Wykład 14
Algorytmy grafowe – metoda zachłanna

1. Właściwości algorytmu zachłannego:

1. W przeciwieństwie do metody programowania dynamicznego nie występuje etap dzielenia na mniejsze realizacje z wykorzystaniem właściwości rekurencyjnej.
2. Algorytm zachłanny znajduje rozwiązanie w wyniku podjęcia szeregu decyzji, z których każda wydaje się w danym momencie najlepsza. Oznacza to, że każdy wybór jest optymalny lokalnie.
3. Takie podejście nie zapewnia, że rozwiązanie globalne będzie optymalne.
4. Czasem takie rozwiązanie optymalne globalnie istnieje, jednak należy to zawsze dodatkowo wykazać.

2. Ogólna charakterystyka algorytmu zachłannego

1. Utwórz zbiór pusty.
2. Powtarzaj, aż otrzymasz rozwiązanie realizacji problemu:
   2.1. Wykonaj **procedurę wyboru**, która służy do wybrania kolejnego elementu dodawanego do zbioru. Wybór jest wykonywany zgodnie z kryterium zachłannym, który spełnia warunek optymalny lokalnie.
   2.2. Wykonaj **procedurę sprawdzenia wykonalności**, która służy do określenia, czy zbiór jest akceptowalny dzięki sprawdzeniu, czy istnieje możliwość uzupełnienia go w taki sposób, który zapewni osiągnięcie realizacji problemu.
   2.3. Wykonaj **procedurę sprawdzania rozwiązania**, służąca do określenia, czy ów zbiór stanowi rozwiązanie realizacji problemu.

Definicje:

1. **Droga** w grafie nieskierowanym: sekwencja wierzchołków, w którym każdy wierzchołek u i jego następnik v łączy krawędź.
2. Graf nazywamy **spójnym**, gdy między każdą parą wierzchołków istnieje droga.
3. **Cyklem prostym** w grafie nieskierowanym nazywamy drogę prowadzącą z wierzchołka do niego samego, która zawiera co najmniej trzy wierzchołki i wierzchołki pośrednie są różne.
4. Graf nieskierowany, który nie zawiera żadnych cykli prostych jest określano jako **graf acykliczny.**
5. **Drzewo swobodne** jest acyklicznym spójnym grafem nieskierowanym. Żaden wierzchołek nie jest wyróżniony jako korzeń.
6. **Drzewo korzeniowe** posiada wyróżniony wierzchołek zwany korzeniem i jest kojarzone z potocznym rozumieniem drzewa.
7. **Drzewo rozpinające** grafu G jest to spójny podgraf, który zawiera wszystkie wierzchołki należące do G i jest drzewem.
8. **Minimalne drzewo rozpinające** grafu posiada najmniejszą wagę. Jeśli krawędzie grafu posiadają związane ze sobą wartości, są one nazywane **wagami**, a graf określa się jako **graf ważony**.

3. Algorytmy grafowe
3.1. Algorytm Prima: Minimalne drzewo rozpinające

Problem: określ minimalne drzewo rozpinające
Zastosowanie: budowa sieci dróg
Dane wejściowe: spójny ważony graf nieskierowany zawierający n wierzchołków, gdzie *n>=2*. Graf jest reprezentowany przez macierz sąsiedztwa *Graf*, gdzie element *Graf[i][j]* reprezentuje wagę krawędzi łączącej wierzchołki *i* oraz *j*.

G= {V, E}, gdzie V jest zbiorem wierzchołków (węzłów) i E zbiorem krawędzi

odwiedzony – numer wierzchołka – przodka *u* danego wierzchołka *v*

klucz – minimalna waga spośród wag krawędzi łączących dany wierzchołek *v* z wierzchołkami drzewa

r – wierzchołek, od którego budowane jest minimalne drzewo rozpinające

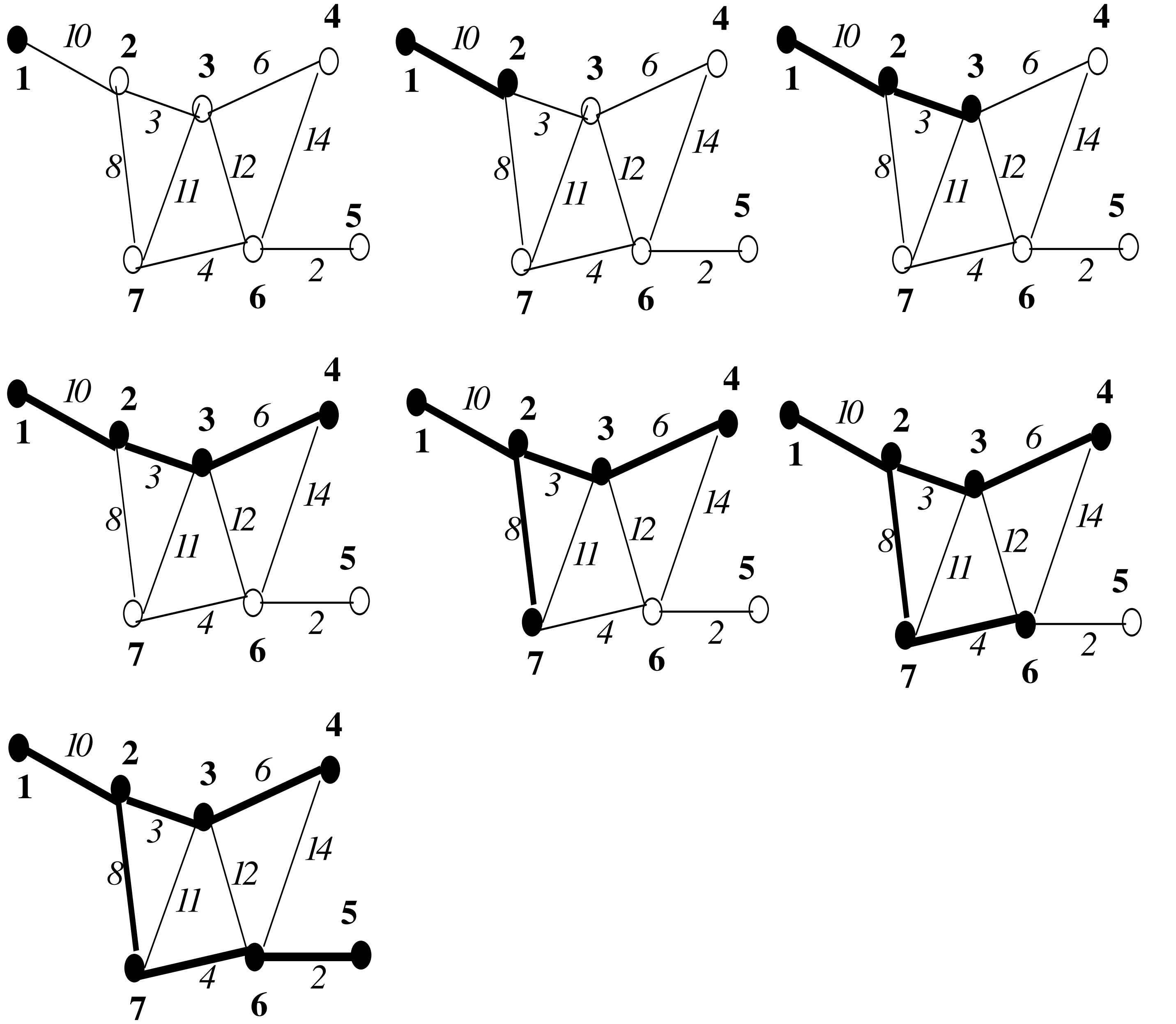
Kolejka_P – kolejka priorytetowa zawierająca wszystkie wierzchołki spoza rozrastającego się drzewa rozpinającego zwracająca element o najmniejszej wadze

Tablica *wierzcholki* będzie zawierać elementy zawierające zbiór krawędzi drzewa rozpinającego: (v, wierzcholki[v].odwiedzony)

```
wierzcholek
  { odwiedzony;
    klucz;}

wierzcholek wierzcholki[V[G]]
```

Graf(u, v) – macierz sąsiedztwa; wagi gałęzi (u,v) testowane w celu znalezienia ***najlżejszej gałązi*** należącej do przekroju rozrastającego się minimalnego drzewa rozpinającego. **Przekrojem drzewa** są wszystkie gałęzie łączące wierzchołki czarne z białymi.

```
PRIM (Graf(G), r)
1. Załaduj wszystkie wierzchołki grafu G do Kolejka_P
2. dla każdego wierzchołka u należącego do Kolejka_P
        wierzcholki[u].klucz = NS        //NS – nieskończoność
3. wierzcholki[r].klucz = 0
4. wierzcholki[r].odwiedzony = 0
5.  Dopóki Kolejka_P nie jest pusta
   5.1. u = Usun(Kolejka_P)
   5.2. Dla każdego wierzchołka v należącego do macierzy sąsiedztwa
        wierzchołka u (Graf[u])
          jeśli v należy do Kolejka_P i  Graf[u][v] < wierzcholki [v].klucz
          to //znaleziono najlżejszą gałąź z przekroju grafu
           5.2.1. { wierzcholki[v].odwiedzony = u
                    wierzcholki[v].klucz = Graf [u][v] }
```


1
2
3
4
5
6
7
10
3
6
8
11
12
14
4
2

Kolejka priorytetowa Kolejka_P – wszystkie wierzcholki

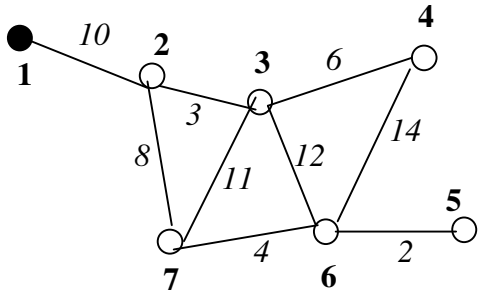
1	2	3	4	5	6	7	Pozycja w kolejce
1	2	3	4	5	6	7	Numer wierzchołka

Tabela wierzcholki

1	2	3	4	5	6	7	Numer wierzchołka
0							odwiedzony
0	NS	NS	NS	NS	NS	NS	klucz

Kolejka priorytetowa Kolejka_P – wszystkie wierzcholki

1	2	3	4	5	6	7	Pozycja w kolejce
1	2	3	4	5	6	7	Numer wierzchołka
0							odwiedzony
0	NS	NS	NS	NS	NS	NS	klucz



Kolejka priorytetowa Kolejka_P – sześć wierzchołków

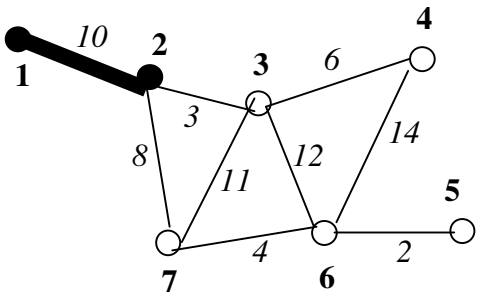
1	2	3	4	5	6	***7***	Pozycja w kolejce
2	3	4	5	6	7	**1**	Numer wierzchołka

Tabela wierzcholki

1	2	3	4	5	6	7	Numer wierzchołka
0	**1**						odwiedzony
0	**10**	NS	NS	NS	NS	NS	klucz

Kolejka priorytetowa Kolejka_P – sześć wierzchołków(wersja bliższa implementacji)

1	2	3	4	5	6	***7***	Pozycja w kolejce
2	3	4	5	6	7	**1**	Numer wierzchołka
1						**0**	odwiedzony
10	NS	NS	NS	NS	NS	**0**	klucz



Kolejka priorytetowa Kolejka_P – pięć wierzchołków

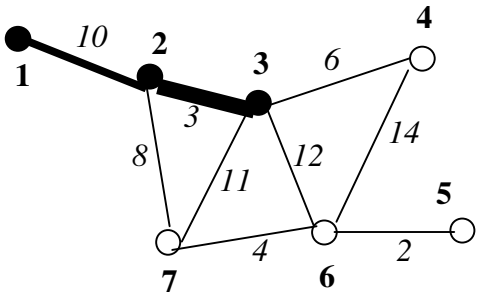
1	2	3	4	5	*6*	***7***	Pozycja w kolejce
3	7	4	5	6	1	**2**	Numer wierzchołka

Tabela wierzcholki

1	2	3	4	5	6	7	Numer wierzchołka
0	1	**2**				**2**	odwiedzony
0	10	**3**	NS	NS	NS	**8**	klucz

Kolejka priorytetowa Kolejka_P – pięć wierzchołków (wersja bliższa implementacji)

1	2	3	4	5	*6*	***7***	Pozycja w kolejce
3	7	4	5	6	1	**2**	Numer wierzchołka
2	2				0	**1**	odwiedzony
3	8	NS	NS	NS	0	**10**	klucz



Kolejka priorytetowa Kolejka_P – cztery wierzchołki

1	2	3	4	*5*	*6*	***7***
4	7	6	5	1	2	**3**

Pozycja w kolejce

Numer wierzchołka

Tabela wierzcholki

1	2	3	4	5	6	7	Numer wierzchołka
0	1	2	**3**		**3**	***2***	odwiedzony
0	10	3	**6**	NS	**12**	***8***	klucz

Kolejka priorytetowa Kolejka_P – cztery wierzchołki (wersja bliższa implementacji)

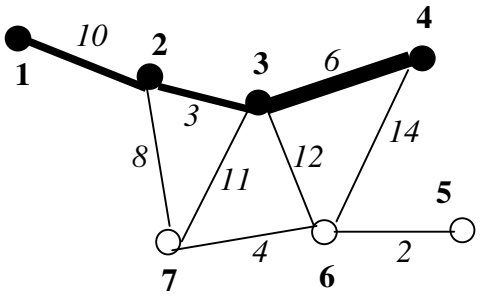
1	2	3	4	*5*	*6*	***7***
4	7	6	5	1	2	**3**
3	2	3		0	1	**2**
6	8	12	NS	0	10	**3**

Pozycja w kolejce

Numer wierzchołka

odwiedzony

klucz



Kolejka priorytetowa Kolejka_P – trzy wierzchołki

1	2	3	*4*	*5*	*6*	***7***
7	6	5	1	2	3	**4**

Pozycja w kolejce

Numer wierzchołka

Tabela wierzcholki

1	2	3	4	5	6	7	Numer wierzchołka
0	1	2	***3***		3	2	odwiedzony
0	10	3	***6***	NS	12	8	klucz

Kolejka priorytetowa Kolejka_P – trzy wierzchołki (wersja bliższa implementacji)

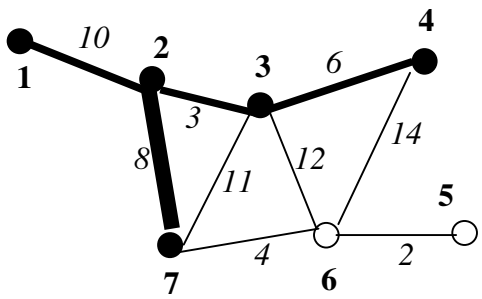
1	2	3	*4*	*5*	*6*	***7***
7	6	5	1	2	3	**4**
2	3		0	1	2	**3**
8	12	NS	0	10	3	**6**

Pozycja w kolejce

Numer wierzchołka

odwiedzony

klucz



Kolejka priorytetowa Kolejka_P – dwa wierzchołki

1	2	*3*	*4*	*5*	*6*	***7***
6	5	1	2	3	4	**7**

Pozycja w kolejce

Numer wierzchołka

Tabela wierzcholki

1	2	3	4	5	6	7	Numer wierzchołka
0	1	2	3		**7**	2	odwiedzony
0	10	3	6	NS	**4**	8	klucz

Kolejka priorytetowa Kolejka_P – dwa wierzchołki (wersja bliższa implementacji)

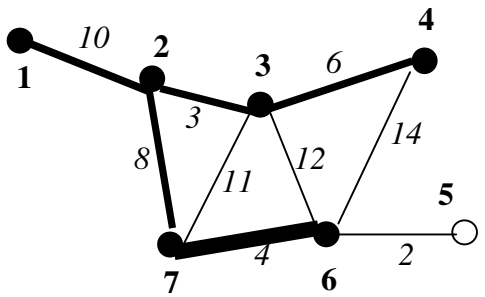
1	2	*3*	*4*	*5*	*6*	***7***
6	5	1	2	3	4	**7**
7		0	1	2	3	**2**
4	NS	0	10	3	6	**8**

Pozycja w kolejce

Numer wierzchołka

odwiedzony

klucz



Kolejka priorytetowa Kolejka_P – jeden wierzchołek

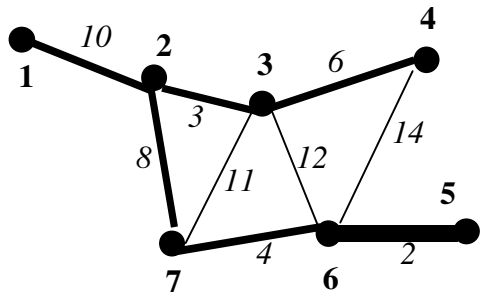
1	*2*	*3*	*4*	*5*	*6*	***7***	Pozycja w kolejce
5	1	2	3	4	7	**6**	Numer wierzchołka

Tabela wierzcholki

1	2	3	4	5	6	7	Numer wierzchołka
0	1	2	3	**6**	7	2	odwiedzony
0	10	3	6	**2**	4	8	klucz

Kolejka priorytetowa Kolejka_P – jeden wierzchołek (wersja zbliżona do implementacji)

1	*2*	*3*	*4*	*5*	*6*	***7***	Pozycja w kolejce
5	1	2	3	4	7	**6**	Numer wierzchołka
6	0	1	2	3	2	**7**	odwiedzony
2	0	10	3	6	8	**4**	klucz



Kolejka priorytetowa Kolejka_P - pusta

1	2	3	4	5	6	7	Pozycja w kolejce
1	2	3	4	7	6	**5**	Numer wierzchołka

Tabela wierzcholki

1	2	3	4	5	6	7	Numer wierzchołka
0	1	2	3	6	7	2	odwiedzony
0	10	3	6	2	4	8	klucz

Czas działania algorytmu Prima

O(VlgV + E lg V) = O(E lgV)

1. Zakłada się, że kolejka priorytetowa Kolejka_P jest zaimplementowana jako kopiec. Oznacza czas działania linii algorytmu 1-4 równy O(V).

2. Czas działania pętli głównej w linii 5 wynosi O(V). Każde wykonanie Usun(Kolejka_P) wynosi O(lgV), stąd łączny czas wynosi O(V lgV).

3. Suma długości wszystkich list sąsiedztwa jest równa 2|E|, czyli czas działania pętli w linii 5.2 wynosi O(E), a operacji w linii 5.2.1 jest równy O(lgV) – działanie na kopcu polegające na zmianie klucza i przebudowie kopca

3.2. Algorytm Dijkstry – najkrótsza ścieżka z jednym źródłem

Problem: określ najkrótsze ścieżki z jednego źródła w ważonym grafie skierowanym, gdy wagi są nieujemne.
Zastosowanie: budowa sieci dróg
Dane wejściowe: spójny ważony graf skierowany zawierający *n* wierzchołków, gdzie *n>=2*. Graf jest reprezentowany przez macierz sąsiedztwa *Graf*, gdzie element *Graf[i][j]* reprezentuje wagę krawędzi łączącej wierzchołki *i* oraz *j*.

G= {V, E}, gdzie V jest zbiorem wierzchołków (węzłów) i E zbiorem krawędzi

S – zbiór zawierający te wierzchołki, dla których wagi najkrótszych ścieżek ze źródła s zostały już obliczone

Kolejka_P – kolejka priorytetowa zawierająca wszystkie wierzchołki spoza rozrastającego się zbioru S (V - S)

odwiedzony – numer wierzchołka – przodka *u* danego wierzchołka *v*

klucz – wartość ograniczająca od góry wagę najkrótszej ścieżki ze źródła s do wierzchołka v.

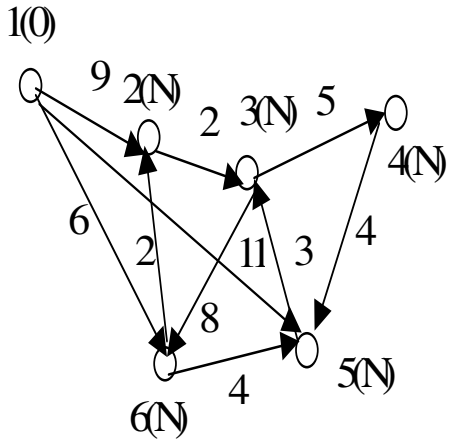
s – wierzchołek, od którego budowane są najkrótsze ścieżki

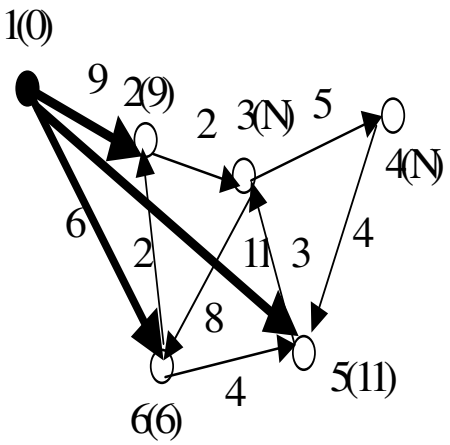
Tablica *wierzcholki* będzie zawierać elementy zawierające zbiór krawędzi należących do najkrótszych ścieżek z s: (v, wierzcholki[v].odwiedzony)

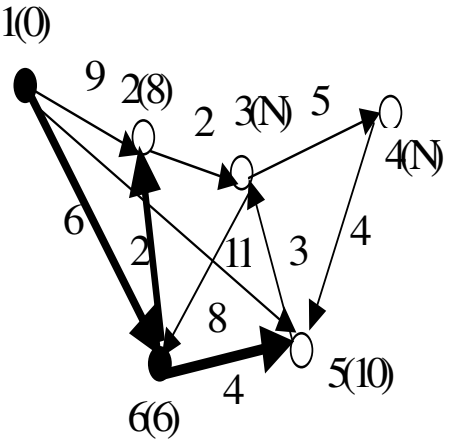
```
wierzcholek
  {  odwiedzony;
     klucz;}

wierzcholek wierzcholki[V[G]]
```

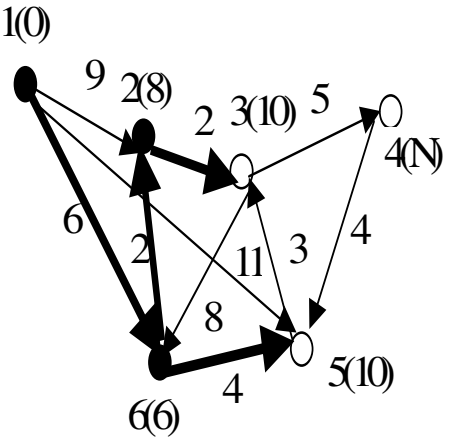
w(u, v) – waga gałęzi (u, v).

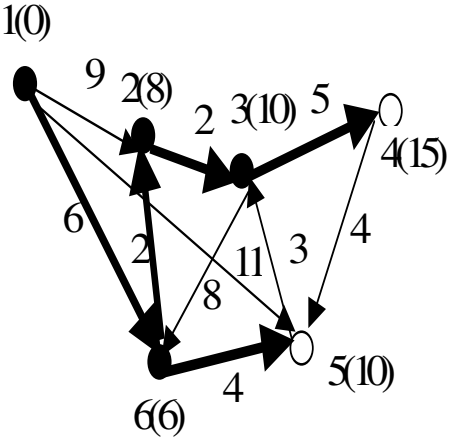
```
Dijkstra (Graf, s)
1. Dla każdego wierzchołka v należącego do zbioru wierzchołków V
       { wierzcholki[v].klucz = NS   //NS – nieskończoność
         wierzcholki[v].odwiedzony = 0 }
2. wierzcholki[v].klucz = 0
3. Zbiór S inicjuje się jako pusty
4. Załaduj wszystkie wierzcholki grafu G do kolejki priorytetowej Kolejka_P
5. Dopóki Kolejka_P  nie jest pusta
     5.1.  u = Usun(Kolejka_P)        //usuwanie wierzchołka z najmniejszym kluczem
     5.2.  Do zbioru S dodaj wierzchołek u
     5.3.  Dla każdego wierzcholka v należącego do  macierzy sąsiedztwa
           wierzcholka u (Graf[u])
              Jeżeli wierzcholki[v].klucz > wierzcholki[u].klucz + Graf[u][v],
              to
                 { wierzcholki[v].klucz = wierzcholki[u].klucz + Graf[u][v]
                   wierzcholki[v].odwiedzony = u }
```

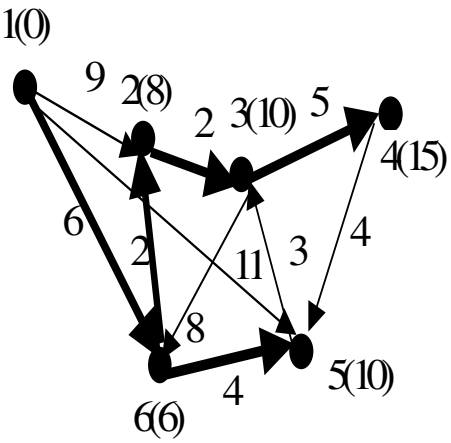












Czas trwania algorytmu:
O((V+E)lgV) lub **O(E lgV)**

1. Każdy wierzchołek jest dokładnie raz usuwany z kolejki Kolejka_P i wstawiany do zbioru S, czyli liczba iteracji jest równa |V|

2. Do zbioru S jest dodawany najlżejszy wierzchołek z Kolejka_P z szybkością kopca O(lgV) – Usun(Kolejka_P). Wykonuje się |V| takich operacji

3. Przypisanie: wierzcholki[v].klucz = wierzcholki[u].klucz + Graf[u][v] wykonywane jest z szybkością O(lgV). Jest to operacja, która przebudowuje kopiec dla węzła v. Takich operacji jest co najwyżej |E|. Łączny czas wynosi O((V+E)lgV). Jeśli wszystkie wierzchołki są osiągalne ze źródła, to czas ten jest równy O(E lgV)