Esercizi su grafi in linguaggio Python

Corso Ottimizzazione Combinatoria (IN440)

Prof. M. Liverani

Programmazione a oggetti

- La programmazione orientata agli oggetti (OOP – *Object Oriented Programming*) capovolge il rapporto tra procedura e strutture dati esistente nella programmazione procedurale
- Programmazione procedurale:
  - l'**algoritmo** è codificato in una **procedura** (**funzione** in C)
  - le **strutture dati** sono **definite** e **istanziate** (allocate in memoria) all'interno delle procedure
  - una procedura può richiamarne un'altra passandogli i **riferimenti** alle strutture dati istanziate nella propria area di memoria
- Programmazione ad oggetti:
  - si definisce una struttura dati denominata «**classe di oggetti**», indicando il tipo di ogni **attributo** della struttura dati
  - nella «classe» si definiscono anche delle procedure per operare sulla struttura dati: i «**metodi**»
  - i metodi possono essere:
    - **costruttori**: per creare una istanza di un oggetto e inizializzarne gli attributi
    - **setter** / **getter**: per impostare o acquisire i valori degli attributi della struttura dati (oggetto)
    - **altri metodi** per eseguire operazioni specifiche sull'oggetto

Programmazione a oggetti

Esempi in Python:

- `list`: classe di oggetti per gestire una struttura dati di array o lista
- `a = list()`: metodo costruttore, crea una istanza di un oggetto della classe «`list`» e ne assegna il riferimento ad `a`; da questo momento `a` non è una variabile, ma un oggetto
- `a.append(x)`: metodo che aggiunge l'elemento `x` all'oggetto lista `a`
- `a.sort()`: metodo che ordina gli elementi dell'oggetto lista `a`

Proprietà della programmazione a oggetti

- Incapsulamento: non tutti gli attributi di un oggetto sono accessibili dall'esterno, ma solo attraverso metodi specifici; non tutti i metodi di una classe di oggetto sono richiamabili dall'esterno, ma solo da altri metodi della classe
- Ereditarietà: è possibile definire classi di oggetti che ereditano gli attributi e i metodi di un'altra classe (e poi ne specificano o estendono alcune caratteristiche)
- Polimorfismo: uno stesso metodo può essere ridefinito più volte per operare su tipi di dato differenti

Costruzione di un grafo C_n

Costruzione del grafo ciclo $C_n = (V, E)$ con n vertici $V = \{1, ..., n\}$ e n spigoli $E = \{(1,2), (2,3), ..., (n,1)\}$

Algoritmo 48 CYCLEGRAPH(n)

Input: Un intero n

Output: Il grafo $G = C_n$

1: **per** $i = 1, \ldots, n$ **ripeti**
2: $V(G) = V(G) \cup \{i\}$
3: **fine-ciclo**
4: **per** $i = 1, \ldots, n-1$ **ripeti**
5: $E(G) = E(G) \cup \{(i, i+1)\}$
6: **fine-ciclo**
7: $E(G) = E(G) \cup \{(n, 1)\}$

Costruzione di un grafo C_n

```
from pythonds import *

def cycleGraph(g, n):
    for v in range(1, n+1):
        g.addVertex(v)
    for v in range(1, n):
        g.addEdge(v, v+1)
        g.addEdge(v+1, v)
    g.addEdge(1, n)
    g.addEdge(n, 1)
    return
```

```
G = Graph()
n = int(input("N. vertici:"))
cycleGraph(G, n)

print("Vertici:", G.getVertices())
print("Spigoli: ")
for u in G:
    for v in u.getConnections():
        print("(%s,%s)" % (u.getId(),v.getId()))
```

Costruzione di un grafo completo bipartito $K_{p,q}$

Costruzione di un grafo completo bipartito $G = K_{p,q}$ con $G = (V, E)$, $|V| = p + q$, $V = V_1 \cup V_2$ dove $V_1 = \{1, ..., p\}$, $V_2 = \{p + 1, ..., p + q\}$ e $(u, v) \in E$ se e solo se $u \in V_1$ e $v \in V_2$

Algoritmo 49 BIPARTITECOMPLETEGRAPH(p, q)

Input: Due interi p e q

Output: Il grafo bipartito completo $G = K_{p,q}$

```
1: per i = 1,...,p+q ripeti
2:     V(G) = V(G) ∪ {i}
3: fine-ciclo
4: per i = 1,...,p ripeti
5:     per j = p+1,...,p+q ripeti
6:         E(G) = E(G) ∪ {(i,j)}
7:     fine-ciclo
8: fine-ciclo
```

Costruzione di un grafo completo bipartito $K_{p,q}$

```
from pythonds import *

def bipartiteCompleteGraph(g, p, q):
   for v in range(1, p+q+1):
      g.addVertex(v)
   for u in range(1, p+1):
      for v in range (p+1, p+q+1):
         g.addEdge(u, v)
         g.addEdge(v, u)
   return
```

```
G = Graph()
p = int(input("N. vertici primo insieme:"))
q = int(input("N. vertici secondo insieme:"))
bipartiteCompleteGraph(G, p, q)

print("Vertici:", G.getVertices())
print("Spigoli: ")
for u in G:
   for v in u.getConnections():
      print("(%s,%S)" % (u.getId(),
                                 v.getId()))
```

Costruzione di un grafo random con probabilità p

Dato $n > 0$ e $p \in [0, 1]$, costruire il grafo random $G = (V, E)$ con $V = \{1, ..., n\}$ e, per $u, v \in V$, $(u, v) \in E$ con probabilità p

Algoritmo 50 RANDOMGRAPH(p, n)

Input: Un intero $n > 0$ e una probabilità $0 \leq p \leq 1$
Output: Il grafo $G = (V, E)$

```
1: per i = 1,...,n ripeti
2:     V(G) = V(G) ∪ {i}
3: fine-ciclo
4: per u = 1,...,n ripeti
5:     per v = 1,...,n ripeti
6:         sia x un numero casuale in [0,1]
7:         se u ≠ v and x ≤ p allora
8:             E(G) = E(G) ∪ {(u,v)}
9:         fine-condizione
10:    fine-ciclo
11: fine-ciclo
```

NOTA:

il modulo «`random`» offre la funzione `random()` che restituisce un numero casuale in $[0, 1]$

Costruzione di un grafo random con probabilità *p*

```
from pythonds import *
from random   import *

def randomGraph(g, n, p):
   for v in range(1, n+1):
      g.addVertex(v)
   for u in range(1, n+1):
      for v in range (1, n+1):
         x = random()
         if x <= p and u != v:
            g.addEdge(u, v)
   return
```

```
G = Graph()
n = int(input("N. vertici:"))
p = float(input("Probabilita':"))
randomGraph(G, n, p)

print("Vertici:", G.getVertices())
print("Spigoli: ")
for u in G:
   for v in u.getConnections():
      print("(%s,%s)" % (u.getId(), v.getId()))
```

Visualizzazione in output di un grafo

Dato $n > 0$ e $p \in [0, 1]$, costruire il grafo random $G = (V, E)$ con $V = \{1, ..., n\}$ e, per $u, v \in V$, $(u, v) \in E$ con probabilità p. Visualizzare l'insieme dei vertici e degli spigoli e una rappresentazione grafica del grafo G

Visualizzazione in output di un grafo

```
from pythonds import *
from graphics import *
from random   import *
import numpy as np

def randomGraph(g, n, p):
   for v in range(1, n+1):
      g.addVertex(v)
   for u in range(1, n+1):
      for v in range (1, n+1):
         x = random()
         if x <= p and u != v:
            g.addEdge(u, v)
   return

def printGraph(g):
   print("Vertici: ", g.getVertices())
   print("Spigoli:")
   for u in g:
      for v in u.getConnections():
         print("(%s,%s)" % (u.getId(), v.getId()))
```

```
def plotGraph(g):
   win = GraphWin("Grafo", 800, 600)
   win.setCoords(-400, -300, 400, 300)
   n = len(g.getVertices())
   for i in range(1,n+1):
      a = Circle(Point(np.sin(6.28/n*i)*200, np.cos(6.28/n*i)*200), 5)
      a.setFill("red")
      a.draw(win)
      a = Text(Point(np.sin(6.28/n*i)*215, np.cos(6.28/n*i)*215), str(i))
      a.draw(win)
   for i in range(1, n+1):
      u = Point(np.sin(6.28/n*i)*200, np.cos(6.28/n*i)*200)
      for x in g.getVertex(i).getConnections():
         v = Point(np.sin(6.28/n*x.getId())*200, np.cos(6.28/n*x.getId())*200)
         a = Line(u, v)
         a.draw(win)
   win.getMouse()
   win.close()
```

```
g = Graph()
n = int(input("N. vertici:  "))
p = float(input("Probabilita': "))
randomGraph(g, n, p)
printGraph(g)
plotGraph(g)
```