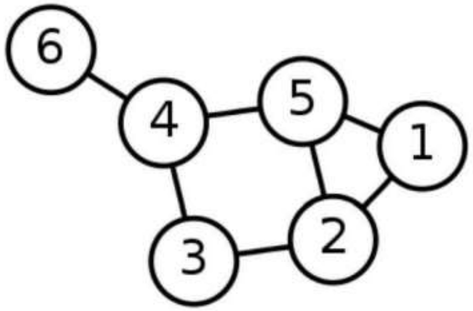
C7

miercuri, 24 ianuarie 2024 23:24

Informal, un graf reprezintă o mulțime de *obiecte* (*noduri, vârfuri, puncte etc.*) între care există anumite *legături* (*linii*, *muchii, arce etc.*).



Imagine: http://en.wikipedia.org/wiki/File:6n_graf.svg 11

Ce e un graf?

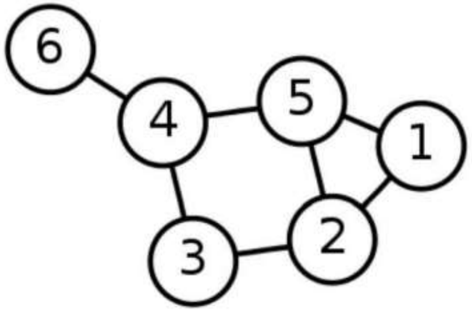
Formal, un graf G e o pereche ordonată $G=(V, E)$

V - *mulțimea nodurilor (eng. Vertices)* și

E - *mulțimea muchiilor (eng. Edges)*

E - *mulțimea muchiilor (eng. Edges)*

- o mulțime de perechi $(u, v) \in V \times V$



Imagine: http://en.wikipedia.org/wiki/File:6n_graf.svg

12

Ce e un graf?

Mulțimea nodurilor trebuie să fie o mulțime *finită* și *nevidă*, deci nu e posibil să avem un graf fără noduri, dar e posibil să avem un graf fără muchii.

Așadar, un graf poate fi reprezentat sub forma unei *figuri geometrice* alcătuite din *puncte* (care corespund vârfurilor/nodurilor) și din *linii* drepte sau curbe care unesc aceste puncte (care corespund muchiilor sau arcelor).

sau curbe care unesc aceste puncte (care corespund muchiilor sau arcelor).

Grafuri – noțiuni generale

Se numește *ordin al unui graf* numărul de noduri al grafului.

Un nod *v* este *incident* cu o muchie *r* dacă muchia r atinge nodul v - v ∈ r.

Două noduri se numesc *adiacente* dacă există o muchie care le unește.

Două muchii sunt *adiacente* dacă există un nod care să fie incident cu ambele muchii.

Grafuri – noțiuni generale

Se numeste grad al unui nod, *n*umărul de muchii

Grafuri – noțiuni generale

Se numește grad al unui nod, *n*umărul de muchii care sunt incidente la acel nod.

Dacă se adună *gradele tuturor nodurilor* din graful G se obține de două ori numărul de muchii.

Un graf e *orientat* dacă muchiile sale sunt perechi *ordonate*

Un graf e *neorientat* dacă muchiile sale sunt perechi *neordonate* (nu contează sensul parcurgerii)

Mulțimea muchiilor unui graf formează o *relație* $E \in V \times V$ pe mulțimea nodurilor.

Un graf *neorientat* poate fi reprezentat printr-o relație *simetrică:*

$$\forall u, v \in V .\ (u, v) \in E \rightarrow (v, u) \in E$$

Într-un graf *orientat*, E e o relație oarecare (nu trebuie să fie simetrică, dar poate fi)

Reciproc, *orice relație binară* poate fi văzută ca un *graf orientat* pentru $(u, v) \in E$ introducem o muchie $u \rightarrow v$

Un drum are un *nod inițial* x_0 și un *nod final* x_n.

Lungimea unui drum e numărul de muchii parcurse.
în particular, poate fi zero (un nod x_0, fără niciun fel de muchii)

Un *ciclu* e un drum de *lungime nenulă* în care nodurile de început și sfârșit sunt identice (aceleași).

Adeseori, lucrăm cu *cicluri simple*:

- cicluri în care muchiile și nodurile *nu apar de mai multe ori* (cu excepția nodului inițial care e și cel final).

Un graf e *conex* dacă are un drum *de la orice nod la orice nod*. (definiție generală, depinde de noțiunea de *drum* – în graf orientat sau neorientat)

Pentru grafuri *neorientate*:

O *componentă conexă* e un subgraf conex maximal.

- deci are un drum între oricare două noduri
- nu s-ar mai putea adăuga alte noduri păstrând-o conexă

Un graf cu n noduri și e muchii are un număr de componente conexe $\geq n - e$. Se poate demonstra prin inducție.

componente conexe ≥ $n - e$. Se poate demonstra prin inducție.

Un graf *orientat* e *slab conex* dacă are un drum *neorientat* de la orice nod la orice nod, și *tare conex* dacă are un drum *orientat* de la orice nod la orice nod.

O *componentă tare conexă* e un *subgraf* tare conex maximal. Componentele tare conexe sunt *disjuncte*:

- *R*(*u, v*) : *drum*(*u, v*) și *drum*(*v, u*) e o *relație de echivalență*, și componentele tare conexe sunt *clase de echivalență*

Determinarea componentelor conexe (graf neorientat)

Componentele conexe sunt *clase de echivalență*

- orice nod e în componenta proprie - *reflexivitate*

- un drum de la *u* la *v* e și drum de la *v* la *u* - *simetrie*

- *drum*(*u, v*) ∧ *drum*(*v, w*) → *drum*(*u, w*) - *tranzitivitate*

Determinăm componentele conexe parcurgând muchiile grafului:

Determinăm componentele conexe parcurgând muchiile grafului:

- inițial, fiecare nod e în propria componentă
- pentru o muchie (*u, v*) *unim* componentele lui *u* și *v*

Drumuri Euleriene (în grafuri neorientate)

Gradul unui nod (într-un graf neorientat) e numărul de muchii care ating nodul.

Un *drum eulerian* e un *drum* care conține *toate* muchiile unui graf exact o dată.

Un *ciclu eulerian* e un *ciclu* care conține *toate* muchiile unui graf exact o dată.

Un graf conex neorientat are un *ciclu eulerian* dacă și numai dacă *toate nodurile au grad par*.

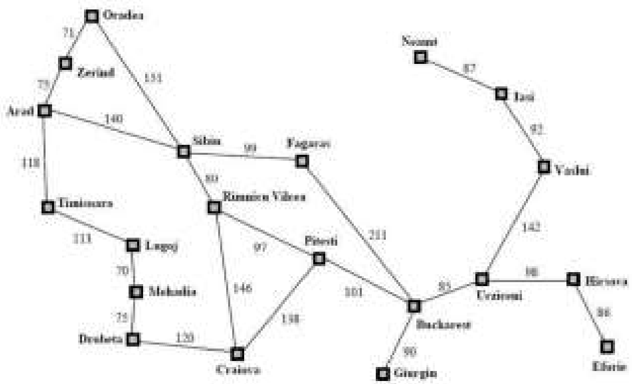
Un graf conex neorientat are un *ciclu eulerian* dacă și numai dacă *toate nodurile au grad par*.

Un graf conex neorientat are un *drum* (dar nu și un ciclu) *eulerian* dacă și numai dacă *exact două noduri au grad impar*.

(primul și ultimul nod din drum)

Exemple: hărțile ca și grafuri ponderate

Graf ponderat: fiecare muchie are asociată o valoare numerică numită *cost* (poate reprezenta lungime, capacitate, etc.)



Hartă (inexactă) din Russell & Norvig, Introduction to AI

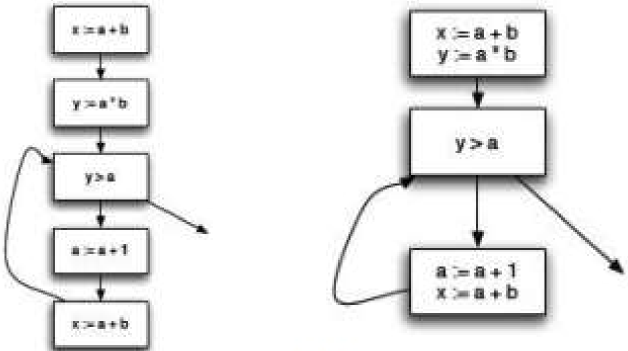
37

Exemple: Graful fluxului de control (control flow graph)

Reprezentarea programelor în compilatoare, analizoare de cod, etc.

- nodurile: *instrucțiuni* sau *secvențe liniare* de instrucțiuni (*basic blocks*)
- muchiile: descriu secvențierea instrucțiunilor (*fluxul de control*)

```
x := a + b;
y := a * b;
while (y > a) {
  a := a + 1;
  x := a + b
}
```



http://vinaytech.wordpress.com/2008/10/04/abstract-syntax-tree/

38

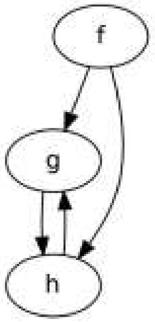
Exemple: Graful de apel al funcțiilor (call graph)

Introducem o muchie $f \rightarrow g$ dacă funcția f apelează pe g

Graful de apel e ciclic dacă există funcții (direct sau indirect) recursive

def g(x):

```
def g(x):
        return 0 if x==0 else 1+h(x-1)
def h(x):
        return 1 if x==0 else 2*g(x-1)
def f(x):
        return h(x) + g(x)
```



39

Parcurgerea în adâncime (depth-first)

Parcurgerea grafului în adâncime e o traversare în *preordine.*

După vizitarea nodului se parcurg (recursiv) toți vecinii (dacă nu au fost vizitați încă)

Acționează ca și cum vecinii ar fi introduși într-o *stivă.*

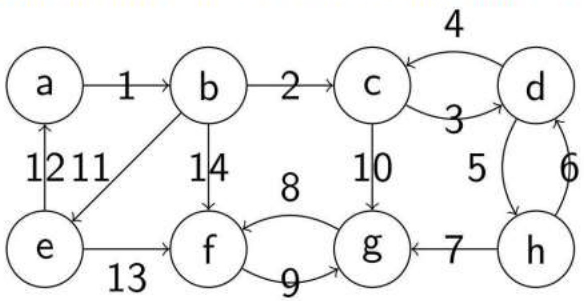
44

Parcurgerea în adâncime (depth-first)

Parcurgerea în adâncime (depth-first)

Fie graful de mai jos, cu listele de adiacență ordonate după litere.

Ordinea muchiilor parcurse de la *a* în adâncime e cea indicată:



Se poate programa: funcție recursivă, acumulând mulțimea nodurilor vizitate

45

Parcurgerea prin cuprindere (breadth-first)

Parcurgerea prin cuprindere vizitează nodurile în ordinea *distanței minime* de nodul de plecare

(în "valuri" care se depărtează de la nodul de pornire)

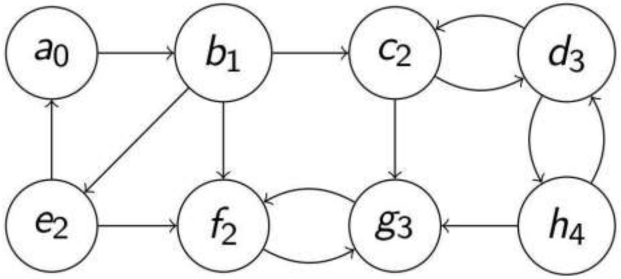
Nodurile încă nevizitate se pun într-o *coadă*

Nodurile încă nevizitate se pun într-o *coadă*.

46

Parcurgerea prin cuprindere (breadth-first)

In figura de mai jos, se indică distanța minimă de la nodul a (nodurile cu distanță mai mare sunt parcurse mai târziu)



O implementare: funcție recursivă,
acumulând: mulțimea tuturor nodurilor vizitate
frontiera: mulțimea nodurilor noi atinse în runda curentă

Afișarea muchiilor unui graf

import functools

Afișarea muchiilor unui graf

```
import functools

def afisare_muchii(graf, muchii = set()):
   def functie(acc,elem):
      cheie, valoare = elem
      def f_multime(acc2,elem2):
         muchii.add((cheie,elem2))
      functools.reduce(f_multime, valoare, 0)
   functools.reduce(functie, graf.items(), 0)
   return muchii

print(afisare_muchii(graf))
# {('a', 'c'), ('d', 'e'), ('a', 'b'), ('e', 'd'), ('b', 'a'), ('b', 'd'), ('c', 'a'),
('c', 'd')}
```

53

Adaugarea unui nod nou

```
graf = {
   "a" : {"b","c"},
   "b" : {"a", "d"},
   "c" : {"a", "d"},
   "d" : {"e"},
   "e" : {"d"}
}
def adaugare_nod(graf, nod):
   if(not nod in graf):
```

```
def adaugare_nod(graf, nod):
  if(not nod in graf):
    graf[nod] = set()
  return graf
print(adaugare_nod(graf, "f"))             # {'a': {'c', 'b'},
'b': {'d', 'a'}, 'c': {'d', 'a'}, 'd': {'e'}, 'e': {'d'}, 'f': set()}
```

55

Adaugarea unei muchii noi

```
def adaugare_muchie_orientat(graf, muchie):
  if (muchie[0] in graf):
    graf[muchie[0]].add(muchie[1])
  else:
    graf[muchie[0]]={muchie[1]}
  if (not muchie[1] in graf):
    graf[muchie[1]] = set()
  return graf

print(adaugare_muchie_orientat(graf,("a","d")))
print(adaugare_muchie_orientat(graf,("f","g")))
# {'a': {'b', 'c', 'd'}, 'b': {'d', 'a'}, 'c': {'d', 'a'}, 'd': {'e'}, 'e': {'d'}}
# {'a': {'b', 'c', 'd'}, 'b': {'d', 'a'}, 'c': {'d', 'a'}, 'd': {'e'}, 'e': {'d'}, 'f':
{'g'}, 'g': set()}
```

57

Exerciții

1. Fie un graf reprezentat de mulțimea perechilor de noduri adiacente. Să se creeze structura de date care reține informațiile despre graf într-un dicționar.

Exemplu:

Input: {(1, 3), (1, 2), (2, 4), (4, 1)}

Output: {2: {4}, 4: {1}, 1: {2, 3}, 3: set()}

59

Exerciții

```
import functools
def constructie_graf(multime, dictionar = {}):
   def functie(acc, elem):
      if (elem[0] in dictionar):
         dictionar[elem[0]].add(elem[1])
      else:
         dictionar[elem[0]] = set({elem[1]})
```

```
        else:
            dictionar[elem[0]] = set({elem[1]})
        if(not elem[1] in dictionar):
            dictionar[elem[1]] = set()
    functools.reduce(functie, multime, 0)
    return dictionar
print(constructie_graf({(1, 3), (1, 2), (2, 4), (4, 1)}))
```

60