Algorytmy grafowe

Przypomnienie. Graf możemy reprezentować w pamięci na dwa sposoby:

- macierz sąsiedztwa
- listy sąsiedztwa

W algorytmach nie będziemy jawnie odwoływać się do tych struktur danych. Będziemy dopuszczać wyrażenia typu u–$v \in E$ oraz konstrukcje **for all** $v \in X$, na oznaczenie pętli, która wykonuje się $|X|$ razy, za każdym razem zmienna v oznacza inny wierzchołek ze zbioru X. Podczas całego wykładu przez n będziemy oznaczać liczbę wierzchołków grafu, przez m liczbę jego krawędzi.

Przykład problemu grafowego. Dla danego grafu nieskierowanego obliczyć liczbę spójnych składowych i każdemu wierzchołkowi przypisać numer jego spójnej składowej.

Przeszukiwanie grafu

Często przy rozwiązywaniu problemów grafowych konieczne jest zebranie informacji o strukturze grafu. W tym celu najczęściej przeszukujemy graf, tzn. „wędrujemy" po grafie przechodząc wzdłuż krawędzi i odwiedzając kolejne wierzchołki.

Podczas przeszukiwania grafu wierzchołki będą odwiedzane w kolejności charakterystycznej dla wybranego algorytmu przeszukiwania. Nie jest obojętne, jaki algorytm wybierzemy – dobieramy najwłaściwszy algorytm do problemu. Poznamy dwie najważniejsze metody przeszukiwania grafu:

- Przeszukiwanie w głąb (DFS, Depth First Search)
- Przeszukiwanie wszerz (BFS, Breadth First Search).

Przeszukiwanie w głąb

Idea przeszukiwania w głąb polega na tym, że od pewnego wierzchołka idziemy ścieżką po wierzchołkach jeszcze nieodwiedzonych tak *głęboko* jak się da. Gdy nie możemy już przedłużyć ścieżki – wycofujemy się do poprzedniego wierzchołka i próbujemy z niego wyruszyć dalej po wierzchołkach nieodwiedzonych. Spójrzmy na algorytm 1.

Za każdym razem, gdy wykonuje się obrót pętli **for all** $x \in Sasiedzi[u]$, sprawdzana jest jedna krawędź grafu (od u do x). Jeśli wierzchołek x nie jest jeszcze odwiedzony, przechodzimy od u do x. Algorytm sprawdza każdą krawędź grafu co najwyżej dwa razy. Dlatego złożoność czasowa algorytmu jest $\Theta(n + m)$.

Zastosowanie: spójne składowe Algorytmy przeszukiwania są często szkieletami innych algorytmów rozwiązujących konkretne problemy. Rozważmy np. problem spójnych składowych. Algorytm 2 pokazuje, jak rozszerzyć DFS aby znajdować spójne składowe.

Algorithm 1 DFS

```
1: for all v ∈ V do
2:     Odwiedzony[u] ← False
3: for all v ∈ V do
4:     if not Odwiedzony[v] then
5:         OdwiedźDFS(v)
```

procedure OdwiedźDFS(u)

```
1: Odwiedzony[u] ← True
2: for all x ∈ Sasiedzi[u] do
3:     if not Odwiedzony[x] then
4:         OdwiedźDFS(x)
```

Algorithm 2 Zastosowanie DFS: znajdowanie spójnych składowych.

```
1: for all v ∈ V do
2:     Odwiedzony[u] ← False
3: Numer ← 0 {zmienna globalna}
4: for all v ∈ V do
5:     if not Odwiedzony[v] then
6:         Numer ← Numer + 1
7:         OdwiedźDFS(v)
```

procedure OdwiedźDFS(u)

```
1: Odwiedzony[u] ← True
2: Skladowa[u] ← Numer
3: for all x ∈ Sasiedzi[u] do
4:     if not Odwiedzony[x] then
5:         OdwiedźDFS(x)
```

Po zakończeniu Algorytmu 2 zmienna $Numer$ zawiera liczbę spójnych składowych grafu. Dla każdego wierzchołka grafu u, w komórce tablicy $Skladowa[u]$ znajduje się numer spójnej składowej, do której należy u.

Przeszukiwanie wszerz

Podczas przeszukiwania wszerz nie zawsze odwiedzane są wszystkie wierzchołki grafu. Odwiedzamy jedynie wierzchołki *osiągalne* z pewnego ustalonego wierzchołka s, tzn. takie wierzchołki, do których istnieje ścieżka z wierzchołka s. W przypadku grafu nieskierowanego oznacza to po prostu, że odwiedzamy wszystkie wierzchołki w spójnej składowej zawierającej s.

Idea algorytmu: najpierw odwiedzamy wierzchołki odległe o 1 od s, potem odległe o 2, 3, 4, itd...

Natychmiastowe zastosowanie przeszukiwania wszerz: Dla każdego wierzchołka x w grafie

znajdziemy odległość od s do x; jej wartość zapiszemy w tablicy $d[x]$. W tablicy $Skad[x]$ zapiszemy numer wierzchołka, z którego weszliśmy do x.

Algorithm 3 BFS

```
1: for all v ∈ V − {s} do
2:    Odwiedzony[v] ← False
3:    d[v] ← +∞
4:    Skad[v] ← nil
5: Odwiedzony[s] ← True
6: d[s] ← 0
7: Skad[s] ← nil
8: Q.Utwórz {tworzy pustą kolejkę FIFO Q}
9: Q.Dodaj(s)
10: while not Q.Pusta do
11:    u ← Q.Usuń
12:    for all x ∈ Sasiedzi[u] do
13:       if not Odwiedzony[x] then
14:          Odwiedzony[x] ← True
15:          d[x] ← d[u] + 1
16:          Skad[x] ← u
17:          Q.Dodaj(x)
```

Lemat 1. *Niech u będzie wierzchołkiem na początku kolejki Q w dowolnym momencie wykonania algorytmu BFS. Za wierzchołkiem u w kolejce pojawia się pewna liczba wierzchołków v takich że $d[v] = d[u]$, a w dalszej kolejności aż do końca kolejki następują wierzchołki v takie że $d[v] = d[u] + 1$.*

Uzasadnienie. Pokażemy, że warunek opisany w lemacie jest niezmiennikiem pętli **while**. Na początku warunek jest spełniony: w kolejce jest tylko jeden wierzchołek s. Załóżmy teraz, że właśnie rozpoczyna się kolejny obrót pętli **while** i niezmiennik jest spełniony. Pokażemy, że po kolejnym obrocie pętli niezmiennik pozostanie spełniony.

Jest jasne, że warunek pozostaje spełniony po usunięciu u z kolejki na początku pętli **while**. Jeśli v jest ostatnim wierzchołkiem w kolejce Q to $d[v] = d[u]$ lub $d[v] = d[u] + 1$. Zobaczmy co się dzieje, gdy wstawiany jest do kolejki wierzchołek x, nieodwiedzony jeszcze sąsiad u. Wtedy $d[x] = d[u] + 1$. A więc po wstawieniu x (i wszystkich innych nieodwiedzonych sąsiadów u) niezmiennik pozostanie spełniony. □

Wniosek 1. *Wartości w tablicy $d[v]$ dla kolejnych wierzchołków wstawianych do kolejki nie maleją.*

Twierdzenie. *Algorytm BFS dla każdego wierzchołka x poprawnie oblicza odległość z s do x i zapamiętuje ją w tablicy $d[x]$.*

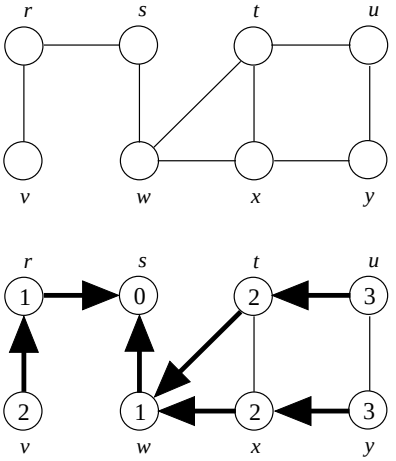
Uzasadnienie. Jest jasne, że algorytm poprawnie oblicza odległość z s do s (równą 0). Załóżmy teraz, że algorytm poprawnie oblicza odległość z s do wierzchołków odległych o k od s. Pokażemy, że dla wierzchołków odległych o $k+1$ obliczenia także są poprawne*.

Niech v będzie dowolnym wierzchołkiem odległym od s o $k+1$. Rozważmy, moment, w którym v jest wstawiany do kolejki. Niech x będzie dowolnym wierzchołkiem odległym od s o k. Założyliśmy, że dla x odległość jest poprawnie obliczana, a więc po wstawieniu x do kolejki $d[x] = k$. Z drugiej strony łatwo zobaczyć, że $d[v] \geq k+1$ (inaczej istnieje ścieżka od s do v krótsza niż $k+1$). Ponieważ wartości tablicy d nie maleją (wniosek 1), x musiał być wstawiony do kolejki przed v. A więc wszystkie wierzchołki odległe o k od s zostały wstawione przed v.

Niech y będzie wierzchołkiem bezpośrednio przed v na najkrótszej ścieżce od s do v. Jeśli takich najkrótszych ścieżek jest wiele wybieramy tą, dla której y został najwcześniej stawiony do kolejki Q. Oczywiście $d(s,y) = k = d[y]$†. Gdy wierzchołek y był usuwany z kolejki, v nie mógł być odwiedzony (mógł być odwiedzony tylko z wierzchołka odległego o k od s, a y jest pierwszym takim wierzchołkiem w kolejce). W takim razie po usunięciu y wierzchołek v będzie wstawiony do kolejki i $d[v] = d[y]+1 = k+1 = d(s,v)$, czyli odległość do v też jest poprawnie obliczana. □

Wniosek 2. *Niech $v_0 v_1 \ldots v_k$ będzie najkrótszą ścieżką od s do v ($v_0 = s$, $v_k = v$). Wtedy $v_{k-1} = Skad[v]$, $v_{k-2} = Skad[v_{k-1}]$, ... itd, tzn. $v_j = Skad[v_{j+1}]$, czyli można ją odtworzyć w czasie $\Theta(k)$.*

Drzewo najkrótszych ścieżek. Na rysunku 1 przedstawiono graf po przetworzeniu przez algorytm BFS. Strzałki odpowiadają zawartości tablicy $Skad$. Widzimy, że tworzą one drzewo o korzeniu w s. Nazywamy je *drzewem najkrótszych ścieżek*. Z takiego drzewa morzemy łatwo odczytać najkrótsze ścieżki od s do innych wierzchołków.



Rysunek 1: Drzewo najkrótszych ścieżek

Analiza złożoności czasowej. Jak zwykle oznaczamy przez n liczbę wierzchołków, przez m liczbę krawędzi. Operacje na kolejce zajmują czas $O(n)$, bo każdy wierzchołek może być tylko

*Innymi słowy dowód jest przez indukcję po odległości od s do x

†$d(s,y)$ oznacza odległość od s do y.

raz wstawiony do kolejki i raz z niej usunięty. Zauważmy też, że listę sąsiedztwa dowolnego wierzchołka x przeglądamy co najwyżej raz, po usunięciu x z kolejki. Liczba wszystkich elementów na listach sąsiedztwa wynosi $2m$ a więc przeglądanie list sąsiedztwa zajmuje natomiast czas $O(m)$. Złożoność czasowa algorytmu przeszukiwania wszerz wynosi więc $O(n+m)$.

Problem najkrótszej ścieżki

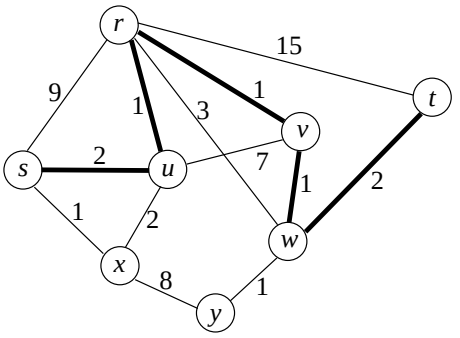
Dane:

- $G = (V, E)$ graf nieskierowany,
- funkcja $w : E \to [0, \infty)$, tzn. dla każdej krawędzi e mamy jej „długość", czyli nieujemną liczbę rzeczywistą $w(e)$,
- wyróżniony wierzchołek s.

W takim grafie za długość ścieżki przyjmiemy sumę długości jej krawędzi. Zmienia wtedy swój sens także pojęcie odległości dwóch wierzchołków (długości najkrótszej ścieżki łączącej wierzchołki). Taką „nową" odległość będziemy oznaczać przez $d_w(u, v)$.

Problem: Dla każdego $v \in V$ obliczyć $d_w(s, v)$. Obliczyć drzewo najkrótszych ścieżek (w sensie odległości d_w).

Przykład: najkrótsza ścieżka. Patrz rys. 2. Najkrótsza ścieżka od s do t (złożona z pogrubionych krawędzi) ma długość 7.



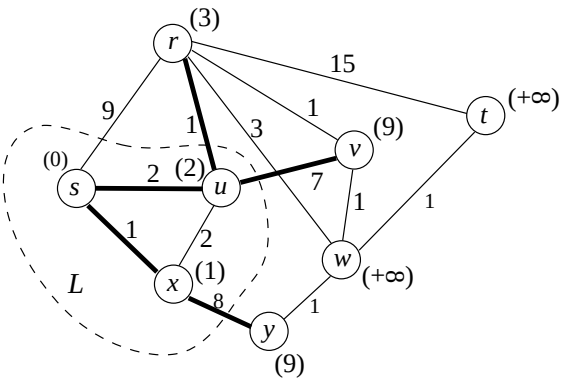
Rysunek 2: Graf z wagami na krawędziach i najkrótsza ścieżka od s do t.

Algorytm Dijkstry, zasada działania (rys. 3). W każdym momencie zbiór V podzielony jest na dwa zbiory rozłączne: $V = L \cup P$.

- L – wierzchołki, dla których już znamy odległość od s
- P – pozostałe wierzchołki.

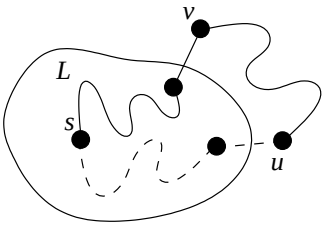
Dodatkowo wierzchołki L leżą bliżej s niż wierzchołki z P, tzn. jeśli $l \in L$ i $p \in P$ to $d_w(s, l) \leq d_w(s, p)$. Co więcej, dla każdego wierzchołka z L najkrótsza ścieżka z s przechodzi *tylko* po wierzchołkach z L. Na początku $L = \{s\}$, $P = V - \{s\}$. Algorytm w każdym kroku przenosi jeden wierzchołek z P do L.

W tablicy $LOdl[v]$ będziemy dla każdego wierzchołka v przechowywać długość najkrótszej ścieżki z s do v przechodzącej *tylko* po wierzchołkach L, lub $+\infty$ jeśli takiej drogi nie ma. Zauważmy, że wartości tablica $LOdl$ będzie zawierać liczby rzeczywiste dla wierzchołków z L (będzie to prawdziwa odległość od s) oraz dla tych wierzchołków z P, które mają choć jednego sąsiada w L. Dla pozostałych wierzchołków $LOdl$ będzie zawierała $+\infty$.



Rysunek 3: Działanie algorytmu Dijkstry. Pogrubiono najlepsze znalezione dotąd ścieżki z s. W nawiasach wartości $LOdl$.

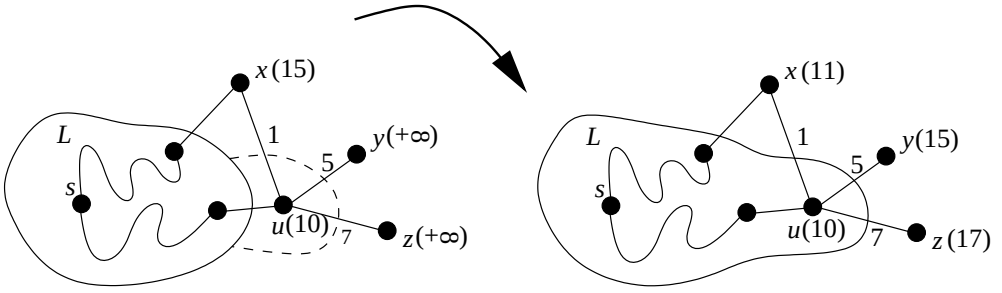
Lemat 2. *Niech zbiory L, P i tablica $LOdl$ mają własności takie, jak opisano powyżej. Niech $u \in P$ będzie takim wierzchołkiem, że $LOdl[u]$ jest najmniejsze, tzn. dla każdego $v \in P$ $LOdl[v] \geq LOdl[u]$. Wtedy $LOdl[u] = d_w(s,u)$ i u można przenieść do L.*



Rysunek 4: Lemat 2. Ścieżka zaznaczona linią przerywaną ma długość $LOdl[u]$.

Uzasadnienie. Zobaczmy, co by było, gdyby $LOdl[u] > d_w(s,u)$, tzn. istniałaby ścieżka od s do u, przechodząca przez wierzchołek spoza L, krótsza niż $LOdl[u]$. Niech v będzie pierwszym wierzchołkiem spoza L na tej ścieżce (patrz rys. 4). Ten fragment ścieżki, od s do v, jest równocześnie najkrótszą ścieżką od s do v i przechodzi tylko przez wierzchołki z L. W takim razie ma długość $LOdl[v]$. Ale $LOdl[v] \geq LOdl[u]$ czyli nasza ścieżka nie może być krótsza niż $LOdl[u]$. □

Uwaga. Po przeniesieniu u z P do L musimy być może odświeżyć niektóre wartości w tablicy $LOdl$. Zmienić mogły się tylko wartości $LOdl$ dla sąsiadów u. Dla każdego takiego sąsiada x sprawdzamy, czy nowa ścieżka od s do u przedłużona o krawędź u–x jest krótsza niż najkrótsza dotąd znaleziona ścieżka do u. Jeśli tak jest zapamiętujemy długość nowej ścieżki w $LOdl[x]$ (patrz rys 5).



Rysunek 5: Odświeżanie wartości tablicy $LOdl$ po przeniesieniu wierzchołka u. W nawiasach podano wartości w tablicy $LOdl$.

Algorithm 4 Algorytm Dijkstry.

```
1: L ← {s}; P ← V − {s}
2: LOdl[s] ← 0; Skad[s] ← nil
3: for all u ∈ P do
4:    if s–u ∈ E then
5:       LOdl[u] ← w(s–u)
6:       Skad[u] ← s
7:    else
8:       LOdl[u] ← +∞
9:       Skad[u] ← nil
10: for i ← 2 to n do
11:    u ← wierzchołek z P t.ż. LOdl[u] jest najmniejsze
12:    P ← P − {u}
13:    L ← L ∪ {u}
14:    for all x ∈ Sasiedzi[u] do
15:       if x ∈ P and LOdl[u] + w(x–u) < LOdl[x] then
16:          LOdl[x] ← LOdl[u] + w(x–u)
17:          Skad[x] ← u
```

Analiza złożoności. Złożoność algorytmu Dijkstry zależy w istotny sposób od użytych struktur danych. Potrzebna jest struktura przechowująca elementy zbioru P i udostępniająca następujące operacje:

- Znalezienie w P wierzchołka o najmniejszej wartości $LOdl$,

- Usunięcie takiego wierzchołka,
- Zmniejszenie wartości $LOdl$.

Taka specyfikacja odpowiada dokładnie kolejce priorytetowej dla zbioru P, w którym kluczami elementów są wartości z tablicy $LOdl$. Dwie pierwsze operacje to `min` oraz `UsuńMin`. Większość kolejek priorytetowych udostępnia także trzecią operację, operację zmniejszenia klucza. Na przykład żeby zmniejszyć klucz elementu $A[k]$ w kopcu binarnym zapamiętanym w tablicy A, wystarczy wpisać w $A[k]$ nową wartość klucza i wywołać procedurę `DoGóry (A, k)`.

Wniosek 3. *Algorytm Dijkstry, w którym do implementacji kolejki priorytetowej użyto kopców binarnych ma złożoność* $O((m+n)\log n)$.

Dowód. Za operacje dominujące możemy przyjąć opracje na kolejce priorytetowej. Wszystkie operacje na kopcu binarnym zawierającym nie więcej niż n elementów wykonują się w czasie $O(\log n)$. Każdy wierzchołek jest raz wstawiany i raz usuwany z kopca, a więc wszystkie wstawienia i usunięcia zajmują czas $O(n \log n)$. Wartość klucza jest zmniejszana tylko raz dla każdej krawędzi, a więc w sumie te operacje zajmują czas $O(m \log n)$. □

Uwaga. Algorytm Dijkstry pozostaje nie zmieniony, jeśli dany graf jest skierowany (tzn. wierzchołki są połączne strzałkami z wagami).

Problemy najkrótszych ścieżek stanowią jeden z najintensywniej badanych obszarów algorytmicznej teorii grafów. Przez pewien czas algorytm Dijkstry był główną motywacją badań nad kolejkami priorytetowymi. Użycie zaawansowanych kolejek priorytetowych (kopców Fibonacciego) zmniejsza złożoność algorytmu Dijkstry do $O(n \log n + m)$.