Algoritmi su grafi in linguaggio Python

Marco Liverani*

Aprile 2019

1 Introduzione

Questa breve guida illustra l'utilizzo del linguaggio di programmazione Python per la codifica di algoritmi che operano sulle strutture dati di grafo e di albero. Viene impiegato il modulo **pythonds**, definito per la gestione di numerose strutture dati in Python, per la rappresentazione di grafi (mediante liste di adiacenza), alberi, liste, code, code di priorità e pile.

Il modulo `pythonds` (*Python Data Structures*) è una collezione di classi di oggetti con cui vengono definite le principali strutture dati utili per la codifica efficiente di algoritmi su grafi. È stato scritto da Brad Miller e David Ranum ed è descritto estensivamente, corredato da numerosissimi esempi, sul sito web

`http://interactivepython.org/runestone/static/pythonds/index.html`

Con il modulo `pythonds` e le numerose classi di oggetti definite al suo interno, abbiamo a disposizione metodi per operare in modo estremamente semplice sulle strutture dati astratte, come liste, code, pile, alberi binari e grafi. Rispetto ad un programma equivalente scritto in linguaggio C, sebbene quest'ultimo possa risultare probabilmente più veloce, il programma in linguaggio Python, con l'uso di `pythonds`, sarà molto più compatto e leggibile e, come vedremo negli esempi riportati nelle pagine seguenti, assai vicino allo pseudo-codice di un algoritmo.

Il modulo `pythonds` è progettato per la versione 3.x del linguaggio Python. Può essere installato utilizzando il programma `pip`, dedicato all'installazione di moduli e package del linguaggio Python, attraverso il seguente comando:

```
python3 -m pip install pythonds
```

L'uso del modulo deve essere dichiarato all'inizio del programma, specificando quali classi del modulo si intende utilizzare. Ad esempio:

```
from pythonds.graphs import Graph
```

oppure, per importare tutti gli oggetti definiti nel modulo `pythonds`:

```
from pythonds import *
```

Nel seguito di questa breve guida viene prima fornita una descrizione delle classi e dei metodi resi disponibili dal modulo `pythonds` e, successivamente, vengono forniti alcuni esempi di implementazione in Python di algoritmi su grafi.

*E-mail: liverani@mat.uniroma3.it; ultima modifica: 6 aprile 2019. Questa guida è disponibile su Internet all'indirizzo `http://www.mat.uniroma3.it/users/liverani/doc/pythonGraphs.pdf`

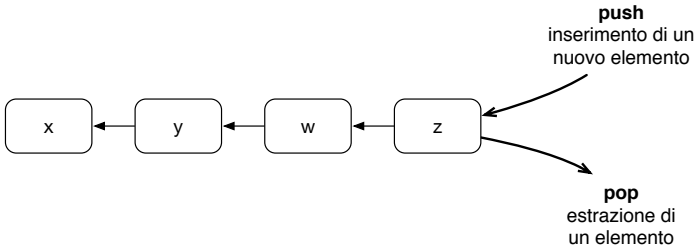

Figura 1: Una struttura dati di Pila / *Stack*

2 Classi

Il modulo pythonds mette a disposizione le seguenti classi principali:

- **Stack**: struttura dati sequenziale per la rappresentazione di una *pila* (struttura dati di tipo LIFO: *last in first out*);
- **Queue**: struttura dati sequenziale per la rappresentazione di una *coda* (struttura dati di tipo FIFO: *first in first out*);
- **Deque**: struttura dati simile alla coda, ma con due punti di ingresso e due punti di uscita;
- **BinarySearchTree**: struttura dati per la rappresentazione di alberi binari di ricerca;
- **BinHeap**: struttura dati per la rappresentazione di heap binari;
- **Graph**: struttura dati per la rappresentazione di grafi e alberi generici.

3 Strutture dati sequenziali

3.1 La classe Stack

Una *pila* (in inglese *stack*) è una struttura dati sequenziale (come una lista) utilizzata in modo tale che gli elementi vengano aggiunti e rimossi sempre a partire dal primo elemento della sequenza; per questo motivo sono dette strutture LIFO (*last in first out*), visto che l'ultimo elemento inserito nella struttura dati, è anche il primo ad essere estratto.

I metodi per definire ed operare su un oggetto della classe Stack sono descritti brevemente di seguito.

- **Stack()**: è il costruttore della classe; crea un nuovo stack vuoto e restituisce un riferimento all'oggetto. Ad esempio:

```
a = Stack()
```

- **push(*item*)**: aggiunge un nuovo elemento alla struttura dati inserendolo all'inizio della pila, come primo elemento (vedi Figura 1). L'elemento da aggiungere allo stack viene passato come argomento del metodo. Ad esempio le seguenti istruzioni creano lo stack a ed inseriscono in cima alla pila un elemento con il valore 17:

```
a = Stack()
a.push(17)
```

- **pop()**: restituisce l'elemento in cima alla pila e lo rimuove dallo stack. Ad esempio:

```
b = a.pop()
```

- **peek()**: restituisce il valore dell'elemento in cima alla pila, ma non lo rimuove dalla struttura dati; lo stack non viene modificato. Ad esempio:

```
b = a.peek()
```

- **isEmpty()**: verifica se la pila è vuota e restituisce True se è vuota, False se contiene almeno un elemento. Ad esempio:

```
if a.isEmpty() == True:
    print("La pila è vuota")
else:
    print("La pila contiene almeno un elemento")
```

- **size()**: restituisce il numero di elementi presenti nella pila; restituisce un numero intero e non richiede alcun parametro. Ad esempio:

```
n = a.size()
if n>0:
    print("La pila contiene", n, "elementi")
else:
    print("La pila è vuota")
```

3.2 La classe Queue

Una *coda* (in inglese *queue*) è una struttura dati astratta, sequenziale, che si distingue da una generica lista e da una pila, per la modalità con cui vengono effettuate le operazioni di inserimento ed estrazione degli elementi. In una coda gli elementi entrano dall'ultima posizione ed escono dalla prima. Per questo motivo la coda è una struttura dati di tipo FIFO (*first in first out*): il primo elemento inserito nella struttura dati sarà anche il primo ad essere estratto, mentre gli elementi inseriti successivamente vengono accodati in fondo alla struttura dati stessa.

I metodi per definire ed operare su un oggetto della classe Queue sono descritti brevemente di seguito.

- **Queue()**: è il metodo costruttore per istanziare gli oggetti della classe Queue; questo metodo crea una coda vuota e restituisce un riferimento alla struttura dati. Ad esempio:

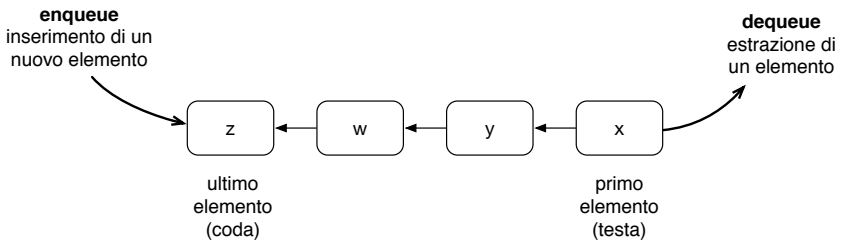

Figura 2: La struttura dati di coda (*queue*)

```
q = Queue()
```

- **enqueue(***item***)**: accoda un nuovo elemento, aggiungendolo in fondo alla struttura dati come ultimo elemento. L'elemento da aggiungere alla coda deve essere passato come argomento. Ad esempio:

```
q = Queue()
q.enqueue('gatto')
```

- **dequeue()**: restituisce l'elemento al primo posto nella coda e lo elimina dalla coda stessa. Non richiede alcun parametro e restituisce l'elemento rimosso dalla coda; con questo metodo la coda viene modificata e si riduce di un elemento. Ad esempio:

```
x = q.dequeue()
```

- **isEmpty()**: verifica se la coda è vuota, ossia priva di elementi; se la coda è vuota restituisce il valore booleano True, altrimenti, se la coda contiene almeno un elemento, restituisce False. Ad esempio:

```
if q.isEmpty() == True:
    print("La coda è vuota")
else:
    print("La coda contiene almeno un elemento")
```

- **size()**: restituisce il numero di elementi presenti nella coda; restituisce un numero intero e non richiede alcun parametro. Ad esempio:

```
n = q.size()
if n>0:
    print("La coda contiene", n, "elementi")
else:
    print("La coda è vuota")
```

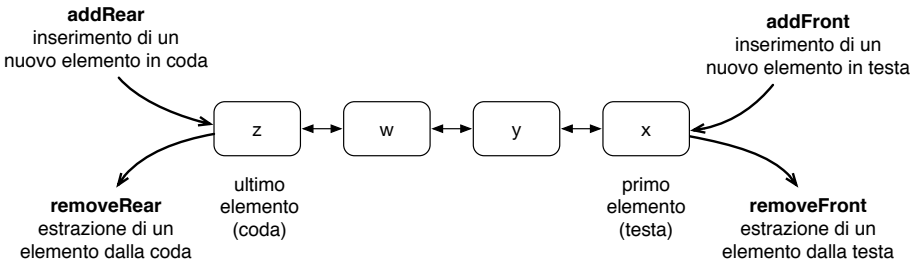

Figura 3: Una struttura dati di tipo *Deque* (*double-ended queue*)

3.3 La classe Deque

La struttura dati *deque* (abbreviazione di *double-ended queue*) è una coda in cui è possibile inserire ed estrarre elementi sia dall'inizio che dalla fine della struttura dati. Su un oggetto della classe `Deque` è possibile operare con i metodi descritti di seguito.

- **`Deque()`**: è il metodo costruttore della classe; crea un nuovo oggetto della classe `Deque` vuoto e restituisce un riferimento all'oggetto creato. Non richiede alcun parametro. Ad esempio:

```
d = Deque()
```

- **`addFront(`***item***`)`**: aggiunge all'inizio della struttura dati l'elemento specificato come argomento; non restituisce nulla. Ad esempio:

```
d = Deque()
d.addFront('Primo elemento della lista')
```

- **`addRear(`***item***`)`**: aggiunge alla fine della struttura dati l'elemento specificato come argomento, come ultimo elemento della lista; non restituisce nulla. Ad esempio:

```
d.addRear('Ultimo elemento della lista')
```

- **`removeFront()`**: restituisce il primo elemento della lista e lo elimina dalla struttura dati. Non richiede nessun argomento; con questo metodo la struttura dati viene modificata e si riduce di un elemento. Ad esempio:

```
x = d.removeFront()
```

- **`removeRear()`**: restituisce l'ultimo elemento della lista e lo elimina dalla struttura dati. Non richiede nessun argomento; con questo metodo la struttura dati viene modificata e si riduce di un elemento. Ad esempio:

```
x = d.removeRear()
```

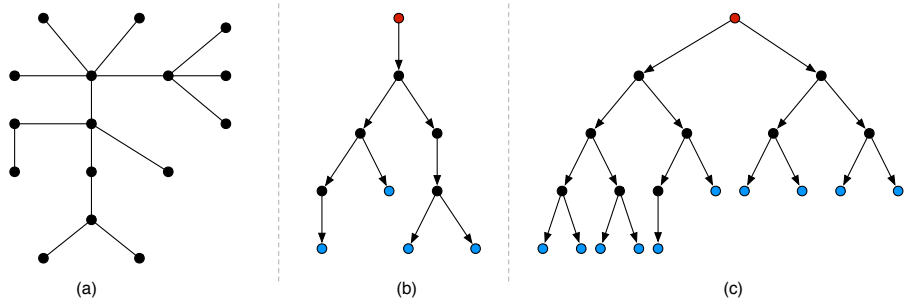

Figura 4: Un albero libero (a), un albero binario (b) e un albero binario completo (c)

- **isEmpty()**: verifica se la struttura dati è vuota e restituisce un valore booleano: restituisce True se la lista è vuota, altrimenti restituisce False. Non richiede alcun parametro. Ad esempio:

```
if d.isEmpty() == True:
    print("La deque è vuota")
else:
    print("La deque contiene almeno un elemento")
```

- **size()**: restituisce il numero di elementi presenti nella struttura dati; restituisce un numero intero e non richiede alcun parametro. Ad esempio:

```
n = d.size()
if n>0:
    print("La deque contiene", n, "elementi")
else:
    print("La deque è vuota")
```

4 Strutture dati per alberi binari

Un *albero* è un grafo connesso e aciclico. Un *albero radicato* è un albero con gli spigoli orientati in modo naturale da un vertice detto *radice* verso i vertici "terminali", detti *foglie*; in questo modo si crea una relazione "padre/figlio" tra i vertici del grafo: la radice è l'unico vertice del grafo privo di padre, mentre le foglie sono i vertici del grafo privi di figli. In un albero con radice i vertici del grafo possono essere disposti su livelli contenenti i vertici equidistanti dalla radice: la radice occupa il primo livello (livello 0), i figli della radice sono disposti sul livello 1, i figli dei figli sul livello 2 e così via. In un *albero binario* ogni vertice ha al massimo due figli. L'albero binario è *completo* se su ogni livello dell'albero, tranne al più l'ultimo, è presente il massimo numero di vertici.

Dal momento che l'albero binario, a partire dalla radice, si sviluppa su più livelli equidistanti dalla radice con un numero di vertici pari al massimo al doppio dei vertici del livello precedente, possiamo dire che, indicando con il livello 0 dell'albero quello in cui è presente la radice, con il livello 1 quello con i due figli della radice e

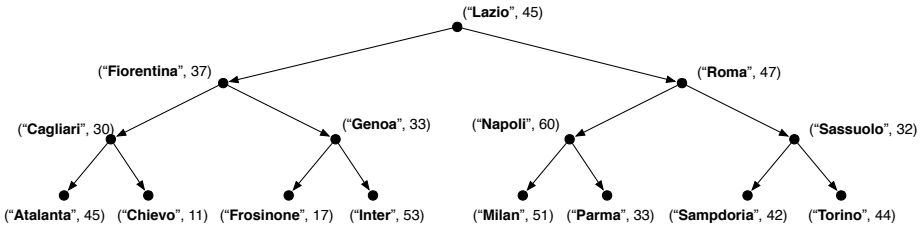

Figura 5: Un albero binario di ricerca con alcune squadre del campionato di calcio di serie A e i loro punteggi

così via, il livello $k > 0$ di un albero binario ha al massimo 2^k vertici. Ne consegue che, se h è la profondità di un albero binario completo con n vertici, allora $h \leq \log_2 n$.

In Figura 4 sono rappresentati un albero generico non orientato (albero *libero*), un albero binario e un albero binario completo (ogni livello ha il massimo numero di vertici tranne l'ultimo). Nei due alberi binari rappresentati in figura, il vertice colorato di rosso è la radice, mentre i vertici colorati di azzurro sono le foglie.

4.1 La classe BinarySearchTree

Un *albero binario di ricerca* (in inglese *binary search tree*) è un albero binario i cui vertici sono delle coppie «chiave/valore»; se v è un vertice dell'albero, indicheremo con $v.key$ e con $v.value$ rispettivamente la chiave e il valore corrispondenti; indichiamo inoltre con $left(v)$ e $right(v)$ rispettivamente il figlio sinistro e il figlio destro di v.

Le chiavi assegnate ai vertici dell'albero binario di ricerca determinano la dislocazione dei vertici nell'albero, in modo tale che se v è un vertice dell'albero, risulta $left(v).key \leq v.key < right(v).key$.

In uno stesso albero possono essere presenti più elementi con la stessa chiave, anche se associati a valori diversi.

In Figura 5 riportiamo la rappresentazione di un albero binario di ricerca: a ciascun vertice dell'albero è associata una coppia $(key, value)$ in cui la chiave è costituita dal nome di una squadra del campionato di calcio di serie A, mentre il valore è il punteggio in campionato. I vertici sono quindi collocati nell'albero in figura rispettando la relazione d'ordine basata sulle chiavi di ciascun vertice: se nella radice dell'albero c'è la "Lazio", la "Roma" si trova alla sua destra (la parola "Roma" è maggiore della parola "Lazio" in ordine alfabetico), mentre la "Fiorentina" si trova a sinistra della radice. La regola poi si ripete per ogni altro vertice del grafo e i suoi figli.

Questa struttura dati è utile perché permette di realizzare funzioni di ricerca di un elemento e di inserimento di un nuovo elemento molto efficienti: se l'albero binario di ricerca è "bilanciato" (l'ideale è un albero binario di ricerca completo), l'operazione di inserimento di un elemento e di ricerca di un elemento avvengono entrambe con un tempo dell'ordine dell'altezza dell'albero, che, nel migliore dei casi è pari a $\log_2 n$.

La classe `BinarySearchTree` mette a disposizione i seguenti metodi:

- **BinarySearchTree()**: è il metodo costrutture della classe; crea un nuovo oggetto vuoto della classe BinarySearchTree e restituisce un riferimento all'oggetto stesso. Ad esempio

```
a = BinarySearchTree()
```

- **put(*chiave*,*valore*)**: aggiunge un elemento alla struttura dati collocandolo nella posizione corretta nell'ambito dell'albero binario di ricerca; ciascun elemento è composto da una coppia: la *chiave* è l'attributo che determina la posizione nell'albero binario di ricerca, mentre il *valore* è l'informazione che si intende memorizzare nella struttura dati. Ad esempio:

```
a.put('Roma', 47)
```

- **get(*chiave*)**: restituisce il valore del primo elemento presente nell'albero identificato con la chiave specificata come argomento. L'elemento non viene rimosso dalla struttura dati. Se invece non esiste un elemento con tale chiave, il metodo restituisce `None`. Ad esempio:

```
puntiDellaRoma = a.get('Roma')
```

- **delete(*chiave*)**: elimina dalla struttura dati il primo elemento presente nell'albero identificato dalla chiave specificata come argomento. Se non esiste alcun elemento identificato dalla chiave specificata, il metodo produce un errore e blocca il programma. Ad esempio:

```
a.delete('Inter')
```

- **length()**: restituisce il numero di elementi presenti nell'albero binario di ricerca; se l'albero è vuoto, restituisce 0. Ad esempio:

```
n = a.length()
```

- **inorder()**: esegue una visita in profondità dell'albero binario a partire dalla radice; per ogni vertice v dell'albero, visualizza le chiavi dei nodi visitati nel seguente ordine:

  1. visualizza le chiavi del sotto-albero con radice in $left(v)$
  2. visualizza la chiave di v
  3. visualizza le chiavi del sotto-albero con radice in $right(v)$

  In questo modo le chiavi vengono visualizzate in ordine crescente. Ad esempio:

```
a.inorder()
```

- **postorder()**: esegue una visita dell'albero binario in "post-ordine" a partire dalla radice; per ogni vertice v dell'albero, visualizza le chiavi dei nodi visitati nel seguente ordine:

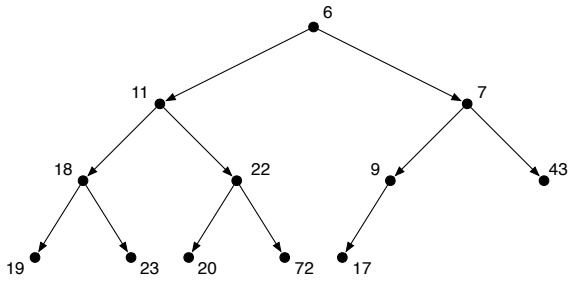

Figura 6: Un esempio di *heap binario*: ciascun vertice è minore o uguale dei suoi figli

1. visualizza le chiavi del sotto-albero con radice in $right(v)$
2. visualizza le chiavi del sotto-albero con radice in $left(v)$
3. visualizza la chiave di v

```
a.postorder()
```

- **in**: l'operatore restituisce True se una determinata chiave è presente nell'albero, altrimenti restituisce False. Ad esempio:

```
if 'Sampdoria' in a:
    print("La Sampdoria c'e' nell'albero")
else:
    print("La Sampdoria non c'e' nell'albero")
```

4.2 La classe BinHeap

Un *heap binario* è una struttura dati di albero binario completo, con la proprietà che il valore associato ad ogni vertice è minore o uguale al valore assegnato ai suoi figli. Naturalmente, se ogni vertice è minore o uguale dei suoi figli, allora sarà anche minore o uguale di tutti i suoi discendenti. In questo modo il valore minimo, fra tutti gli elementi inseriti nell'albero, si trova certamente sulla radice, mentre il valore massimo è certamente una delle foglie dell'albero. Non c'è nessuna relazione che lega fra loro i vertici "fratelli" di uno stesso vertice padre, se non il fatto che il valore di entrambi è maggiore o uguale a quello del padre.

In Figura 6 rappresentiamo un esempio di heap binario. L'elemento minimo (6) è sulla radice, mentre il massimo (72) è una delle foglie dell'albero. Siccome l'heap binario è un albero binario completo, allora, se l'albero contiene n vertici, la sua altezza è pari a $\log_2 n$. Per questo motivo, è possibile implementare le operazioni di inserimento di un elemento e di estrazione dell'elemento minimo con una complessità computazionale di $O(\log_2 n)$.

Gli heap binari vengono generalmente utilizzati come struttura dati efficiente per la gestione di *code di priorità*, ossia strutture dati di tipo coda, in cui però si tiene conto anche della priorità degli elementi: il primo elemento ad essere estratto non è necessariamente il primo ad entrare nella struttura dati, ma quello con la massima priorità (il valore minimo).

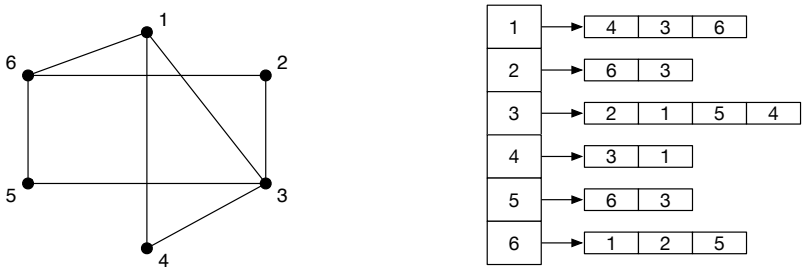

Figura 7: Un grafo $G = (V, E)$ e la corrispondente rappresentazione con liste di adiacenza

Il modulo pythonds ci fornisce la classe BinHeap per definire heap binari; la classe ci offrei i seguenti metodi per operare sugli oggetti:

- **BinHeap()**: è il metodo costruttore della classe; crea un nuovo oggetto vuoto della classe BinHeap e restituisce un riferimento all'oggetto stesso. Ad esempio:

```
a = BinHeap()
```

- **insert(*k*)**: inserisce l'elemento *k* nell'*heap*. Ad esempio:

```
a.insert(27)
```

- **delMin()**: restituisce l'elemento minimo presente nell'*heap* e lo elimina dalla struttura dati. Ad esempio:

```
x = a.delMin()
```

5 Strutture dati per grafi e alberi

La classe Graph definisce una struttura dati per un grafo, rappresentato con liste di adiacenza. Il grafo è definito mediante una collezione di *vertici*; ciascun vertice è associato ad una lista dei suoi vertici adiacenti, definendo in questo modo gli spigoli del grafo stesso. In Figura 7 riportiamo il disegno di un grafo $G = (V, E)$ con sei vertici e la corrispondente rappresentazione con le liste di adiacenza.

È possibile definire un grafo "pesato", assegnando un valore numerico (detto *peso* o *costo*) ad ogni spigolo del grafo.

La classe Graph consente di rappresentare *grafi orientati*, ossia grafi in cui gli spigoli hanno un verso: escono da un vertice u ed entrano in un altro vertice v. È possibile rappresentare anche *grafi non orientati*, aggiungendo oltre agli spigoli in un verso, anche gli spigoli nel verso opposto.

La classe Graph mette a disposizione i seguenti metodi per costruire ed operare su un oggetto di questo tipo:

- **Graph()**: è il metodo costruttore della classe, crea un oggetto Graph privo di vertici e di spigoli e restituisce un riferimento all'oggetto stesso. Ad esempio:

```
g = Graph()
```

- **addVertex(**v**)**: aggiunge il vertice identificato dal valore v al grafo. Alla struttura dati di grafo viene così aggiunto un oggetto della classe Vertex che è identificato dal valore v. Ad esempio:

```
g.addVertex(27)
```

- **addEdge(**u**,** v**)**: aggiunge lo spigolo (u, v) al grafo; lo spigolo è orientato: uscente dal vertice identificato dal valore u ed entrante nel vertice identificato dal valore v. Ad esempio:

```
g.addEdge(3,7)
```

- **addEdge(**u**,** v**,** w**)**: aggiunge lo spigolo (u, v) al grafo; lo spigolo è orientato da u a v ed ha un "peso" (o "costo") di w. Ad esempio:

```
g.addEdge(3, 7, 100)
```

- **getId()**: se il vertice v a cui si applica il metodo appartiene al grafo, restituisce l'identificativo del vertice. Ad esempio:

```
for v in g:
    print(v.getId)
```

- **getVertex(**x**)**: se presente, restituisce un riferimento all'oggetto Vertex con identificativo x. Ad esempio:

```
v = g.getVertex(1)
```

- **getVertices()**: restituisce la lista degli identificativi dei vertici del grafo. Ad esempio:

```
V = g.getVertices()
print("Vertici:", V)
```

- **in**: restituisce il valore booleano True se il vertice con l'identificativo specificato appartiene al grafo, altrimenti restituisce False. Ad esempio:

```
if v in g:
    print("Il vertice", v, "appartiene al grafo")
else:
    print("Il vertice", v, "non appartiene al grafo")
```

I vertici aggiunti al grafo sono tipicamente (ma non necessariamente) identificati da un numero naturale e sono oggetti della classe Vertex, che offre alcuni metodi utili per assegnare o rilevare alcuni attributi specifici dei vertici del grafo, utili per l'implementazione di alcuni algoritmi ben noti.

- **getColor()**: restituisce il "colore" (numero intero o stringa) assegnato al vertice del grafo con il metodo setColor(). Ad esempio:

```
x = v.getColor()
print("Il vertice", v.getId(), "ha il colore", x)
```

- **getConnections()**: restituisce la lista di adiacenza del vertice (la lista dei vertici adiacenti). Ad esempio:

```
Nv = v.getConnections()
print("Vertici adiacenti a", v.getId(), ":", Nv)
```

- **getDistance()**: restituisce il valore di una distanza assegnato al vertice con il metodo setDistance(). Ad esempio:

```
d = v.getDistance()
print("Distanza di", v.getId(), ":", d)
```

- **getPred()**: restituisce il vertice "predecessore" (padre) del vertice a cui si applica il metodo; questa informazione viene impostata con il metodo setPred() e può essere utile per tenere traccia di un cammino nella visita di un grafo o nella costruzione di un albero ricoprente. Se il vertice non ha un predecessore il metodo restituisce la costante None. Ad esempio:

```
u = v.getPred()
if u != None:
    print("Il predecessore di", v.getId(), "e'", u.getId())
else:
    print("Il vertice", v.getId(), "non ha un predecessore")
```

- **setColor(*colore*)**: assegna il colore specificato al vertice; il colore può essere indifferentemente un numero o una stringa di caratteri e rappresenta semplicemente un attributo informativo che caratterizza il vertice. Ad esempio:

```
v.setColor('rosso')
```

- **setDistance(d)**: assegna la distanza d al vertice; anche in questo caso si tratta di un attributo informativo che caratterizza il vertice, il cui significato dipende dal contesto in cui lo si utilizza. Ad esempio:

```
v.setDistance(1000)
```

- **setPred(u)**: assegna il vertice u come predecessore del vertice a cui si applica il metodo. Si tratta comunque di un attributo informativo che caratterizza il vertice, il cui significato dipende dal contesto in cui lo si utilizza. Ad esempio:

```
v.setPred(u)
```

6 Algoritmi su grafi

Riportiamo di seguito, a titolo di esempio e per agevolare la comprensione del modo in cui possono essere utilizzati gli oggetti e i metodi visti nelle pagine precedenti, alcuni algoritmi classici per compiere semplici operazioni su una struttura dati di grafo.

6.1 Generazione di grafi

6.1.1 Generazione di un grafo casuale

Un grafo casuale (*random graph*) $G = (V, E)$, può essere costruito specificando il numero n di vertici del grafo e la probabilità P con cui viene creato lo spigolo tra due vertici u e v del grafo. Naturalmente risulta $0 \leq P \leq 1$; se la probabilità è $P = 1$ otterremo il grafo completo, mentre se al contrario $P = 0$, si ottiene il grafo privo di spigoli.

Algoritmo 1 RANDOMGRAPH(n, P)

Input: Il numero di vertici del grafo G e la probabilità P con cui vengono aggiunti gli spigoli al grafo
Output: Un grafo con n vertici

1: **per** $v = 1, \ldots, n$ **ripeti**
2: aggiungi il vertice v al grafo G
3: **fine-ciclo**
4: **per** $u = 1, \ldots, n-1$ **ripeti**
5: **per** $v = u+1, \ldots, n$ **ripeti**
6: $x =$ random$(0, \ldots, 1)$
7: **se** $x > 0$ e $x \leq P$ **allora**
8: aggiungi lo spigolo (u, v) al grafo G
9: **fine-condizione**
10: **fine-ciclo**
11: **fine-ciclo**
12: **restituisci** G

L'Algoritmo 1 presenta lo pseudo codice di una funzione per la creazione di un grafo casuale. Di seguito riportiamo il programma Python che implementa la funzione definita nell'Algoritmo.

Con le istruzioni alle righe 1 e 2 vengono importate tutte le definizioni (classi e metodi) presenti nei moduli pythonds e random. Quest'ultimo modulo contiene le funzioni per la generazione di numeri casuali. In particolare la funzione random() genera un numero *floating point* pseudo-casuale compreso tra 0 e 1.

```
1 from pythonds import *
2 from random import *
3
4 def randomGraph(G, n, P):
5     for v in range(1,n+1):
6         G.addVertex(v)
7     for u in range(1,n):
```

```
8           for v in range(u+1,n+1):
9               x = random()
10              if x>0 and x<=P:
11                  G.addEdge(u,v)
12                  G.addEdge(v,u)
13      return
14
15  g = Graph()
16  n = int(input("Numero di vertici: "))
17  p = float(input("Probabilita' [0-1]: "))
18  randomGraph(g, n, p)
19  print("Vertici del grafo: ", g.getVertices())
20  print("Spigoli del grafo:")
21  for u in g:
22      for v in u.getConnections():
23          print("(%s,%s)" % (u.getId(), v.getId()))
```

Il programma inizia a riga 16 con l'istruzione con cui viene creato un nuovo oggetto g della classe Graph. Quindi, dopo aver acquisito in input il numero di vertici del grafo n e la probabilità p, viene richiamata la funzione randomGraph() definita da riga 4 a riga 14.

La funzione accetta come argomento il grafo (l'oggetto della classe Graph), il numero di vertici n e la probabilità p. Quindi, con le istruzioni alle righe 5 e 6 implementa le istruzioni 1 e 2 dell'Algoritmo, aggiungendo gli n vertici al grafo, identificati con i numeri $1, 2, \ldots, n$, con il metodo addVertex().

I due cicli for nidificati, alle righe 7 e 8 consentono di produrre tutte le coppie ordinate di vertici del grafo (con $u < v$); con l'istruzione a riga 9 viene generato un numero pseudo-casuale compreso tra 0 e 1; se tale numero è maggiore si zero e minore o uguale alla probabilità P, viene creato lo spigolo (u, v) nel grafo G. Nel programma vengono utilizzate due chiamate al metodo addEdge() per creare uno spigolo con entrambi i versi, in modo da definire un grafo non orientato.

Con l'istruzione a riga 19, al termine dell'esecuzione della funzione randomGraph(), viene visualizzata la lista di tutti i vertici del grafo, ottenuta con il metodo getVertices(). Quindi, per ogni vertice u del grafo (riga 21) viene calcolata la lista di adiacenza di u con il metodo getConnections() e, per ogni vertice v adiacente ad u, viene visualizzato lo spigolo, stampando l'identificativo di u e di v (riga 23).

6.1.2 Generazione di un grafo completo

La generazione di un grafo completo, non orientato, con n vertici è un'operazione molto semplice, che può essere facilmente ricavata dal programma precedente. Ne riportiamo di seguito una codifica.

```
1   from pythonds import *
2
3   def completeGraph(G, n):
4       for v in range(1,n+1):
5           G.addVertex(v)
6       for u in range(1,n):
7           for v in range(u+1,n+1):
```

```
8               G.addEdge(u,v)
9               G.addEdge(v,u)
10      return
11
12  g = Graph()
13  n = int(input("Numero di vertici: "))
14  completeGraph(g, n)
15  print("Vertici del grafo: ", g.getVertices())
16  print("Spigoli del grafo:")
17  for u in g:
18      for v in u.getConnections():
19          print("(%s,%s)" % (u.getId(), v.getId()))
```

La funzione completeGraph() aggiunge al grafo i vertici $\{1,2,\ldots,n\}$ con il ciclo alle riga 4–5; con i due cicli nidificati alle righe 6–9 e 7–9, al variare di u da 1 a $n-1$ e di v da $u+1$ a n, aggiunge al grafo gli spigoli (u,v) e (v,u).

6.1.3 Generazione di un grafo ciclico

Un grafo ciclico C_n con n vertici è formato dall'insieme di vertici $V(C_n) = \{1,2,\ldots,n\}$ e dall'insieme degli spigoli $E(C_n) = \{(1,2),(2,3),\ldots,(n-1,n),(n,1)\}$. Il programma per costruire un grafo di questo genere è riportato di seguito.

```
1   from pythonds import *
2
3   def cycleGraph(G, n):
4       for v in range(1,n+1):
5           G.addVertex(v)
6       for v in range(1,n):
7           G.addEdge(v, v+1)
8           G.addEdge(v+1, v)
9       G.addEdge(1, n)
10      G.addEdge(n, 1)
11      return
12
13  g = Graph()
14  n = int(input("Numero di vertici: "))
15  cycleGraph(g, n)
16  print("Vertici del grafo: ", g.getVertices())
17  print("Spigoli del grafo:")
18  for u in g:
19      for v in u.getConnections():
20          print("(%s,%s)" % (u.getId(), v.getId()))
```

Anche in questo caso con le istruzioni alle righe 7–8 e 9–10, vengono creati i due spigoli nei due versi opposti tra le coppie di vertici v e $v+1$, per costruire un grafo non orientato.

6.1.4 Generazione di un grafo bipartito completo

Un grafo $G = (V, E)$ con $|V(G)| = n$ vertici è bipartito completo è possibile determinare una partizione di V in due sottoinsiemi V_1 e V_2 tali che $V_1 \cap V_2 = \emptyset$ e $V_1 \cup V_2 = V$ e tale che per ogni spigolo $(u, v) \in E(G)$ risulta $u \in V_1$ e $v \in V_2$.

Costruire un grafo di questo genere è quindi molto semplice, una volta acquisiti in input $n_1 = |V_1|$ e $n_2 = |V_2|$. Di seguito riportiamo il codice del programma che costruisce il grafo bipartito completo K_{n_1,n_2} con i vertici $V_1 = \{1, 2, \ldots, n_1\}$ e $V_2 = \{n_1 + 1, n_1 + 2, \ldots, n_1 + n_2.\}$

```
from pythonds import *

def bipartiteCompleteGraph(G, n1, n2):
    for v in range(1, n1+n2+1):
        G.addVertex(v)
    for u in range(1, n1+1):
        for v in range(n1+1, n1+n2+1):
            G.addEdge(u, v)
            G.addEdge(v, u)
    return

g = Graph()
n1 = int(input("Numero di vertici di V1: "))
n2 = int(input("Numero di vertici di V2: "))
bipartiteCompleteGraph(g, n1, n2)
print("Vertici del grafo: ", g.getVertices())
print("Spigoli del grafo:")
for u in g:
    for v in u.getConnections():
        print("(%s,%s)" % (u.getId(), v.getId()))
```

6.2 Visita di un grafo

La visita di un grafo è un procedimento che consente di raggiungere ogni vertice del grafo, partendo da un vertice assegnato (la *sorgente* della visita), percorrendo al massimo una volta gli spigoli del grafo. La visita di un grafo produce un *albero di visita* che collega, utilizzando alcuni degli spigoli del grafo, tutti i vertici raggiunti dal procedimento di visita. Se il grafo è non orientato e connesso, l'albero di visita è un *albero ricoprente* (in inglese: *spanning tree*).

6.2.1 Visita in ampiezza

L'algoritmo BFS (*breadth first search*) rappresenta un famosissimo procedimento di visita in ampiezza di un grafo $G = (V, E)$. Nel procedimento di visita in ampiezza, a partire da un vertice $s \in V$ detto *sorgente* della visita, vengono raggiunti prima i vertici con distanza 1 da s, poi quelli con distanza 2 e così via; in generale i vertici con distanza k da s vengono visitati solo dopo che sono stati visitati tutti i vertici di distanza minore di k. La distanza tra due vertici su un grafo è la lunghezza del cammino più breve che li unisce. L'albero di visita prodotto dall'algoritmo BFS è

un albero con radice in s composto dai cammini minimi per raggiungere ciascun vertice di G a partire da s.

L'Algoritmo 2 riporta una pseudo-codifica dell'algoritmo BFS per la visita in ampiezza di un grafo G a partire dal vertice s.

Algoritmo 2 BFS($G = (V, E)$, s)

Input: Un grafo $G = (V, E)$ e un vertice $s \in V(G)$
Output: L'albero di visita T del grafo, la distanze $d(v)$ di ogni vertice dalla radice.

```
1: per ogni v ∈ V(G) ripeti
2:     d(v) = ∞, colora v di bianco
3: fine-ciclo
4: Q = {s}, T = ({s}, ∅), d(s) = 0
5: colora s di grigio
6: fintanto che Q ≠ ∅ ripeti
7:     estrai un elemento u dalla coda Q
8:     per ogni v ∈ N(u) ripeti
9:         se v non è colorato allora
10:            colora v di grigio
11:            aggiungi v alla coda Q
12:            aggiungi il vertice v e lo spigolo (u, v) all'albero T
13:            d(v) = d(u) + 1
14:        fine-condizione
15:    fine-ciclo
16:    colora u di nero
17: fine-ciclo
```

L'algoritmo colora i vertici del grafo per tenere traccia del loro stato di visita e, nel caso in cui il grafo contenga dei cicli, per evitare di visitare più volte lo stesso vertice. I vertici non ancora visitati sono colorati di bianco, i vertici incontrati per la prima volta nel procedimento di visita sono colorati di grigio e, infine, i vertici visitati, di cui siano stati visitati anche tutti i vertici adiacenti, sono colorati di nero.

Per tenere traccia dell'ordine con cui sono stati incontrati i vertici nel corso della visita, per poi prenderli in esame nello stesso ordine e visitarne i vertici adiacenti, viene utilizzata una coda Q. Questo fatto è cruciale per il corretto funzionamento dell'algoritmo di visita in ampiezza, dal momento che abbiamo detto che intendiamo visitare i vertici di distanza k dalla sorgente, solo dopo aver completato la visita dei vertici di distanza inferiore a k.

Nel corso della visita viene calcolato l'albero di visita T, a cui man mano vengono aggiunti i vertici visitati e gli spigoli attraverso cui sono stati raggiunti tali vertici, e viene anche calcolata la distanza $d(v)$ di ciascun vertice v dalla sorgente s; si pone inizialmente $d(v) = \infty$ per ogni vertice (riga 2 dell'Algoritmo 2) e $d(s) = 0$ per la sorgente della visita (riga 4 dell'Algoritmo).

Riportiamo di seguito la codifica dell'algoritmo in linguaggio Python; per la gestione della coda Q viene utilizzata la classe `Queue`, mentre vengono utilizzate le proprietà degli oggetti della classe `Vertex` per memorizzare il colore dei vertici (metodi `setColour()` e `getColour()`) e la loro distanza dalla sorgente della visita (metodi `setDistance()` e `getDistance()`). Per brevità, nel programma seguente non viene costruito esplicitamente l'albero di visita T.

```
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):
        for v in range(u+1,n+1):
            x = random()
            if x<=P:
                G.addEdge(u,v)
                G.addEdge(v,u)
    return

def BFS(G, s):
    n = len(G.getVertices())
    for v in G:
        v.setColor("bianco")
    Q = Queue()
    Q.enqueue(s)
    s.setDistance(0)
    s.setColor("grigio")
    while Q.isEmpty() == False:
        u = Q.dequeue()
        for v in u.getConnections():
            if v.getColor() == "bianco":
                v.setColor("grigio")
                Q.enqueue(v)
                v.setDistance(u.getDistance()+1)
        u.setColor("nero")
    return

g = Graph()
n = int(input("Numero di vertici: "))
p = float(input("Probabilita': "))
randomGraph(g, n, p)
print("Vertici del grafo: ", g.getVertices())
print("Spigoli del grafo:")
for u in g:
    for v in u.getConnections():
        print("(%s,%s)" % (u.getId(), v.getId()))
x = int(input("Sorgente della visita: "))
BFS(g,g.getVertex(x))
for v in g:
    print("d(", v.getId(), ") = ", v.getDistance())
```

6.2.2 Visita in profondità

La visita in profondità di un grafo $G = (V, E)$ è un procedimento di visita che cerca di allontanarsi più rapidamente possibile dalla sorgente della visita. Riportiamo nell'Algoritmo 3 la pseudo-codifica dell'algoritmo DFS (*depth first search*) per la visita in profondità di un grafo. La visita vera e propria è demandata alla funzione ricorsiva VISITA.

Algoritmo 3 DFS(G)

Input: Il grafo G ed una sorgente $s \in V(G)$
Output: La sequenza di vertici visitati su G a partire da s

```
1: per ogni u ∈ V(G) ripeti
2:     colora u di bianco: c(u) = 0
3: fine-ciclo
4: per ogni u ∈ V(G) ripeti
5:     se u non è marcato allora
6:         VISITA(G, u)
7:     fine-condizione
8: fine-ciclo
```

VISITA(G, u)

```
1: colora u di grigio: c(u) = 1
2: per ogni v ∈ N(u) ripeti
3:     se v non è marcato allora
4:         VISITA(G, v)
5:     fine-condizione
6: fine-ciclo
7: colora u di nero: c(u) = 2
```

Riportiamo di seguito la codifica in Python dell'algoritmo DFS. Nel programma viene generato un grafo casuale mediante la funzione `randomGraph()` e poi viene avviata la visita in profondità del grafo a partire dal primo vertice di colore bianco. Grazie al ciclo alle righe 26–28, al termine della visita a partire da un determinato vertice, se sono ancora presenti altri vertici bianchi, la visita ricomincia dal vertice bianco successivo, fino ad aver esaurito i vertici bianchi nel grafo.

Nel corso della visita in profondità implementata nel programma, viene costruito l'albero orientato di visita, composto dall'unione dei cammini con radice nel vertice da cui la visita ha avuto inizio. Se però il grafo è non connesso o se l'orientazione degli spigoli non permette di visitare tutti i vertici del grafo a partire dal primo, allora l'algoritmo produce una foresta di alberi ricoprenti. Ciascuna radice di uno degli alberi che compongono la foresta ricoprente del grafo, è costituita dai vertici privi di predecessore.

Il predecessore di ciascun vertice nell'albero di visita, viene impostato attraverso il metodo `setPred()` nell'istruzione a riga 27 del programma Python. La stessa informazione viene poi visualizzata in output con le istruzioni a riga 43, 46 e 47, utilizzando il metodo `getPred()`; tale metodo, se il vertice a cui si applica non ha predecessori, restituisce la costante `None`.

Come è noto questo algoritmo, a differenza dell'algoritmo BFS per la visita in ampiezza del grafo, non consente di calcolare la lunghezza dei cammini minimi per raggiungere un determinato vertice a partire dalla sorgente della visita. Tuttavia so-

no comunque numerosissime le applicazioni di questo algoritmo, a partire dal calcolo del “tempo” di inizio e di fine visita di ciascun vertice del grafo, da cui si può anche ottenere un ordinamento topologico dei vertici, se il grafo è privo di cicli.

```
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):
        for v in range(u+1,n+1):
            x = random()
            if x<=P:
                G.addEdge(u,v)
                G.addEdge(v,u)
    return

def DFS(G):
    for v in G:
        v.setColor("bianco")
    for v in G:
        if v.getColor() == 'bianco':
            visita(G, v)
    return

def visita(G,u):
    u.setColor('grigio')
    for v in u.getConnections():
        if v.getColor() == "bianco":
            v.setPred(u)
            visita(G, v)
    u.setColor('nero')
    return

g = Graph()
n = int(input("Numero di vertici: "))
p = float(input("Probabilita': "))
randomGraph(g, n, p)
print("Vertici del grafo: ", g.getVertices())
print("Spigoli del grafo:")
for u in g:
    for v in u.getConnections():
        print("(%s,%s)" % (u.getId(), v.getId()))
DFS(g)
for v in g:
    if v.getPred() == None:
        x='-'
    else:
```

```
46          x = v.getPred().getId()
47      print("padre(", v.getId(), ") = ", x)
```

Riferimenti bibliografici

[1] Marco Beri, *Python 3*, ed. Apogeo, Milano, 2010.

[2] Marco Buttu, *Programmare con Python. Guida Completa*, ed. LSWR, 2014.

[3] Mark Lutz, *Imparare Python*, ed. O'Reilly – Tecniche Nuove, 2011.

Di seguito sono riportati i riferimenti di alcune risorse disponibili gratuitamente in rete:

[4] Allen Downey, *Pensare in Python. Come pensare da informatico*, Green Tea Press, 2015, `https://github.com/AllenDowney/ThinkPythonItalian/blob/master/thinkpython_italian.pdf`

[5] Marco Liverani, *Breve introduzione al linguaggio Python*, una guida molto sintetica in italiano per programmatori che non conoscono il Python `http://www.mat.uniroma3.it/users/liverani/doc/pythonIntro.pdf`

[6] Brad Miller, David Ranum, *Problem Solving with Algorithms and Data Structures using Python*, Franklin Beedle Publishers, `http://interactivepython.org/runestone/static/pythonds/index.html`

[7] *MatPlotLib*, libreria per la visualizzazione di grafici 2D con i moduli `pylab` e `pyplot`: `https://matplotlib.org`

[8] *NumPy* è una libreria per il calcolo scientifico in Python, che offre un ampio insieme di strutture dati e funzioni matematiche: `http://www.numpy.org`

[9] John Zelle, *graphics.py*, una libreria estremamente semplice e potente per la visualizzazione di componenti grafiche: `https://mcsp.wartburg.edu/zelle/python/graphics/graphics.pdf`