# 1 Problem dynamicznych drzew

W problemie dynamicznych drzew należy zbudować strukturę danych reprezentującą las drzew ukorzenionych. Wierzchołek w takim drzewie może mieć dowolną liczbę synów, na których nie jest ustalony żaden porządek. Struktura danych powinna udostępniać następujące operacje:

- MakeTree() – Zwraca wierzchołek nowego, jednoelementowego drzewa.
- Cut($v$) – Usuwa krawędź łączącą $v$ z ojcem.
- Link($v, w$) – Przy założeniu, że $v$ jest korzeniem pewnego drzewa oraz $w$ jest węzłem *innego* drzewa, podłącza $v$ jako syna $w$.
- Root($v$) – Zwraca korzeń drzewa zawierającego $v$.

W zastosowaniach przydaje się rozszerzona wersja problemu, gdzie wierzchołki drzewa mogą dodatkowo przechowywać wagi (liczby rzeczywiste) oraz dostępne są pewne operacje umożliwiające modyfikację wag oraz zwracające pewne statystyki dotyczące wag (patrz rozdział 4.1).

# 2 Drzewa Link-Cut

Opiszemy drzewa Link-Cut, które rozwiązują problem dynamicznych drzew udostępniając operacje Cut, Link i Root w czasie $O(\log n)$, gdzie $n$ jest liczbą wierzchołków we wszystkich drzewach. Drzewa Link-Cut oraz ich zastosowania do problemu znajdowania maksymalnego przepływu oraz LCA podali Sleator i Tarjan w pracy [1]. Na tym wykładzie przedstawimy jednak prostszą wersję drzew Link-Cut, wykorzystującą drzewa splay, opisaną w pracy [2]. Jak zobaczymy, struktura drzew Link-Cut jest podobna do struktury drzew Tango, które poznaliśmy na jednym z poprzednich wykładów (chociaż ze względu na chronologię powstawania tych wyników należałoby raczej powiedzieć odwrotnie). Mianowicie, podobnie jak Tango, drzewo Link-Cut jest drzewem drzew typu splay. Jedno drzewo Link-Cut reprezentuje jedno drzewo z lasu, który utrzymujemy, jego struktura jest jednak zupełnie inna. Drzewo odpowiadające drzewu Link-Cut będziemy nazywać *drzewem reprezentowanym*. Drzewo Link-Cut zawiera takie same węzły co drzewo reprezentowane, a jego struktura zależy od struktury drzewa reprezentowanego i ciągu dotychczas wykonywanych operacji. W szczególności z drzewa Link-Cut można odtworzyć drzewo reprezentowane.

## 2.1 Drzewa reprezentowane jako zbiory ścieżek

W tym podrozdziale jest mowa jedynie o drzewach reprezentowanych. Wprowadzimy pewne pojęcia które będą później pomocne przy opisie drzew Link-Cut. W każdej chwili drzewo reprezentowane można podzielić na pewien zbiór ścieżek, a podział ten zależy od wcześniej

wykoywanych operacji. (Ważne jest jednak żeby pamiętać, że drzewa reprezentowane *nie* są przechowywane w pamięci komputera).

Powiemy, że wierzchołek drzewa reprezentowanego został *dotknięty*, jeśli był przekazany do jednej z operacji Cut, Link lub Root jako argument. *Preferowanym synem* węzła $v$ nazwiemy takiego syna $w$, który jest przodkiem ostatnio dotkniętego węzła w poddrzewie $v$. W szczególności $v$ nie ma preferowanego syna (przyjmiemy że preferowany syn jest równy Null), gdy nie było takiego dotknięcia lub gdy to wierzchołek $v$ został najpóźniej dotkniety spośród węzłów swojego poddrzewa. *Preferowana krawędź* to krawędź łącząca węzeł z jego preferowanym synem. *Preferowana ścieżka* to maksymalna spójna ścieżka preferowanych krawędzi w drzewie, lub pojedyńczy wierzchołek, jeśli nie jest on incydentny z żadną preferowaną krawędzią. Zauważmy, że każdy wierzchołek leży na dokładnie jednej preferowanej ścieżce.

## 2.2 Struktura drzew Link-Cut

Podział drzewa reprezentowanego na preferowane ścieżki jednoznacznie wyznacza strukturę drzewa Link-Cut. Drzewo Link-Cut składa się z wielu *drzew pomocniczych* połączonych dodatkowymi krawędziami w jedno drzewo. Każde drzewo pomocnicze reprezentuje dokładnie jedną preferowaną ścieżkę. Drzewa pomocnicze są drzewami typu splay (a więc BST). Węzły drzewa pomocniczego odpowiadają węzłom reprezentowanej ścieżki, ale klucz węzła drzewa pomocniczego jest *głębokością* odpowiadającego mu węzła w drzewie reprezentowanym. Stąd dla dowolnego wierzchołka $v$ w drzewie pomocniczym wszystkie elementy jego lewego poddrzewa w drzewie pomocniczym leżą powyżej $v$ w drzewie reprezentowanym. Drzewa pomocnicze są połączone w jedno duże drzewo wskaźnikami, które będziemy nazywać *wskaźnikami ścieżka-ojciec*. Każde drzewo pomocnicze ma jeden taki wskaźnik, przechowywany w korzeniu. Wskazuje on na ojca (w drzewie reprezentowanym) najwyższego węzła na ścieżce reprezentowanej przez drzewo pomocnicze.

## 2.3 Operacja Dotknij

Drzewa Link-Cut realizują wszystkie operacje problemu drzew dynamicznych za pomocą operacji Dotknij($v$)*. Przebudowuje ona drzewo Link-Cut aby wyglądało tak, jakby właśnie dotknięto wierzchołka $v$ w drzewie reprezentowanym. Zauważmy, że w efekcie takiego dotknięcia w drzewie reprezentowanym $R$ znajduje się ścieżka preferowana o początku w korzeniu $R$ i końcu w $v$. Wierzchołki na tej ścieżce (łącznie z $v$) mogły podczas tej operacji zmienić swoich preferowanych synów. Zauważmy, że operacja zmiany preferowanego syna odpowiada podziałowi pewnej ścieżki preferowanej na dwie ścieżki (a więc podziałowi odpowiedniego drzewa pomocniczego w drzewie Link-Cut) oraz połączeniu górnej z tych dwóch ścieżek z pewną ścieżką (a więc będziemy też łączyć drzewa pomocnicze).

Będziemy „budować” ścieżkę preferowaną od dołu (czyli od $v$) do góry (czyli do korzenia $R$). Oczywiście przez budowanie rozumiemy tu modyfikacje w odpowiadającym drzewie

*Ta operacja nazywa się w oryginale Access($v$)

Link-Cut.

Zaczynamy od tego, że w $v$ ustawiamy preferowanego syna na Null (a preferowany syn może istnieć, $v$ jest wtedy w środku poprzedniej preferowanej ścieżki). Musimy więc w drzewie pomocniczym odciąć wierzchołki, które leżały poniżej $v$ w reprezentowanej ścieżce. W tym celu wykonujemy splay($v$), $v$ staje się korzeniem drzewa pomocniczego i możemy odciąć jego prawe poddrzewo (korzeniowi tego poddrzewa ustawiamy wskaźnik ścieżka-ojciec na $v$).

Następnie idziemy w drzewie pomocniczym od $v$ do góry po krawędziach preferowanych. W miejscu, w którym okaże się, że krawędź wzwyż do pewnego wierzchołka $w$ nie jest preferowana (jest wskaźnikiem ścieżka-ojciec), musimy ojcu zmienić preferowanego syna. Co to oznacza w drzewie Link-Cut? Zauważmy że po splay wierzchołek $v$ jest w korzeniu swojego drzewa pomocniczego, a więc przechowuje wskaźnik ścieżka-ojciec. Jest to wskaźnik do wierzchołka $w$. Najpierw odłączmy dolną część ścieżki preferowanej zawierającej $w$ od $w$, podobnie jak to zrobiliśmy dla $v$ (przez operację splay w drzewie pomocniczym i odłączenie prawego poddrzewa). Następnie po prostu podłączamy $v$ jako prawego syna $w$ w drzewie splay (zauważmy, że porządek BST jest zachowany). Powtarzamy to postępowanie aż dojdziemy do korzenia drzewa (kolejna krawędź do zamiany jest wskaźnikiem ścieżka-drzewo przechowywanym w wierzchołku $w$ itd). Zauważmy, że na samym końcu wierzchołek $v$ jest w tym samym drzewie pomocniczym co korzeń drzewa Link-Cut (podobnie jak wszystkie wierzchołki na ścieżce od $v$ do korzenia w drzewie reprezentowanym). Na koniec wykonujemy jeszcze raz splay($v$). Oto pseudokod Dotknij($v$):

- Wykonaj splay($v$) w drzewie pomocniczym zawierającym $v$.
- Usuń preferowanego syna $v$:
  - path-parent(right($v$)) $\leftarrow v$
  - right($v$) $\leftarrow$ Null
- $x \leftarrow v$
- while $x$ nie jest korzeniem drzewa reprezentowanego
  - $w \leftarrow$ path-parent($x$)
  - splay($w$)
  - Zmień preferowanego syna $w$:
    - path-parent(right($w$)) $\leftarrow w$
    - right($v$) $\leftarrow x$
    - path-parent($x$) $\leftarrow$ Null
  - $x \leftarrow w$
- splay($v$)

## 2.4 Operacja ROOT($v$)

Najpierw wykonujemy DOTKNIJ($v$). W efekcie $v$ jest korzeniem swojego drzewa pomocniczego. Wtedy korzeń $r$ drzewa reprezentowanego jest skrajnie lewym potomkiem $v$ w drzewie pomocniczym. Podążamy skranie lewą ścieżką, znajdujemy $r$, wykonujemy splay($r$) i zwracamy $r$.

## 2.5 Operacja CUT($v$)

Wykonujemy DOTKNIJ($v$). W rezultacie wszystkie elementy wyższe niż $v$ drzewie reprezentowanym pojawiają się w lewym poddrzwie $v$. Wystarczy już tylko odciąć lewe poddrzewo $v$ (staje się ono osobnym drzewem Link-Cut).

## 2.6 Operacja LINK($v, w$)

Wykonujemy DOTKNIJ($v$) i DOTKNIJ($w$). Wierzchołek $v$ jest korzeniem drzewa reprezentowanego a więc w drzewie Link-Cut ma puste lewe poddrzewo. Po wykonaniu operacji $v$ powinien być preferowanym synem $w$ w drzewie reprezentowanym (natomiast $v$ nie powinien mieć preferowanego syna). A więc wystarczy podłączyć $w$ jako jako lewego syna $v$ w drzewie Link-Cut.

## 2.7 Analiza złożoności

Jest jasne, że złożoność operacji LINK i CUT jest taka sama jak operacji DOTKNIJ, natomiast dla operacji ROOT jest to czas DOTKNIJ plus zamortyzowany czas $O(\log n)$, co wynika z własności drzew splay. A więc w dalszym ciągu skupimy się na analizie złożoności operacji DOTKNIJ.

### 2.7.1 Ograniczenie $O(\log^2 n)$

Operacja DOTKNIJ($v$) wykonuje $k + 1$ operacji splay, gdzie $k$ jest liczbą zmienionych preferowanych synów na ścieżce od $v$ do korzenia. Najpierw pokażemy, że zamortyzowana liczba takich zmian jest $O(\log n)$ – da to od razu ograniczenie $O(\log^2 n)$, które poźniej poprawimy.

Użyjemy ciekawej techniki dekompozycji ciężko-lekkiej. Niech $v$ będzie synem $w$ oraz oznaczmy przez size($x$) rozmiar poddrzewa o korzeniu w $x$. Powiemy, że krawędź od $(v, w)$ jest *ciężka*, gdy $\text{size}(v) > \frac{1}{2}\text{size}(w)$. Gdy krawędź nie jest ciężka to jest *lekka*. Oczywiście dowolny wierzchołek ma co najwyżej jedną ciężką krawędź do syna. Mówiąc o krawędziach lekkich i ciężkich odnosimy się tu do drzew reprezentowanych.

Zauważmy, że podczas jednej operacji DOTKNIJ co najwyżej $\log_2 n$ lekkich krawędzi stało się preferowanymi, ponieważ na dowolnej ścieżce od korzenia w dół może być co najwyżej $\log_2 n$ lekkich krawędzi. A więc całkowita liczba zmian krawędzi preferowanych na krawędź lekką jest $O(m \log n)$, gdzie $m$ to liczba wykonanych operacji. Pozostaje obliczyć

liczbę zmian krawędzi preferowanych na ciężkie. Zastanówmy się nad życiem pewnej krawędzi $e$ od syna $v$ do ojca $u$. Rozważmy dwa kolejne zdarzenia „$e$ była ciężka i stała się preferowana". Między tymi zdarzeniami musiało zajść jedno z trzech zdarzeń:

(A) $e$ była preferowana i przestała być ciężka,

(B) $e$ była ciężka i przestała być preferowana, bo inny syn $u$ stał się preferowany

(C) $e$ była ciężka i przestała być preferowana, bo $u$ przestał mieć preferowanego syna.

Zauważmy, że całkowita liczba zmian krawędzi preferowanych na ciężkie jest ogracznona przez całkowitą liczbę zdarzeń (A), (B) i (C) powiększoną o całkowitą liczbę krawędzi podczas ciagu operacji, czyli $m$. Jeśli zaszło zdarzenie (A) to oznacza, że usunięto pewne poddrzewo w poddrzewie $v$ (sytuacja gdy dodano pewne poddrzewo w poddrzewie innego syna $u$ odpada, bo wtedy $e$ nie jest preferowana!). Ale podczas takiego usunięcia wszystkie pojawiające się zdzarzenia (A) są na jednej ścieżce do korzenia, a ponieważ może być $\leq \log n$ lekkich krawędzi na takiej ścieżce, całkowita liczba zdarzeń (A) jest $O(m \log n)$. Jeśli zaszło zdarzenie (B), to wiemy, że równocześnie pewna lekka krawędź stała się preferowana, a liczbę takich zdarzeń już oszacowaliśmy przez $O(m \log n)$. Wreszcie, zdarzenie (C) ma miejsce tylko raz podczas operacji Dotknij, a więc całkowita liczba tych zdzarzeń jest $O(m)$. Tak więc dostaliśmy całkowitą liczbę zmian krawędzi preferowanych $O(m \log n)$.

### 2.7.2 Ograniczenie $O(\log n)$

Użyjemy metody potencjału. Podobnie jak przy analizie drzew splay, niech $s(v)$ będzie rozmiarem poddrzewa $v$ w drzewie Link-Cut (krawędzie drzew splay i krawędzie ścieżka-korzeń traktujemy tu tak samo). Potencjał definiujemy jako $\Phi = \sum_v \log s(v)$. Operacja Dotknij wykonuje pewną liczbę operacji splay, idąc w górę drzewa. Ponieważ potencjał zdefiniowaliśmy tak samo jak w drzewach splay, możemy użyć Lematu o Dostępie (patrz wykład o drzewach splay) – to nic, że tutaj drzewo nie jest biarne, możemy udawać że jest binarne dodając wagę drzew podczepionych wskaźnikami ścieżka-ojciec do wierzchołków drzewa binarnego. A więc koszt jednej operacji splay($v$) jest równy:

$$\text{cost} = 3(\log s(u) - \log s(v)) + 1$$

gdzie $u$ jest korzeniem w drzewie pomocniczym zawierającym $v$. Zauważmy, że operacja zmiany preferowanego syna nie zmienia struktury (topologicznej) drzewa Link-Cut, a więc potencjał też pozostaje ten sam. Zawuażmy, że jeśli $w$ jest ojcem w drzewie Link-Cut korzenia drzewa pomocniczego zawierającego $v$ i $u$, to $s(v) \leq s(u) < s(w)$, a więc jeśli dodamy koszty wszystkich operacji splay to dostaniemy sumę teleskopową ograniczoną przez

$$3(\log s(\text{korzeń drzewa repr.}) - \log s(v)) + O(\text{liczba zmian pref. krawędzi})$$

a to jest z kolei ograniczone przez $O(\log n)$. Żeby zakończyć analizę potrzeba jeszcze sprawdzić co się dzieje z potencjałem podczas podczepiania i odłączania drzew w operacjach Link

i Cut. Przy podłączaniu rośnie rozmiar tylko jednego węzła (ojca), o co najwyżej $n$, a więc potencjał rośnie o $O(\log n)$. Podczas odłączania potencjał maleje. A więc zamortyzowany czas Link i Cut także jest taki sam jak zamortyzowany czas Dotknij, czyli $O(\log n)$.

# 3 Zastosowanie do problemu LCA

Drzewa link-cut można zastosować do dynamicznej wersji problemu LCA: w tej wersji struktura danych ma utrzymywać *dynamiczny* las (tzn. dostępne są operacje Link i Cut) oraz przetwarzać zapytania LCA($u$,$v$) (znajdź najniższego wspólnego przodka węzłów $u$ i $v$. W tym celu używamy zwykłych drzew Link-Cut. Operację LCA implementujemy w ten sposób, że wykonujemy Dotknij($u$), a następnie Dotknij($v$). Zwracamy pierwszy wierzchołek odwiedzony podczas wykonywania Dotknij($v$), który znajduje się w drzewie pomocniczym zawierającym $u$. W ten sposób otrzymujemy strukturę danych o rozmiarze $O(n)$, na której można wykonywać wszystkie operacje w czasie $O(\log n)$. (Należy to porównać ze statyczną strukturą danych LCA, która miała rozmiar $O(n)$ i przetwarzała zapytania w czasie stałym).

# 4 Zastosowanie do problemu maksymalnego przepływu

## 4.1 Rozszerzenie problemu

Na potrzeby problemu maksymalnego przepływu rozszerzymy nasz problem drzew dynamicznych i pokażemy, że łatwo jest wzbogacić drzewa Link-Cut tak, żeby rozwiązywały także rozszerzony problem.

Dodatkowo zakładamy, że każdy wierzchołek w lesie przechowuje pewną liczbę rzeczywistą, którą będziemy nazywać *kosztem*. Do istniejących operacji dodajemy:

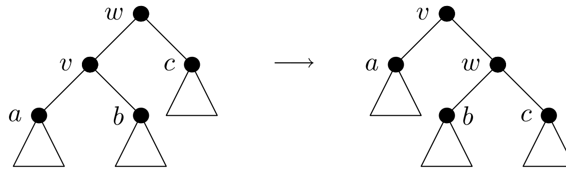
- Cost($v$) – Zwraca koszt wierzchołka $v$.
- Min($v$) – Zwraca wierzchołek o najmniejszym koszcie na ścieżce od $v$ do korzenia drzewa zawierającego $v$. Spośród wielu wierzchołków o najmniejszym koszcie wybiera ten najbliższy korzenia (to ważne!).
- AddCost($v, x$) – Dodaje $x$ do kosztów wszystkich wierzchołków na ścieżce od $v$ do korzenia.

Niech cost($x$) oznacza koszt $x$, a mincost($x$) oznacza najmniejszy koszt wierzchołka w poddrzewie $x$. Zauważmy, że przechowując mincost w węzłach drzewa Link-Cut operacja sc Min($v$) staje się prosta: po wykonaniu Dotknij($v$) wystarczy iść w dół w lewym poddrzewie $v$ wybierając zawsze gałąź z mniejszą wartością mincost, a w przypadku remisów iść w lewo. Ale to nie ułatwia nam operacji AddCost. Zamiast tego, każdy wierzchołek drzewa Link-Cut przechowuje dwie liczby:

- $\Delta\text{cost}(x)$, który jest równy $\text{cost}(x) - \text{cost}(p(x))$ gdy $x$ nie jest korzeniem drzewa pomocniczego i $\text{cost}(x)$ gdy jest.
- $\Delta\text{min}(x) = \text{cost}(x) - \text{mincost}(x)$.

Zauważmy, że przy tej reprezentacji także możemy efektywnie wykonać MIN, bo poruszając się w dół od korzenia możemy w czasie stałym wyliczać wartości cost i mincost odwiedzanych węzłów i ich synów. Teraz addcost($v$,$x$) sprowadza się do wykonania DOTKNIJ($v$) i dodania $x$ do $\Delta\text{cost}(v)$.

Pozostaje jeszcze opisać jak uaktualniać wartości $\Delta$cost i $\Delta$min po wykonaniu rotacji i operacji zmiany preferowanego syna. Rozważmy najpierw operację rotacji $v$ i $w$ jak na rysunku poniżej:




Wtedy:

$$
\begin{aligned}
\Delta\text{cost}'(v) &= \Delta\text{cost}(v) + \Delta\text{cost}(w),\\
\Delta\text{cost}'(w) &= -\Delta\text{cost}(v),\\
\Delta\text{cost}'(b) &= \text{cost}(b) - \text{cost}(w) = \Delta\text{cost}(b) + \text{cost}(v) - \text{cost}(w) = \Delta\text{cost}(b) + \Delta\text{cost}(v),\\
\Delta\text{min}'(w) &= \max\{\text{cost}(w) - \text{cost}(w), \text{cost}(w) - \text{mincost}(b), \text{cost}(w) - \text{mincost}(c)\}\\
&= \max\{0, \Delta\text{min}(b) - \Delta\text{cost}'(b), \Delta\text{min}(c) - \Delta\text{cost}(c)\},\\
\Delta\text{min}'(v) &= \max\{0, \Delta\text{min}(a) - \Delta\text{cost}(a), \Delta\text{min}'(w) - \Delta\text{cost}'(w)\}.
\end{aligned}
$$

Opiszmy także jak zmieniają się wartości przechowywanych wartości, podczas zamiany preferowanego syna. Niech $w$ będzie korzeniem drzewa pomocniczego, a $u$ jego prawym synem w drzewie pomocniczym. Niech $v$ będzie korzeniem (innego) drzewa pomocniczego zawierającym wskaźnik ścieżka-ojciec do $w$. Wtedy po zmianie prawego syna $w$ na $v$ należy zmienić jak następuje:

$$
\begin{aligned}
\Delta\text{cost}'(v) &= \Delta\text{cost}(v) - \Delta\text{cost}(w),\\
\Delta\text{cost}'(u) &= \Delta\text{cost}(u) + \Delta\text{cost}(w),\\
\Delta\text{min}'(w) &= \max\{\text{cost}(w) - \text{cost}(w), \text{cost}(w) - \text{mincost}(\text{left}(w)), \text{cost}(w) - \text{mincost}(v)\}\\
&= \max\{0, \Delta\text{min}(\text{left}(w)) - \Delta\text{cost}(\text{left}(w)), \Delta\text{min}(v) - \Delta\text{cost}'(v)\}.
\end{aligned}
$$

## 4.2 Algorytm Dinica

Zakładamy, że czytelnik zna problem znajdowania maksymalnego przepływu i podstawowe pojecia związane z tą tematyką (graf residualny, ścieżka powiększająca, twierdzenie o maksymalnym przepływie i minimalnym przekroju itd.) – są one opisane w książce Cormena,

Leisersona i Rivesta „Wprowadzenie do algorytmów". Przyjmiemy oznaczenia z tej książki, w szczególności:

- Dla dowolnych $u, v$ przez $c(u, v) \geq 0$ oznaczamy przepustowość krawędzi $(u, v)$. (Gdy $(u, v) \notin E$ to $c(u, v) = 0$).
- $f(u, v)$ oznacza przepływ po krawędzi od $u$ do $v$.
- $c_f(u, v) = c(u, v) - f(u, v)$ jest przepustowością residualną, a sieć indukowana przez niezerowe przepustwowści residualne oznaczamy przez $G_f$ i nazywamy siecią residualną.
- *Ścieżka powiększająca* jest dowolną ścieżką od $s$ do $t$ w sieci $G_f$.
- Krawędź $u \to v$ jest *nasycona*, gdy $f(u, v) = c(u, v)$.

### 4.2.1 Graf warstwowy

Niech $G$ będzie siecią o źrodle $s$ i ujściu $t$. Niech $f$ będzie przepływem w $G$. Zdefiniujmy *graf warstwowy* $L_f$ jako podgraf grafu $G_f$ taki, że $(u, v) \in E(L_f)$ wtedy i tylko wtedy gdy $d_{G_f}(s, v) = d_{G_f}(s, u) + 1$. Innymi słowy krawędź $e = (u, v) \in E(L_f)$ wtedy i tylko wtedy gdy $e$ zawiera się w pewnej najkrótszej ścieżce od $s$ do $v$ w grafie $G_f$. Graf $L_f$ łatwo utworzyć w czasie $O(n + m)$ korzystając z BFS-u:

1. Uruchamiamy BFS z $s$ i obliczamy odległości wszystkich wierzchołków od $s$
2. Dla każdej krawędzi $(u, v)$ sprawdzamy, czy $d_{G_f}(s, v) = d_{G_f}(s, u) + 1$ i jeśli tak, dodajemy ją do grafu.

### 4.2.2 Schemat algorytmu Dinica

*Przepływem blokującym* w $L_f$ nazywamy dowolny przepływ w $L_f$ taki, że każda ścieżka od $s$ do $t$ w $L_f$ zawiera krawędź nasyconą. Teraz możemy już podać schemat algorytmu Dinica:

$f \leftarrow 0$;<br>
dopóki w $L_f$ istnieje ścieżka od $s$ do $t$<br>
    znajdź przepływ blokujący $f^*$ w $L_f$<br>
    $f \leftarrow f + f^*$

Możemy powiedzieć, że algorytm Dinica składa się z *faz*, a każda faza polega na skonstruowaniu grafu $L_f$ i znalezieniu w nim przepływu blokującego. Poprawność algorytmu Dinica wynika natychmiast z twierdzenia o maksymalnym przepływie i minimalnym przekroju (po zakończeniu algorytmu nie ma już ścieżek powiększających).

Główna siła algorytmu Dinica wynika z następującego faktu:

**Fakt 1.** Po każdej fazie algorytmu Dinica odległość od $s$ do $t$ w sieci residualnej rośnie.

*Dowód.* (Ćwiczenie.) □

Powyższy fakt gwarantuje, że algorytm Dinica wykona nie więcej niż $n-1$ faz.

### 4.2.3 Znajdowanie przepływu blokującego: algorytm Dinica

Opiszemy teraz oryginalny algorytm Dinica znajdowania przepływu blokującego w sieci warstwowej. Oznaczmy znajdowany przepływ blokujący przez $f^*$. Algorytm będzie obliczał $f^*$ i podczas swojego działania będzie modyfikował sieć $L_f$ (usuwając krawędzie). Algorytm używa zmiennej $v$ (wierzchołek, w którym jesteśmy) oraz w $p$ przechowuje pewną ścieżkę od $s$. Początkowo kładziemy $f^* \leftarrow 0$, $p \leftarrow ()$ oraz $v \leftarrow s$. Następnie, dopóki jest to możliwe wykonujemy jedną z dwóch operacji: Advance lub Retreat. Operację Retreat wykonujemy tylko wtedy, gdy nie możemy już wykonać Advance. Operację Advance możemy wykonać jeśli istnieje krawędź wychodząca z $v$. Operację Retreat możemy wykonać zawsze, o ile $v \neq s$.

```
Advance:
w ← dowolny wierzchołek taki, że (v, w) ∈ L_f
Push(p, w)
v ← w
if w = t /* znaleziono ścieżkę powiększającą */
   while p ≠ ()
      x ← najmniejsza przepustowość c_f na ścieżce p
      u ← Pop(p)
      f*(u, v) ← f*(u, v) + x
      c_f(u, v) ← c_f(u, v) − x
      if c_f(u, v) = 0 then Usuń krawędź (u, v) z grafu
      v ← u
   v ← s
```

```
Retreat:
u ← Pop(p)
Usuń krawędź (u, v) z grafu
v ← u
```

Jest jasne, że powyższy algorytm znajdowania przepływu blokującego jest poprawny (ćwiczenie). Oznaczmy przez $n$ liczbę wierzchołków sieci a przez $m$ liczbę krawędzi. Operacji Retreat wykonujemy co najwyżej $m$. Podczas operacji Advance przechodzimy z wierzchołka $v$ do wierzchołka $w$, który znajduje się w następnej warstwie. Operacje Advance możemy podzielić na dwa rodzaje: takie, że krawędź $(v, w)$ wejdzie w skład ścieżki powiększającej i takie, że krawędź $(v, w)$ zostanie usunięta podczas Retreat. Operacji drugiego

rodzaju jest co najwyżej $m$ (tyle razy krawędzie są usuwane). Ścieżek powiększających znajdujemy co najwyżej $m$ (bo taka ścieżka zawiera przynajmniej jedną krawędź nasyconą, która jest usuwana), natomiast długość takiej ścieżki to co najwyżej $n$. Stąd operacji pierwszego rodzaju jest $O(mn)$.

**Wniosek 1.** *Algorytm Dinica znajduje przepływ blokujący w czasie $O(mn)$. Cały algorytm znajdowania maksymalnego przepływu działa wtedy w czasie $O(mn^2)$.*

### 4.2.4 Znajdowanie przepływu blokującego: algorytm Tarjana i Sleatora

Sleator i Tarjan zaproponowali, aby przepływ blokujący znajdować wykorzystując drzewa Link-Cut. Wierzchołki sieci warstwowej będą węzłami drzew Link-Cut (w szczególności jeden wierzchołek należy do co najwyżej jednego drzewa). W danej chwili dla dowolnego wierzchołka $v$ istnieje co najwyżej jedna krawędź wychodząca z $v$, która znajduje się w pewnym drzewie Link-Cut (chodzi o drzewo reprezentowane). Każdy wierzchołek $v$ różny od korzenia ma koszt równy przepustowości aktualnej residualnej krawędzi $(v, p(v))$, gdzie $p(v)$ jest ojcem $v$ w drzewie reprezentowanym.

Na początku każdy wierzchołek jest osobnym drzewem. Podczas działania algorytmu drzewa łączone są w większe drzewa. Od momentu wstawienia pewnej krawędzi do drzewa (za pomocą LINK), jej aktualna przepustowość zapamiętana jest jedynie w drzewie Link-Cut. W chwili, gdy powstanie drzewo, którego liściem jest $s$ a korzeniem $t$, takie drzewo zawiera ścieżkę powiększającą, po której przesyłany jest przepływ. Przesłanie przepływu realizujemy przez zmniejszenie przepustowości krawędzi residualnych na ścieżce (operacją ADDCOST) i krawędzie o przepustowości 0 są usuwane (FINDMIN i CUT). W ten sposób drzewo rozpada się na kilka (przynajmniej 2) drzew. W sytuacji, gdy z korzenia $v$ pewnego drzewa nie ma już w grafie krawędzi wychodzących, obliczamy koszty (aktualne przepustowości residualne) wszystkich krawędzi wchodzących do $v$ i usuwamy je z grafu. Dla każdej takiej krawędzi $(u, v)$ różnica początkowej przepustowości $c_f(u, v)$ i aktualnej przepustowości daje na końcu wartość przepływu blokującego $f^*(u, v)$. Algorytm ponownie wyrazimy jako parę operacji Advance i Retreat.

Jak już napisaliśmy, zaczynamy od utworzenia drzewa Link-Cut dla każdego wierzchołka. Następnie wykonujemy:

```
Algorytm:
f* ← 0
while True
   v ← Root(s)
   if v = t /* jest ścieżka powiększająca */
      w ← Min(s)
      AddCost(s, −Cost(w))
      while Cost(w) = 0
         Usuń (w, p(w)) z L_f.
         f*(w, p(w)) ← c_f(w, p(w))
         Cut(w)
         w ← Min(s)
   else
      if (v, w) ∈ L_f dla pewnego w
         Advance
      else /* nie ma krawędzi wychodzących z v */
         if v = s
            dla każdej krawędzi (u, v) w każdym drzewie Link-Cut:
               f*(u, v) ← c_f(u, v)−Cost(u)
            ZATRZYMAJ algorytm.
         else
            Retreat
```

```
Advance:
w ← dowolny wierzchołek taki, że (v, w) ∈ L_f
AddCost(v, c_f(v, w))
Link(v,w)
```

```
Retreat:
for each (u, v) ∈ L_f
   f*(u, v) ← c_f(u, v)−Cost(u)
   Cut(u)
   Usuń (u, v) z L_f.
```

**Fakt 2.** Algorytm Tarjana-Sleatora znajduje przepływ blokujący w czasie $O(m \log n)$. Cały algorytm znajdowania maksymalnego przepływu działa wtedy w czasie $O(mn \log n)$.

(Dowód jako ćwiczenie).

# Literatura

[1] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comp. System Sci.*, 26(3):362–391, 1983.

[2] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. A.C.M.*, 32(3):652–686, 1985.