2) ebenfalls `true`, da (u_1, u_2) die Flags von e bekommen hat. Wegen der Korrektheit der SHARC-Vorberechnung sind die Arc-Flags in G so gesetzt, dass der kürzeste Weg für jedes Paar $v, w \in V$ von Knoten gefunden wird. Insbesondere bedeutet das, dass der kürzeste Weg von u_2 nach t ebenfalls gefunden wird, das heißt Flaggen entlang dieses Weges auf `true` gesetzt sind.

Es bleibt zu zeigen, dass der Weg $P_{u_2,t}$ nicht die Kante e enthielt. Wegen $\text{len}\, P_{u_1,u_k} = \text{len}\, e$ ist P ein kürzester Weg von u_1 nach u_k. Damit der kürzeste u_2-u_k-Weg also die Kante e benutzt, muss gelten $\text{dist}(u_2, u_1) = \text{len}(u_1, u_2) = 0$. In diesem Fall darf die Kante e nicht gelöscht werden, andernfalls benutzte der kürzeste Weg $P_{u_2,t}$ nicht die Kante e, und er wird nach Löschen von e weiterhin gefunden.

Aufgabe 2: Transit-Node Routing & Arc-Flags

Gegeben seien ein Graph $G = (V, E, \text{len})$ und Mengen $\mathcal{T}_1 \subseteq \mathcal{T}_0$ von Transit-Knoten, wobei $\mathcal{T}_0 = V$. Sei weiterhin $\mathcal{P} = \{C_1, \ldots, C_k\}$ eine Partitionierung der Transitknoten $\mathcal{T}_1$.

(a) Zu einer s-t-Anfrage seien $\overrightarrow{A}(s)$ und $\overleftarrow{A}(t)$ die Vorwärts- bzw. Rückwärts-Access-Nodes zu s und t. Des Weiteren sei $\text{cell} : \mathcal{T}_1 \to \{1, \ldots, k\}$ die Funktion die jedem Transit-Knoten seine Zelle zuweist.

Sei nun der kürzeste Weg von s nach t gegeben durch die Knoten $P_{s,t} := [s, a, a', t]$. Für den extremen Fall, dass jeder Transit-Knoten zu einer eigenen Zelle in der Partition gehört (also $|\mathcal{P}| = |\mathcal{T}_1|$ gilt), wäre ausschließlich der Table-Lookup zwischen a und a' nötig. Im Allgemeinen sind aber alle Table-Lookups zwischen Knoten aus

$$\overrightarrow{A}(s) \cap C_{\text{cell}(a)} \qquad \text{und} \qquad \overleftarrow{A}(t) \cap C_{\text{cell}(a')}$$

notwendig. Anders ausgedrückt ist ein Tabel-Lookup zwischen zwei Access-Nodes $a \in \overrightarrow{A}(s)$ und $a' \in \overleftarrow{A}(t)$ ist überflüssig, wenn der kürzeste s-t-Weg *nicht* einen Access-Node aus $\text{cell}(a)$ bzw. $\text{cell}(a')$ benutzt.

Wir speichern uns also an jeden Transit-Knoten $a \in \mathcal{T}_1$ einen Bitvektor $\overrightarrow{f}_a$ der Länge k, wobei der i-te Eintrag genau dann auf `true` gesetzt wird, wenn es Knoten $s, t \in V$ gibt so dass der kürzeste s-t-Weg $P = [s, a, a', t]$ einen Transit-Knoten $a' \in C_i$ benutzt (in diesem

Fall sind natürlich $a \in \overrightarrow{A}(s)$ und $a' \in \overleftarrow{A}(t)$). Analog speichern wir für für jeden Transit-Knoten $a' \in \mathcal{T}_1$ einen Rückwärts-Bitvektor $\overleftarrow{f}_{a'}$ wobei das i-te Flag auf `true` gesetzt wird genau dann wenn es $s, t \in V$ gibt mit wobei der kürzeste s-t-Weg $P = [s, a, a', t]$ einen Access-Node $a \in C_i$ benutzt.

(b) Die Vorberechnung der "Arc-Flags" (korrekterweise müsste es Transit-Node-Flags heißen) läuft in folgenden zwei Schritten ab.

Overlaygraph konstruieren.

Zunächst konstruieren wir einen möglichst minimalen (in der Anzahl Kanten) Overlaygraphen $\bar{G} = (\mathcal{T}_1, \bar{E})$ auf den Transit-Knoten. Für diesen gilt für jedes Paar von Knoten $a, a' \in \mathcal{T}_1$ dass die Distanz $\text{dist}_{\bar{G}}(a, a')$ im Overlaygraphen genau der Distanz $\text{dist}_G(a, a')$ im ursprünglichen Graphen entspricht. Da wir durch die Distanztabelle D bereits alle $\text{dist}_G(a, a')$ gegeben haben, lässt sich durch sukzessives Anwenden der Edge-Reduction ausgehend vom vollständig verbundenen Graphen die Kantenmenge rasch reduzieren (Es sind keine lokalen Suchen notwendig, da alle Distanzen vorberechnet sind).

Randknoten bestimmen.

Sei nun $B \subseteq \mathcal{T}_1$ die Menge an Randknoten bezüglich der Partition $\mathcal{P}$ in $\bar{G}$. Durch die vorhergehende Kantenreduktion in $\bar{G}$ wird erreicht dass die Menge B möglichst klein ausfällt.

Flags initialisieren.

Wir initialisieren nun die Flags $\overrightarrow{f}_a(i)$ und $\overleftarrow{f}_a(i)$ mit `false` für alle $a \in \mathcal{T}$ und für alle $1 \leq i \leq k$.

Flags setzen.

Bezüglich der Vorwärts-Flags wird nun für jeden Knoten $s \in V$ und alle Randknoten $b \in B$ derjenige Access-Node $a \in \overrightarrow{A}(s)$ gesucht, der den Abstand zu b minimiert. Formal ist a gegeben durch

$$a := \operatorname*{argmin}_{a \in \overrightarrow{A}(s)} \{\text{dist}(s, a) + \text{dist}(a, b)\}.$$

Dabei können wir die vorberechneten Abstände $\text{dist}(s, a)$ und $\text{dist}(a, b)$ benutzen. Der kürzeste Weg von s zum Randknoten b führt also über a, das heißt wir setzen das Flag $\overrightarrow{f}_a(\text{cell}(b)) = \texttt{true}$.

Analog werden für das Setzen der Rückwärts-Flags für jeden Knoten $t \in V$ und jeden Randknoten $b \in B$ der Rückwärts-Access-Node von t gesucht der

$$a' := \operatorname*{argmin}_{a' \in \overleftarrow{A}(t)} \{\text{dist}(b, a') + \text{dist}(a', t)\}$$

minimiert. Für diesen setzen wir $\overleftarrow{f}_{a'}(\text{cell}(b)) := \texttt{true}$.

(c) Der Transit-Node-Routing Anfrage-Algorithmus lässt sich wie folgt modifizieren um von den vorberechneten Informationen gebrauch zu machen.

Zu einer s-t-Anfrage werden wie gewohnt die Access-Nodes $\overrightarrow{A}(s)$ und $\overleftarrow{A}(t)$ bestimmt.

Zunächst wird nun ein temporärer Flag-Vector $\overleftarrow{f}_t$ berechnet indem für jeden Access-Node $a' \in \overleftarrow{A}(t)$ das Flag $\overleftarrow{f}_t(\text{cell}(a)) := \texttt{true}$ gesetzt wird. Es reicht nun aus für s nurnoch Access-Nodes zu betrachten die in mindestens eine Region führen, für die mindestens ein Backward-Access-Node aus $\overleftarrow{A}(t)$ existiert das zu der jeweiligen Region gehört.

Da diese Information gerade in $\overleftarrow{f}_t$ enthalten ist, ist für s eine reduzierte Menge $\overrightarrow{A}'(s)$ von Vorwärts-Access-Nodes gegeben durch

$$a \in \overrightarrow{A}'(s) \quad :\Longleftrightarrow \quad a \in \overrightarrow{A}(s) \text{ und } \overrightarrow{f}_a \,\&\, \overleftarrow{f}_t \neq 0.$$

Analog wird ein temporärer Flag-Vektor $\overrightarrow{f}_s$ berechnet mit $\overrightarrow{f}_s(\text{cell}(a)) = \texttt{true}$ für alle $a \in \overrightarrow{A}(s)$. Die reduzierte Menge von Rückwärts-Access-Nodes ist dann analog gegeben durch

$$a' \in \overleftarrow{A}'(s) \quad :\Longleftrightarrow \quad a \in \overleftarrow{A}(t) \text{ und } \overleftarrow{f}_a \,\&\, \overrightarrow{f}_s \neq 0.$$

Die Anzahl Table-Lookups reduziert sich dadurch auf $|\overrightarrow{A}'(s)| \times |\overleftarrow{A}'(t)|$.