A Fully Dynamic Reachability Algorithm for Directed Graphs

with an Almost Linear Update Time

Michał Karpiński

8 stycznia, 2013

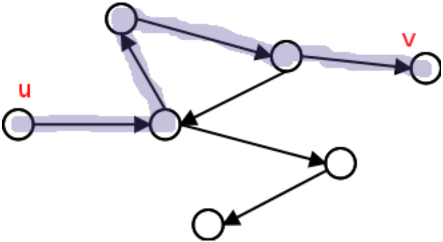
Specyfika problemu (wersja statyczna)

Dane:

- graf skierowany G oraz wierzchołki $u, v \in V(G)$

Wynik:

- Odpowiedź na pytanie: czy w G istnieje ścieżka z u do v?



Specyfika problemu (wersja dynamiczna)

Różnica: graf G zmienia się w czasie!

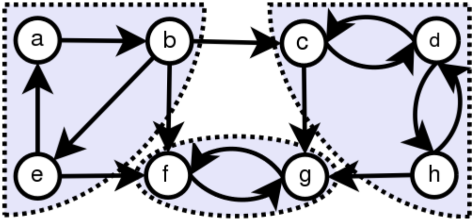
Cel: zbudowanie struktury danych wspierającej operacje:

- *Update* - aktualizuje graf
- *Query* - odpowiada na pytanie o osiągalność

Silnie spójne składowe (SSS)

Definicja

Silnie spójna składowa *grafu skierowanego G to taki maksymalny podgraf H, a jednocześnie jego spójna składowa, taka, że pomiędzy każdymi dwoma jej wierzchołkami istnieje ścieżka.*



Silnie spójne składowe (SSS)

Problem: mając dany graf G, czy $u, v \in V(G)$ należą do tej samej silnie spójnej składowej?

- wersja statyczna - proste
- wersja dynamiczna - zaraz się okaże :)

Uwaga

Struktura rozwiązująca problem dynamiczny SSS jest kluczem do utworzenia struktury rozwiązującej problem dynamiczny osiągalności w grafie skierowanym!

Porządane procedury

- *Insert(E')* - tworzy nową wersję grafu, początkowo identyczną z poprzednią wersją, w której dodajemy zbiór krawędzi E'
- *Delete(E')* - usuwa zbiór krawędzi E' ze **wszystkich** wersji grafu
- *Query(u,v,i)* - sprawdza, czy u i v należą do wspólnego komponentu w i-tej wersji grafu

Porządane procedury

Bardziej formalnie

Algorytm zachowuje komponenty grafów $G_1, G_2 \cdots G_t$, gdzie t jest liczbą operacji *Insert* wykonaną do tej pory. Definiujemy $G_i = \langle V, E_i \rangle$ jako graf utworzony po i-tej operacji *Insert*.

Uproszczenie

Zakładamy, że graf początkowy $G_0 = \langle V, E_0 \rangle$ jest grafem bez krawędzi, czyli $E_0 = \phi$.

Lowest Common Ancestor

Obserwacja

Jeśli (w jakiś sposób) będziemy przechowywać las komponentów dla ciągu $G_0, G_1 \cdots G_t$, to operacja Query może być zredukowana do zapytania LCA na tym lesie.

Przypomnienie

Wiadomo, że można wykonać preprocessing na lasie o liczbie wierzchołków $O(n)$, w czasie $O(n)$ tak, aby zapytania *LCA* wykonywać w czasie stałym $O(1)$. Źródła:

D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestros. SIAM Journal on Computing, 13:338-355, 1984.

B. Shieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. SIAM Journal on Computing, 17:1253-1262, 1988.

Dynamic edge partitioning

Definicja

Mamy dane zbiory krawędzi $E_1 \subseteq E_2 \subseteq \cdots \subseteq E_t$. *Dla każdego* $i = 1 \dots t$, ***dynamiczny zbiór krawędzi*** H_i *grafu* G_i *jest zdefiniowany jako:*

$$H_i = \{(u,v) \in E_i | Query(u,v,i) \wedge (\neg Query(u,v,i-1) \vee (u,v) \notin E_{i-1})\}$$

oraz

$$H_{t+1} = E_t \setminus \cup_{i=1}^{t} H_i$$

Dynamic set partitioning

- $E_t = \cup_{i=1}^{t+1} H_i$
- $H_i \cap H_j = \phi$ dla każdego $1 \leqslant i < j \leqslant t + 1$
- Zbiór H_i jest złożony ze wszystkich krawędzi łączących dwa różne komponenty w G_{i-1} lub takich, które nie znajdują się w G_{i-1}, ale łączą dwa różne komponenty w G_i.

Taki podział jest **dynamiczny** ze względu na to, że każda zmiana w lesie komponentów może powodować przeniesienie krawędzi z H_i do H_j, dla $i < j$.

Algorytm

Struktury:

- *parent* - tablica przechowująca dla każdego wierzchołka w lesie, wskaźnik do jego ojca
- *version* - tablica przechowująca dla każdego wierzchołka w której wersji grafu po raz pierwszy pojawił się komponent związany z tym wierzchołkiem

Algorytm - procedury

Init:
1. $t \leftarrow 0$
2. $H_1 \leftarrow \phi$
3. for each $v \in V$ do
4. $\quad parent[v] \leftarrow null$
5. $\quad version[v] \leftarrow 0$

Obserwacja

Dwa wierzchołki u i v należą do tego samego komponentu w G_i wtedy i tylko wtedy, gdy wersja ich najniższego wspólnego przodka jest mniejsza bądź równa i.

***Query*(u, v, i)**:
1. return $(version[LCA(u, v)] \leq i)$

Algorytm - procedury

***Insert*(E'):**

1. $t \leftarrow t + 1$
2. $H_t \leftarrow H_t \cup E'$
3. $FindScc(H_t, t)$
4. $Shift(H_t, H_{t+1})$
5. $Pre\text{-}LCA(parent)$

- *FindScc(H,i)* -
- *Shift(H_1,H_2)* -
- *Pre-LCA(parent)* -

Algorytm - procedury

***FindScc(H, i)*:**

1. $H' \leftarrow \{(Find(u), Find(v)) \mid (u, v) \in H\}$
2. $\mathcal{C} \leftarrow SCC(H')$
3. for each $C = \{w_1, w_2, \ldots, w_{|C|}\} \in \mathcal{C}$ do
4.   if $|C| > 1$ then
5. $c \leftarrow NewNode$
6. $version[c] \leftarrow i$
7.   for $j \leftarrow 1$ to $|C|$
8.   if $j > 1$ then $Union(w_1, w_j)$
9. $parent[w_j] \leftarrow c$

***Shift*(H_1, H_2):**

1. for each $(u, v) \in H_1$
2.   if $Find(u) \neq Find(v)$ then
3. $H_1 \leftarrow H_1 \setminus \{(u, v)\}$
4. $H_2 \leftarrow H_2 \cup \{(u, v)\}$

Algorytm - procedury

***Delete*($\boldsymbol{E'}$):**

1. for each $v \in V$ do
2. $\quad parent[v] \leftarrow null$
3. for $i \leftarrow 1$ to t
4. $\quad H_i \leftarrow H_i \setminus E'$
5. $\quad FindScc(H_i, i)$
6. $\quad Shift(H_i, H_{i+1})$
7. $H_{t+1} \leftarrow H_{t+1} \setminus E'$
8. $Pre\text{-}LCA(parent)$