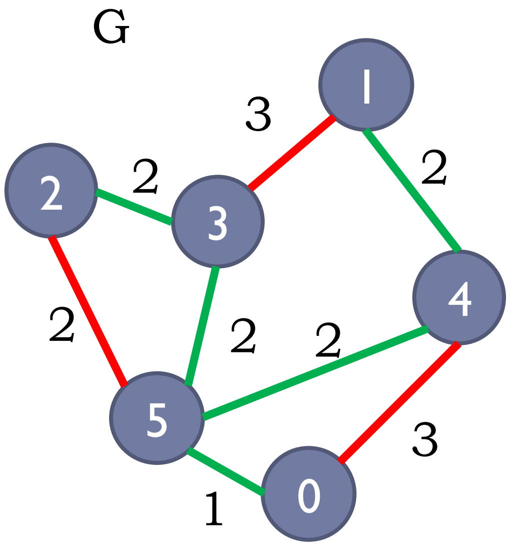


Minimalne drzewa rozpinające (lub spinające) MST

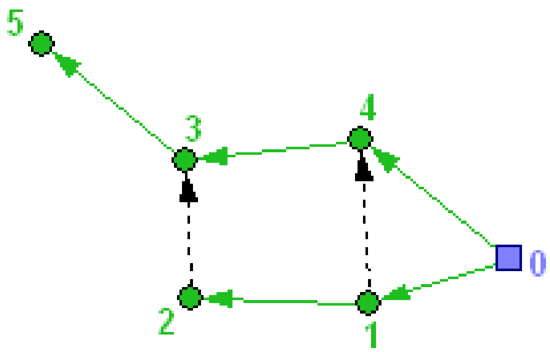
dr Anna Beata Kwiatkowska, UMK Toruń

Drzewa w różnych problemach

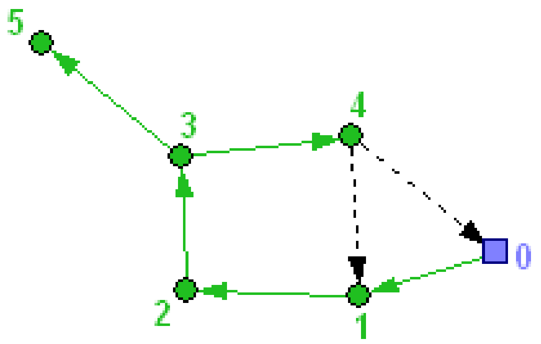
Drzewa pojawiały się do tej pory w różnych sytuacjach:

- drzewo najkrótszych co do liczby krawędzi dróg - przeszukiwanie grafu wszerz (bfs)
- drzewo poszukiwań z nawrotami, odwiedzanie wierzchołków w porządku preorder, postorder – przeszukiwanie grafu w głąb (dfs)
- szukanie najkrótszych dróg w sieciach, drzewa najkrótszych dróg – algorytm Dijkstry, itd.
- sortowanie na drzewie – sortowanie na kopcu

Wszerz BFS



W głąb DFS



Drzewo rozpinające

Rozważmy graf spójny G(V, E). Możemy budować drzewo grafu G zawierające wszystkie jego wierzchołki:

- Jeśli G nie jest drzewem, tzn. ma za dużo krawędzi, można je wyrzucać bez rozspojenia, usuwając cykle.
- Można też postępować inaczej – startując od pustego zbioru krawędzi dodawać krawędzie tak, aby nie powstał cykl.

Definicja 8.1

Dla dowolnego grafu spójnego G (V,E) każde drzewo T(V, F) takie, że F $\subseteq$ E, nazywamy drzewem rozpinającym (spinającym) grafu G.

Drzewo rozpinające grafu G jest jego podgrafem spójnym o minimalnej liczbie krawędzi. Zapewnia więc osiągalność każdego wierzchołka grafu G.

Sieci - przypomnienie

Definicja 6.3

W grafie G (V,E) (lub digrafie) definiujemy funkcją na jego krawędziach (łukach), określającą wagę każdej z nich:

$$w : E \longrightarrow R$$

Graf (digraf) G z tak określoną funkcją wag nazywamy grafem ważonym (siecią).

Definicja 6.4.

Dla podgrafu H (V_H, E_H) grafu G określamy długość d podgrafu H, jako sumę wag wszystkich jego krawędzi e:

$$d(H) = \sum_{e \in E_H} w(e)$$

Mając tak określoną funkcję d możemy mówić np. o długości drogi w grafie, o długości drzewa itd.

Drzewa rozpinające grafu (tak samo digrafy)

Graf może mieć wiele drzew rozpinających. Rozważmy zbiór wszystkich drzew rozpinających grafu G.

$$\mathcal{T}_G = \{ T:\ T \text{ drzewo rozpinające grafu } G \}$$

Definicja 8.2

Rozważmy graf spójny G(V, E), z funkcją wag $w : E \longrightarrow R$.

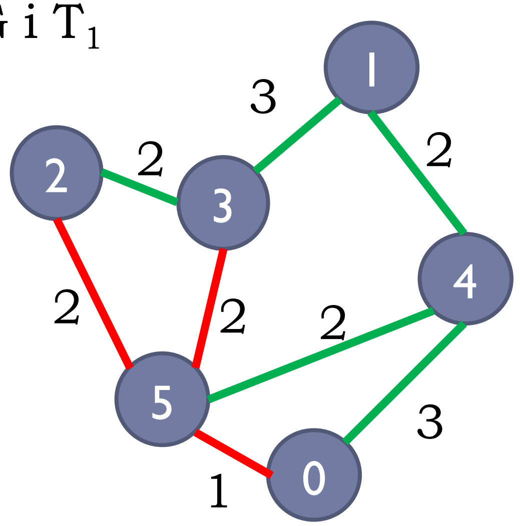
Minimalnym drzewem rozpinającym grafu G(V,E) (MST, *ang. minimum spanning tree*) nazywamy drzewo rozpinające T(V, F), $F \subseteq E$, grafu G, takie, że wartość d(T) jest najmniejsza.

Graf i jego przykładowe drzewa rozpinające

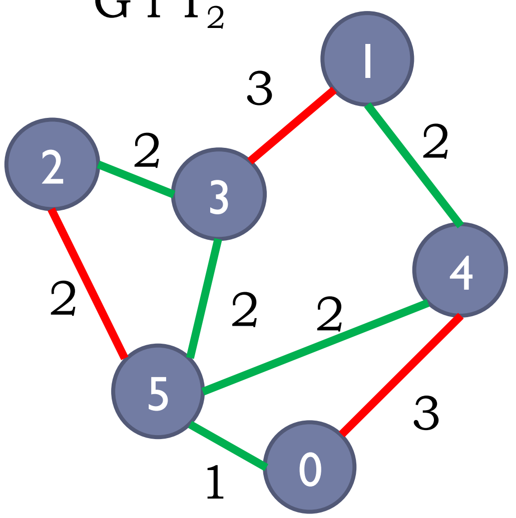
Przykład:

Graf G i jego drzewo rozpinające T_1 oraz minimalne drzewo rozpinające T_2.

G i T_1



G i T_2



MST - własności

Własność 8.1

Jeśli wagi wszystkich krawędzi grafu G=(V,E) są różne, to istnieje tylko jedno minimalne drzewo rozpinające tego grafu.

Dowód:

Sortujemy krawędzie grafu rosnąco względem wag. Budujemy MST przeglądając krawędzie wg rosnących kosztów. Do drzewa zaliczamy krawędzie nietworzące cyklu z dotychczas wybranymi krawędziami. Koszty krawędzi są różne, więc wybór jest jednoznaczny.

□

MST - własności

Własność 8.2

W grafie G=(V,E) dla dowolnego podzbioru wierzchołków $V_1 \subset V$, $V_1 \neq \emptyset$, $V_1 \neq V$, krawędź e={v,u}, $v \in V_1$, u$\in V \setminus V_1$ o minimalnej wadze spośród krawędzi łączących wierzchołki zbioru V_1 z wierzchołkami $V \setminus V_1$ należy do minimalnego drzewa rozpinającego.

Dowód:

Niech V_1 jest takie, że można wybrać krawędź e={v,u}, $v \in V_1$, u$\in V \setminus V_1$ o minimalnym koszcie, która nie należy do minimalnego drzewa rozpinającego.

Niech e_1={s,t}, s$\in V_1$, t $\in V \setminus V_1$ będzie krawędzią o większej wadze niż waga krawędzi e, która należy do minimalnego drzewa rozpinającego. W drzewie tym istnieje tylko jedna ścieżka łącząca wierzchołki v i u, która przechodzi przez krawędź e_1.

Dodanie do tego minimalnego drzewa krawędzi e powoduje utworzenie cyklu zawierającego e i e_1. Usuwając krawędź e_1 otrzymamy drzewo o niższym koszcie.

□

Minimalne drzewo rozpinające - MST

Dane:

Spójny graf ważony $G=(V, E)$, $|V| = n$, $|E| = m$, $w : E \longrightarrow R$

Wynik:

Znaleźć najkrótsze drzewo rozpinające grafu G, tzn. znaleźć drzewo T*, takie że:

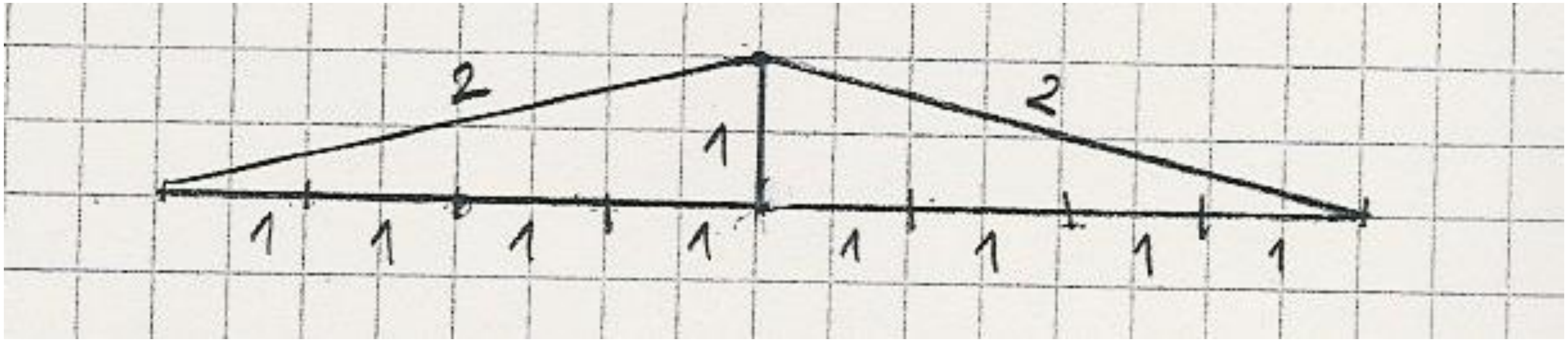
$$d(T^*) = \min \{ d(T) : T \in \mathcal{T}_G \}$$

Zastosowanie:

Znaleźć połączenie zbioru terminali za pomocą sumarycznie najkrótszej sieci kabli.

Drzewo najkrótszych dróg a minimalne drzewo rozpinające

Przykład grafu z obciążonymi krawędziami, dla którego żadne drzewo najkrótszych dróg nie jest minimalnym drzewem rozpinającym.



Algorytm Kruskala 1956 – opis działania

Algorytm znajduje minimalne drzewo rozpinające dla spójnej sieci **metodą zachłanną.**

- Drzewo T tworzymy rozpatrując krawędzie w porządku ich niemalejacych wag.
- Jeśli badana krawędź tworzy cykl z krawędziami dotychczas wybranymi do T, to ją pomijamy. W przeciwnym razie krawędź ta jest dołączana do T.
- Postępowanie to przerywamy, gdy zostało wybranych n-1 krawędzi lub gdy zostało rozpatrzonych wszystkich m krawędzi.
- Jeśli rozważana sieć nie jest spójna, to otrzymujemy najkrótszy las rozpinający.

Algorytm Kruskala - pseudokod

$T \leftarrow \emptyset$; $E' \leftarrow E$;
while $|T| < n - 1$ and $E' \neq \emptyset$ do
begin
    e ← najkrótsza krawędź w E';
    $E' \leftarrow E' - \{e\}$;
    if $T \cup \{e\}$ nie zawiera cyklu then $T \leftarrow T \cup \{e\}$;
end;
If $|T| < n - 1$ then write ('sieć nie jest spójna');

Algorytm jest bardzo prosty, trzeba jednak:

- opracować szczegóły implementacji,
- wybrać odpowiednie struktury danych

tak, aby otrzymać efektywna realizacje komputerową.

Algorytm Kruskala - efektywność

Elementami decydującymi o efektywności algorytmu Kruskala są:

Wybór krawędzi w kolejności niemalejących wag.

Jeśli na początku uporządkujemy wszystkie m krawędzi, może okazać się, że zrobiliśmy to niepotrzebnie.

np. Dla $|V|=100$ mamy $|E|<4951$, a wystarczy, że zbadamy tylko kilkaset krawędzi.

Wystarczy więc, jeśli krawędzie są tylko częściowo uporządkowane, dlatego umieszczamy je na kopcu, z najkrótszą krawędzią w jego korzeniu.

Zbudowanie kopca początkowego z m krawędzi wymaga $O(m)$ działań, każda następna krawędź może być otrzymana z kopca w czasie $O(\log m)$. Wybór krawędzi może więc być zrealizowany w czasie $O(m+p*\log m)$, gdzie p jest liczba wszystkich krawędzi zbadanych w celu utworzenia drzewa MST.

Metoda reprezentowania wybranych krawędzi.

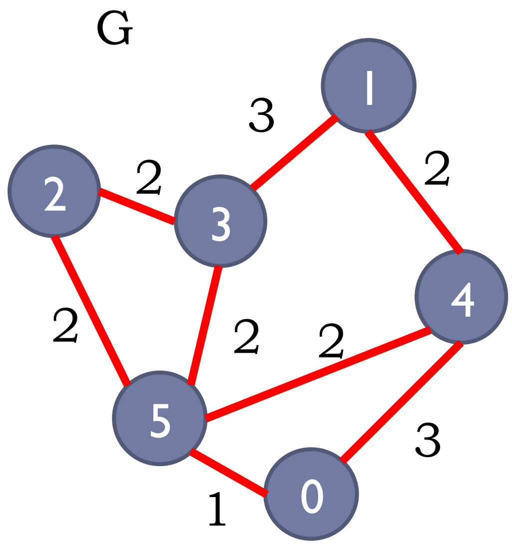
Metoda powinna szybko dawać odpowiedź na pytanie, czy dana krawędź zamyka cykl. W trakcie badania i dołączania krawędzi do T, rozwiązanie częściowe jest lasem rozłącznych drzew. Krawędź e tworzy cykl z krawędziami w T, gdy oba jej końce należą do tego samego poddrzewa. Aby dodać krawędź e do T wystarczy więc zbadać, czy e łączy dwa różne poddrzewa w T.

Algorytm Kruskala i operacje FIND i UNION

Tworzymy zbiory rozłączne zawierające po jednym wierzchołku. Każdy wierzchołek reprezentuje samego siebie (wskaźnik na siebie).

Przykład:

{0}, {1}, {2}, {3}, {4}, {5}

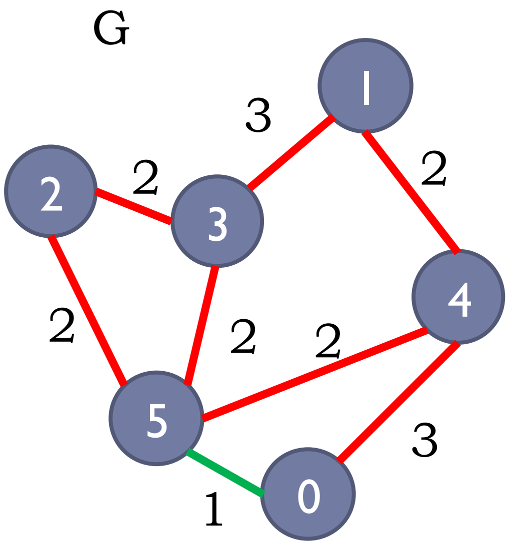


Algorytm Kruskala i operacje FIND i UNION

Wybieramy najlżejszą co do wagi w kolejności krawędź (na kopcu. Porównujemy wyniki funkcji find, która znajduje reprezentanta wierzchołka, dla wierzchołków z krawędzi.

Jeśli reprezentanci są różni, możemy wybrać krawędź, więc wszystkim elementom mniej licznego zbioru przypisujemy wskaźnik bezpośrednio (kompresja ścieżek) do reprezentanta z bardziej licznego zbioru. Jeśli zbiory są równoliczne, nie ma to znaczenia. W ten sposób łączymy zbiory (union).

5 ma teraz jako reprezentanta 0.



Przykład:

{0}, {1}, {2}, {3}, {4}, {5}

po wybraniu (0,5) mamy {0,5}, {1}, {2}, {3}, {4}

Algorytm Kruskala i operacje FIND i UNION

Wybieramy najlżejszą co do wagi w kolejności krawędź (na kopcu. Porównujemy wyniki funkcji find, która znajduje reprezentanta wierzchołka, dla wierzchołków z krawędzi.

Jeśli reprezentanci są różni, możemy wybrać krawędź, więc wszystkim elementom mniej licznego zbioru przypisujemy wskaźnik bezpośrednio (kompresja ścieżek) do reprezentanta z bardziej licznego zbioru. Jeśli zbiory są równoliczne, nie ma to znaczenia. W ten sposób łączymy zbiory (union).

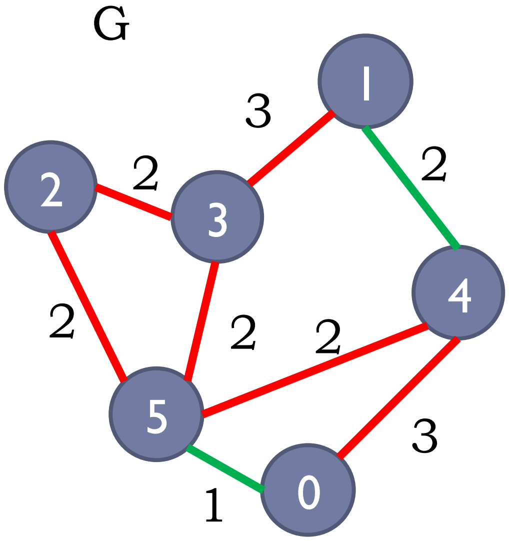
4 ma teraz jako reprezentanta 1.

Przykład:

{0}, {1}, {2}, {3}, {4}, {5}

po wybraniu (0,5) mamy {0,5}, {1}, {2}, {3}, {4}

po wybraniu (4,1) mamy {0,5}, {1,4}, {2}, {3}

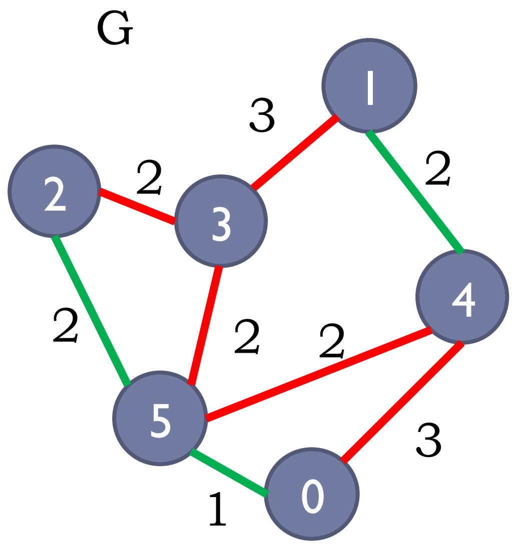


Algorytm Kruskala i operacje FIND i UNION

Wybieramy najlżejszą co do wagi w kolejności krawędź (na kopcu. Porównujemy wyniki funkcji find, która znajduje reprezentanta wierzchołka, dla wierzchołków z krawędzi.

Jeśli reprezentanci są różni, możemy wybrać krawędź, więc wszystkim elementom mniej licznego zbioru przypisujemy wskaźnik bezpośrednio (kompresja ścieżek) do reprezentanta z bardziej licznego zbioru. Jeśli zbiory są równoliczne, nie ma to znaczenia. W ten sposób łączymy zbiory (union).

2 ma teraz jako reprezentanta 0.



Przykład:

{0}, {1}, {2}, {3}, {4}, {5}

po wybraniu (0,5) mamy {0,5}, {1}, {2}, {3}, {4}

po wybraniu (4,1) mamy {0,5}, {1,4}, {2}, {3}

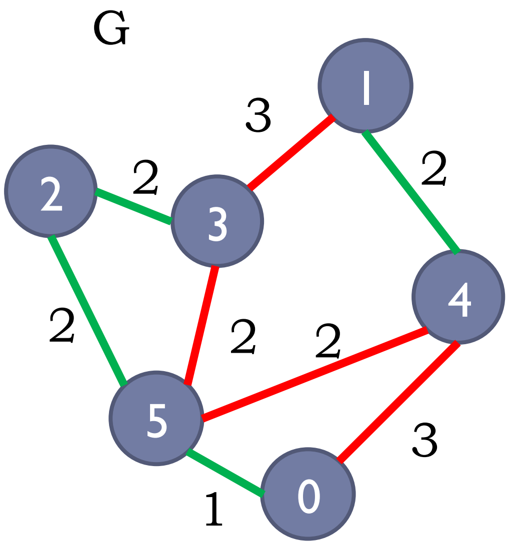
po wybraniu (2,5) mamy {0,5,2}, {1,4}, {3}

Algorytm Kruskala i operacje FIND i UNION

Wybieramy najlżejszą co do wagi w kolejności krawędź (na kopcu. Porównujemy wyniki funkcji find, która znajduje reprezentanta wierzchołka, dla wierzchołków z krawędzi.

Jeśli reprezentanci są różni, możemy wybrać krawędź, więc wszystkim elementom mniej licznego zbioru przypisujemy wskaźnik bezpośrednio (kompresja ścieżek) do reprezentanta z bardziej licznego zbioru. Jeśli zbiory są równoliczne, nie ma to znaczenia. W ten sposób łączymy zbiory (union).

3 ma teraz jako reprezentanta 0.



Przykład:

{0}, {1}, {2}, {3}, {4}, {5}

po wybraniu (0,5) mamy {0,5}, {1}, {2}, {3}, {4}

po wybraniu (4,1) mamy {0,5}, {1,4}, {2}, {3}

po wybraniu (2,5) mamy {0,5,2}, {1,4}, {3}

po wybraniu (2,3) mamy {0,5,2,3}, {1,4}

Algorytm Kruskala i operacje FIND i UNION

Krawędzi (3,5) nie wybieramy, bo 3 i 5 mają tego samego reprezentanta, więc utworzylibyśmy cykl.

Przykład:

{0}, {1}, {2}, {3}, {4}, {5}

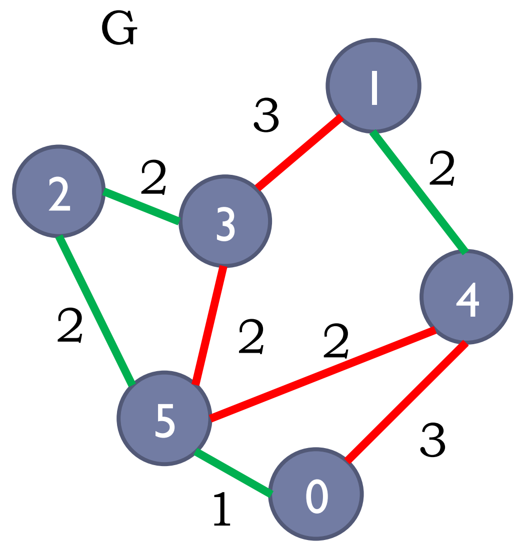
po wybraniu (0,5) mamy {0,5}, {1}, {2}, {3}, {4}

po wybraniu (4,1) mamy {0,5}, {1,4}, {2}, {3}

po wybraniu (2,5) mamy {0,5,2}, {1,4}, {3}

po wybraniu (2,3) mamy {0,5,2,3}, {1,4}

Krawędzi (3,5) nie wybieramy



Algorytm Kruskala i operacje FIND i UNION

Elementom zbioru {1,4} przypisujemy bezpośrednio reprezentanta zbioru mniej licznego, czyli zbioru {0,5,2,3}, którym jest 0.

Przykład:

{0}, {1}, {2}, {3}, {4}, {5}

po wybraniu (0,5) mamy {0,5}, {1}, {2}, {3}, {4}

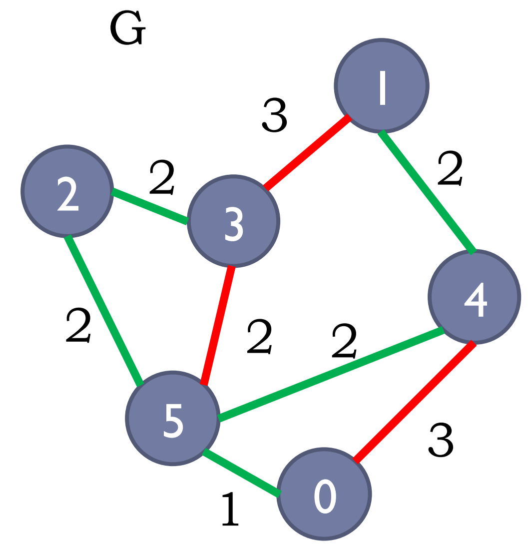
po wybraniu (4,1) mamy {0,5}, {1,4}, {2}, {3}

po wybraniu (2,5) mamy {0,5,2}, {1,4}, {3}

po wybraniu (2,3) mamy {0,5,2,3}, {1,4}

Krawędź (3,5) nie wybieramy

po wybraniu (5,4) mamy {0,5,2,1,4}



Algorytm Kruskala i operacje FIND i UNION

Przykład:

{0}, {1}, {2}, {3}, {4}, {5}

po wybraniu (0,5) mamy {0,5}, {1}, {2}, {3}, {4}

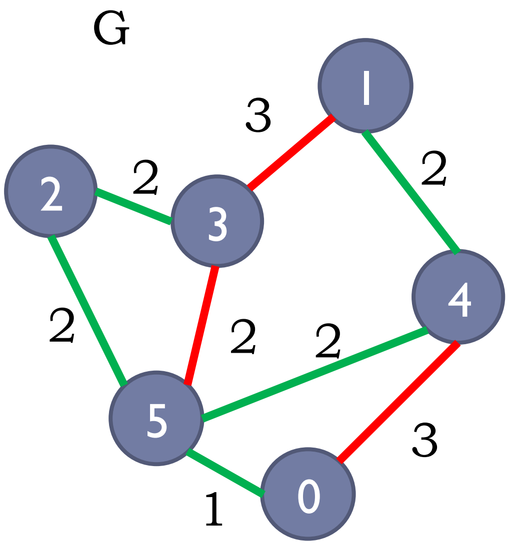
po wybraniu (4,1) mamy {0,5}, {1,4}, {2}, {3}

po wybraniu (2,5) mamy {0,5,2}, {1,4}, {3}

po wybraniu (2,3) mamy {0,5,2,3}, {1,4}

Krawędź (3,5) nie wybieramy

po wybraniu (5,4) mamy {0,5,2,1,4}

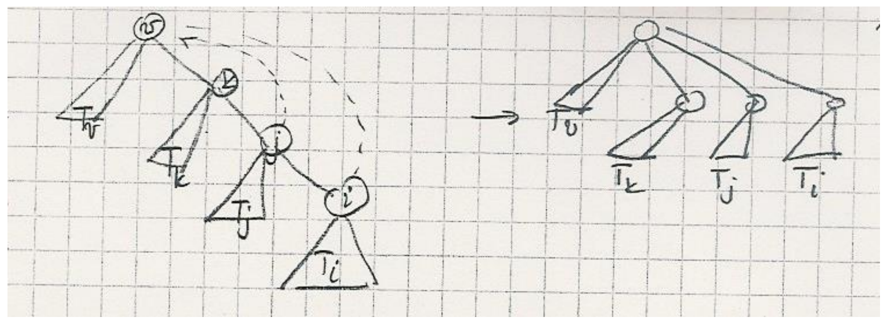


Podsumowując:

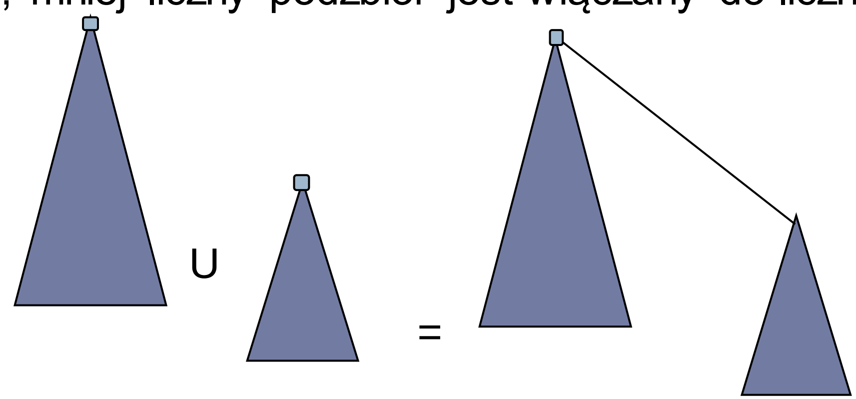
Nową krawędź dołączamy do rozwiązania tworząc sumę (ang. **union**) odpowiednich podzbiorów, a sprawdzamy, czy tworzy cykl znajdując (ang. **find**), czy oba końce badanej krawędzi należą do tego samego podzbioru. Elementy wskazują na reprezentanta bezpośrednio (**kompresja ścieżek**).

Algorytm Kruskala – implementacja

Operacja FIND przebiega z kompresją ścieżek – naszym celem jest by poszukiwanie reprezentanta, do którego należy dany wierzchołek, było możliwie krótkie.



Operacja UNION jest realizowana w wersji zbalansowanej – aby otrzymać sumę dwóch podzbiorów, mniej liczny podzbiór jest włączany do liczniejszego.



Algorytm Kruskala – struktury danych

- Sieć pamiętamy w postaci tablicy krawędzi z wagami (trzy tablice).
- Każdy podzbiór reprezentowany jest przez drzewo z korzeniem (root) identyfikującym ten podzbiór.
- Implementacją tych podzbiorów jest tablica wskaźników father, które wskazują na poprzednika. Wskaźnik father dla korzenia jest pusty.
- Wygodnie jest przyjąć:

  father(root) = liczba wierzchołków w drzewie ze znakiem przeciwnym.

Algorytm Kruskala - pseudokod szczegółowo

Inicjalizacja

for v ∈ V do father(v)= -1;

utworzyć kopiec początkowy ze wszystkich m krawędzi;

ecount ← 0; {liczba zbadanych krawędzi grafu}

tcount ← 0; {liczba krawędzi w T}

T ← Ø;

Iteracja

```
while  tcount < n -1  and  ecount < m  do
begin
        e ← krawędź (u, v) z wierzchołaka kopca;
        usunąć e z kopca i przywrócić warunek kopca;

        ecount ← ecount +1;
        r1 ←FIND(u); r2 ← FIND(v);
        if  r1 ≠ r2 then
        begin
              T ← T U {e};
              tcount ← tcount + 1;
              UNION(r1, r2);
        end;
end;
If  tcount< n -1  then write (‘sieć nie jest spójna’);
```

Algorytm Prima-Dijkstry – opis działania

Algorytm wyznacza najkrótsze drzewo rozpinające **metodą najbliższego sąsiada.**

MST tworzymy od dowolnie wybranego wierzchołka s, łącząc go z najbliższym wierzchołkiem do niego przyległym, nich to będzie y. Czyli najkrótsza krawędź (s, y) incydentna z s zostaje jako pierwsza zaliczona do MST.

Następnie spośród wszystkich krawędzi incydentnych z s lub y wybieramy najkrótszą krawędź prowadzącą do trzeciego wierzchołka i dołączamy ja do rozwiązania.

Kontynuujemy to postępowanie, aż wszystkie wierzchołki osiągalne z s zostaną włączone do rozwiązania.

Algorytm Prima-Dijkstry – opis działania

Podstawowym krokiem w algorytmie jest znajdywanie następnej krawędzi, dołączanej do rozwiązania MST.

near – tablica taka, że near(u) dla $u \in (V - V_T)$ jest wierzchołkiem w V_T najbliższym u.

dist – tablica taka, że dist(u) jest bieżącą odległością wierzchołka u od V_T

Aby określić następny wierzchołek, który ma być dołączony do V_T , porównujemy wszystkie niezerowe wartości w tablicy dist i wybieramy najmniejszą. Dla i-tego wierzchołka wystarczy wykonać $n - i - 1$ porównań.

Sieć jest dana jako macierz wag W, w której nieistniejące krawędzie mają wagi ∞.

Algorytm Prima-Dijkstry - pseudokod

Inicjalizacja:

wybrać dowolny wierzchołek
początkowy s;

near(s) ← 0;

for v ∈ V – {s} do

begin

near(v) ← s;

dist(v) ← w[s,v];

end;

V_T ← {s}; {zb. wierzchołków rozwiązania}

E_T ← ø; {zb. krawędzi rozwiązania}

Iteracja:

while |V_T| < n do

begin

u ← wierzchołek w V - V_T o najmniejszej wartości dist (u);

if dist(u) ≥ ∞ then write ('graf nie jest spójny'); Exit;

E_T ← E_T ∪ {(u, near(u))};

V_T ← V_T ∪ {u};

for v ∈ (V - V_T) do

if w[u,v]<dist(v) then

begin

dist(v) ← w[u,v];

near(v) ← u;

end;

end;

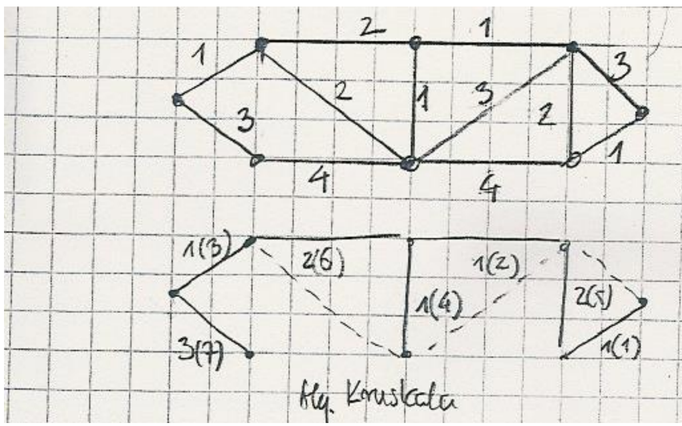
Złożoność algorytmu Prima-Dijkstry

- Implementacja algorytmu Prima-Dijkstry z użyciem macierzy sąsiedztwa ma złożoność czasową $O(n^2)$.
- Używając kopca binarnego i reprezentacji grafu za pomocą list sąsiedztwa mamy złożoność O(q logn), gdzie q jest liczbą rozpatrywanych krawędzi.
- Stosując kopiec Fibonacciego redukujemy złożoność czasową do O(q+n logn).
- Dla grafów o mniejszej gęstości znany jest algorytm o mniejszej złożoności czasowej.

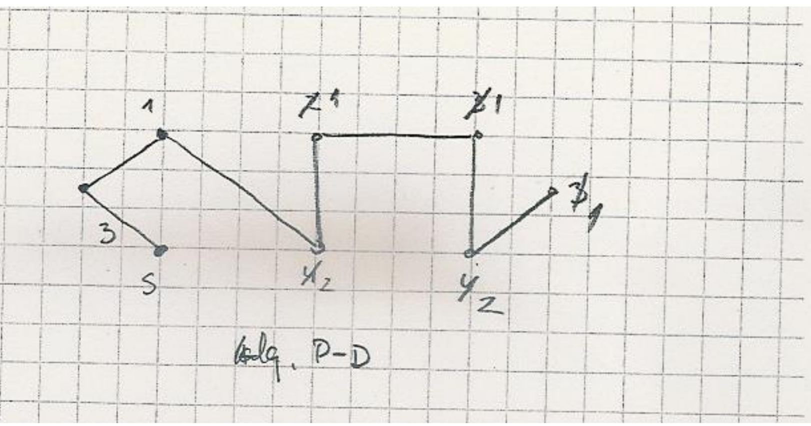
Algorytmy poszukujące najkrótszych drzew

Porównanie drzew wynikowych MST

Algorytm Kruskala



Algorytm Prima-Dijkstry



Dziękuję z uwagę

dr Anna Beata Kwiatkowska

aba@mat.umk.pl

tel. 602 184 813