Wykład 4 – grafy

Grafem nazywamy strukturę **G = (V, E):**

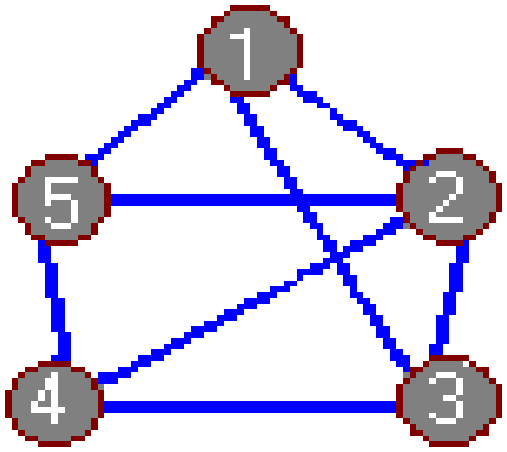
V – zbiór węzłów lub wierzchołków,

E – zbiór krawędzi,

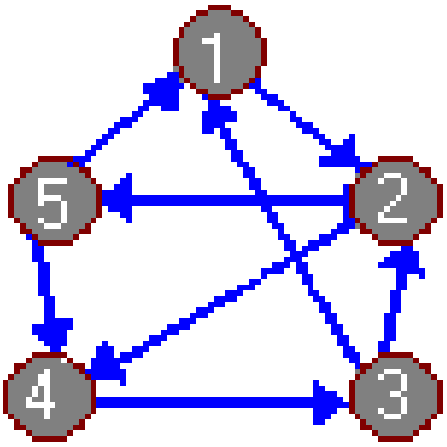
Grafy dzielimy na grafy skierowane i nieskierowane:

Formalnie, w grafach skierowanych E jest podzbiorem V x V, czyli podzbiorem zbioru par: {(x,y) | x,y należą do V}
Krawędzie oznaczamy strzałkami.

W grafach nieskierowanych E jest podzbiorem zbioru dwu-elementowych podzbiorów V: {{x,y} | x,y należą V }
Krawędzie oznaczamy odcinkami.



Rys.1. Graf nieskierowany



Rys.2. Graf skierowany

Graf może być etykietowany elementami zbioru etykiet A.

Wtedy jest trójką (V,E,M)

M: E → A

Rozmiar grafu jest równy sumie |V| + |E|

Graf nieskierowany: $|E| \leq (|V| \times |V|) \setminus 2$

Graf skierowany: $|E| \leq |V| \times |V|$

Podstawowe implementacje grafu G

Macierz sąsiedztwa:

M[x,y] = 1 jeśli (x,y) należy do E
0 jeśli (x,y) nie należy do E

Budujemy tablicę o rozmiarach |V|*|V|, gdzie |V| - liczba wierzchołków.

Wypełniamy ją:
Zerem - jeśli dwa wierzchołki nie są połączone krawędzią i
Jedynką- jeśli dwa wierzchołki są połączone.

Oto macierz sąsiedztwa dla grafu z rysunku 1:

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

W tej implementacji jest potrzebna pamięć $O(n^2)$ dla n = |V|.

Lista incydencji (sąsiedztwa)

Dla każdego wierzchołka x budujemy listę (oznaczaną L[x]) dla wierzchołków y połączonych krawędzią z x.

Lista dla grafu z rysunku 1 wygląda następująco:

1: 2, 3, 5
2: 1, 3, 4, 5
3: 1, 2, 4
4: 2, 3, 5
5: 1,2,4

W tej implementacji jest potrzebna pamięć O(|V| + |E|).

Macierz incydencji

Tablica o rozmiarach |V|*|E|.

Jeśli krawędź wychodzi z danego wierzchołka to piszemy w odpowiedniej kolumnie (-1), jeśli do niego wchodzi piszemy (+1), jeśli wierzchołek nie należy do krawędzi piszemy 0, jeśli jest to pętla własna piszemy 2.

Oto przykład dla grafu z rys. 2.

	1	**2**	**3**	**4**	**5**
1 - 2	-1	1	0	0	0
2 - 4	0	-1	0	1	0
2 - 5	0	-1	0	0	1
3 - 2	0	1	-1	0	0
4 - 3	0	0	1	-1	0
5 - 1	1	0	0	0	-1
5 - 4	0	0	0	1	-1

W tej implementacji jest potrzebna pamięć O(|V| * |E|).

Dodatkowe pojęcia dla grafów

Stopień wierzchołka - liczba krawędzi do niego przyległych

Graf regularny - graf, w którym każdy wierzchołek ma taki sam stopień

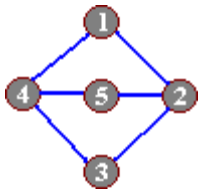
Graf planarny - graf, który można przedstawić na płaszczyźnie tak, by żadne dwie krawędzie się nie przecinały

f-graf - graf z ograniczonym stopniem wierzchołka, tzn. jego stopień nie może być większy niż f.

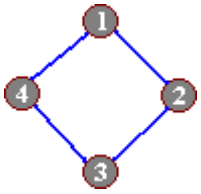
Graf prosty - to graf bez pętli własnych i krawędzi równoległych

Niezmiennik grafu - to liczba lub ciąg liczb, który zależy tylko od struktury grafu a nie od sposobu jego poetykietowania (np. liczba wierzchołków, liczba krawędzi)

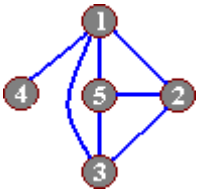
Liczba chromatyczna grafu - najmniejsza liczba kolorów do pokolorowania tak, by żadne dwa przyległe wierzchołki nie były tego samego koloru.



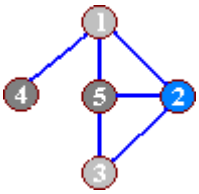
stopień wierzchołka *1* =2, wierzchołka *2* =3, graf planarny, f-graf dla f=3



graf **2**-regularny, planarny



graf planarny, f-graf dla f=4



graf planarny, liczba chromatyczna wynosi 3

Algorytm przechodzenia (przeszukiwania) grafu

Wychodząc od wierzchołka p, należy odwiedzić każdy wierzchołek i każdą krawędź, które są osiągalne z p.

Dozwolonym ruchem jest przejście krawędzią grafu wychodzącą z odwiedzonego już wierzchołka.

W jednym kroku będziemy odwiedzać jeden z wierzchołków oraz jedną z krawędzi grafu i zaznaczać je jako odwiedzone.

Na początku wszystkie wierzchołki i wszystkie krawędzie są zaznaczone jako nie odwiedzone.

1. Odwiedź wierzchołek p i zaznacz go jako odwiedzony.

2. Dopóki z jednego z odwiedzonych wierzchołków wychodzi nie odwiedzona jeszcze krawędź, wykonuj podane czynności:

   a) wybierz odwiedzony wierzchołek, powiedzmy v, z którego wychodzi nie odwiedzona krawędź;

   b) wybierz nie odwiedzoną krawędź, powiedzmy (v, w), wychodzącą z wierzchołka v;

   c) zaznacz krawędź (v, w) jako odwiedzoną;

   d) jeśli wierzchołek w nie został odwiedzony, odwiedź go i zaznacz jako odwiedzony.

Formalnie można pokazać, że powyższy algorytm jest poprawny.

Nie biorąc pod uwagę samej procedury odwiedzania wierzchołków i krawędzi złożoność algorytmu jest rzędu $O(|V| + |E|)$

Aby otrzymać konkretny algorytm należy wykonać podane tu kroki.

1. Przyjąć odpowiednią reprezentację grafu, na przykład V = {1. 2 •...• n} i listy sąsiedztwa L[v] dla v należącego do V.

2. Określić co to znaczy "zaznacz wierzchołek jako odwiedzony". Można na przykład dołączyć do każdego wierzchołka v pole visited[v] i przyjąć, że ‘false’ oznacza „wierzchołek nie odwiedzony", a ‘true’ "wierzchołek odwiedzony".

3. Określić sposób wyboru odwiedzanego wierzchołka, z którego wychodzi nie odwiedzona krawędź. Można na przykład przechowywać takie wierzchołki w **kolejce lub na stosie**

4. Określić sposób odróżniania (dla danego wierzchołka) krawędzi odwiedzonych od nie odwiedzonych. Można na przykład trzymać na liście sąsiedztwa danego wierzchołka v wskaźnik current[v] do pierwszej nie odwiedzonej krawędzi (current[v] = nil oznacza, że wszystkie krawędzie wychodzące z danego wierzchołka zostały odwiedzone).

Bardziej uszczegółowiony schemat przechodzenia grafu (visit jest procedurą "odwiedzania" wierzchołka; nie ma procedury odwiedzania krawędzi).

```
for v := 1 to n do
begin
      visited[v] := false;
       ustaw wskaźnik current[v] na pierwszy wierzchołek na liście L[v]
 end;
for v := 1 to n do if visited[v] = false then przeszukaj(v);
```

```
przeszukaj(p);
visit(p); visited[p] := true;
if current[p] <> nil then
begin
    S := {p};
    while S <> 0 do
    {S zawiera wszystkie odwiedzone do tej pory wierzchołki, z
    których wychodzą nie odwiedzone jeszcze krawędzie}
    begin
    wybierz wierzchołek v ze zbioru S;
    niech w będzie wierzchołkiem wskazywanym przez current[v];
    przesuń current[v] do następnego wierzchołka na liście L[v];
    if current[v] = nil then S := S - {v};
    if not visited[w] then
       begin
          visit(w);
          visited[w] := true;
           if current[w] <> nil then S := S u {w}
       end
   end
end;
```

Przyjęte implementacje zbioru S to stos i kolejka.

Najpierw uszczegółowimy schemat przechodzenia grafu, przedstawiając S za pomocą stosu i interpretując operacje na S jako operacje na stosie.

Przyjmujemy, że operacje pop i push będą realizowane jako procedury, a front jako funkcja.

Otrzymujemy algorytm przechodzenia grafu w głąb (metoda DFS).

```
for v := 1 to n do
begin
      visited[v] := false;
       ustaw wskaźnik current[v] na pierwszy wierzchołek na liście L[v]
 end;
for v := 1 to n do if visited[v] = false then dfs(v);

procedure dfs(p);
visit(p); visited[p] := true;
if current[p] <> nil then
begin
    S := [ ]; push(S,p);
    while S <> 0 do
    {stos S zawiera wszystkie odwiedzone do tej pory wierzchołki, z
    których wychodzą nie odwiedzone jeszcze krawędzie}
    begin
    v := front(S);
    niech w będzie wierzchołkiem wskazywanym przez current[v];
    przesuń current[v] do następnego wierzchołka na liście L[v];
    if current[v] = nil then pop(S);
    if not visited[w] then
       begin
          visit(w);
          visited[w] := true;
           if current[w] <> nil then push(S,w)
       end
   end
end;
```

Zapiszemy teraz schemat przechodzenia grafu, przedstawiając S jako kolejkę i interpretując operacje na S jako operacje na kolejce. (Przyjmujemy, że operacje pop i inject są realizowane jako procedury, a front jako funkcja).

Algorytm przechodzenia grafu wszerz (metoda BFS)

```
for v := 1 to n do
begin
      visited[v] := false;
       ustaw wskaźnik current[v] na pierwszy wierzchołek na liście L[v]
 end;
for v := 1 to n do if visited[v] = false then bfs(v);

procedure bfs(p);
visit(p); visited[p] := true;
if current[p] <> nil then
begin
    S := [ ]; inject(S,p);
    while S <> 0 do
    {kolejka S zawiera wszystkie odwiedzone do tej pory wierzchołki,
    z których wychodzą nie odwiedzone jeszcze krawędzie}
    begin
    v := front(S);
    niech w będzie wierzchołkiem wskazywanym przez current[v];
    przesuń current[v] do następnego wierzchołka na liście L[v];
    if current[v] = nil then pop(S);
    if not visited[w] then
       begin
          visit(w);
          visited[w] := true;
          if current[w] <> nil then inject(S,w)
       end
   end
end;
```

DFS z czasami odwiedzenia d[] i przetworzenia f[]

```
for v := 1 to n do
begin
      visited[v] := false;
       ustaw wskaźnik current[v] na pierwszy wierzchołek na liście L[v]
 end;
for v := 1 to n do if visited[v] = false then dfs(v);

procedure dfs(p);
t := 0;
visit(p); visited[p] := true;
t := 1; d[p] := t;
if current[p]  =  nil then begin t := 2; f[p] := t end;
if current[p] <> nil then
begin
S := [ ]; push(S,p);
    while S <> 0 do
    {stos S zawiera wszystkie odwiedzone do tej pory wierzchołki, z
    których wychodzą nie odwiedzone jeszcze krawędzie}
    begin
    v := front(S);
    niech w będzie wierzchołkiem wskazywanym przez current[v];
    przesuń current[v] do następnego wierzchołka na liście L[v];
    if current[v] = nil then begin pop(S); t := t+1; f[v] := t end;
    if not visited[w] then
       begin
          visit(w);
          visited[w] := true;
           t := t+1; d[w] := t;
          if current[w] <> nil then push(S,w)
           else begin t := t+1; f[v] := t end;
       end
   end
end;
```

Silnie spójne składowe

Maksymalny podgraf G’ danego grafu taki, że dla każdej pary wierzchołków istnieje droga w G’ nazywamy silnie spójną składową G.

Transpozycja grafu

Dany graf $G = (V,E)$

Graf transponowany $G^T = (V,E')$,

gdzie (x,y) należy do E’ wtedy i tylko wtedy (y,x) należy do E.

Dla reprezentacji listowej algorytm obliczający G^T ma złożoność $O(|V| + |E|)$.

FAKT: Każda silnie spójna składowa grafu G jest po transpozycji silnie spójną składową grafu G^T.

Algorytm znajdowania SSS:

1. wykonaj dfs(G) w celu przypisania każdemu wierzchołkowi czasu przetworzenia f[u],
2. oblicz G^T
3. wykonaj dfs(G^T), ale w głównej pętli procedury dfs rozważaj wierzchołki w kolejności malejących wartości f[u],
4. wypisz wierzchołki z każdego drzewa w lesie przeszukiwania w głąb z kroku 3 jako oddzielną silnie spójną składową.

Złożoność obliczeniowa procedury SSS(G) wynosi O(|V| + |E|).

Główna pętla:

```
for v := 1 to n do
begin
     visited[v] := false;
      ustaw wskaźnik current[v] na pierwszy wierzchołek na liście L[v]
 end;
for  v := 1 to n do if visited[v] = false then dfs(v);
```