Zaawansowane struktury danych Zima 2010

Grafy dynamiczne — 6 grudnia, 2010

Paweł Rzechonek *Paweł Kacprzak, Dawid Ryznar*

1 Wprowadzenie

W wielu problemach grafowych, grafy poddawane są dynamicznym zmianom, takim jak dodawanie, usuwanie wierzchołków lub krawędzi.

W ostatnich 20 latach rosło zainteresowanie grafami, które zmieniają swoją strukturę w czasie. W wyniki czego powstało wiele efektywnych algorytmów i struktur danych, które rozwiązują dynamiczne problemy grafowe.

Definicja **Uaktualnieniem grafu** będziemy nazywać operacje, która wstawia lub usuwa krawędzie czy wierzchołki lub zmienia wartości z nimi związane np. wagi.

Definicja **Graf dynamiczny** jest to graf poddany sekwencji uaktualnień.

Celem dynamicznych algorytmów grafowych jest efektywne uaktualnianie rozwiązania problemu po zmianie struktury grafu, zamiast każdorazowego budowania rozwiązania od początku używając statycznego algorytmu. Chcemy także minimalizować czas pojedynczej operacji uaktualnienia.

W naszych rozważaniach, n będzie oznaczało liczbę wierzchołków w grafie. Grafy będziemy oznaczać literą G, lasy literą F, a drzewa literą T.

Usunięcie wierzchołka będziemy rozumieć jako usunięcie wierzchołek oraz wszystkich krawędzi z nim incydentnych.

2 Podział problemów

Dynamiczne problemy grafowe możemy podzielić według rodzaju operacji, które są dozwolone:

Definicja Problem nazywamy **w pełni dynamicznym**, jeśli dozwolone jest zarówno wstawianie i usuwanie wierzchołków oraz krawędzi.

Definicja Problem nazywamy **wstępującym**, jeśli dozwolone jest tylko wstawianie.

Definicja Problem nazywamy **zstępującym**, jeśli dozwolone jest tylko usuwanie.

Zaprezentujemy jeden problem w pełni dynamiczny oraz jeden zstępujący. Wstępujący problem spójności grafu można rozwiązać używając struktury danych UNION-FIND w czasie $O(\alpha n)$, gdzie α to liczba uaktualnień i zapytań.

3 Plan wykładu

W pierwszej części zaprezentujemy strukturę danych - ET drzewo, na której będą bazować algorytmy przedstawiane w dalszej części wykładu:

- W pełni dynamiczny algorytm rozwiązujący problem spójności grafu.
- Zstępujący algorytm znajdujący minimalne drzewo spinające grafu.

4 ET drzewa

4.1 Wprowadzenie

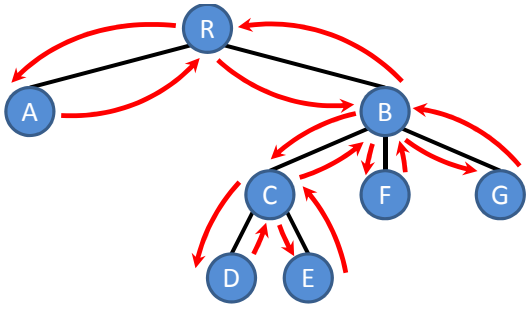
Oba zaprezentowane algorytmy będą używały ET drzew jako struktury danych. ET drzewa zostały wynalezione przez M.Henzinger i V.King w ich wspólnej pracy [1].

Drzewo, które będziemy chcieli reprezentować przy pomocy ET drzewa będziemy nazywać drzewem reprezentowanym.

Dla przypomnienia, cyklem Eulera w grafie nazywamy cykl, który przechodzi każdą krawędź dokładnie raz. Jako cykl Eulera po ukorzenionym drzewie T, będziemy rozumieć cykl Eulera po grafie G, powstałym z T przez zastąpienie każdej krawędzi przez dwie krawędzie skierowane z przeciwnymi zwrotami. Cykl ten zaczyna się i kończy w korzeniu T.

Cykl Eulera po drzewie T, przechodzi każdą krawędź dokładnie dwa razy: raz wchodząc do poddrzewa do którego prowadzi krawędź, a drugi raz wychodząc z tego poddrzewa. Możemy myśleć o cyklu Eulera po drzewie T jako o DFS, który zaczyna się i kończy w korzeniu **Rysunek 1**.

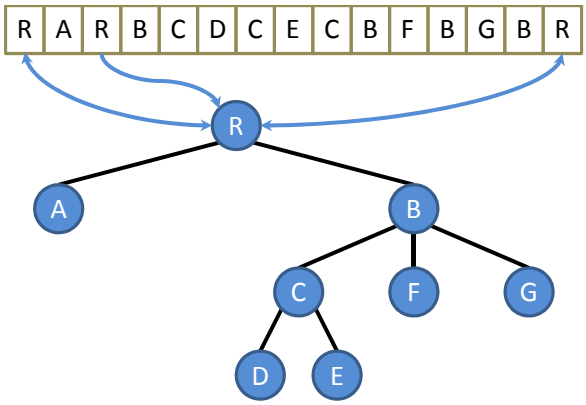
Wniosek ET drzewo będzie miało $O(n)$ wierzchołków.



Rysunek 1: Przykładowe drzewo T oraz jego cykl Eulera. Porządek w jakim wierzchołki są odwiedzane pokazują czerwone strzałki - zaczynamy od korzenia R. Sekwencja kolejnych odwiedzeń to: $R\ A\ R\ B\ C\ D\ C\ E\ C\ B\ F\ B\ G\ B\ R$.

ET drzewo reprezentuje sekwencję odwiedzin kolejnych wierzchołków w cyklu Eulera po drzewie T. Sekwencję będziemy przechowywać używając zbalansowanego drzewa binarnego, wzbogaconego o operacje na listach - `SPLIT(v)` oraz `JOIN(v, w)`. Kluczem węzła w tym drzewie będzie jego czas odwiedzenia w cyklu Eulera po drzewie reprezentowanym. Łatwą implementacją jest drzewo splay.

Każdy wierzchołek w drzewie T, które chcemy reprezentować będzie zawierał wskaźnik do pierwszego oraz ostatniego odwiedzenia w sekwencji reprezentowanej przez ET drzewo **Rysunek 2**.



Rysunek 2: ET drzewo roprezentuje sekwencję odwiedzeń wierzchołków, która zaczyna sie i kończy w korzeniu. Wskaźniki z tej sekwencji prowadzą do wierzchołków reprezentowanego drzewa. Każdy wierzchołek z reprezentowanego drzewa ma wskaźniki do pierwszej i ostatniej wizytacji w sekwencji. (na rysunku zaznaczone są jedynie wskaźniki prowadzące od i do korzenia).

ET drzewa przechowują w łatwy sposób informacje o poddrzewach reprezentowanego drzewa. Rozważmy wierzchołek v z drzewa, które chcemy reprezentować. Wszystkie wierzchołki z poddrzewa ukorzenionego w v znajdują się w ET drzewie pomiędzy pierwszym, a ostatnim odwiedzeniem v.

4.2 Operacje

Chcemy na ET drzewie wykonywać 3 podstawowe operacje:

- `FINDROOT(v)` - znajduje korzeń drzewa, które zawiera v. W cyklu Eulera po drzewie T korzeń jest odwiedziany pierwszy oraz ostatni. Wystarczy więc wyszukać minimum lub maksimum w ET drzewie.
- `CUT(v)` - Chcemy odciąć poddrzewo ukorzenione w v. Cykl Eulera po poddrzewie ukorzenionym w v, jest jednym spójnym blokiem sekwencji wizytacji zaczynającym się i kończącym w v. Wystarczy więć wykonać operacje listową `SPLIT` przed pierwszym odwiedzeniem v oraz po ostatnim odwiedzeniu v. W wyniku tych operacji dostaniemy ET drzewo poddrzewa ukorzenionego w v oraz 2 ET drzewa, które pokrywają sekwencję odwiedzenia przez wejściem do poddrzewa v oraz po wyjściu z niego. Wykonujemy `JOIN` na tych 2 ostatnich sekwencjach. W wyniku dostaniemy ET drzewo odciętego poddrzewa ukorzenionego w v oraz ET drzewa T bez poddrzewa ukorzenionego w v.
- `LINK(v,w)` - Chcemy umieścić krawędź pomiędzy dwoma drzewami, które zawierają odpowiednio v oraz w. Wierzchołek v ma zostać synem w. W ET drzewie taka operacja musi umieszczać sekwencję przejścia poddrzewa ukorzenionego w v bezpośrednio po pierwszym odwiedzeniu w oraz bezpośrednio przed jego drugim odwiedzeniem. Możemy to zrobić wykonując SPLIT na ET drzewie zawierającym w zaraz po pierwszym odwiedzeniu w. Wystarczy teraz połaczyć cześć do pierwszego odwiedzenia w, ET drzewo w którym znajduję się v, drzewo złożone z singletona w oraz cześć po pierwszym odwiedzeniu w.

Wszystkie powyższe 3 operacje wymagają stałej ilości standardowych operacji, które działają na zbalansowanym drzewie binarnym w czasie $O(\log n)$, stąd każda z nich wymaga czasu $O(\log n)$

5 Problem spójności dynamicznego grafu

Chcemy odpowiadać na pytania postaci:

- `CONNECTED(v,w)` - zwraca `TRUE` wtedy i tylko wtedy gdy istnieje ścieżka z v do w w grafie.
- `CONNECTED(G)` - zwraca `TRUE` wtedy i tylko wtedy gdy graf G jest spójny.

Przedstawimy algorytm, który uaktualnia strukture grafu w czasie $O(\log^2 n)$ oraz odpowiada na powyższe pytania w czasie $O(\log n)$.

Gdybyśmy rozważali wariant wstępujący tego problemu, możemy rozwiązać go w następujący sposób:

- zbudować las spinający F grafu G.
- zastosować ET drzewo do zarządzania tym lasem.

W pełni dynamicznej wersji problemu, nie jest już tak łatwo.

Idea: Będziemy przechowywać $\log n$ lasów spinających $F_0, F_1, \ldots, F_{\log n}$. Każdy jako ET drzewo.

Każdej krawędzi przypiszemy rangę. Ranga będzie liczbą naturalną z przedziału od 0 do $\log n$. Będzie ona mogła jedynie zmniejszać się w czasie.

Definicja G_i będzie grafem składającym się z krawędzi o randze nie większej niż i. Wtedy $G = G_{\log n}$.

Definicja F_i będzie lasem spinającym grafu G_i.

W czasie działania algorytmu będziemy utrzmywać 2 niezmienniki:

Niezmiennik 1 Każda spójna składowa G_i ma co najwyżej 2^i wierzchołków.

Niezmiennik 2 $F_{\log n}$ jest minimalnym lasem spinającym grafu G, używając rangi krawędzi jako wagi. $F_i = G_i \cap F_{\log n}$.

Chcemy, żeby nasza struktura składająca się z $\log n$ lasów spinających implementowała poniższe 4 operacje:

- `INSERT(e=(v,w))`:

  Chcemy umieścić w grafie krawędź (v, w). Ustawiamy rangę tej krawędzi na $\log n$ oraz uaktualniamy listy sąsiedztwa v oraz w, dodając odpowiedni wierzchołek do nich. Dodatkowo wywołujemy procedury `FINDROOT(v)` oraz `FINDROOT(w)`. Jeśli ich wyniki są różne, to w wyniku dodania krawędzi drzewa zawierające v oraz w połączą się. Jeśli tak jest, dodajemy krawędź e do $F_{\log n}$.

- `DELETE(e=(v,w))`: Najpierw usuń krawędź e z list sąsiedztwa w oraz v, następnie:
  - jeśli e należy do $F_{\log n}$:
    - usuń e z każdego F_i dla $i \geqslant ranga(e)$.
    - poszukaj krawędzi zastępczej, która połączy v oraz w:
      - krawędź zastępcza nie może mieć rangi mniejszej niż $ranga(e)$ z niezmiennika 2 - każdy F_i jest MST względem rangi krawędzi.
    - for $i = ranga(e)$ to $\log n$
      - niech T_v będzie drzewem zawierającym v, a T_w drzewem zawierającym w.
      - w razie potrzeby zamieńmy v z w aby zachodziło $|T_v| \leqslant |T_w|$.
      - z niezmiennika 1, wiemy, że $|T_v| + |T_w| \leqslant 2^i$, stąd $|T_v| \leqslant 2^{(i+1)}$, a to oznacza, że możemy zdegradować wszystkie krawędzie z T_v do poziomu $i-1$, zachowując dalej niezmiennik.
      - dla każdej krawędzi $e' = (x, y)$, gdzie x należy do T_v oraz $ranga(e') = i$:
        - jeśli y należy do T_w: dodaj e' do $F_i, F_{(i+1)}, \ldots, F_{\log n}$.
        - w.p.p. zmniejsz range e' do $i-1$.

- `CONNECTED(v, w)`:

  Odpowiada na pytanie, czy istnieje ścieżka z v do w:
  return FINDROOT(v) == FINDROOT(w) w $F_{\log n}$

- `CONNECTED(G)`

  r = `PATH-UP(v)`, gdzie v jest dowolnym wierzchołkiem z G.
  if size[v] = n then
  return TRUE
  else
  return FALSE

  – `PATH-UP(v)` zwraca korzeń ET drzewa zwierającego v - nie mylić z korzeniem drzewa reprezentowanego. Wędruje w górę ET drzewa.

  – `size[v]` zwraca ilość wierzchołków w poddrzewie ET drzewa zakorzenionego w v wraz z v. Zarządzamy tą wartością przez wzbogacenie zbalansowanego BST, w którym przechowujemy ET drzewo. Nie zmienia to asymptotycznego kosztu operacji na tym BST.

Dodatkowo musimy wzbogacić nasza strukturę danych, aby móc wykonywać `DELETE` w rządanym czasie:

- chcemy sprawdzać czy $|T_v| \leqslant |T_w|$ w $O(1)$, możemy to zrobić przez standardowe wzbogacenie struktury przez związanie rozmiarów poddrzew z wierzchołkami.
- dla każdego wierzchołka w poddrzewie ukorzenionym w v w F_i: chcemy sprawdzać czy poddrzewo ukorzenione w v zawiera jakieś wierzchołki incydentne do krawędzi o $randze = i$. Szukamy następnej krawędzie o randze i incydentnej do jakiegoś x należącego do T_v w $O(\log n)$ znajdując nastepnika w drzewie i skacząc po pustych poddrzewach.

Analiza kosztu

- `INSERT`: - stała liczba podstawowych operacji na ET drzewie, każda działa w czasie $O(\log n)$ stąd czas działania `INSERT` wynosi $O(\log n)$.
- `CONNECTED(v,w)`: - dwa wywołania `FINDROOT`, każde działa w czasie $O(\log n)$ stąd cała procedura działa w czasie $O(\log n)$.
- `CONNECTED(G)` - jedno przejście w górę drzewa o wysokości $O(\log n)$ - czas $O(\log n)$.
- `DELETE`
  - Główny koszt to usuwanie krawędzi z co najwyżej $\log n$ lasów spinających, każde usunięcie zajmuje czas $O(\log n)$, stąd koszt to $O(\log^2 n)$.
  - Szukanie krawędzi zastępczej. Tutaj analiza jest zamortyzowana i opiera się na fakcie, że ranga każdej krawędzi jest zmniejszana co najwyżej $\log n$ razy, a najbardziej zewnętrzna pętla, w której to robimy wykonuje się też $\log n$ razy, stąd zamortyzowany koszt szukania krawędzie zastępczej wynosi $O(\log^2 n)$.
  - Całkowity koszt `DELETE` jest zamortyzowany i wynosi $O(\log^2 n)$.

Poprawienie czasu

Czysto teoretycznie, możemy poprawić czas odpowiadania na pytania oraz czas wykonywania uaktualnień grafu. Osiągniemy to używając do przechowywania naszych lasów spinających B-drzew o stopniu $O(\log n)$ zamiast drzew BST. W rezultacie otrzymujemy poniższe czasy:

- `CONNECTED` - $O(\log n / \log\log n)$.
- `INSERT` - $O(\log^2 n / \log\log n)$.
- `DELETE` - $O(\log^2 n / \log\log n)$.

6 Zstępujący algorytm MST

Zaprezentuję algorytm i strukturę danych do utrzymywania lasu rozpinającego. Będąc precyzyjnym algorytm i struktura danych zużywa $O\left(n^{\frac{1}{3}} \log n\right)$ zamortyzowanego czasu na operację uaktualnienia grafu. Dodatkowo jest w stanie odpowiadać na pytanie w $O(1)$ jest to czas najgorszego przypadku. Rozważmy problem utrzymania minimalnego lasu rozpinającego (*msf*) podczas wykonywania operacji usuwania krawędzi. Załóżmy że mamy ważony graf $G = \langle V, E \rangle$, gdzie V to zbiór wierzchołków ($|V| = n$), a E to zbiór krawędzi ($|E| = m$). Chcemy utrzymywać minimalny las rozpinający F podczas wykonywania operacji uaktualniania grafu:

- *delete*(u, v): Usuwamy krawędź $\{u, v\}$ z grafu G. Jeśli $\{u, v\} \in F$, to:
  - usuwamy $\{u, v\}$ z F.
  - zwracamy najlżejszą krawędź $e \in G \setminus F$ która połączy na nowo F, jeśli taka krawędź e nie istnieje zwracamy *null*.

Co więcej struktura będzie pozwalała odpowiadać na następujące pytanie:

- *connected*(u, v): zwraca *true* jeśli między wierzchołkami istnieje ścieżka oraz *false* jeśli taka ścieżka nie istnieje.

W 1985 Fredrickson wprowadził strukturę danych *topology trees*, która rozwiązywała ten probelm w czasie $O(\sqrt{m})$ jest to czas potrzebny na przeprowadzenie aktualizacji. Czas poświęcony na odpowiedź na pytanie czy istnieje ścieżka pomiędzy dwoma wierzchołkami to $O(1)$. W 1992 Eppstein osiągną czas $O(\sqrt{n})$ używając *sparsification technique*. W 1997 Monika Rauch Henzinger i Valerie King zaprezentowały strukturę danych, która porzebuje $O\left(n^{\frac{1}{3}} \log n\right)$ zamortyzowanego czasu na aktualizację i odpowiada na pytania w $O(1)$ jest to czas najgorszego przypadku. Ten algorytm zostanie przedstawiony w poniższym dokumencie.

7 Utrzymywanie minimalnego lasu rozpinajacego

Przedstawię teraz algorytm służący do utrzymywania minimalnego lasu rozpinającego z dozwoloną operacją usuwania krawędzi. Bez straty ogólności załóżmy, że wagi na krawędziach są rozróżnialne. Na początku wyliczamy minimalny las rozpinający F dla grafu G. Możemy użyć do tego algorytm Prima używającego kopców fibonacciego. Czas poświęcony na wyliczenie msf to $O(m + n \log n)$. Niech $m_{in}^{'}$ oznacza liczbę krawędzi nie będących w F a będącecyh w G (*niedrzewnych*), mam tutaj na myśli stan F i G na początku (przed wykonaniem aktaulizacji na strukturze). Ponadto niech $k = m_{in}^{'\frac{1}{3}} \log n$. Sortujemy niedrzewne krawędzie względem wagi i rozmieszczamy je w $m_{in}^{'}/k$ poziomach, każdy poziom ma rozmiar k. K najlżejszych znajdzie się na poziomie 0, kolejne k najlżejszych znajdzię się na poziomie 1 i tak dalej. Zbiór krawędzi z poziomu i oznaczmy przez E_i. Co więcej wszystkie krawędzie należące do F (drzewne) umieszczamy w E_0.

Podczas wykonywania algorytmu, poziom przypisany do krawędzi pozostaje niezmieniony. F oznacza minimalny las rozpinający dla całego grafu G. Dla każdego $i = 0, 1, .., (m_{in}^{'}/k) - 1$ niech F_i oznacza minimalny las rozpinający dla grafu ze zbiorem wierzchołków V i zbiorem krawędzi $\bigcup_{j \leqslant i} E_j$. Na początku dla każdego i, $F_i = F$, jednakże podczas wykonywania aktualizacji struktury może się to pozmieniać, ponieważ niedrzewne krawędź z dowlonego poziomu może stać się krawędzią drzewną. Mamy następujący niezmienik $F_0 \subseteq F_1 \subseteq \ldots \subseteq F_{(m_{in}^{'}/k)-1} = F$, który zostaje zachowany podczas działania algorytmu. Niech $T_i(x)$ oznacza drzewo w F_i zawierające x, a $T(x)$ oznacza drzewo w F zawierające x.

7.1 Główna idea

Jeśli usuwamy krawędz $e = \{u, v\}$ taką że $e \notin F$, to F nie zmienia się, nie jest to zbyt ciekawy przypadek. Załóżmy więc, że usuwamy krawędź e, która na i - tym poziomie jest krawędzią drzewną. Dla każdego lasu F_j, gdzie $j \geqslant i$ drzewo zawierające tą krawędź zostanie podzielone na dwa drzewa $T_j(u)$ i $T_j(v)$. Szukamy najlżejszej krawędzi niedzewnej która połączy na nowo $T(u)$ i $T(v)$, zbierając i testując zbiór S kandydujących krawędzi z poziomu i. Jeśli takiej krawędzi nie znajdziemy powtarzamy tą procedurę na poziomie $i + 1$ aż do wyczerpania wszystkich poziomów.

Teraz opiszemy operację uaktualniania:

delete(u,v): usuwamy krawędź $\{u, v\}$ z każdego miejsca w strukturze w którym występuje. Jeśli $\{u, v\}$ jest krawędzią drzewną na poziomie i wtedy wywołujemy procedurę *Replace(i,u,v)*.

W algorytmie przedstawionym poniżej procedura *Search*$(T_i(u))$ zwraca wszystkie niedrzewne krawędzie z poziomu i, które są incydentne z drzewem $T_i(u)$. Procedura *Search* zostanie dokładnie opisana w puźniejszej części tego dokumentu.

Replace(i, u, v)

1. Wykonujemy *Search*$(T_i(u))$, *Search*$(T_i(v))$.
   - Jeden z *Search - ów* się zatrzymał i zwrócił niepusty wynik, niech S oznacza wynik zwrócony przez ten *Search*.
   - *Search - e* zwróciły puste wyniki wtedy S będzie zbiorem wszystkich niedrzewnych krawędzi z poziomu i.
2. Sprawdzamy każdą krawędź z S czy łączy $T_i(u)$ i $T_i(v)$.
   - znaleźlismy krawędź która łączy $T_i(u)$ i $T_i(v)$, wprowadzamy tą krawędź do F oraz do każdego lasu F_j, gdzie $j \geqslant i$.
   - w S nie znaleźliśmy szukanej krawędzi. Jeśli i nie jest ostatnim poziomem wywołujemy *Replace*$(i + 1, u, v)$. Jeśli i jest ostatnim poziomem procedura *Replac* kończy się, nie znajdujemy zastępczej krawędzi.

7.2 Struktura danych

Każde drzewo w dowolnym F_i i w F będzie reprezentowane jako ET-drzewo, zaowocuje to dwiema korzyściami:

- Stały czas potrzebny na sprawdzenie czy jakaś krawędź łączy dwa drzewa.
- Szybko możemy odnaleść niedrzewne krawędzie w E_i, które są incydentne z zadanym drzewem.

Aby zmniejszyć koszty będziemy trzymać tylko te F_i w pamięci, których i jest wielokrotnoscią $m_{in}^{'\frac{1}{3}} / \log n$. Każde ET-drzewa jest przechowywane w B-drzewie o stopniu d. Dzięki temu będziemy mogli:

- Zaimplementować usuwanie i dodawanie krawędzi w F w czasie $O(d \log_d n)$ (*split* lub *join* wykonany na ET-drzewie).
- Sprawdzać czy dwa wierzchołki są w tym samym drzewie w czsie $O(\log_d n)$

Jesli przyjmiemy że $d = n^{\alpha}$, gdzie α - dodatnia stała to:

- $O(d \log_d n) \Rightarrow O(d)$.

- $O(\log_d n) \Rightarrow O(1)$.

Struktura danych będzie składać się z następujących elementów:

1. Każda krawędź ma przypisany swój poziom i, oraz bit który informuje czy jest to krawędź drzewna czy też nie.
2. Niech $k' = max\left\{m_{in}'^{\frac{1}{3}} \log n, n^{\epsilon}\right\}$, gdzie $\epsilon \in (0, \frac{1}{3}]$. Dla każdego drzewa w F reprezentujemy je jako ET-drzewo, które jest zaimplementowane za pomocą B-drzewa o stopniu k'.
3. Niech $c = m_{in}'^{\frac{1}{3}} \log n$. Mapujemy każdy poziom i na poziom j taki, że $f(i) = j$, gdzie $f(i) = c\lfloor \frac{i}{c} \rfloor$. Dla każdego poziomu j takiego że $c|j$:
   (a) Każde drzewo z F_j jest reprezentowane jako ET-drzewo i jest trzymane w B-drzewie o stopniu 2.
   (b) Dla każdego wierzchołka v tworzymy liste $L_j(v)$, która zawiera:
      - Wszystkie niedrzewne krawędzie incydentne do v które są na jakimkolwiek poziomie i, gdzie $i \in f^{-1}(j)$.
      - Wszystkie drzewne krawędzie incydentne do v które są na jakimkolwiek poziomie $i > j$, gdzie $i \in f^{-1}(j)$.
   (c) Zaznaczamy każde wystąpienie wierzchołka v którego lista $L_j(v)$ nie jest pusta. Każdy wierzchołek w ET-drzewie jest zaznaczony jeżeli drzewo zakorzenione w nim zawiera zaznaczony wierzchołek.

7.3 Procedura *Search*

Search($T_i(u)$) - zwraca wszystkie niedrzewne krawędzie z poziomu i incydentne z $T_i(u)$. Zaczynamy od przeszukania $T_{f(i)}(u)$, które jest poddrzewem $T_i(u)$. Sprawdzamy wszystkie krawędzie należace do $L_i(v)$ dla każdego wierzchołka v w drzewie $T_i(u)$. Niedrzewne krawędzie z poziomu i są po kolei wybierane, a wszystkie drzewne krawędzie z poziomu i', $f(i) < i' \leqslant i$ prowadzą do innych drzew wchodzących w skład $F_{f(i)}$. Wszystkie takie krawędzie drzewne prowadzą do innych drzew $F_{f(i)}$, które są poddrzewami $T_i(u)$. Wszystkie poddrzewa $T_i(u)$ zostaną sprawdzone w ten sposób.

Procedura *Search* w pseudokodzie:

Search($T_i(u)$)

1. $S' \leftarrow \emptyset$;
2. *listadrzew* $\leftarrow T_{f(i)}(u)$;
3. Powtarzamy dopóki *listadrzew* $\neq \emptyset$;
   (a) Usuwamy jedno ET-drzewo z *listadrzew*;

(b) Dla każdego zaznaczonego wierzchołka x w ET-drzewie oraz dla każdej krawędzi $\{x, y\} \in L_{f(i)}(x)$:

- if $\{x, y\}$ jest krawędzią niedrzewną dodaj ją do $S^{'}$;
- else if $\{x, y\}$ jest krawędzią drzewną na poziomie $l \leqslant i$ to dodaj $T_{f(i)}(y)$ do *lista-drzew*;

4. Return $S^{'}$;

7.4 Analiza złożoności

Niech t oznacza liczbę krawędzi w F przed wykonaniem jakiejkolwiek aktualizacji w grafie. W koszt stworzenia struktury wliczamy kolejno:

1. Obliczenie minimalnego lasu rozpinającego. Zajmie to czas $O(m + n \log n)$.
2. Posortowanie i podzielenie wszystkich krawędzi niedrzewnych $O(m \log m)$.
3. Stowrzenie ET-drzew dla F, oraz dla każdego lasu F_j, gdzie $c|j$. Zajmuje to czas proporcjonalny do wilkości każdego lasu czyli $O\left(m_{in}^{'} + t\right)$.
4. Stworzenie wszystkich list L zajmuje czas propocjonalny do liczby niedrzewnych krawędzi $O\left(m_{in}^{'}\right)$.

Koszt usunięcia niedrzewnej krawędzi: Usunięcie niedrzewnej krawędzi z któregokolwiek poziomu może spowodować konieczność zrestartowania znacznika na wierzchołkach w niektórych ET-drzewach. W najgorszym przypadku będzie to restartowanie znacznika na wszystkich wierzchołkach na ścieżce aż do korzenia. Zajmie to maksymalnie czas $O(\log n)$.

Usunięcie i wstawienie drzewnej krawędzi do struktury:

W czas poświęcony na usunięcie drzewnej krawędzi wchodzi:

- Usunięcie krawędzi z odpowiedniego ET-drzewa wchodzącego w skład F. Czas $O\left(k^{'}\right)$
- Usunięcie krawędzi z pojedynczego ET-drzewa $O(\log n)$. Trzeba to zrobić dla każdego lasu F_j, gdzie $c|j$. W sumie daje to czas $O\left(\left(\left(m_{in}^{'}/k\right)/c\right)\log n\right)$

Czas poświęcony na dodanie drzewnej krawędzi jest propocjonalny do czasu potrzebnego na usunięcie drzewnej krawędzi.

Szukanie zastępczej krawędzi: Niech $w(T)$ dla pewnego drzewa T w pewnym lesie F_i oznacza wagę drzewa $\left(w(T) = \sum_{v \in T} |L_{f(i)}(v)|\right)$. Aby przejść od korzenia do liścia w ET-drzewie w celu odnalezienia zaznaczonego wierzchołka, albo przejść od zaznaczonego liścia do korzenia potrzeba poświęcić $O(\log n)$ czasu. Dlatego koszt *Search*$(T_i(x))$ to $O(w(T_i(x)) \log n)$. Koszt wykonywania *Replace* to koszt *Search* plus koszt sprawdzenia każdej krawędzi ze zbioru S czy łączy na nowo dwa drzewa. Sprawdzenie odbywa się w czasie $O(1)$. Wielkość zbioru S to $O(min\{k, w(T)\})$. Zauważamy że jest co najwyżej k krawędzi na każdym poziomie. Każda lista $L_j(v)$ zawiera co

najwyżej krawędzie z c poziomów. Wnioskujemy więc, że $w(T) \leqslant ck$. Podsumuwując koszt szukania zastępczej krawędzi to $O\left(m'^{\frac{1}{3}} \log n + n^{\epsilon}\right)$.

Zauważamy również że odpowidź na pytanie *connected(u,v)* zostaje znaleziona w czasie $O(1)$ używając ET-drzew dla przechowywania drzew z lasu F.

Literatura

[1] Monika Rauch Henzinger, Valerie King, Maintaining Minimum Spanning Trees in Dynamic Graphs.

[2] Camil Demetrescu, Irene Finocchi, Giuseppe F. Italiano, Dynamic Graph Algorithms.

[3] Robert E. Tarjan, Renato F. Werneck, Dynamic Trees in Practice.

[4] Camil Demetrescu, Giuseppe F. Italiano, Dynamic Shortest Paths and Transitive Closure: Algorithmic Techniques and Data Structures.

[5] Daniele Frigioni, Alberto Marchetti-Spaccamela, Umberto Nanni, Fully Dynamic Algorithms for Maintaining Shortest Paths Trees.

[6] Daniele Frigioni, Alberto Marchetti-Spaccamela, Umberto Nanni, Incremental Algorithms for the Single-Source Shortest Path Problem.

[7] Stephen Alstrup, Jacob Halm, Kristian de Lichtenberg, Mikkel Thorup, Maintaining Information in Fully Dynamic Trees with Top Trees.

[8] Daniel D. Seator, Robert Endre Tarjan, A Data Structure for Dynamic Trees.

[9] Stephen Alstrup, Jacob Halm, Kristian de Lichtenberg, Mikkel Thorup, Minimizing Diameters of Dynamic Trees.

[10] Greg N. Frederickson, Data Structures for On-Line Updating of Minimum Spanning Trees.

[11] Greg N. Frederickson, Ambivalent Data Structures for Dynamic 2-Edge Connectivity and k - Smallest Spanning Trres.