Podstawowe algorytmy sekwencyjne: grafowe, geometryczne, tekstowe.

1. Algorytmy grafowe

REPREZENTACJE GRAFÓW

Graf G = (V, E), gdzie V = {1, 2, …, n} – zbiór wierzchołków, E = {e_1, …, e_m} – zbiór krawędzi

- Macierz sąsiedztwa

$$A[i,j] = \begin{cases} 0, \text{jeśli } \{i,j\} \notin E \\ 1 \text{ w przeciwnym razie} \end{cases}$$

- Macierz incydencji

$$B[i,j] = \begin{cases} -1, \text{jeśli krawędź j wychodzi z wierzchołka i} \\ 1, \text{jeśli krawędź j wchodzi do wierzchołka i} \\ 0 \text{ w pozostałych przypadkach} \end{cases}$$

- Lista sąsiedztwa

Do *Adj(u)* należą wszystkie wierzchołki *v* takie, że $(u, v) \in E$.

Przeszukiwanie grafu w głąb i wszerz

Dane: G = (V, E) w postaci list sąsiedztwa
Wynik: kolor(v)

PRZESZUKIWANIE W GŁĄB

Odwiedza każdy wierzchołek grafu. Badane są krawędzie ostatnio odwiedzonego wierzchołka v, z którego wychodzą jeszcze niezbadane krawędzie. Gdy wszystkie krawędzie opuszczające v są zbadane, przeszukiwanie wraca do wierzchołka, z którego v został odwiedzony. Na początku wszystkie wierzchołki mają kolor biały, szary otrzymują przy pierwszym odwiedzeniu, a czarny – gdy lista sąsiedztwa wierzchołka zostanie zbadana.

```
DFS (G):

for każdy u ∈ V(G)

     kolor (u) := BIAŁY

     π(u) := NIL

for każdy u ∈V(G)

     if kolor (u) = BIAŁY

          DFS-VISIT (u)
```

```
DFS-VISIT (u):

kolor (u) := SZARY

for każdy v ∈ Adj(u)

     if kolor (v) = BIAŁY

          DFS-VISIT (v)

kolor (u) := CZARNY
```

Animowany przykład: http://informatyka.wroc.pl/sites/default/files/user_files/u387/anim_dfs.gif
Wybrane zastosowania:

- Szukanie cykli w grafie
- Algorytm Kruskala
- Algorytm Prima

PRZESZUKIWANIE WSZERZ

Jeśli G jest spójny, to odwiedza każdy wierzchołek grafu, jeśli niespójny, to odwiedza każdy wierzchołek składowej spójności zawierającej wierzchołek s. Wierzchołki odwiedzone mają kolor szary (odwiedzony, ale jego lista sąsiedztwa nie została jeszcze w całości przejrzana) lub czarny (wierzchołki z listy sąsiedztwa rozpatrywanego wierzchołka są już zbadane). Wierzchołek koloru białego nie był jeszcze odwiedzony.

Dla każdego wierzchołka v osiągalnego z s, ścieżka w drzewnie przeszukiwania wszerz od s do v odpowiada najkrótszej ścieżce od s do v w grafie G (zawiera najmniejszą możliwą liczbę krawędzi).

```
BFS (G, s):

for każdy u ∈ V(G) – {s}

      kolor(u) := BIAŁY

      d(u) := ∞

      π(u) := NIL

kolor(s) := SZARY

d(s) := 0

π(s) := NIL

Q := {s}

while Q <> ∅

      u := head(Q)

            for każdy v ∈ Adj(u)

                  if kolor(v) = BIAŁY

                        kolor(v) = SZARY

                        d(v) := d(u) + 1

                        π(v) := u

                        ENQUEUE(Q, v)

            DEQUEUE(Q)

            kolor(u) = CZARNY
```

Wybrane zastosowania:

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

WYZNACZANIE MINIMALNEGO DRZEWA SPINAJĄCEGO GRAFU

Minimalne drzewo spinające (MST) grafu *G* to acykliczny podzbiór $T \subseteq E$, który łączy wszystkie wierzchołki z *V* i którego całkowita waga

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

jest najmniejsza.

Oba przedstawione algorytmy są zachłanne – w każdym kroku dodają do MST krawędź o najniższej możliwej wadze.

ALGORYTM PRIMA

Wyznacza zbiór krawędzi $ET = \{\{v, \pi(v)\}: v \in V(G) - \{r\}\}$ minimalnego drzewa spinającego grafu G.

Korzysta z:

- kolejki priorytetowej w postaci kopca binarnego typu MIN,
- EXTRACT-MIN(Q) – usuwa korzeń (wierzchołek ze szczytu kopca), przywracając własność kopca typu MIN.

```
MST-PRIM (G, w, r):

Q := V(G) for każdy u ∈ V(G)

      key(u) := ∞

      π(v) := NIL

key(r) := 0

π(r) := NIL

while Q <> ∅

      u := EXTRACT-MIN(Q)

      for każdy v ∈ Adj(u)

            if v ∈ Q i w(u, v) < key(v)

                  π(v) := u

                  key(v) := w(u,v)
```

ALGORYTM KRUSKALA

Korzysta z:

- *MAKE-SET(v)* – tworzy drzewo jednowierzchołkowe.
- *FIND-SET(u)* – zwraca reprezentanta zbioru zawierającego *u*. Jeżeli FIND-*SET(u) = FIND-SET(v)*, to wierzchołki *u* i *v* należą do tego samego drzewa.
- *UNION(u, v)* – operacja łączy dwa drzewa.

```
MST-KRUSKAL(G,w):

ET:= ∅

for każdy v ∈ V(G)

     MAKE-SET(v)

Posortuj krawędzie z E(G) niemalejąco względem wag w

for każda krawędź (u,v) ∈ E(G), w kolejności niemalejących wag

     if FIND-SET(u) <> FIND-SET(v)

          ET:=ET U{(u,v)}

          UNION(u,v)
```

ALGORYTM DIJKSTRY

Wyznacza odległość każdego wierzchołka grafu G od wierzchołka s, w: E(G) →R ≥ 0
Korzysta z:

- Kolejki priorytetowej (kluczem jest aktualnie wyliczona odległość od wierzchołka źródłowego s),
- *RELAX(u, v, w)* – sprawdza, czy przechodząc przez wierzchołek *u* można znaleźć krótszą od dotychczas najkrótszej ścieżki do *v* (szacuje wagę i aktualizuje ją jeżeli potrzeba).

```
RELAX(u ,v, w):
     if d(v) > d(u) + w(u, v) //oszacowanie wagi najkrótszej ścieżki
          d(v) := d(u) + w(u, v)
          π(v) := u
```

```
DIJKSTRA (G, w, s):

for każdy v ∈V(G)

     d(v) := ∞

     π(v) := NIL

d(s) := 0

S := ∅

Q := V(G)

while Q <> ∅

     u := EXTRACT-MIN(Q)

     S := S U {u}

     for każdy v ∈Adj(u)

          RELAX(u, v, w)
```

ALGORYTM BELLMANA-FORDA

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

```
BELLMAN-FORD (G, w, s):

for każdy v ∈V(G)

     d(v) := ∞

     π(v) := NIL

d(s) := 0

for i:= 1 to |V(G)|-1

     for każda krawędź (u, v) ∈ E(G)

          RELAX(u, v, w)

for każda krawędź (u, v) ∈E(G)

     if d(v) > d(u) + w(u, v)

          return FALSE

return TRUE
```

ALGORYTM FLOYDA-WARSHALLA

Wyznacza macierz D(n) odległości między każdą parą wierzchołków grafu reprezentowanego przez macierz wag W, zdefiniowaną następująco:

$$W_{ij} = \begin{cases} 0 \;\; dla \; i = j \\ w(i,j) \; dla \; i \neq j \; oraz \; (i,j) \in E \\ \infty \; dla \; i \neq j \; oraz \; (i,j) \notin E \end{cases}$$

```
FLOYD-WARSHALL (W):

D(0):=W

for k:=1 to n do

     for i:=1 to n do

          for j:=1 to n do

               D_ij(k) = min{D_ij(k-1), D_ik(k-1) + D_kj(k-1)}

return D(n)
```

2. Algorytmy geometryczne

ZNAJDOWANIE OTOCZKI WYPUKŁEJ ZBIORU PUNKTÓW NA PŁASZCZYŹNIE

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

ALGORYTM GRAHAMA

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.

GRAHAM-SCAN (Q):

1. Wyznacz punkt p0 = (x0, y0) (x_0= min{x_i: p_i = (x_i, y_0) ∈ Q}, y_0 = min{y_i: p_i = (x_i, y_i) ∈ Q})
2. 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)
3. Oznacz otrzymany zbiór Q'={p_0, p_1, …,p_m}
4. Utwórz stos pusty S
5. PUSH(p_0, S)
6. PUSH(p_1, S)
7. PUSH(p_2, S)
8. for i:=3 to m

```
    while przejście NEXT-TO-TOP(S), TOP(S), pᵢ nie oznacza skrętu w lewo
        POP(S)
    PUSH(pᵢ, S)
return S
```

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

S jest skończonym zbiorem odcinków na płaszczyźnie S = {S_1, ..., S_n}, S_i= [k_i^L, k_i^P], i = 1, ..., 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.

Korzysta z:

- *INSERT(T, s)* – wstawia odcinek *s* to *T*,
- *DELETE(T, s)* – usuwa odcinek *s* to *T*,
- *ABOVE(T, s)* – zwraca odcinek bezpośrednio powyżej *s* w *T*,
- *BELOW(T, s)* – zwraca odcinek bezpośrednio poniżej *s* w *T*.

```
ANY-SEGMENTS_INTERSECT (S):

T := Ø

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

     if p jest lewym końcem odcinka s

          INSERT (T,s)

          If ABOVE (T,s) istnieje i przecina s

          lub BELOW (T,s) istnieje i przecina s

               return TRUE

     if p jest prawym końcem odcinka s

          if oba odcinki (ABOVE (T,s) i (BELOW (T,s) istnieją

          i ABOVE (T,s) przecina BELOW (T,s)

               return TRUE

          DELETE (T,s);

return FALSE
```

3. Algorytmy tekstowe

WYSZUKIWANIE WZORCA

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

ALGORYTM RABINA-KARPA

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.

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

$h := 10^{m-1}$

```
p := 0

t := 0

for i := 1 to m

     p := (10 * p + P[i])

     t := (10 * t + T[i])
```

```
for s := 0 to n-m
      if p = t
            write("Wzorzec występuje z przesunięciem", s)
      if s < n-m
            t := 10 * (t –T[s+1] * h ) + T[s+m+1]
```

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.

h := d^{m-1} mod q

```
p := 0
t := 0
for i := 1 to m
      p :=(d * p + P[i]) mod q
      t := (d * t + T[i]) mod q
for s := 0 to n-m
      if p = t
            if P[1..m] = T[s+1..s+m]
                  write(„Wzorzec występuje z przesunięciem", s)
      if s < n-m
            t := (d * (t –T[s+1] * h) + T[s+m+1]) mod q
```