VII Algorytmy i struktury danych

Podstawowe algorytmy sekwencyjne: grafowe, geometryczne, tekstowe.

1.Algorytmy grafowe:

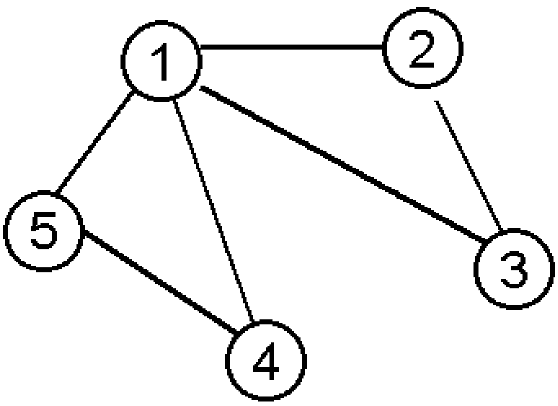
REPREZENTACJE GRAFÓW

$G=(V,E)$ $V=\{1,2,\ldots,n\}$ $E=\{e_1,\ldots,e_m\}$

- Macierz sąsiedztwa: A[1..n,1..n]

$$A[i,j]=\begin{cases} 1 \text{ (true) jeśli } \{i,j\} \in E \\ 0 \text{ (false) jeśli } \{i,j\} \notin E \end{cases}$$

Rząd złożoności pamięciowej: $O(n^2)$

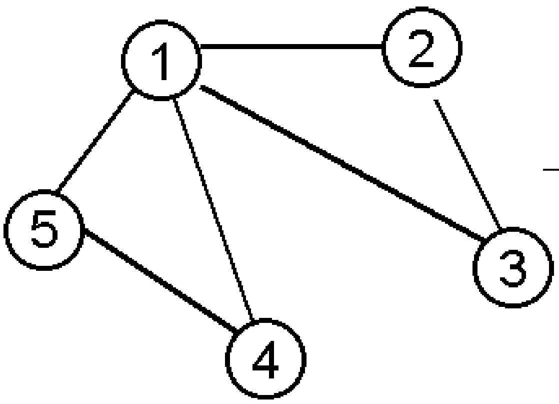


	1	2	3	4	5
1	0	1	1	1	1
2	1	0	1	0	0
3	1	1	0	0	0
4	1	0	0	0	1
5	1	0	0	1	0

- Macierz incydencji : B[1..n,1..m]

$$B[i,j]=\begin{cases} 1 \text{ (true) jeśli } i \in e_j \\ 0 \text{ (false) jeśli } i \notin e_j \end{cases}$$

Rząd złożoności pamięciowej: $O(mn)$



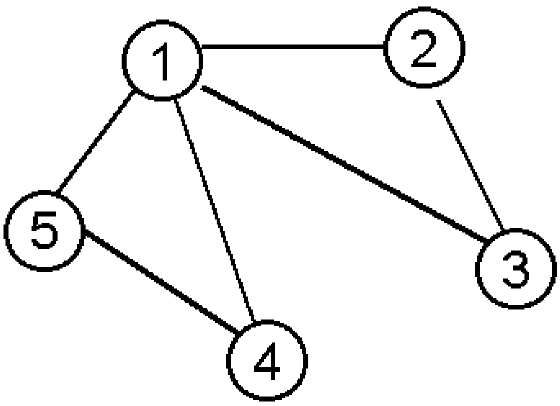
	1	2	3	4	5	6
1	1	1	0	1	1	0
2	0	1	1	0	0	0
3	0	0	1	1	0	0
4	0	0	0	0	1	1
5	1	0	0	0	0	1

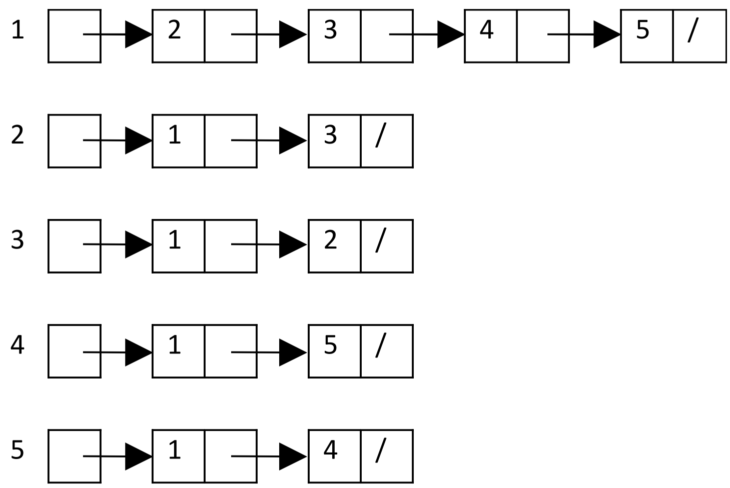
- Listy sąsiedztwa

  L [1..n], { Adj(i), i=1,…,n }

  L[i] = wskaźnik na początek listy sąsiadów wierzchołka i (Adj(i))

Rząd złożoności pamięciowej: O(m)





Przeszukiwanie grafu w głąb i wszerz

Dane: G= (V, E) w postaci list sąsiedztwa: { Adj(v): v ∈ V }

Wynik: kolor(v): v ∈ V

PRZESZUKIWANIE W GŁĄB

{"Odwiedza" każdy wierzchołek grafu ; Wierzchołki “odwiedzone” mają kolor czarny}

```
DFS-VISIT (u);
{"Odwiedza" wierzchołki składowej
spójności zawierającej wierzchołek u}
   begin
     kolor (u) := SZARY;
     for każdy v ∈ Adj(u) do
       if kolor (v) = BIAŁY
       then
           DFS-VISIT (v);
     kolor (u) := CZARNY;
   end;
```

```
DFS (G);
  begin
    for każdy u ∈ V(G) do
       kolor (u) := BIAŁY;
    for każdy u ∈V(G) do
       if kolor (u) = BIAŁY
         then DFS-VISIT (u);
  end;
```

PRZESZUKIWANIE W WSZERZ

{"Odwiedza" każdy wierzchołek grafu ; Wierzchołki “odwiedzone” mają kolor czarny}

```
DFS-VISIT (u);
{"Odwiedza" wierzchołki składowej
spójności zawierającej wierzchołek u}
   begin
     kolor (u) := SZARY;
     for każdy v ∈ Adj(u) do
       if kolor (v) = BIAŁY
       then
           begin
             π(v):=u;
             DFS-VISIT (v);
           end;
     kolor (u) := CZARNY;
   end;
```

```
DFS (G);
  begin
    for każdy u ∈ V(G) do
     begin
       kolor (u) := BIAŁY;
       π(v):=-1;
     end;
    for każdy u ∈ V(G) do
       if kolor (u) = BIAŁY
         then DFS-VISIT (u);
  end;
```

PRZESZUKIWANIE WSZERZ

BFS (G, s);

{Jeśli G jest spójny, to odwiedza każdy wierzchołek grafu;
jeśli niespójny, to odwiedza każdy wierzchołek składowej spójności zawierającej wierzchołek s .Wierzchołki “odwiedzone” mają kolor czarny}

```
begin
  for każdy u ∈ V(G) – {s} do
    kolor (u) := BIAŁY;

  kolor (s) := SZARY;
```

```
while Q <> Ø do
 begin
   u:=head(Q);
   for każdy v ∈ Adj(u) do
     begin
```

```
Q := {s};
```

```
        if kolor(v) = BIAŁY
          then begin
                 kolor(v)= SZARY;
                 ENQUEUE(Q,v);
               end;
      end;
    DEQUEUE(Q);
    kolor(u)=CZARNY;
 end;
end;
```

PRZESZUKIWANIE WSZERZ

BFS (G, s);

{Jeśli G jest spójny, to odwiedza każdy wierzchołek grafu;
jeśli niespójny, to odwiedza każdy wierzchołek składowej spójności zawierającej wierzchołek s
.Wierzchołki "odwiedzone" mają kolor czarny}

```
begin
  for każdy u ∈ V(G) – {s} do
    begin
      kolor (u) := BIAŁY;
      π(v):=-1;
    end;

  kolor (s) := SZARY;

  Q := {s};
```

```
while Q <> Ø do
 begin
   u:=head(Q);
   for każdy v ∈ Adj(u) do
    begin
      if kolor(v) = BIAŁY
        then begin
               kolor(v)= SZARY;
               π(v):=u;
               ENQUEUE(Q,v);
             end;
    end;
   DEQUEUE(Q);
   kolor(u)=CZARNY;
 end;
end;
```

Wybrane zastosowania:

- Testowanie czy dany graf jest spójny
- Wyznaczanie składowych spójności
- Znajdowanie drogi

Algorytmy grafowe (cd)

MST –PRIM (G, w, r);

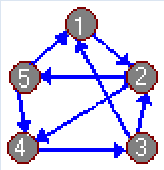
{Wyznacza zbiór krawędzi ET = { {v, π(v)}: v ∈V(G) –{r} }
minimalnego drzewa spinającego grafu G }

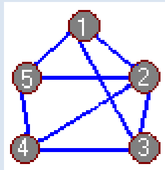
pseudokod:

```
begin
        for każdy u ∈ V(G) do
                        {
                          key(u):= ∞;
                          π(v):=NIL;
                        }
        key(r):=0;
        π(r):=-1;
        utwórz kolejkę priorytetową
        Q=( V(G),key() );

        while Q <> Ø do
         begin
                u:=EXTRACT-MIN(Q);
                for każdy v ∈ Adj(u) do
                        if v ∈ Q and w(u,v)<key(v)
                                then
                                  begin
                                        π(v):=u;
                                        key(v):=w(u,v);
                                   end;
         end;
end
```

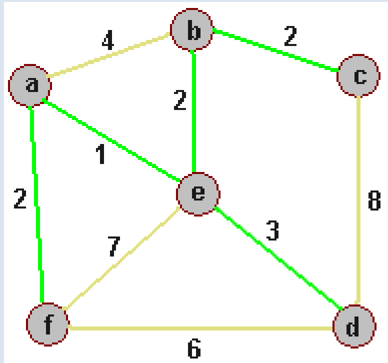
Zrozumienie algorytmu wymaga znajomości pojęć **grafu** oraz **minimalnego drzewa rozpinającego.** Algorytm ten jest oparty o **metodę zachłanną**.

Graf:
W informatyce **grafem** nazywamy strukturę G=(V, E) składającą się z węzłów (wierzchołków, oznaczanych przez V) wzajemnie połączonych za pomocą krawędzi (oznaczonych przez E).
Grafy dzielimy na grafy skierowane i nieskierowane:



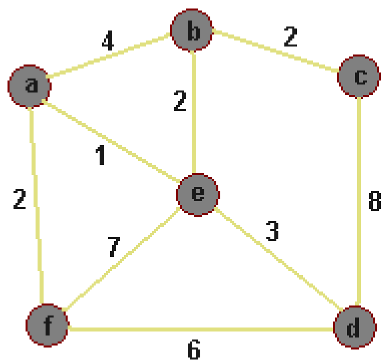


Minimalne drzewo rozpinające:
Niech będzie dany spójny graf nieskierowany o wierzchołkach V i krawędziach E. Ponad to z każdą krawędzią będzie związana tzw. **waga**, określona przez funkcję *w*, która oznaczają długość danej krawędzi. Jeżeli znajdziemy taki podzbiór T zawarty w zbiorze krawędzi E, który łączy wszystkie wierzchołki i taki, że suma wszystkich wag krawędzi wchodzących w skład T jest możliwie najmniejszy, to znajdziemy tzw. **minimalne drzewo rozpinające**.
Oto minimalne drzewo rozpinające dla przykładowego grafu:
Suma wag dla tego drzewa wynosi **10**.

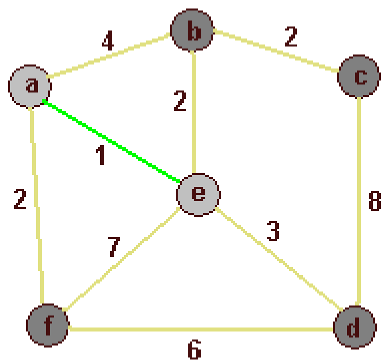


Budowę minimalnego drzewa rozpinającego zaczynamy od dowolnego wierzchołka, np. od pierwszego. Dodajemy wierzchołek do drzewa a wszystkie krawędzie incydentne umieszczamy na posortowanej wg. wag liście. Następnie zdejmujemy z listy pierwszą krawędź (o najmniejszej wadze). Sprawdzamy, czy drugi wierzchołek tej krawędzi należy do tworzonego drzewa. Jeżeli tak, to nie ma sensu dodawać takiej krawędzi (bo oba jej wierzchołki znajdują się w drzewie), porzucamy krawędź i pobieramy z listy następną. Jeżeli jednak wierzchołka nie ma w drzewie, to należy dodać krawędź do drzewa, by wierzchołek ten znalazł się w drzewie rozpinającym. Następnie dodajemy do posortowanej listy wszystkie krawędzie incydentne z dodanym wierzchołkiem i pobieramy z niej kolejny element.
Jednym zdaniem: zawsze dodajemy do drzewa krawędź o najmniejszej wadze, osiągalną (w przeciwieństwie do Kruskala) z jakiegoś wierzchołka tego drzewa.
Można zauważyć, że kluczową, z punktu widzenia wydajności algorytmu, czynnością jest zaimplementowanie listy posortowanej, gdyż po każdym dodaniu krawędzi, lista musi być nadal posortowana.

Przykład:
Przyjmijmy następującą notację: [u_i, u_j, w] oznacza krawędź łączącą wierzchołki (u_i, u_j) o wadze *w*.
Dany jest spójny graf nieskierowany:
Przebieg algorytmu:



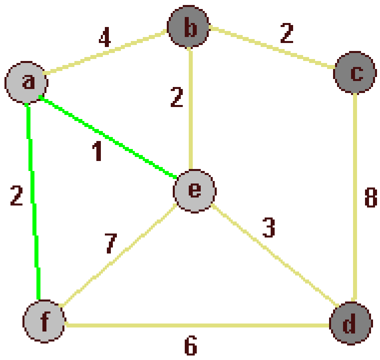
Wybieramy wierzchołek *a*.

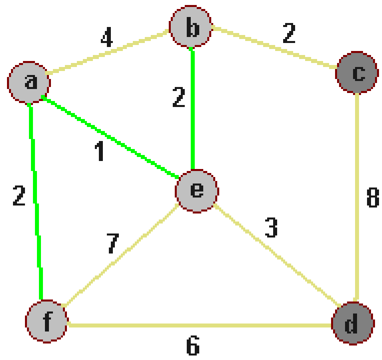


Dodajemy nowe krawędzie:

Tworzymy posortowaną listę
L={[a,e,1],[a,f,2],[a,b,4]}
Wybieramy krawędź (a,e).

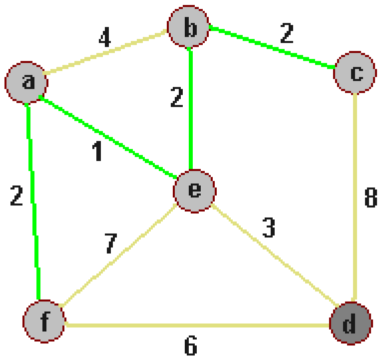
L={[a,f,2],[e,b,2],[e,d,3],[a,b,4],[e,f,7]}
Wybieramy krawędź (a,f).

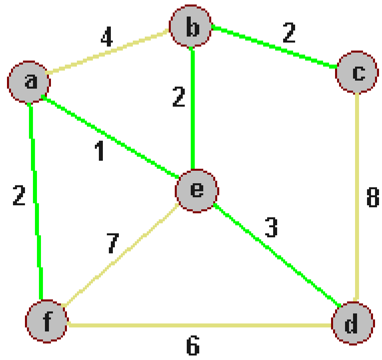




Krawędź [f,e,7] jest już na liście:
L={[e,b,2],[e,d,3],[a,b,4],[f,d,6],[e,f,7]}
Wybieramy krawędź (e,b).

Dodajemy krawędź (b,c):
L={[b,c,2],[e,d,3],[a,b,4],[f,d,6],[e,f,7]}
Wybieramy krawędź (b,c).





Dodajemy krawędź (c,d):
L={[e,d,3],[a,b,4],[f,d,6],[e,f,7],[c,d,8]}
Wybieramy krawędź (e,d).

Drzewo utworzone.

MST-KSRUSKAL(G,w);

{ Wyznacza zbiór ET krawędzi minimalnego drzewa spinającego grafu G}

pseudokod:

begin

ET:= Ø;

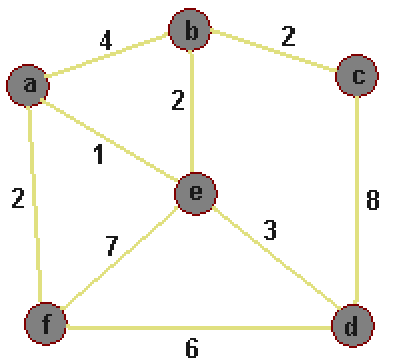
for każdy v ∈ V(G) **do**

```
            MAKE-SET(v);
        Posortuj krawędzie z E(G) niemalejąco względem wag

        for każda krawędź {u,v} ∈ E(G) (w kolejności niemalejących wag) do
            if FIND-SET(u) <> FIND-SET(v)
            then
                begin
                  ET:=ET U{{u,v}}
                  UNION(u,v)
                end;

  return ET;
end
```

Zrozumienie algorytmu wymaga znajomości pojęć **grafu** oraz **minimalnego drzewa rozpinającego**. Jest to algorytm oparty o metodę zachłanną. Algorytm polega na łączeniu wielu poddrzew w jedno za pomocą krawędzi o najmniejszej wadze. W rezultacie powstałe drzewo będzie minimalne. Na początek należy posortować wszystkie krawędzie w porządku niemalejącym. Po tej czynności można przystąpić do tworzenia drzewa. Proces ten nazywa się rozrastaniem lasu drzew. Wybieramy krawędzie o najmniejszej wadze i jeśli wybrana krawędź należy do dwóch różnych drzew należy je scalić (dodać do lasu). Krawędzie wybieramy tak długo, aż wszystkie wierzchołki nie będą należały do jednego drzewa.

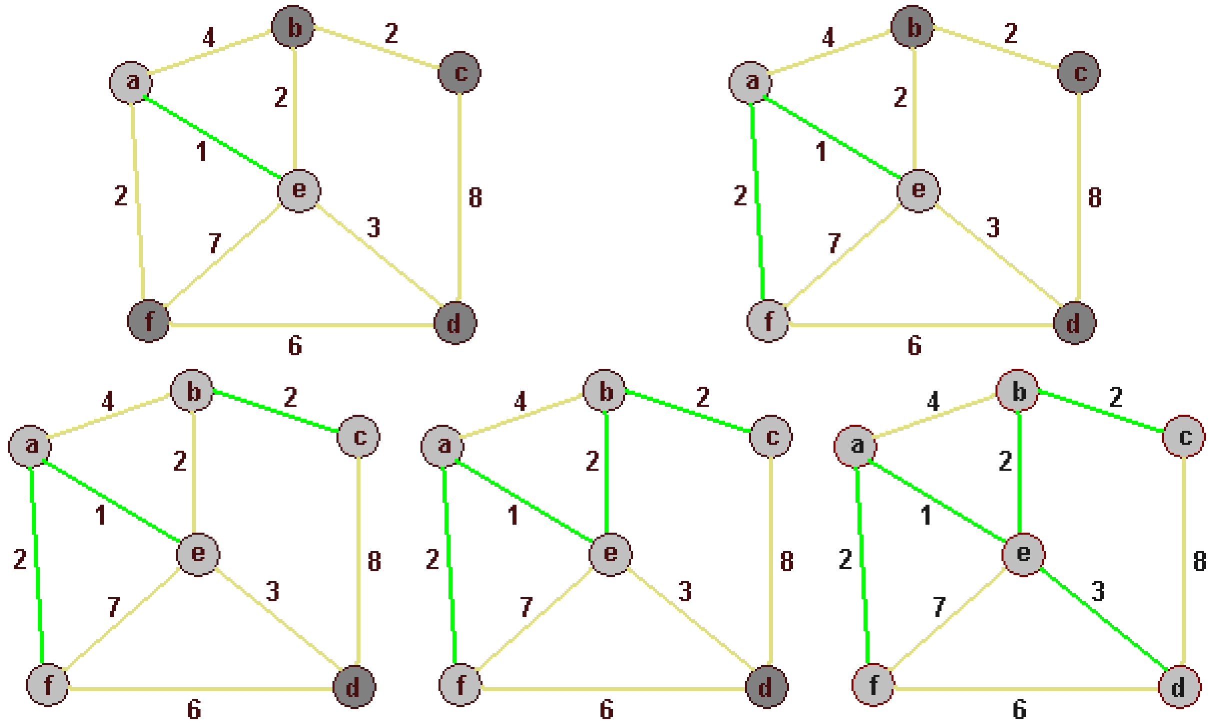
Oto przykład:

Dany jest spójny graf nieskierowany:



Po posortowaniu krawędzi wg. wag otrzymamy:

1. Krawędź ae=1
2. Krawędź af=2
3. Krawędź bc=2
4. Krawędź be=2
5. Krawędź de=3
6. Krawędź ab=4
7. Krawędź fd=6
8. Krawędź ef=7
9. Krawędź cd=8



Wszystkie wierzchołki należą do jednego drzewa- minimalnego drzewa rozpinającego.
Suma wag krawędzi wchodzących w skład drzewa wynosi 10.

CONNECTED_COMPONENT(G);

{ Wyznacza rodzinę zbiorów rozłącznych; każdy zbiór zawiera wierzchołki jednej składowej spójności grafu G }

```
begin
        for każdy wierzchołek v ∈ V(G) do MAKE-SET(v);
        for każda krawędź {u,v} ∈ E(G)
         do
                if FIND-SET(u)<>FIND-SET(v)
                then UNION(u,v);
end
```

SAME-COMPONENT(S, u, v);

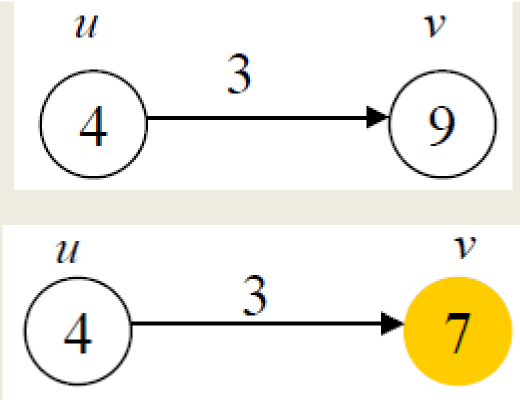
{ Testuje czy wierzchołki u i v należą do tej samej składowej spójności grafu }

```
begin
        if FIND-SET(u)= FIND-SET(v)
        then return true;
        else return false;
end
```

RELAX (u, v, w(u,v));

{ Może zmniejszyć wartość d(v) i zmienić π(v) }

```
begin
   if d(v)>d(u)+ w(u,v)
      then
               begin
                       d(v)=d(u)+ w(u,v)
                       {π(v) := u};
               end;
end
```



DIJKSTRA (G, w, s);

{Wyznacza odległość każdego wierzchołka grafu G od wierzchołka s ;
$w : E(G) \rightarrow R \geq 0$}

pseudokod:

```
a)
Dijkstra(G,w,s)
   dla każdego wierzchołka v w V[G] wykonaj
      d[v] := nieskończoność
      poprzednik[v] := niezdefiniowane
   d[s] := 0
   Q := V
   dopóki Q niepuste wykonaj
      u := Zdejmij_Min(Q)
      dla każdego wierzchołka v – sąsiada u wykonaj
         jeżeli d[v] > d[u] + w(u, v) to
            d[v] := d[u] + w(u, v)
            poprzednik[v] := u
b)
begin
        for każdy v∈V(G) do
                begin
                  d(v):=∞;
                  { π(v):=NIL };
                end
        d(s):=0;
        S:=Ø;
        Q:=( V(G), d); kolejka priorytetowa
```

```
        while Q <> Ø do
                begin
                        u:=EXTRACT-MIN(Q);
                        S:=S ∪ {u};

                        for każdy v∈Adj(u) do
                                RELAX(u,v,w);
                end
end
```

- Stwórz tablicę *d* odległości od źródła dla wszystkich wierzchołków grafu. Na początku *d*[*s*] = 0, zaś *d*[*v*] = *nieskończoność* dla wszystkich pozostałych wierzchołków.
- Utwórz kolejkę priorytetową *Q* wszystkich wierzchołków grafu. Priorytetem kolejki jest aktualnie wyliczona odległość od wierzchołka źródłowego *s*.
- Dopóki kolejka nie jest pusta:
  - Usuń z kolejki wierzchołek *u* o najniższym priorytecie (wierzchołek najbliższy źródła, który nie został jeszcze rozważony)
  - Dla każdego sąsiada *v* wierzchołka *u* dokonaj *relaksacji* poprzez *u*: jeśli *d*[*u*] + *w*(*u*,*v*) < *d*[*v*] (poprzez *u* da się dojść do *v* szybciej niż dotychczasową ścieżką), to *d*[*v*]: = *d*[*u*] + *w*(*u*,*v*).

Na końcu tablica *d* zawiera najkrótsze odległości do wszystkich wierzchołków.

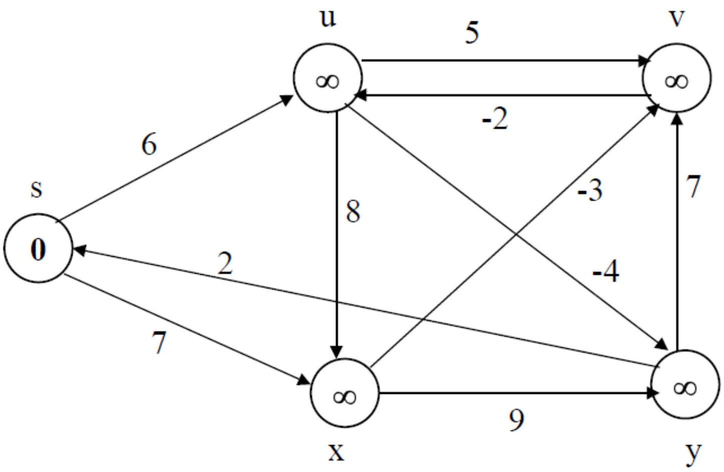
BELLMAN –FORD (G, w, s);

{ Testuje czy graf ma cykle ujemnej długości osiągalne z s; jeśli nie ma, to znajduje odległość każdego wierzchołka grafu G od wierzchołka s ; **w : E(G) → R**}

```
begin
        for każdy v∈V(G) do
                begin
                        d(v):=∞;
                        { π(v):=NIL };
                end;
        d(s):=0;

        for i:=1 to |V(G)|-1 do
                for każda krawędź {u,v} ∈ E(G) do
                        RELAX(u,v,w);
        for każda krawędź {u,v}∈E(G) do
```

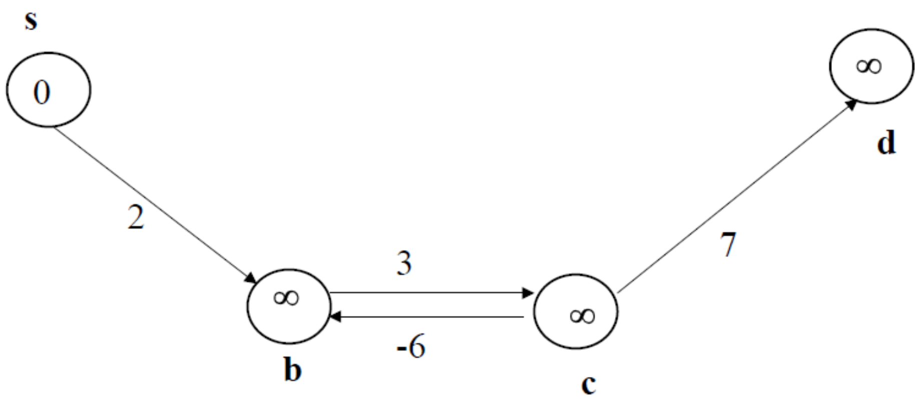
```
                if d(v)>d(u)+w(u,v)
                        then return FALSE;
  return TRUE;
end;
```

a)



Kolejność krawędzi:
(u,v), (u,x), (u,y), (v,u), (x,v), (x,y), (y,v), (y,s), (s,u), (s,x)
Algorytm zwraca TRUE

b)



Kolejność krawędzi:
(s,b), (b,c), (c,b), (c,d)
Algorytm zwraca FALSE

FLOYD-WARSHALL (W);

{ Wyznacza macierz **D(n)**odległości miedzy każdą parą wierzchołków grafu reprezentowanego przez macierz wag W }

```
begin
        D(0):=W;
        for k:=1 to n do
                for i:=1 to n do
                        for j:=1 to n do
                                Dij(k):=min{ Dij(k-1), Dik(k-1)+Dkj(k-1) };
        return D(n)
end;
```

2.Algorytmy geometryczne

ZNAJDOWNIE OTOCZKI WYPUKŁEJ ZBIORU PUNKTÓW NA PŁASZCZYŹNIE

DANE : Q = {p1,...,pn}, n≥3 , $pi \in R^2$

- zbiór punktów na płaszczyźnie

WYNIK : Otoczka wypukła zbioru Q.

DEF.

Otoczką wypukłą zbioru punktów Q na płaszczyźnie nazywamy najmniejszy zbiór wypukły na płaszczyźnie zawierający Q.

FAKT.

Jeśli |Q| < $\aleph_0$, to otoczką wypukłą zbioru Q jest **najmniejszy wielokąt wypukły P**, którego wierzchołki należą do Q oraz każdy punkt ze zbioru Q leży albo na brzegu, albo w jego wnętrzu.

GRAHAM-SCAN (Q)

{Q = {p1,...,pn}, n≥3 , jest danym zbiorem punktów na płaszczyźnie.}

Algorytm wyznacza stos S zawierający wszystkie wierzchołki otoczki wypukłej CH(Q) uporządkowane przeciwnie do ruchu wskazówek zegara .

Procedura korzysta z funkcji TOP(S) , której wartością jest element na szczycie stosu oraz NEXT-TO-TOP(S) , której wartością jest element poprzedzający wierzchołek stosu ; obie procedury nie modyfikują S }

begin

- Wyznacz punkt $p_0=(x_0,y_0)$
  gdzie:
  $y_0 = \min \{ y_i : p_i = (x_i, y_i) \in Q \}$
  $x_0 = \min \{ x_i : p_i = (x_i, y_0) \in Q \}$;
- Posortuj punkty ze zbioru $Q - \{ p_0 \}$ ze względu na współrzędną kątową w biegunowym układzie współrzędnych o środku w p_0 przeciwnie do ruchu wskazówek zegara;
  Jeśli więcej niż jeden punkt ma taką samą współrzędną kątową, usuń wszystkie z wyjątkiem punktu położonego najdalej od p_0
- Oznacz otrzymany zbiór $Q' = \{ p_0, p_1, \ldots, p_m \}$;
- Utwórz stos pusty S;
- PUSH(p_0,S);
- PUSH(p_1,S);
- PUSH(p_2,S); { punkty p_0, p_1, p_2 są na stosie S }
- for i:=3 to m do

```
begin
        while(przejście od NEXT-TO-TOP(S) przez TOP(S) do pi
                nie oznacza skrętu w lewo w TOP(S)) do
          POP(S);
        PUSH(pi, S);
end;
return S;
```

end;

T(n)=O(nlgn)

ALGORYTM TESTUJĄCY CZY W ZBIORZE ODCINKÓW NA PŁASZCZYŹNIE ISTNIEJE PARA ODCINKÓW PRZECINAJĄCYCH SIĘ

ANY-SEGMENTS_INTERSECT (S);

{S jest skończonym zbiorem odcinków na płaszczyźnie

$S = \{ S_1, \dots S_n\}$; $S_i = [k_i^L, k_i^P]$ $i = 1,\dots,n$

Zakładamy, że żaden odcinek nie jest prostopadły do osi Ox oraz żadne trzy różne odcinki nie przecinają się w jednym punkcie. Procedura sprawdza czy w S istnieje co najmniej jedna para odcinków przecinających się }

```
begin

T := ∅;

Utwórz listę LS posortowanych końców odcinków z S w kolejności
od najmniejszej współrzędnej x do największej ,
w przypadku równych –najpierw punkt o mniejszej współrzędnej y;
dla każdego zaznacz czy jest lewym czy prawym końcem odcinka

        for każdy punkt p z listy LS do
         begin
                if p jest lewym końcem odcinka s then
                  begin
                        INSERT (T,s);
                        if(ABOVE (T,s) istnieje i przecina s )
                                or( BELOW (T,s) istnieje i przecina s )
                          then returnTRUE
                end;

                if p jest prawym końcem odcinka s then
                begin
                        if oba odcinki (ABOVE (T,s ) and (BELOW (T,s)) istnieją
                                and(ABOVE (T,s) przecina BELOW (T,s) )
                        then returnTRUE;
                        DELETE (T,s);
                end;
        end;
    return FALSE;
end;
```

ZNAJDOWANIE NAJMNIEJ ODLEGŁEJ PARY PUNKTÓW W ZBIORZE PUNKTÓW NA PŁASZCZYŹNIE

DANE: $n \in N$, $n \geq 2$;

$Q = \{p_1, \dots p_n\} : p_i \in R^2$, $p_i = (x_i, y_i)$ $1 \leq i \leq n$

WYNIK: $p, p' \in Q : \delta^{**} (p, p') = \min \{ d(p,q); p,q \in P \}$

ZNAJDŹ-PARĘ-NAJMNIEJ-ODLEGŁYCH-PUNKTÓW(P,X,Y);

{ Algorytm stosuje metodę "dziel i zwyciężaj" Każde wywołanie rekurencyjne algorytmu otrzymuje jako dane wejściowe :

•$P \in Q$

•tablicę X zawierającą punkty ze zbioru P posortowane **według współrzędnej x.**

•tablicę Y zawierającą punkty ze zbioru P posortowane **według współrzędnej y. }**

(1) Sprawdź czy **|P|** $\leq$**3**

TAK : wyznacz δ= **min { d(p,q), p,q** $\in$ P **}**

porównując odległości wszystkich par punktów w P;

niech p, q: d(p, q) = δ

NIE : DZIEL (2)

(2) DZIEL

- Podziel P na dwa "równoliczne" podzbiory P_L, P_R

  tzn.: $|P_L|$ = sup(|P| / 2), $|P_R|$ = inf(|P| / 2)

  przy pomocy prostej x = k , prostopadłej do osi Ox(ozn. prosta k) tak, że :

  wszystkie punkty z P_L leżą na lewo od prostej klub na prostej k

  a wszystkie punkty z P_R leżą na prawo od prostej klub na prostej k
- Podziel X na tablice X_L, X_R

  zawierające ,odpowiednio, punkty z P_L, P_R

  ,posortowane według współrzędnej x ;
- Podziel Y na tablice Y_L, Y_R

  zawierające ,odpowiednio, punkty z P_L, P_R

  posortowane według współrzędnej y ;

(3) ZWYCIĘŻAJ

Dwa wywołania rekurencyjne :

- ZNAJDŹ-PARĘ-NAJMNIEJ-ODLEGŁYCH-PUNKTÓW (P_L, X_L, Y_L);

  WYNIK : δ_L, p_L, q_L
- ZNAJDŹ-PARĘ-NAJMNIEJ-ODLEGŁYCH-PUNKTÓW (P_R, X_R, Y_R);

  WYNIK : δ_R, p_R, q_R

(4) POŁĄCZ :

- Oblicz : δ= min {δ_L, δ_R}, niech p, q: d(p, q) = δ

  Oblicz δ^{**}= min {δ, δ* }

  gdzie δ* = min { d(p_L, p_R) ; $p_L \in P_L$, $p_R \in P_R$ }

  Aby wyznaczyć δ* wystarczy :

  Rozpatrywać punkty z P_L o współrzędnych x z przedziału [k -δ, k]

  i punkty z P_R o współrzędnych x z przedziału [k, k + δ]

(i) Utwórz tablicę Y^{δ} z tablicy Y zawierającą te punkty uporządkowane według współrzędnej y;

(ii) Dla każdego $p \in Y^{\delta}$:

znajdź punkty w Y^{δ} w kole o promieniu δ i środku w p,
oblicz odległości tych punktów od p
i wyznacz najmniejszą(δ*) i punkty p*, q*, (tzn.: δ*= d(p*, q*))

Istotna obserwacja:

Dla każdego $p \in Y^{\delta}$ wystarczy przeglądać7 punktów z Y^{δ} powyżej p, tzn :
jeśli p= $Y^{\delta}[i]$, to obliczamy odległości p od $Y^{\delta}[i+1]$,..., $Y^{\delta}[i+8]$

3.Algorytmy tekstowe

WYSZUKIWANIE WZORCA

DANE : T , P∈Σ*, Σ-skończony zbiór (alfabet)
(T -tekst , P -wzorzec).

- Sprawdź czy wzorzec **P** występuje w tekście **T**
- Znajdź wzorzec **P** w tekście **T**
- Znajdź wszystkie wystąpienia wzorca **P** w tekście **T**
- Policz ile razy występuje wzorzec **P** w tekście **T**
- Zamień wszystkie wystąpienia słowa (wzorca) **P** na słowo **P’** w tekście **T**

RABIN-KARP-MATCHER (T ,P, 10);

{Alfabet = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} T[1..n] -tekst, P[1..m] -wzorzec;

Dla “niedużych” P , sprawdza czy P występuje w T }

```
begin
          h := 10m-1;
          p := 0 ;
          t := 0 ;

          for i := 1 to m do
                    begin
                              p := (10 * p + P[i]);
                              t := (10 * t + T[i]);
                    end;

          for s := 0 to n-m do
             begin
                     If p = t then
                              write( “Wzorzec występuje z przesunięciem”, s);
                    ifs < n-m
                              then  t := 10 * (t –T[s+1] * h ) + T[s+m+1];
              end;
end;
```

RABIN-KARP-MATCHER (T,P,d,q);

{ Alfabet = {0, 1, ..., d } T[1..n] -tekst, P[1..m] -wzorzec. Sprawdza czy P występuje w T }

```
begin
          h :=d^(m-1) mod q ;
          p := 0 ;
          t := 0 ;
          for i := 1 to m do
          begin
                    p :=(d * p + P[i]) mod q;
                    t := (d * t + T[i]) mod q;
          end;


          for s := 0 to n-m do
          begin
                    If p = t then
                              If P[1..m] = T[s+1..s+m] then
                                        write( “Wzorzec występuje z przesunięciem”, s);
                    if s < n-m then
                              t := (d * (t –T[s+1] * h ) + T[s+m+1] ) mod q;
          end;
```

```
end;
```

KODOWANIE

PRZYKŁAD

DANE : $\Sigma = \{ A, B, C, D, E, F \}$

	A	B	C	D	E	F
f_P	45	13	12	16	9	5

$P \in \Sigma^*$,

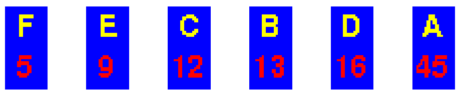
$f_P : \Sigma \rightarrow N$

WYNIK : Optymalny statyczny kod prefiksowy , tzn. :

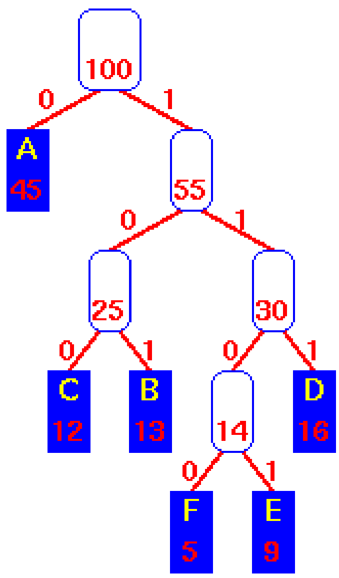
K : Σ   {0,1}*   taki, że K(P) ma najmniejszy "rozmiar" wśród wszystkich statycznych kodów prefiksowych K' dla Σ

Rozmiar K(P) = $\Sigma f_P (x) * |K(x)|$

- Posortowany ciąg lub
- Kolejka priorytetowa



DRZEWO:



HUFFMAN (Σ, f);

{Tworzy regularne drzewo binarne reprezentujące optymalny statyczny kod prefiksowy dla Σ}

```
begin
        n := |Σ|;
        Q := Σ;
        { Kolejka priorytetowa:
                V(a ∈ Σ) f(a) jest kluczem}

        for i := 1 to n-1 do
          begin
                New(w);
                x := EXTRACT-MIN(Q);
                y := EXTRACT-MIN(Q);
                w.left:= x;
                w.right:= y;
                w.f := x.f + y.f;
                INSERT(Q,w);
          end;
       returnEXTRACT-MIN(Q);
  {zwraca korzeń drzewa}
end;
```