

Algoritmi avanzati e modelli computazionali (modulo A)

Giacomo Fiumara
gfiumara@unime.it

2023-2024

Descrizione del corso (in generale)

- Nozioni di base su Python
- Teoria dei grafi
- Scienza delle reti

Schema del corso

Nozioni di base su Python: alcuni dettagli

- Introduzione alla programmazione Python
- Funzioni
- Introduzione alla programmazione orientata agli oggetti

Schema del corso

Teoria dei grafi: alcuni dettagli

- Introduzione e definizioni fondamentali
- Strutture dati
- Esplorazione del grafico
- Percorsi più brevi
- Alberi di copertura minimi

Schema del corso

Scienza delle reti: alcuni dettagli

- Introduzione e definizioni fondamentali
- Modelli di reti casuali
- Reti reali
- Modelli dinamici
- Comunità
- Diffusione delle epidemie

Teoria dei grafi

- Le connessioni a coppie tra gli elementi svolgono un ruolo fondamentale in un vasto numero di applicazioni computazionali
- La relazione implicita in queste connessioni porta ad alcune domande naturali:
 - C'è un modo per collegare un elemento all'altro?
 - Quanti altri elementi sono collegati a un determinato elemento?
 - Qual è la catena più breve di connessioni tra questo oggetto e quest'altro oggetto?

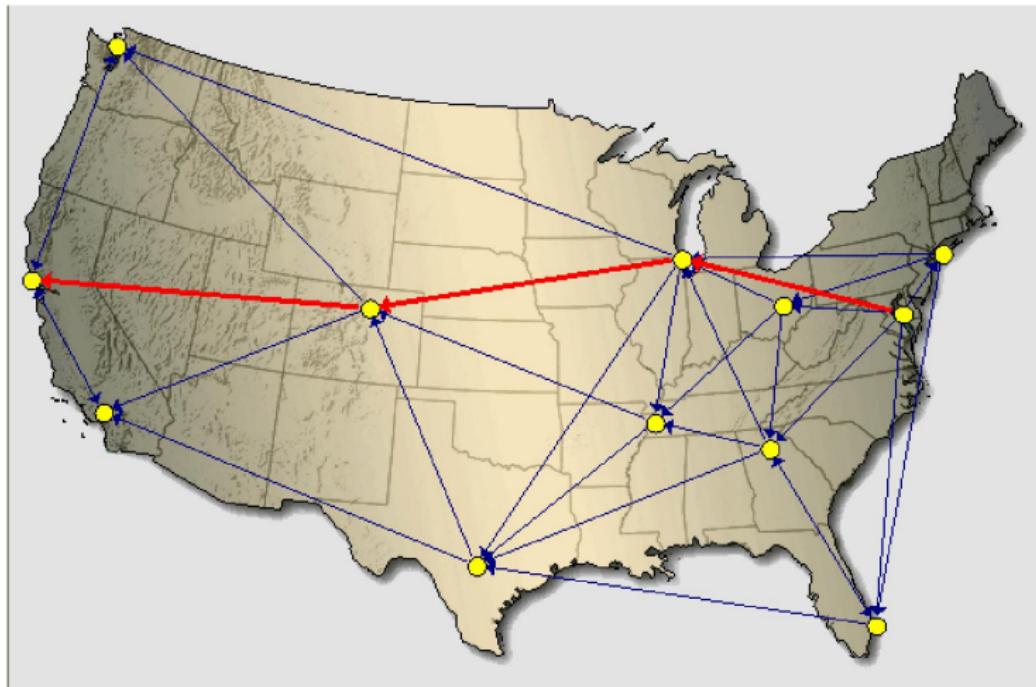
Teoria dei grafi

Un esempio: le mappe

- Una persona che sta pianificando un viaggio potrebbe dover rispondere a domande come “Qual è il percorso più breve da A a B?”
- Un viaggiatore esperto che ha sperimentato rallentamenti dovuti al traffico sul percorso più breve potrebbe porre la domanda “Qual è il modo più veloce da A a B?”

Teoria dei grafi

Un esempio: mappe (percorso più breve)



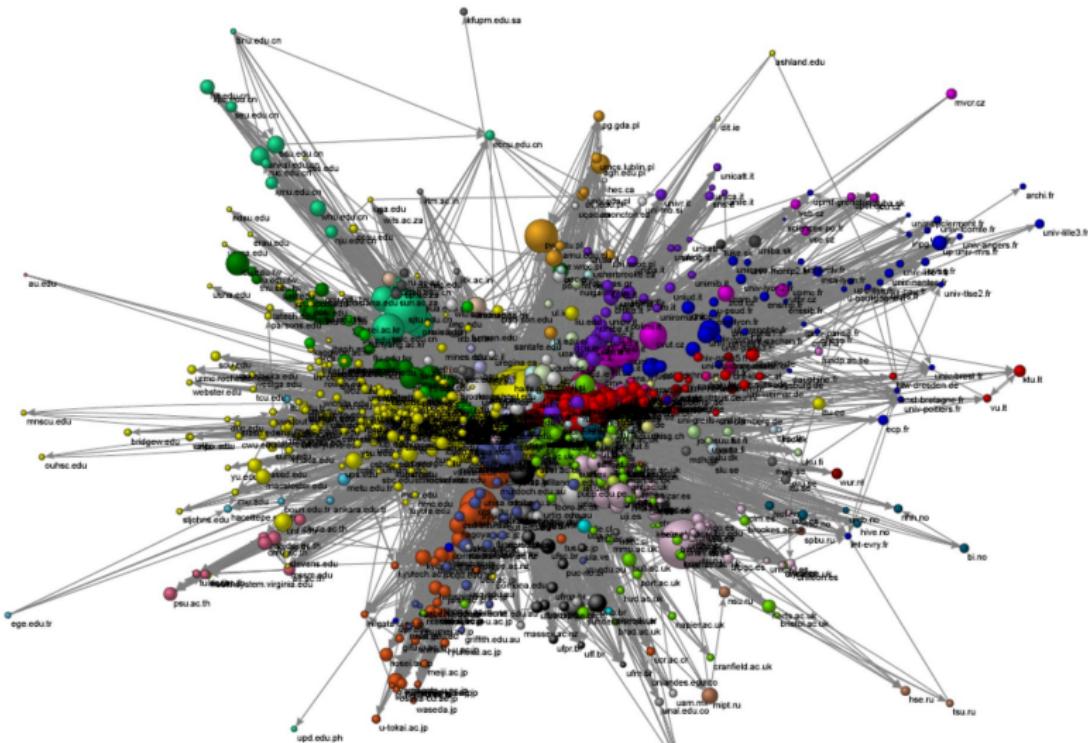
Teoria dei grafi

Un esempio: i contenuti web

- Quando si naviga sul Web, è frequente imbattersi in pagine che contengono collegamenti ad altre pagine ed è frequente spostarsi da una pagina all'altra facendo clic sui collegamenti
- L'intero web è un grafico, dove gli elementi sono pagine e le connessioni sono collegamenti

Teoria dei grafi

Un esempio: Contenuti Web (Collegamenti tra siti web accademici)



Teoria dei grafi

Un esempio: i circuiti

- Un circuito elettrico è composto da elementi come transistor, resistori e condensatori collegati insieme
- È necessario rispondere a domande del tipo "È presente un cortocircuito?" oppure "È possibile disporre questo circuito su un chip senza incrociare i fili?"
- La prima domanda dipende dalle proprietà delle connessioni (fili)
- La risposta alla seconda domanda richiede informazioni dettagliate sui cavi, sui dispositivi che tali cavi collegano e sui vincoli fisici del chip

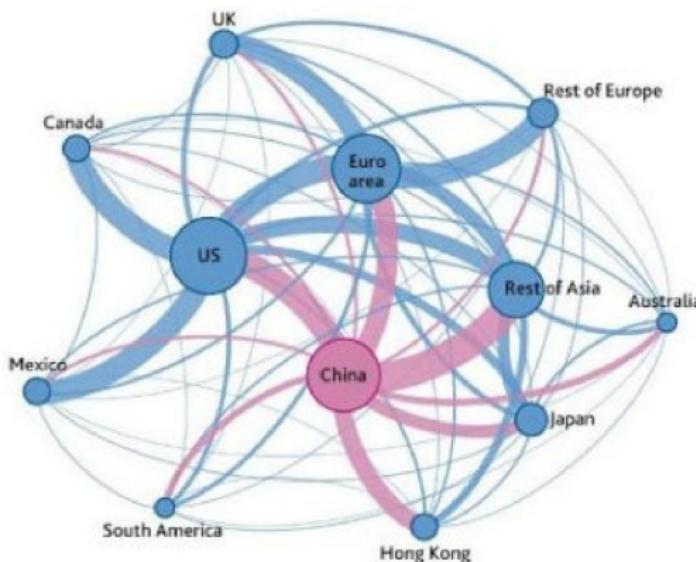
Teoria dei grafi

Un esempio: il commercio

- I rivenditori e le istituzioni finanziarie acquistano/vendono in un mercato
- In questa situazione una connessione rappresenta il trasferimento di contanti e beni tra un'istituzione e un cliente
- La conoscenza della natura della struttura di connessione in questo caso può migliorare la comprensione della natura del mercato

Un esempio: il commercio

Rete del commercio globale di beni, 2018



Un esempio: i social network

- Quando un utente utilizza un social network, crea connessioni esplicite con i suoi amici
- Gli oggetti corrispondono alle persone; le connessioni sono verso amici o follower
- Comprendere le proprietà di queste reti è un'applicazione di grande interesse non solo per le aziende che supportano tali reti, ma anche in politica, diplomazia, intrattenimento, istruzione, marketing e molti altri settori.

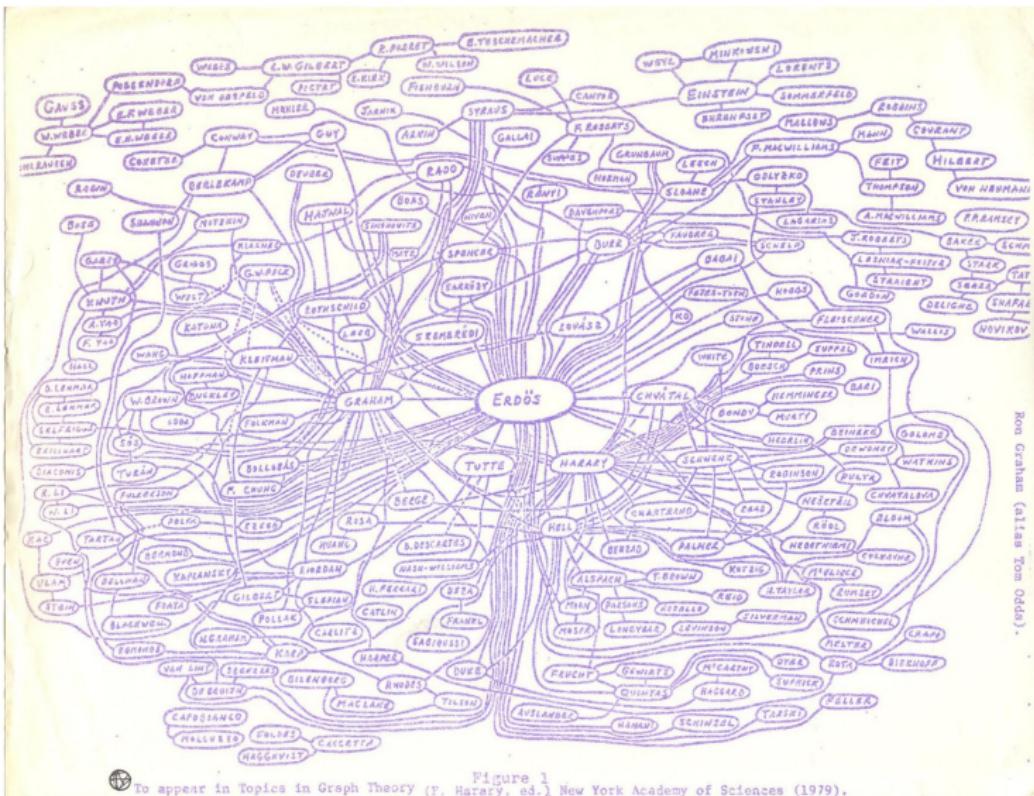
Un esempio: i social network

Distribuzione spaziale di Facebook (dal 2010)



Un esempio: i social network

Grafico delle relazioni scientifiche con Erdos



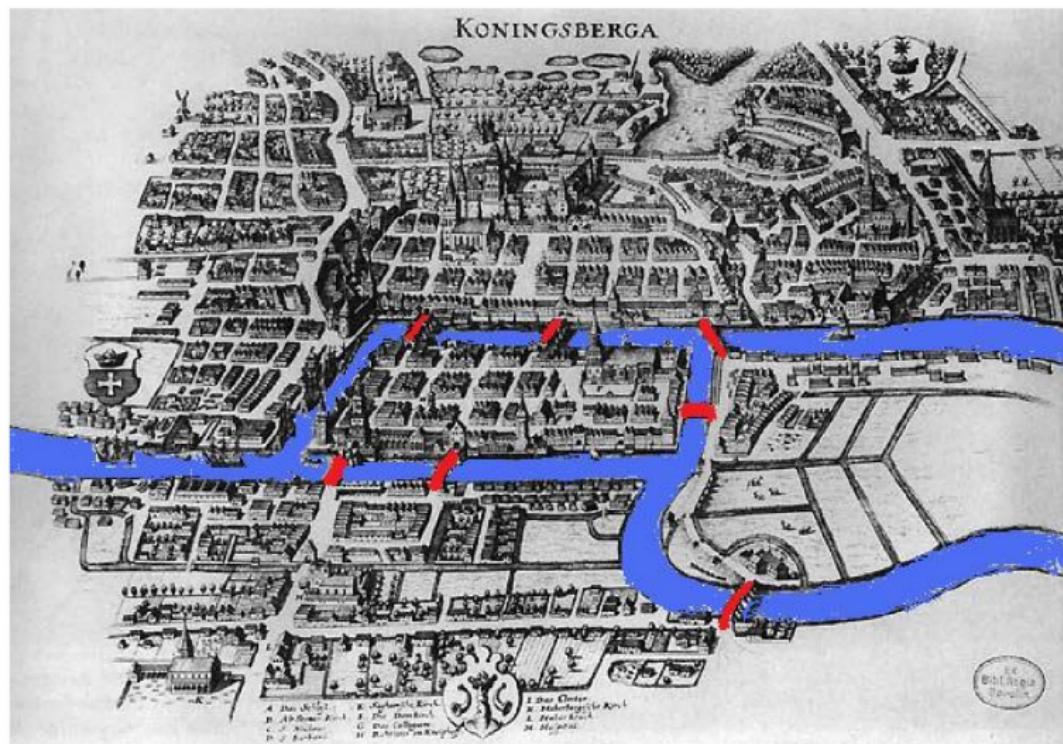
Teoria dei grafi

Il primo problema della teoria dei grafi

- Cittadini di Königsberg (ora Kaliningrad, Russia, allora in Prussia) la domenica passeggiava per le vie della città e attraversava i ponti sul fiume Pregel (e i suoi affluenti)
- Secondo la leggenda chiedevano un sentiero chiuso che attraversasse una ed una sola volta ogni ponte
- Il problema fu sottoposto al famoso matematico Leonardo Eulero

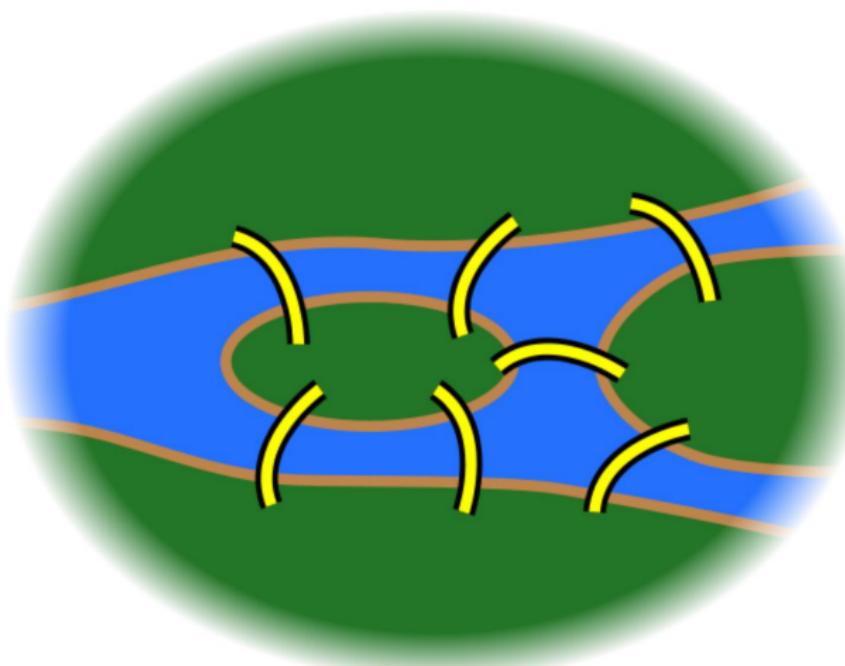
Teoria dei grafi

Il primo problema della teoria dei grafi



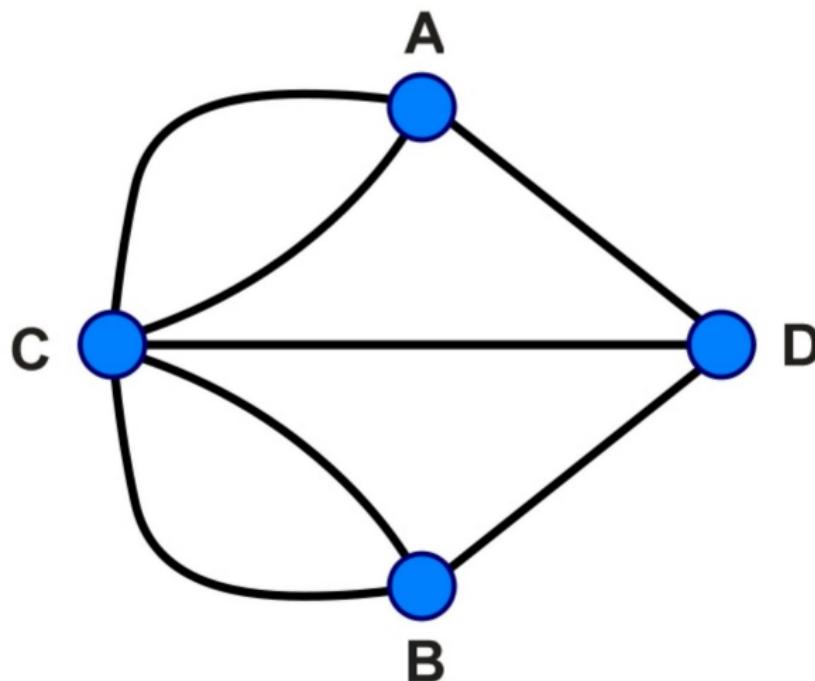
Teoria dei grafi

Il primo problema della teoria dei grafi



Teoria dei grafi

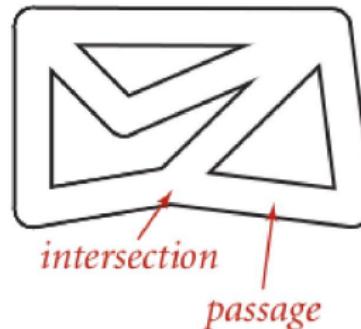
Il primo problema della teoria dei grafi



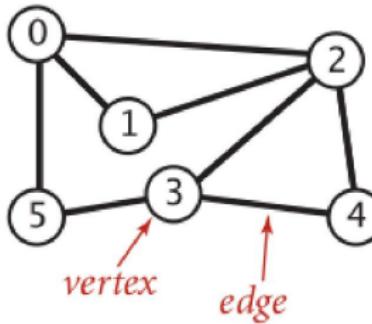
Teoria dei grafi

L'astrazione dalle rotte al grafico

maze



graph



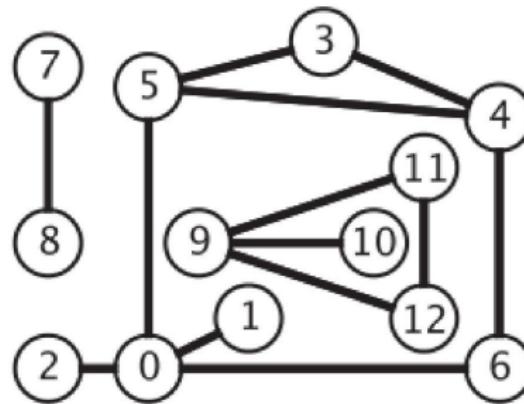
Teoria dei grafi

Tipi di modelli grafici

- Grafici non orientati
- Grafi diretti (o digrafi)
- Grafici ponderati per i bordi
- Grafici diretti ponderati per i bordi

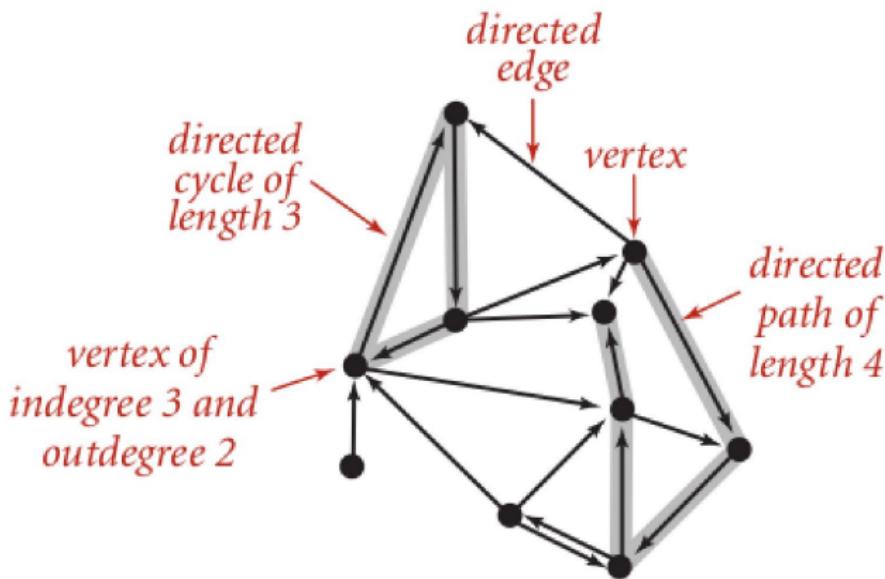
Teoria dei grafi

Un esempio di grafo non orientato



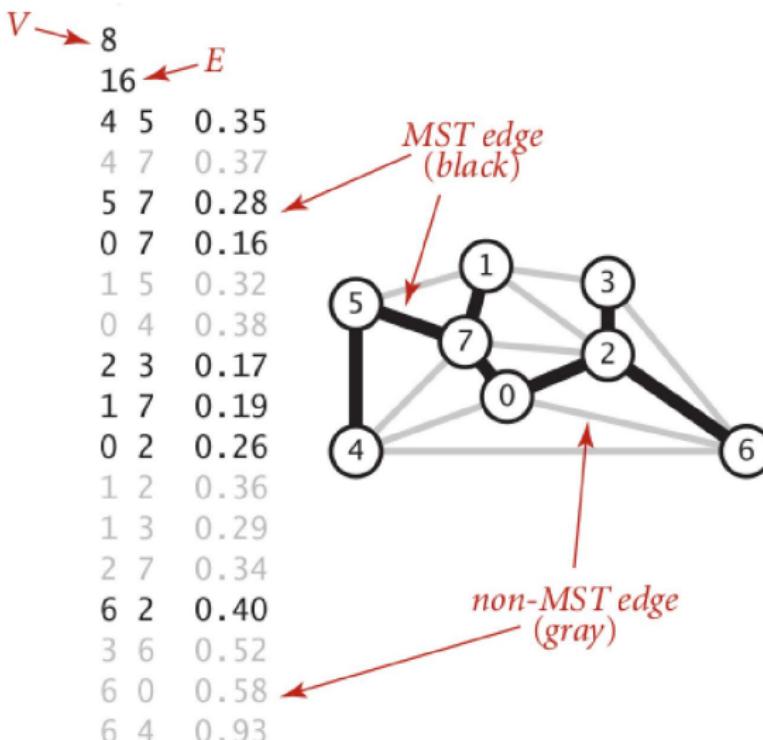
Teoria dei grafi

Un esempio di grafo diretto



Teoria dei grafi

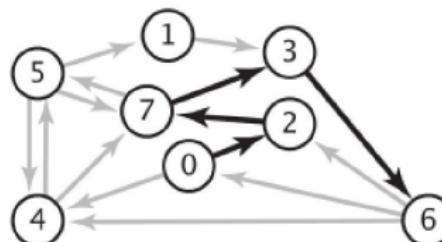
Un esempio di grafo non orientato con ponderazione sugli archi



Teoria dei grafi

Un esempio di grafico diretto con ponderazione sugli spigoli

4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



shortest path from 0 to 6

0->2	0.26
2->7	0.34
7->3	0.39
3->6	0.52

Riferimenti

- Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, strutture dati e algoritmi in Python
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduzione agli algoritmi, terza edizione, MIT Press
- Albert László Barabási, Scienza delle reti
- Letture aggiuntive

Istruzioni per l'esame finale (solo Modulo A)

- Frequentia regolarmente e attivamente le lezioni
- 3 incarichi (10% ciascuno)
- Progetto finale (40%)
- Presentazione del progetto (30%)

Voto finale:

$$G = \frac{GA + GB}{2}$$

Progetto finale (preliminari)

Preparare una presentazione (esattamente 5 diapositive)

- Quali sono i tuoi nodi e collegamenti
- Come raccoglierai i dati
- Dimensione prevista della rete (numero di nodi, numero di collegamenti)
- Quali domande intendi porre (puoi cambiarle man mano che la lezione prosegue)
- Perché questa rete è importante

Utensili

- I tuoi appunti e i tuoi libri
- Il tuo laptop/desktop
- Python3
- Anaconda3 (un ricco set di moduli Python)
- NetworkX (un modulo per le reti)
- In alternativa: utilizza Google Colab (non è necessaria alcuna installazione)

Primer di Python

Primer di Python

- Il linguaggio di programmazione Python è stato sviluppato da Guido von Rossum all'inizio degli anni '90
- Ora è diventata una lingua ampiamente utilizzata nell'industria e nel mondo accademico
- La seconda versione principale, Python 2, è stata rilasciata nel 2000
- La terza versione principale, Python 3, è stata rilasciata nel 2008
- Sono entrambi mantenuti e disponibili gratuitamente su www.python.org

L'interprete Python

- Python è formalmente un linguaggio interpretato
- Ciò significa che i comandi vengono eseguiti tramite un software chiamato interprete che riceve un comando, lo valuta e riporta il risultato del comando
- L'interprete può essere utilizzato in modo interattivo (ad esempio, per scopi di debug), mentre un programmatore in genere definisce in anticipo una serie di comandi e li salva in un file di testo semplice (file sorgente)
- I file sorgente Python sono convenzionalmente archiviati in un file denominato con il suffisso .py
- L'interprete Python viene richiamato digitando python dalla riga di comando

Oggetti in Python

- Python è un linguaggio orientato agli oggetti
- Le classi costituiscono la base per tutti i tipi di dati
- Nel modello a oggetti di Python, alcune classi integrate sono la classe **int** per gli interi, la **float** per i valori in virgola mobile e la **str** per le stringhe di caratteri

La dichiarazione di assegnazione

- Il più importante di tutti i comandi Python è l'istruzione di assegnazione
- Per esempio:

temperatura = 98,6

- Questo comando stabilisce la temperatura come identificatore e poi la associa ad un oggetto espresso a destra del segno uguale, in questo caso un valore in virgola mobile con valore 98,6.

Identifieri

- Gli identifieri in Python fanno distinzione tra maiuscole e minuscole
- Può essere composto da quasi tutte le combinazioni di lettere, numeri e caratteri di sottolineatura
- Un identificatore non può iniziare con un numero
- Ci sono 33 parole appositamente riservate:

Parole riservate

Falso	Nessuno	Vero	e	come	affermare
interruzione	della	lezione	continuare	def	elif
altro	tranne	infine	per	da	globale
Se	l'importazione		È	lambda	non locale
non	O	passaggio	aumentare	il rendimento	Tentativo
mentre	con		cedere		

Identifieri

- Ogni identificatore è implicitamente associato all'indirizzo di memoria dell'oggetto a cui si riferisce
- Python è un linguaggio tipizzato dinamicamente: non esiste una dichiarazione anticipata che associa un identificatore a un particolare tipo di dati
- Attenzione: sebbene un identificatore non abbia un tipo dichiarato, l'oggetto a cui si riferisce ha un tipo definito

Identifieri

- Un programmatore può stabilire un alias assegnando un secondo identificatore a un oggetto esistente
- Per esempio:

originale = temperatura

- Una volta stabilito un alias, è possibile utilizzare entrambi i nomi per accedere all'oggetto sottostante
- Se l'oggetto supporta azioni che influiscono sul suo stato, le modifiche apportate tramite un alias saranno evidenti quando si utilizza l'altro alias
- Tuttavia, se uno dei nomi viene riassegnato a un nuovo valore utilizzando una successiva istruzione di assegnazione, ciò non influisce sull'oggetto con alias (l'alias viene interrotto).

Identifieri

- Per esempio:

temperatura = temperatura + 5,0

- Il risultato dell'espressione a destra viene memorizzato come una nuova istanza in virgola mobile e il nome a sinistra (temperatura) viene riassegnato al risultato
- La conseguenza è che questo comando non ha alcun effetto sul valore dell'istanza float esistente a cui l'identificatore originale continua a fare riferimento

Identifieri

```
In [1]: temperature = 98.6
```

```
In [2]: original = temperature
```

```
In [3]: temperature = temperature + 5.0
```

```
In [4]: temperature
```

```
Out[4]: 103.6
```

```
In [5]: original
```

```
Out[5]: 98.6
```

Istanziazione

- Il processo di creazione di una nuova istanza di una classe è noto come istanziamento
- La sintassi per istanziare un oggetto è invocare il costruttore della classe
- Molte delle classi integrate di Python supportano quella che è conosciuta come forma letterale per designare nuove istanze
- Ad esempio, l'incarico:

temperatura = 98,6

risulta nella creazione di una nuova istanza della classe float, il termine 98.6 è la forma letterale

Metodi di chiamata

- Python supporta le funzioni tradizionali che vengono invocate con una sintassi come

ordinati (dati)

- in cui data è un parametro inviato alla funzione
- Le classi di Python possono anche definire uno o più metodi (noti anche come funzioni membro) che vengono invocati su un'istanza specifica di una classe utilizzando l'operatore punto
- Per esempio:

dati.sort()

Metodi di chiamata

- Alcuni metodi restituiscono informazioni sullo stato di un oggetto, ma non modificano tale stato
- Si chiamano accessori
- Per esempio:

`dati.iniziacon('y')`

- Altri metodi, come il metodo `sort` della classe `list`, modificano lo stato dell'oggetto
- Si chiamano mutatori

Le classi integrate di Python

- Una classe è immutabile se ogni oggetto di quella classe ha un valore fisso valore al momento dell'istanziazione che non può essere successivamente modificato
- Ad esempio, la classe float è immutabile, poiché once an l'istanza è stata creata, il suo valore non può essere modificato

Classe	Descrizione	Immutabile
bool	Valore booleano	sì
int	intero	sì
galleggiante	numero in virgola mobile	sì
elenco	sequenza mutabile di oggetti	N
tupla	sequenza immutabile di oggetti stringa	sì
stra	di caratteri insieme	sì
impostato	non ordinato di oggetti distinti frozenset	N
forma immutabile della classe insieme		sì
dict	mappatura associativa (aka dizionario)	N

La classe bool

- La classe bool viene utilizzata per manipolare i valori booleani
 - Le uniche due istanze della classe sono espresse come valori letterali Falso e Vero
 - Il costruttore predefinito, `bool()` restituisce False
-

```
goofy = bool() pluto  
= Falso
```

- Python consente la creazione di un valore booleano da un tipo non booleano utilizzando la sintassi `bool(huey)` per il valore huey
- L'interpretazione dipende dal tipo di parametro
 - I numeri restituiscono False se zero e True se diverso da zero
 - Le stringhe e le liste vengono valutate False se vuote e True se non vuote

La classe int

- La classe int è progettata per rappresentare valori interi con grandezza arbitraria
- A differenza di Java e C++ che supportano diversi tipi interi con diversa precisione, Python sceglie automaticamente la rappresentazione interna di un intero in base alla grandezza del suo valore
- In alcuni contesti può essere utile esprimere un valore intero utilizzando il formato binario, ottale o esadecimale. Per esempio:

0b110101

0o755

0x9ab

La classe int

```
In [6]: a = 0b1101
```

```
In [7]: a
```

```
Out[7]: 13
```

```
In [8]: b = 0o755
```

```
In [9]: b
```

```
Out[9]: 493
```

```
In [10]: c = 0xff
```

```
In [11]: c
```

```
Out[11]: 255
```

La classe int

- Il costruttore int() restituisce il valore 0 per impostazione predefinita
- Può però essere utilizzato solo per costruire un valore intero a partire da un valore esistente di altro tipo
- Per esempio

```
In [12]: int(goofy)
```

```
...
NameError
st)
<ipython-input-12-798d88bb7518> in <module>()
    ---> 1 int(goofy)

NameError: name 'goofy' is not defined
```

Traceback (most recent call last)

```
In [13]: goofy = 12.4
```

```
In [14]: int(goofy)
```

```
Out[14]: 12
```

```
In [15]: int()
```

```
Out[15]: 0
```

La classe int

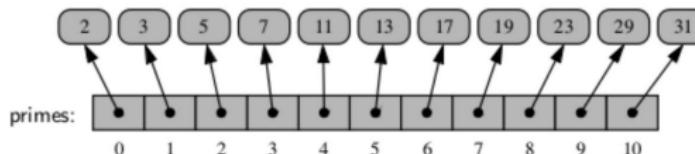
- Se `goofy` rappresenta un valore in virgola mobile, `int(goofy)` produce il valore troncato di `goofy`
- Il costruttore può essere utilizzato per analizzare una stringa che si presume rappresenti un valore intero: `int('150')` produce il valore intero 150
- Per impostazione predefinita, la stringa deve utilizzare la base 10
- Se si desidera la conversione da un'altra base, quella base può essere indicata come secondo parametro opzionale, ad esempio: `int('7f', 16)` restituisce l'intero 127

La classe float

- La classe float è l'unico tipo a virgola mobile in Python con precisione fissa
- Le forme letterali sono 98.6 e 10e3
- La forma del costruttore di float() restituisce 0.0. Quando viene fornito un parametro, il costruttore tenta di restituire il valore a virgola mobile equivalente
- Ad esempio, la chiamata float(2) restituisce 2.0
- Se il parametro del costruttore è una stringa (float('14.3')), tenta di analizzare la stringa come valore a virgola mobile

La classe elenco

- Un'istanza di elenco memorizza una sequenza di oggetti
- Una lista è una struttura referenziale, poiché tecnicamente memorizza una sequenza di riferimenti ai suoi elementi
- Gli elenchi sono sequenziati basati su array e indicizzati a zero (un elenco di lunghezza n ha elementi indicizzati da 0 a $n - 1$ compreso)
- Gli elenchi sono probabilmente i tipi di contenitori più utilizzati in Python, hanno molti comportamenti preziosi, inclusa la capacità di espandere e contrarre dinamicamente le proprie capacità secondo necessità



La classe elenco

- Python usa i caratteri [e] come delimitatori per una lista letterale, dove [] è una lista vuota. Per esempio:

[‘uno due tre’]

- Se sono stati stabiliti gli identificatori a, b, c, la seguente sintassi è corretta:

[a, b, c]

La classe elenco

```
In [16]: a = 1
```

```
In [17]: b = 2.89
```

```
In [18]: c = 'hello everybody'
```

```
In [19]: list1 = [a,b,c]
```

```
In [20]: list1
```

```
Out[20]: [1, 2.89, 'hello everybody']
```

La classe elenco

- Il costruttore `list()` produce per impostazione predefinita un elenco vuoto
- Tuttavia, il costruttore accetterà qualsiasi parametro di tipo iterabile (ad esempio, stringhe, elenchi, tuple, insiemi, dizionari)
- Per esempio:

```
In [21]: list('hello everybody')
```

```
Out[21]: ['h', 'e', 'l', 'l', 'o', ' ', 'e', 'v', 'e', 'r', 'y', 'b', 'o', 'd', 'y']
```

La classe tupla

- La classe tupla fornisce una versione immutabile di una sequenza
- Python usa i caratteri (e) per delimitare una tupla, dove () è una tupla vuota
- Per esprimere una tupla di lunghezza uno come valore letterale, è necessario inserire una virgola dopo l'elemento (tra parentesi).
Per esempio:

```
In [22]: aa = 22
```

```
In [23]: bb = (22,)
```

```
In [24]: aa
```

```
Out[24]: 22
```

```
In [26]: bb
```

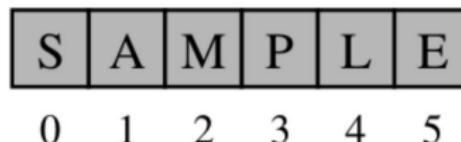
```
Out[26]: (22,)
```

La classe str

- La classe str è progettata per rappresentare una sequenza immutabile di caratteri basata sul carattere internazionale Unicode

impostato

- Le stringhe hanno una rappresentazione interna più compatta rispetto alle liste e alle tuple



La classe str

- Le stringhe possono essere racchiuse tra virgolette singole, come in "ciao" o virgolette doppie, come in "ciao"
- La seconda scelta è conveniente, in particolare quando si utilizza un altro dei caratteri delle virgolette come in "L'ultimo"
- In alternativa, il delimitatore delle virgolette può essere designato utilizzando una barra rovesciata (il cosiddetto carattere di escape) come in \L'ultimo
- Naturalmente, anche la barra rovesciata deve essere preceduta dall'escape (se necessario)
- Python supporta anche il delimitatore ... O ...
- Il vantaggio principale di questi delimitatori è che le stringhe contengono naturalmente dei ritorni a capo

La classe fissa

- Rappresenta la nozione matematica di insieme, cioè un insieme di elementi distinti senza un ordine intrinseco
- Ha un metodo altamente ottimizzato per verificare se un elemento specifico è contenuto nel set
- Due restrizioni principali:
 - L'insieme non mantiene gli elementi in un ordine particolare
 - Solo le istanze di elementi immutabili possono essere aggiunte ad a Set Python (interi, numeri in virgola mobile, stringhe di caratteri)
- Python usa { e } come delimitatori per un insieme, ad esempio {1, 4, 2, 9} o {} C'è
- un'eccezione importante: {} rappresenta un dizionario vuoto (non un insieme). Per rappresentare un insieme vuoto è necessario invocare il costruttore set()

La classe fissa

```
In [27]: alist = [1,2,3,4,5]
```

```
In [28]: alist
```

```
Out[28]: [1, 2, 3, 4, 5]
```

```
In [29]: aset = {1,2,3,4,5}
```

```
In [30]: aset
```

```
Out[30]: {1, 2, 3, 4, 5}
```

```
In [31]: set('hello everybody')
```

```
Out[31]: {' ', 'b', 'd', 'e', 'h', 'l', 'o', 'r', 'v', 'y'}
```

La classe detta

- La classe dict rappresenta un dizionario, da un insieme di chiavi distinte a valori associati
- Ad esempio, un dizionario potrebbe mappare da numeri ID studente univoci a record di studenti più grandi (nome, indirizzo, voti del corso)
- La forma letterale {} rappresenta un dizionario vuoto, mentre un dizionario non vuoto è espresso utilizzando una serie di coppie chiave:valore separate da virgole
- Per esempio:

```
telefono = {'giovanni': 4040, 'joe': 4041, 'kate': 4042}
```

Esempi, operatori e precedenza

Operatori logici

- Python supporta i seguenti operatori di parole chiave per i valori booleani:

Descrizione dell'operatore

non	Negazione unaria
E	Condizionale e
o	Condizionale o

- Gli operatori and e or cortocircuitano, nel senso che non valutano il secondo operando se il risultato può essere determinato in base al valore del primo operando

Espresioni, operatori e precedenza

Operatori di uguaglianza

Python supporta i seguenti operatori per testare due nozioni di uguaglianza:

Descrizione dell'operatore	
È	stessa identità
non è	identità diversa
==	equivalente
!=	non equivalente

Espressioni, operatori e precedenza

Operatori di uguaglianza

- L'espressione a **viene** valutata come True proprio quando gli identificatori a e b sono alias per lo stesso oggetto
- L'espressione a == b verifica una nozione più generale di equivalenza:
 - Se gli identificatori a e b si riferiscono allo stesso oggetto, allora a == b restituisce True
 - Se gli identificatori si riferiscono a oggetti diversi che sono equivalenti, allora a == b restituisce True
- La nozione precisa di equivalenza dipende dal tipo di dati: due stringhe sono considerate equivalenti se corrispondono carattere per carattere, mentre due insiemi sono equivalenti se hanno lo stesso contenuto, indipendentemente dall'ordine
- In generale, l'uso di is e not dovrebbe essere riservato alle situazioni in cui è necessario rilevare un vero aliasing

Espressioni, operatori e precedenza

Operatori di confronto

I tipi di dati possono definire un ordine naturale tramite i seguenti operatori

Descrizione dell'operatore	
<	meno di
<=	minore o uguale a
>	più grande di
>=	maggiore o uguale a

- Questi operatori hanno un comportamento previsto per i tipi numerici
- Sono definiti lessicograficamente e con distinzione tra maiuscole e minuscole, per stringhe
- Viene sollevata un'eccezione se gli operandi hanno tipi non comparabili

Espressioni, operatori e precedenza

Operatori di confronto

```
In [32]: a = 1
```

```
In [33]: b = 1
```

```
In [34]: a == b
```

```
Out[34]: True
```

```
In [35]: a != b
```

```
Out[35]: False
```

```
In [36]: a is b
```

```
Out[36]: True
```

```
In [37]: a <= b
```

```
Out[37]: True
```

```
In [38]: a >= b
```

```
Out[38]: True
```

Espressioni, operatori e precedenza

Operatori di confronto

```
In [39]: x = 'Goofy'
```

```
In [40]: y = 'Goofy'
```

```
In [41]: x == y
```

```
Out[41]: True
```

```
In [42]: x < y
```

```
Out[42]: False
```

```
In [43]: x <= y
```

```
Out[43]: True
```

```
In [44]: z1 = 'aa'
```

```
In [45]: z2 = 'ab'
```

```
In [46]: z1 < z2
```

```
Out[46]: True
```

Esempi, operatori e precedenza

Operatori aritmetici

Python supporta i seguenti operatori aritmetici:

Descrizione dell'operatore	
+	aggiunta
-	sottrazione
*	moltiplicazione
/	vera divisione
//	divisione intera
%	operatore modulo

Espressioni, operatori e precedenza

Operatori aritmetici

- L'uso di +, \cdot e \cdot è semplice (se entrambi gli operandi sono int, il risultato è int; se uno o entrambi gli operandi sono float, il risultato sarà float)
- / denota la divisione vera, restituisce il quoziente
- (float) // viene utilizzato per eseguire il calcolo integrale % restituisce il resto della divisione intera

```
In [47]: 21 / 5
```

```
Out[47]: 4.2
```

```
In [48]: 21 // 5
```

```
Out[48]: 4
```

```
In [49]: 21 % 5
```

```
Out[49]: 1
```

Espressioni, operatori e precedenza

Operatori aritmetici

- Python estende la semantica di `//` e `%` ai casi in cui uno o entrambi gli operandi sono negativi
- Supponiamolo

$$n = q \cdot m + r$$

Dove:

- n , è il dividendo
- m , è il divisore
- q , è il quoziente ($q = n/m$)
- r , è il resto ($r = n \% m$)

Espressioni, operatori e precedenza

Operatori aritmetici

$$n = q \cdot m + r$$

- Quando il divisore è positivo, Python garantisce che $0 \leq r < m$
- Di conseguenza, abbiamo che $27//4 = 6$ e $27\%4 = 1$
- Abbiamo infatti che $6 \cdot 4 + 1 = 27$
- Quando il divisore è negativo, Python garantisce che $m < r \leq 0$
- Di conseguenza, abbiamo che $27//(−4) = −6$ e $27\% −4 = −1$
- Abbiamo infatti che $−6 \cdot (−4) + −1 = 27$

Espresioni, operatori e precedenza

Operatori bit a bit

Python supporta i seguenti operatori bit a bit per gli interi:

Descrizione dell'operatore

`~` complemento bit a bit (prefisso operatore unario)

`& |` bit per bit e

`^` bit per bit o

`bit per bit esclusivo-or`

`<<` sposta i bit a sinistra, riempiendoli con zeri

`>>` sposta i bit a destra, riempiendoli con il bit di segno

Espressioni, operatori e precedenza

Operatori di sequenza

Ciascuno dei tipi di sequenza incorporati di Python (str, tuple, list) supportano le seguenti sintassi degli operatori:

Descrizione dell'operatore

s[j] Elemento $s[j]$ all'indice j

s[start:stop] porzione inclusi gli indici [start,stop]

s[start : stop : step] porzione comprendente gli indici start, start+step, start +2·passo, . . . fino a ma senza eguagliare o fermarsi

s+t concatenazione di sequenze

k*s abbreviazione di $s + s + s + \dots$ (k volte)

val a s controllo del contenimento

val non in s controllo di non contenimento

Espressioni, operatori e precedenza

Operatori di sequenza

- Python si basa sull'indicizzazione zero delle sequenze (una sequenza ha n elementi indicizzati da 0 a $n - 1$ compreso)
- Gli indici negativi indicano una distanza dalla fine della sequenza: indice -1 denota l'ultimo elemento, indice -2 il penultimo e così via
- Python usa una notazione di slicing per descrivere le sottosequenze di una sequenza
 - Le fette sono descritte come intervalli semiaperti, con un indice di inizio (incluso) e un indice di fine (escluso)
 - Un indice di passo opzionale, eventualmente negativo, può essere indicato come terzo parametro di una fetta
 - Se un indice di inizio o di fine viene omesso nella notazione di slicing, si presume che designi l'estremo della sequenza originale

Espressioni, operatori e precedenza

Operatori di sequenza

```
In [50]: a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

```
In [51]: a[0]
```

```
Out[51]: 'a'
```

```
In [52]: a[5]
```

```
Out[52]: 'f'
```

```
In [53]: a[0:3]
```

```
Out[53]: ['a', 'b', 'c']
```

```
In [54]: a[:3]
```

```
Out[54]: ['a', 'b', 'c']
```

```
In [55]: a[0:]
```

```
Out[55]: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

```
In [56]: a[0::2]
```

```
Out[56]: ['a', 'c', 'e', 'g', 'i']
```

Espressioni, operatori e precedenza

Operatori di sequenza

- Le liste sono mutabili, pertanto la seguente sintassi viene utilizzata per sostituire un elemento in un dato indice

s[j] = val

- Gli elenchi supportano anche una sintassi per rimuovere l'elemento designato dall'elenco:

del s[j]

- La notazione delle sezioni può essere utilizzata anche per sostituire o eliminare un sottoelenco

Espressioni, operatori e precedenza

Operatori di sequenza

La notazione

val a s

può essere utilizzato per qualsiasi sequenza per vedere se c'è un elemento equivalente a val nella sequenza. Tutte le sequenze definiscono operatori di confronto in base all'ordine lessicografico, eseguendo un confronto elemento per elemento finché non viene trovata la prima differenza. Per esempio:

```
In [1]: ['a', 'b', 'c'] < ['a', 'B', 'c']
```

```
Out[1]: False
```

```
In [2]: ['a', 'b', 'c'] < ['a', 'B', 'c']
```

```
Out[2]: False
```

```
In [3]: ['a', 'b', 'c'] > ['a', 'B', 'c']
```

```
Out[3]: True
```

Espresioni, operatori e precedenza

Operatori di sequenza

Le seguenti operazioni sono supportate dai tipi di sequenza:

Descrizione dell'operazione	
s == t	equivalente (elemento per elemento)
s!=t	non equivalente
s < t	lessicograficamente inferiore a
s <= t	lessicograficamente inferiore o uguale a
s > t	lessicograficamente maggiore di
s >= t	lessicograficamente maggiore o uguale a

Espressioni, operatori e precedenza

Operatori per insiemi e dizionari

I set supportano i seguenti operatori:

L'operatore	Descrizione
inserisce	controllo del contenimento
la chiave e non la controlla di non contenimento	
s1 == s2	s1 equivale a s2
s1 != s2	s1 non è equivalente a s2
s1 <= s2	s1 è un sottoinsieme di s2
s1 < s2	s1 è un sottoinsieme proprio di s2
s1 >= s2	s1 è un superinsieme di s2
s1 > s2	s1 è un soprainsieme corretto di s2
s1 s2	l'unione di s1 e s2
s1 e s2	l'intersezione di s1 e s2
s1 - s2	l'insieme degli elementi in s1 ma non in s2
s1 ^ s2	l'insieme degli elementi esattamente in uno tra s1 o s2

Espressioni, operatori e precedenza

Operatori per insiemi e dizionari

- Gli insiemi non garantiscono un ordine particolare dei loro elementi, quindi gli operatori di confronto (come `<`) non sono lessicografici; piuttosto, si basano sulla nozione matematica di sottoinsieme
- La conseguenza è che gli operatori di confronto definiscono un ordine parziale
- I dizionari, come gli insiemi, non mantengono un ordine ben definito nei loro elementi
- Il concetto di sottoinsieme non è particolarmente significativo per i dizionari, quindi la classe `dict` non supporta operatori come `<`
- I dizionari supportano la nozione di equivalenza, con `d1 == d2` se i due dizionari contengono lo stesso insieme di coppie chiave-valore

Espressioni, operatori e precedenza

Operatori per insiemi e dizionari

Il comportamento più utilizzato nei dizionari è l'accesso a un valore associato ad una particolare chiave k con la sintassi di indicizzazione, $d[k]$. Gli operatori supportati sono i seguenti:

Operatore	Descrizione
$d[key]$	valore associato alla chiave specificata
$d[key] =$	valore imposta (o reimposta) il valore associato alla chiave specificata
$d[key]$	rimuove la chiave e il valore associato dal dizionario
digitare	il controllo di contenimento
chiave non in	controllo di non contenimento
$d1 == d2$	$d1$ equivale a $d2$
$d1 != d2$	$d1$ non è equivalente a $d2$

Espressioni, operatori e precedenza

Espressioni composte e precedenza degli operatori

	Simboli
Accesso membro di tipo 1	membro expr
2 la funzione/metodo chiama	espressione(...)
3 l'esponenziazione	yy
4 operatori unari	+espressione, -espressione
5 moltiplicazione, divisione,	*, /, //, %
6 addizione, sottrazione,	+, -
7 spostamento bit	«, »
8 per bit e	&
9 xor bit per bit	^
10 bit per bit o	
11 confronti (contenimento) è, non è, ==, !=, <, <=, >, >=, in non in	
12 logico non	non espr
13 logico e 14 logico	E
o 15 condizionale	O
	val1 se cond altrimenti val2
16 incarichi	=, +=, -=, *=, ecc.

Flusso di controllo

Condizionali

- I costrutti condizionali forniscono un modo per eseguire un blocco di codice scelto in base alla valutazione in fase di esecuzione di uno o più Espressioni booleane

se prima_condizione:

 primo_corpo

elif seconda_condizione:

 secondo_corpo

elif terza_condizione:

 terzo_corpo

altrimenti:

 quarto_corpo

Flusso di controllo

Condizionali

- Ogni condizione è un'espressione booleana e ogni corpo contiene uno o più comandi eseguiti in modo condizionale
- Potrebbe esserci un numero qualsiasi di clausole elif; la clausola finale else è facoltativa
- I tipi non booleani possono essere valutati come booleani con significati intuitivi, ad esempio

se risposta:

- è equivalente a:

se risposta != ":"

Flusso di controllo

Condizionali

Per esempio:

```
se la porta_è_chiusa:  
    se la porta_è_chiusa:  
        sblocca_porta()  
    apri_porta()  
    anticipa()
```

Il comando finale, advance(), non è rientrato e quindi non fa parte del corpo condizionale. Verrà eseguito incondizionatamente

Flusso di controllo

Cicli: cicli while

La sintassi per un ciclo while in Python è la seguente:

condizione **while** :

corpo

- condizione può essere un'espressione booleana arbitraria
- il corpo può essere un blocco di codice arbitrario (comprese le strutture di controllo nidificate)

Flusso di controllo

Cicli: cicli while

- L'esecuzione di un ciclo while inizia con un test del file Condizione booleana: se la condizione restituisce True, viene eseguito il corpo del ciclo
- Dopo ogni iterazione del corpo, la condizione del ciclo viene nuovamente testata e, se restituisce True, viene eseguita un'altra iterazione del corpo
- Quando la condizione di test risulta False (se mai), il ciclo viene chiuso e il flusso di controllo continua appena oltre il corpo del ciclo

Flusso di controllo

Cicli: cicli while

Un esempio:

```
j = 0
mentre j < len(dati) e dati[j] != 'X':
    j+= 1
```

- La correttezza di questo ciclo dipende dal comportamento dell'operatore e
- In questo frammento di codice, prima viene testato `j < len(data)` per garantire che `j` sia un indice valido e solo dopo l'accesso al relativo elemento di `dati`
- Se la condizione composta fosse scritta nell'ordine opposto, la valutazione di `data[j]` alla fine solleverebbe un `IndexError` quando 'X' non viene trovato.

Flusso di controllo

Loop: per loop

La sintassi per un ciclo for in Python è la seguente:

per elemento in iterabile:

corpo

La sintassi del ciclo for può essere utilizzata su qualsiasi tipo di struttura iterabile, come list, set, dict o file

Un esempio:

più grande = data[0] per

val in data:

**se val > più grande: più
grande = val**

Flusso di controllo

Loop: per loop

- In alcune applicazioni è necessario l'indice di un elemento all'interno della sequenza, ad esempio l'indice dell'elemento più grande di una lista. In questo caso è possibile scorrere tutti i possibili indici della lista. A questo scopo Python fornisce un built-in nella classe denominata range che genera sequenze di interi. La sintassi range(n) genera la serie di n valori da 0 a n - 1. Ad esempio:

```
bigindex = 0
for j in range(len(data)):
    if dati[j] > dati[bigindex]:
        bigindex = j
```

Flusso di controllo

Loop: per loop

- Python supporta un'istruzione break che termina immediatamente un ciclo while o for quando viene eseguita all'interno del suo corpo.
- Per esempio:

```
trovato = Falso
```

```
per l'articolo nei dati:
```

```
se oggetto == destinazione:  
    trovato = Vero  
    rottura
```

- Python supporta anche un'istruzione continue che provoca l'interruzione dell'iterazione corrente del corpo del ciclo, ma con i successivi passaggi del ciclo che procedono come previsto

Funzioni

- Python supporta sia funzioni che metodi
- Il termine funzione si riferisce a una funzione tradizionale e senza stato che viene invocata senza il contesto di una particolare classe o un'istanza di quella classe, come `sorted(data)`
- Il termine metodo viene utilizzato per descrivere una funzione membro che viene richiamata su un oggetto specifico utilizzando una sintassi di passaggio di messaggi orientata agli oggetti, come `data.sort()`

Funzioni

```
def conteggio(dati, destinazione): n = 0  
  
    per l'articolo nei dati:  
        se oggetto == destinazione:  
            n+= 1  
  
    ritorno n
```

- La prima riga stabilisce un nuovo identificatore come il nome della funzione e il numero di parametri che si aspetta
- Ogni volta che viene chiamata una funzione, Python crea un record di attivazione dedicato che memorizza le informazioni rilevanti per la chiamata corrente
- Il record di attivazione include il namespace, per gestire tutti gli identificatori aventi ambito locale all'interno della chiamata corrente

Funzioni

Dichiarazione di reso

- Un'istruzione return indica che la funzione dovrebbe cessare immediatamente l'esecuzione e che un valore espresso dovrebbe essere restituito al chiamante
- Se un'istruzione return viene eseguita senza un argomento esplicito, viene restituito automaticamente il valore None
- Spesso un'istruzione return è il comando finale all'interno del corpo della funzione
- Tuttavia, nella stessa funzione sono ammesse più istruzioni return, con la logica condizionale che controlla quale comando verrà eseguito

```
def contiene(dati, destinazione): for
    elemento in dati:
        se oggetto == destinazione:
            restituisce True
    restituire Falso
```

Funzioni

Passaggio delle informazioni

- Gli identificatori utilizzati per descrivere i parametri attesi sono noti come parametri formali e gli oggetti inviati dal chiamante quando si invocano le funzioni sono conosciuti come parametri effettivi
- Il passaggio dei parametri in Python segue la semantica dell'istruzione di assegnazione standard: quando viene invocata una funzione, ogni identificatore (parametro formale) viene assegnato, nell'ambito locale della funzione, al rispettivo parametro effettivo fornito dal chiamante della funzione

Funzioni

Passaggio delle informazioni

- Per esempio:

```
def conteggio(dati, destinazione):  
    ...  
  
premi = conteggio(voti, 'A')
```



Funzioni

Passaggio delle informazioni

- Appena prima che il corpo della funzione venga eseguito, il parametro effettivo, i voti e "A", vengono assegnati implicitamente ai dati del parametro formale e all'obiettivo
- La comunicazione di un valore di ritorno dalla funzione al chiamante viene realizzata in modo simile come assegnazione
- Un vantaggio del meccanismo di Python per il passaggio di informazioni da e verso una funzione è che gli oggetti non vengono copiati
- Di conseguenza, l'invocazione di una funzione è efficiente, anche nel caso in cui un parametro o un valore restituito sia un oggetto complesso

Funzioni

Valori dei parametri predefiniti

- Python fornisce mezzi affinché le funzioni supportino più di una possibile firma di chiamata (polimorfismo)
- In particolare, le funzioni possono dichiarare uno o più valori predefiniti per i parametri, consentendo così al chiamante di invocare una funzione con un numero variabile di parametri effettivi
- Supponiamo che una funzione sia dichiarata con firma def foo(a, b=15, c=27)
- Ci sono tre parametri, due dei quali offrono valori predefiniti Un chiamante può inviare tre parametri effettivi, ad esempio foo(5, 20, 25), nel qual caso i valori predefiniti non vengono utilizzati Se il chiamante invia un parametro, foo(4), la funzione verrà eseguita con i valori dei parametri a = 4, b = 15, c = 27

Funzioni

Funzioni integrate di Python

Sintassi di chiamata	Descrizione
abs(x)	Restituisce il valore assoluto di un numero
all(iterable)	Restituisce True se bool(e) è True per ogni elemento mento e
any(iterable)	Restituisce True se bool(e) è True per almeno un elemento e
chr(intero)	Restituisce una stringa di un carattere con il punto di codice Unicode fornito
divmod(x, y)	Restituisce $(x//y, x\%y)$ come tupla, se xey lo sono numeri interi
hash(oggetto)	Restituisce un valore hash intero per l'oggetto

Funzioni

Funzioni integrate di Python

Sintassi di chiamata	Descrizione
<code>id(oggetto)</code>	Restituisce la porzione intera univoca come "identità" dell'oggetto
<code>input (richiesta)</code>	Restituisce una stringa da stdin; IL richiesta è facoltativa
<code>isinstance(obj, cls)</code>	Determina se obj è un'istanza di la classe
<code>iter (iterabile)</code>	Restituisce un nuovo oggetto iteratore per il parametro
<code>len(iterabile)</code>	Restituisce il numero di elementi in l'iterazione data

Funzioni

Funzioni integrate di Python

Chiamata alla	Descrizione
sintassi map(f, iter1, iter2, ...)	Restituisce un iteratore che fornisce il risultato delle chiamate di funzione f(e1, e2, ...) per i rispettivi elementi e1 ў iter1, e2 ў iter2, ...
massimo (iterabile)	Restituisce l'elemento più grande di data iterazione
massimo(a, b, c, ...)	Restituisce il più grande degli argomenti ts
min(a, b, c, ...)	Restituisce il più piccolo degli argomenti menti
successivo(iteratore)	Restituisce l'elemento successivo riportato dall'iteratore

Funzioni

Funzioni integrate di Python

Chiamata alla	Descrizione
<code>sintassi open(nomefile, modalità)</code>	Apri un file con il nome indicato e modalità di accesso
<code>ordine(carattere)</code>	Restituisce il punto di codice Unicode di il carattere dato
<code>po(x, y)</code>	Restituisce il valore x_y (come numero intero se x e y sono numeri interi)
<code>po(x, y, z)</code>	Restituisce il valore $x \text{ mod } y$ come un numero intero
<code>print(oggetto1, oggetto2, ...)</code>	Stampa gli argomenti, con spazi di separazione e fine riga finale

Funzioni

Funzioni integrate di Python

Intervallo di sintassi	Descrizione
di chiamata(stop)	Costruire un'iterazione di valori 0, 1, . . . ,ferma \ddot{y} 1
intervallo (inizio, fine)	Costruire un'iterazione di valori inizio,inizio + 1, . . . ,ferma \ddot{y} 1
range(start, stop, step)	Costruisce un'iterazione di valori inizio + passo,inizio + 2 · passo, . . .
invertito (sequenza)	Restituisce un'iterazione della sequenza al contrario
rotondo(x)	Restituisce il valore intero più vicino
giro(x, k)	Restituisce il valore arrotondato a 10 \ddot{y} k più vicini
ordinato (iterabile)	Restituisce una lista contenente elementi dell'iterabile in ordine ordinato

Funzioni

Funzioni integrate di Python

Sintassi di chiamata	Descrizione
<code>sum(iterable)</code>	Restituisce la somma degli elementi in l'iterabile (deve essere numerico)
<code>tipo(oggetto)</code>	Restituisce la classe a cui appartiene l'istanza obj

Funzioni ricorsive

- Una funzione ricorsiva è una funzione che richiama se stessa
-

```
def conto alla rovescia(n):
    se n <= 0:
        print("Vai!")
    altrimenti:
        stampa(n)
        conto alla rovescia(n-1)
```

- Quante volte invoca se stesso?
- Fino al raggiungimento della condizione base

Funzioni ricorsive

Funzione fattoriale

- Il fattoriale di un numero positivo n , solitamente indicato come $n!$, è definito come:

$$N! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1 & \text{se } n \geq 1 \end{cases}$$

Funzioni ricorsive

Funzione fattoriale

- Il fattoriale può essere definito una funzione ricorsiva
- Posto che, per definizione, $n! = n \cdot (n - 1)!$
- Possiamo scrivere:

$$N! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n - 1)! & \text{se } n \geq 1 \end{cases}$$

- Si noti che questo tipo di definizione è comune ad altre ricorsive funzioni
- Di solito, nella definizione di funzione ricorsiva ne abbiamo una o più condizioni di base
- In questo caso l'unica condizione base è che $n = 0$

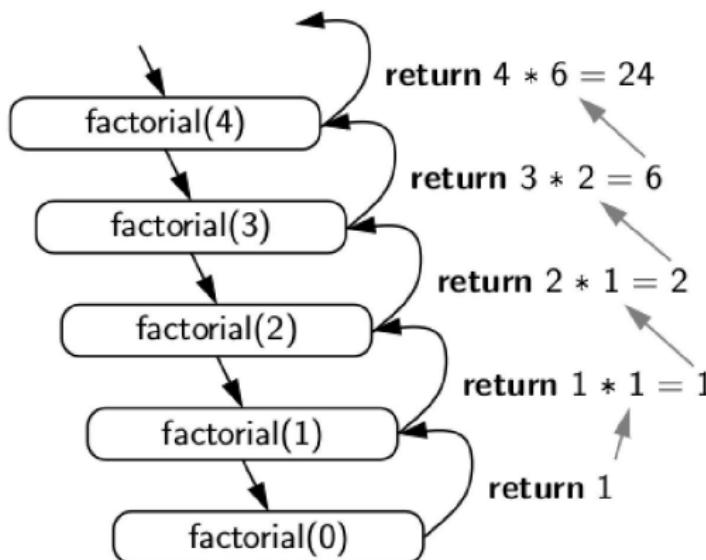
Funzioni ricorsive

Funzione fattoriale

```
def fattoriale(n): se n ==  
    0:  
        ritorno 1  
    altro:  
        rendimento n * fattoriale(n-1)
```

Funzioni ricorsive

Funzione fattoriale



Ingresso e uscita semplici

La funzione di stampa

- La funzione incorporata print viene utilizzata per generare output standard sulla console.
- Stampa una sequenza arbitraria di argomenti, separati da uno spazio e seguiti da un carattere di fine riga finale. Ad esempio: stampa(x, y)
- La funzione di stampa può essere personalizzata come segue:
 - Per impostazione predefinita, nell'output viene inserito uno spazio di separazione tra ogni coppia di argomenti. Il separatore può essere personalizzato fornendo la stringa di separazione desiderata. Ad esempio: print(x, y, sep='--')
 - Per impostazione predefinita, viene emessa una nuova riga finale dopo l'argomento finale. È possibile designare una stringa finale alternativa utilizzando la parola chiave end. Ad esempio: print(x, y, sep=':', end='.)
 - Per impostazione predefinita, l'output viene inviato alla console standard. L'output può essere indirizzato a un file indicando un flusso di file di output utilizzando file come parametro di parola chiave.

Ingresso e uscita semplici

La funzione di ingresso

- La funzione incorporata input viene utilizzata per visualizzare un prompt (se fornito come parametro opzionale) e attendere finché l'utente non immette una sequenza di caratteri
- Il valore di ritorno formale della funzione è la stringa di caratteri immessa rigorosamente prima del tasto Invio
- Quando si legge un input numerico, è necessario utilizzare la funzione input per ottenere la stringa di caratteri e quindi la sintassi int o float per costruire il valore numerico che la stringa di caratteri rappresenta

```
età = int(input("Inserisci la tua età in anni:"))
```

Ingresso e uscita semplici

La funzione di ingresso

- Poiché input restituisce una stringa come risultato, l'uso di tale funzione può essere combinato con la funzionalità della classe string
- Ad esempio, se vengono inserite più informazioni sulla stessa riga, è possibile invocare il metodo di suddivisione:

```
risposta = input("Inserisci xey, separati da spazi: ") pezzi =
risposta.split() x = float(pezzi[0])
y = float(pezzi[1])
```

Ingresso e uscita semplici

File

- In genere in Python si accede ai file iniziando con una chiamata a una funzione integrata, denominata open, che restituisce un proxy per le future interazioni con il file sottostante
- Ad esempio, il comando

```
fp = open('esempio.txt')
```

- tenta di aprire un file denominato sample.txt restituendo un proxy che consente l'accesso in sola lettura al file

Ingresso e uscita semplici

File

- La funzione open accetta un secondo parametro che determina la modalità di accesso
- La modalità predefinita è 'r' (per la lettura); altri modi comuni sono 'w' (scrittura) o 'a' (aggiunta)
- È anche possibile lavorare con file binari utilizzando modalità di accesso come 'rb' o 'wb'
- Durante l'elaborazione di un file, il proxy mantiene una posizione corrente all'interno del file come un offset dall'inizio, misurato in numero di byte
- Quando si apre un file con modalità 'r' o 'w', la posizione è inizialmente 0, se aperto in modalità append, la posizione è inizialmente alla fine del file
- La sintassi fp.close() chiude il file associato al proxy fp assicurando che eventuali contenuti scritti vengano salvati

Ingresso e uscita semplici

File

Chiamata della	Descrizione
sintassi fp.read()	Restituisce il contenuto (rimanente) di un file leggibile come una stringa
fp.read(k)	Restituisce i successivi k byte di un file leggibile come una stringa
fp.readline()	Restituisce la riga corrente di un file leggibile come una stringa
fp.readlines()	Restituisce tutte le righe di un file leggibile come una stringa
for line in fp:	esegue l'iterazione di tutte le righe di un file leggibile
fp.seek(k)	Cambia la posizione attuale in cui essere th al k byte del file
fp.tell()	Restituisce la posizione corrente, misurata come offset in byte dall'inizio

Ingresso e uscita semplici

File

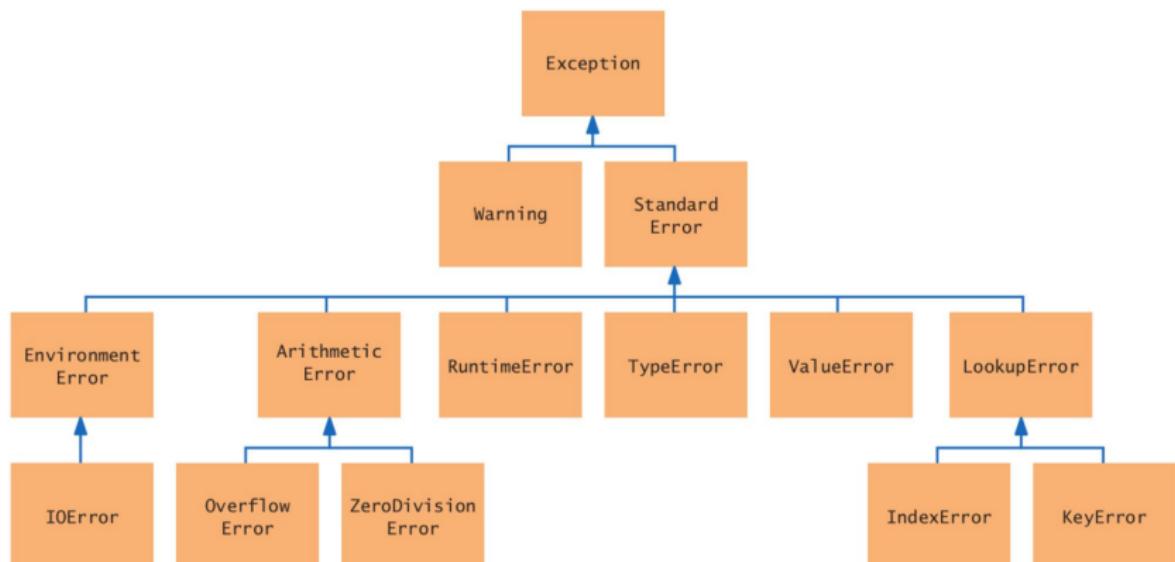
Chiamata della	Descrizione
sintassi fp.tell()	Restituisce la posizione corrente, misurata come offset in byte dall'inizio
fp.write(stringa)	Scrive la stringa data nella posizione corrente del file scrivibile
fp.writelines(seq)	Scrive ciascuna delle stringhe della sequenza data nella posizione corrente del file scrivibile. Questo il comando non inserisce ritorni a capo, oltre a quelli incorporati nel file stringhe
print(..., file=fp)	Reindirizza l'output della funzione di stampazione al file

La gestione delle eccezioni

- Le eccezioni sono eventi imprevisti che si verificano durante l'esecuzione di un programma
- Un'eccezione può derivare da un errore logico o da una situazione imprevista In Python le eccezioni sono oggetti sollevati dal codice che incontra una circostanza
- L'interprete Python può anche sollevare un'eccezione se si verifica una condizione imprevista (ad esempio, esaurimento della memoria)
- Un errore sollevato può essere catturato da un contesto circostante che gestisce l'eccezione in modo appropriato Se non catturata, un'eccezione fa sì che l'interprete interrompa l'esecuzione del programma e riporti un messaggio appropriato alla console

La gestione delle eccezioni

Tipi di eccezioni comuni



La gestione delle eccezioni

Tipi di eccezioni comuni

```
BaseException
  +-- SystemExit
  +-- KeyboardInterrupt
  +-- GeneratorExit
  +-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
    +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
    +-- MemoryError
    +-- NameError
      |    +-- UnboundLocalError
```

La gestione delle eccezioni

Tipi di eccezioni comuni

```
+++ OSError
    +-+ BlockingIOError
    +-+ ChildProcessError
    +-+ ConnectionError
        |   +-+ BrokenPipeError
        |   +-+ ConnectionAbortedError
        |   +-+ ConnectionRefusedError
        |   +-+ ConnectionResetError
    +-+ FileNotFoundError
    +-+ InterruptedError
    +-+ IsADirectoryError
    +-+ NotADirectoryError
    +-+ PermissionError
    +-+ ProcessLookupError
    +-+ TimeoutError
+-+ ReferenceError
+-+ RuntimeError
    +-+ NotImplementedError
    +-+ RecursionError
+-+ SyntaxError
    +-+ IndentationError
        +-+ TabError
```

La gestione delle eccezioni

Tipi di eccezioni comuni

```
.  
+-- SystemError  
+-- TypeError  
+-- ValueError  
|   +-- UnicodeError  
|   |   +-- UnicodeDecodeError  
|   |   +-- UnicodeEncodeError  
|   |   +-- UnicodeTranslateError  
+-- Warning  
    +-- DeprecationWarning  
    +-- PendingDeprecationWarning  
    +-- RuntimeWarning  
    +-- SyntaxWarning  
    +-- UserWarning  
    +-- FutureWarning  
    +-- ImportWarning  
    +-- UnicodeWarning  
    +-- BytesWarning  
    +-- ResourceWarning
```

La gestione delle eccezioni

Tipi di eccezioni comuni

Classe	Descrizione
Eccezione	Una classe base per la maggior parte dei tipi di errore
Errore attributo	Sollevato dalla sintassi obj.foo se obj non ha alcun membro chiamato foo
EOFError	Sollevata se è stata raggiunta la "fine del file". input da console o da file
IOError	Generato in caso di guasto del funzionamento I/O zione
Errore indice	Sollevata se l'indice della sequenza è out di limiti
Errore chiave	Sollevata se è richiesta una chiave inesistente per se o dizionario
KeyboardInterrupt	Sollevato se viene premuto CTRL-C mentre il programma è in esecuzione

La gestione delle eccezioni

Tipi di eccezioni comuni

Classe	Descrizione
NomeErrore	Sollevata se identificatore inesistente usato
Interrompilterazione	Sollevato da next(iteratore) if nessun elemento
TypeError	Sollevato quando a una funzione viene inviato un tipo di parametro errato
ValoreErrore	Sollevato quando il parametro non è valido valore (ad esempio, <code>sqrt(-1)</code>)
ZeroDivisionError	Sollevato quando qualsiasi operatore di divisione utilizzato con 0 come divisore

La gestione delle eccezioni

Sollevare un'eccezione

Viene generata un'eccezione eseguendo l' istruzione **raise** , ad esempio:

```
raise ValueError('x non può essere negativo')
```

- Viene sollevata un'istanza appena creata della classe ValueError, con il messaggio di errore che funge da parametro del costruttore
- Se questa eccezione non viene catturata all'interno del corpo della funzione, l'esecuzione della funzione cessa immediatamente e l'eccezione viene propagata al contesto chiamante

La gestione delle eccezioni

Sollevare un'eccezione

Quando si controlla la validità dei parametri inviati a una funzione, è necessario innanzitutto verificare che il parametro sia del tipo appropriato. Per esempio:

```
def sqrt(x): se
    diverso da isinstance(x, (int, float)):
        raise TypeError('x deve essere numerico') elif x < 0:
            raise ValueError('x non può essere negativo')
```

`isinstance(obj, cls)` restituisce `True` se l'oggetto `obj` è un'istanza della classe `cls` o qualsiasi sottoclasse di quel tipo

La gestione delle eccezioni

Sollevare un'eccezione

Un rigoroso controllo degli errori potrebbe essere scritto come segue:

```
def somma(valori):
    se non isinstance(values, collection.Iterable):
        raise TypeError('il parametro deve essere di tipo iterabile'

    ) totale = 0
    per v nei valori:
        se non è isistanza(v, (int, float)):
            raise TypeEror('gli elementi devono essere numerici') totale =
                totale + v
    restituzione totale
```

La gestione delle eccezioni

Sollevare un'eccezione

Un'implementazione più diretta e chiara può essere scritta come segue:

```
def somma(valori):
    totale = 0
    per v nei valori:
        totale = totale + v
    restituzione totale
```

Va notato che questa semplice implementazione funziona esattamente come la versione integrata Python della funzione.

La gestione delle eccezioni

Sollevare un'eccezione

- Anche senza i controlli esplicativi, le opportune eccezioni vengono sollevate naturalmente dal codice In particolare, se valori non è un tipo iterabile, il tentativo di utilizzare una sintassi for-loop solleva un TypeError segnalando che l'oggetto non è
- iterabile Nel caso in cui un utente invia un tipo iterabile che include un elemento non numerico, come sum([3.14, 'ciao']), viene naturalmente sollevato un TypeError e viene restituito il seguente messaggio di errore:

tipi di operandi non supportati per +: 'float' e 'str'

La gestione delle eccezioni

Rilevare un'eccezione

Prima filosofia: guardare prima di saltare

O:

evitare la possibilità che venga sollevata un'eccezione attraverso l'uso
del test condizionale proattivo

se $y \neq 0$:

rapporto = x / y

altro:

 . . .

La gestione delle eccezioni

Rilevare un'eccezione

Seconda filosofia: è più facile chiedere perdono che ottenere il permesso

O:

non è necessario dedicare ulteriore tempo di esecuzione alla salvaguardia di ogni possibile eccezione

**prova: rapporto =
x / y tranne ZeroDivisionError**

· · ·

La gestione delle eccezioni

Rilevare un'eccezione

Un'altra forma di prova. . . tranne la sintassi:

```
prova: fp = open('sample.txt')
tranne IOError come e:
    print('Impossibile aprire il file: ', e)
```

In questo caso il nome e denota l'istanza dell'eccezione che è stata lanciata e stampandola viene visualizzato un messaggio di errore dettagliato

La gestione delle eccezioni

Rilevare un'eccezione

Il tentativo. . . tranne la sintassi supporta più di un tipo di eccezione, ad esempio:

```
age = -1
while age <= 0: try:
    age
        = int(input('Inserisci la tua età in anni: '))
    if age <= 0: print('La tua età deve
        essere positiva') tranne (ValueError, EOFError) :
    print('Risposta non valida')
```

La tupla (ValueError, EOFError) viene utilizzata per designare i tipi di errore che vogliamo rilevare con la clausola tranne

La gestione delle eccezioni

Rilevare un'eccezione

È possibile fornire risposte diverse a diversi tipi di errori, utilizzando due o più clausole eccetto:

```
età = -1
while età <= 0: try:
    età =
        int(input('Inserisci la tua età in anni: '))
    if età <= 0: print('La tua
        età deve essere
            positiva') tranne ValueError: print(' Questa è
        una specifica di età non
            valida')
    tranne EOFError:
        print('Si è verificato un errore imprevisto durante la lettura dell'input')
        raise
```

La gestione delle eccezioni

Rilevare un'eccezione

- In questa implementazione sono presenti due clausole di eccezione separate per le situazioni ValueError e EOFError
- La clausola per gestire il caso EOFError contiene l'istruzione raise per sollevare nuovamente la stessa eccezione attualmente gestita
- L'istruzione raise permette di interrompere il ciclo while e propagare l'eccezione verso l'alto

La gestione delle eccezioni

Rilevare un'eccezione

- È possibile avere una clausola eccetto finale senza alcun tipo di errore identificato, per rilevare eventuali altre eccezioni che si sono verificate
- Questa possibilità dovrebbe essere utilizzata raramente, poiché è difficile suggerire come gestire un errore di tipo sconosciuto
- Un'istruzione try può avere una clausola final, con un corpo di codice che verrà sempre eseguito nei casi standard o eccezionali
- Questo blocco viene utilizzato per operazioni di pulizia, ad esempio la chiusura di file aperti

Iteratori e generatori

La sintassi for

per elemento in iterabile:

si applica a una serie di tipi di contenitori di base che si qualificano come tipi iterabili (elenco, tupla, set)

Una stringa può produrre un'iterazione dei suoi caratteri, un dizionario può produrre un'iterazione delle sue chiavi e un file può produrre un'iterazione delle sue righe

Iteratori e generatori

In Python, il meccanismo per l'iterazione si basa sulle seguenti convenzioni:

- Un iteratore è un oggetto che gestisce un'iterazione attraverso una serie di valori. Se i identifica un oggetto iteratore, allora ogni chiamata alla funzione integrata, `next(i)`, produce un elemento successivo dalla serie sottostante con un'eccezione `StopIteration` sollevata per indicare che non ci sono altri elementi. Un iterable è un oggetto obj che produce un iteratore tramite la sintassi `iter(obj)`

Iteratori e generatori

Secondo queste definizioni, un'istanza di una lista è un iterabile, ma non un iteratore

Con `data=[1, 2, 4, 8]` non è legale chiamare `next(data)`

```
In [4]: data = [1,2,4,8]
In [5]: i = iter(data)
In [6]: i
Out[6]: <list_iterator at 0x7f5298b27240>
In [7]: next(i)
Out[7]: 1
In [8]: next(i)
Out[8]: 2
In [9]: next(i)
Out[9]: 4
In [10]: next(i)
Out[10]: 8
In [11]: next(i)
...

```

Iteratori e generatori

- La sintassi for-loop in Python automatizza semplicemente questo processo, creando un iteratore per l'iterabile specificato e quindi richiamando ripetutamente l'elemento successivo fino a catturare l'eccezione
- StopIteration. Più in generale, è possibile creare più iteratori basati sullo stesso oggetto iterabile, con ciascun iteratore che mantiene il proprio stato di
- avanzamento Ciascun iteratore manterrà un indice corrente nell'elenco originale, che rappresenta l'elemento successivo da segnalare. Pertanto, se il contenuto dell'elenco originale viene modificato dopo la creazione dell'iteratore (ma prima che l'iterazione sia completata), l'iteratore riporterà il contenuto aggiornato dell'

Iteratori e generatori

classe PowTwo:

```
def __init__(self, massimo = 0):
    auto.max = max
```

```
def __iter__(self): self.n = 0
```

restituire sé stesso

```
def __successivo__(self):
```

se self.n <= self.max:

risultato = 2**self.n

sé.n += 1

risultato restituito

altro:

sollevare StopIteration

Iteratori e generatori

- La tecnica più conveniente per creare iteratori in Python è attraverso l'uso dei generatori
- Un generatore è implementato con una sintassi molto simile a una funzione, ma invece di restituire valori, viene eseguita un'istruzione `yield` per indicare ciascun elemento della serie

Iteratori e generatori

- Una funzione tradizionale:
-

```
def fattori(n): risultati
    = []
    per k in
        range(1, n+1):
            se n%k == 0:
                risultati.append(k)
    restituire risultati
```

- Un generatore:
-

```
def fattori(n): for k in
    range(1, n+1): if n%k == 0:

        rendimento k
```

Iteratori e generatori

- Un esempio:
-

```
def fattori(n): for k in  
    range(1,n+1): if n % k == 0:  
  
        rendimento k
```

n = 100

```
per valore in fattori(n):  
    stampa(valore)
```

Iteratori e generatori

```
|giacomo : ~/Dropbox/Teach/AACM/Lectures/examples >more generator.py
def factors(n):
    for k in range(1,n+1):
        if n % k == 0:
            yield k

n = 100

for value in factors(n):
    print(value)
|giacomo : ~/Dropbox/Teach/AACM/Lectures/examples >python3 generator.py
1
2
4
5
10
20
25
50
100
```

Iteratori e generatori

- Un altro esempio:

```
def gen():
    print("Inizio...") yield "A"

    print("continua...") yield "B"
    print("Fine")

for elemento in gen():
    print("Elemento in gen: ", elemento)
```

Ulteriori comodità

Espressioni condizionali

- Python supporta una sintassi di espressione condizionale che può sostituire una semplice struttura di controllo:

expr1 se condizione altrimenti expr2

Questa espressione composta restituisce expr1 se la condizione è vera,
altrimenti restituisce expr2

Ulteriori comodità

Espressioni condizionali

Per esempio:

se n >= 0:

 param = n **altro**:

 param = -n

risultato = foo(param)

può essere scritto come:

param = n **se** n >= 0 **altrimenti** -n risultato =

foo(param)

Ulteriori comodità

Espressioni condizionali

Oppure, ancora più concisamente:

risultato = foo(n se n >= 0 altrimenti -n)

- La semplice riduzione del codice sorgente è vantaggiosa perché evita la distrazione di una struttura di controllo ingombrante
- Le espressioni condizionali, tuttavia, dovrebbero essere utilizzate solo per migliorare la leggibilità del codice

Ulteriori comodità

Sintassi di comprensione

- Un compito di programmazione molto comune è produrre una serie di valori basati sull'elaborazione di un'altra serie
- Spesso questo compito può essere realizzato utilizzando la cosiddetta sintassi di comprensione
- La comprensione della lista ha la seguente forma generale:

espressione per valore nella condizione iterabile if

- Sia l'espressione che la condizione possono dipendere dal valore
- La clausola if è facoltativa

Ulteriori comodità

Sintassi di comprensione

La comprensione dell'elenco:

espressione per valore nella condizione iterabile if

è logicamente equivalente a:

```
risultato = [] per
valore in iterabile:
    se condizione:
        risultato.append(espressione)
```

Ulteriori comodità

Sintassi di comprensione

Ad esempio, il tradizionale frammento di codice:

```
quadrati = [] for  
k in range(1,n+1):  
    quadrato.append(k*k)
```

può essere sostituito con:

```
quadrati = [k*k per k nell'intervallo (1,n+1)]
```

Ulteriori comodità

Sintassi di comprensione

— Comprensione dell'elenco:

[k*k per k nell'intervallo (1,n+1)]

— Comprensione degli schemi:

{k*k per k nell'intervallo (1,n+1)}

— Comprensione del generatore:

(k*k per k nell'intervallo (1,n+1))

— Comprensione del dizionario:

{k: k*k per k nell'intervallo (1,n+1)}

Ulteriori comodità

Imballaggio e disimballaggio di sequenze

— Compattamento

automatico Se viene fornita una serie di espressioni separate da virgole, verranno trattate come una singola tupla, anche se non vengono fornite parentesi di chiusura. Per esempio:

dati = 2, 4, 6, 8

implica che i dati identificativi siano assegnati alla tupla (2, 4, 6, 8)

Esempio:

restituiscono x, y

Ulteriori comodità

Imballaggio e disimballaggio di sequenze

— Spacchettamento

automatico Oltre al comportamento di compattazione, Python può spacchettare automaticamente una sequenza. Per esempio:

```
x, y, z, t = intervallo(5, 9)
```

```
x, y, z, t = range(5, 9)
```

```
z
```

```
7
```

Ulteriori comodità

Imballaggio e disimballaggio di sequenze

```
x,y,z,t,u = range(5,9)
```

```
-----  
-----  
-----  
ValueError
```

```
Trac
```

```
eback (most recent call last)
```

```
<ipython-input-1-374f81002118> in <module>()
```

```
----> 1 x,y,z,t,u = range(5,9)
```

```
ValueError: not enough values to unpack (expected 5, got 4)
```

Ulteriori comodità

Imballaggio e disimballaggio di sequenze

Questa tecnica può essere utilizzata per decomprimere le tuple restituite da una funzione. Per esempio

quoziente, resto = divmod(a, b)

Questa sintassi può essere utilizzata anche nel contesto di un ciclo for, quando si esegue l'iterazione su una sequenza di iterabili

per x, y in [(7, 2), (5, 8), (6, 4)]:

Ulteriori comodità

Incarichi simultanei

La combinazione di imballaggio e disimballaggio automatico costituisce una tecnica nota come assegnazione simultanea:

x, y, z = 6, 2, 5

In realtà, la parte destra di questo compito viene automaticamente impacchettata in una tupla, e poi automaticamente decompressa con i suoi elementi assegnati a tre identificatori sulla parte sinistra

Quando si utilizza un'assegnazione simultanea, tutte le espressioni vengono valutate sulla destra prima che venga effettuata qualsiasi assegnazione alle variabili di sinistra. Per questo motivo fornisce un mezzo conveniente per scambiare i valori associati a due variabili:

io, j = j, io

Ulteriori comodità

Incarichi simultanei

Per esempio:

```
def fibonacci(): a, b =  
    0, 1 mentre Vero:  
  
        danno aa,  
        b = b, a + b
```

Invece di quello più tradizionale

Generatore:

```
def fibonacci():  
    un = 0  
    b = 1  
    mentre Vero:  
  
        produrre  
        un futuro = a + b  
        un = b  
        b = futuro
```

Ambiti e spazi dei nomi

- Quando si calcola una somma con la sintassi `x + y`, i nomi `x` e `y` devono essere stati precedentemente associati agli oggetti che fungono da valori, altrimenti verrà sollevato un `NameError` se non vengono trovate tali definizioni. La determinazione del valore associato a un identificatore è nota come risoluzione dei nomi
- Ogni volta che un identificatore viene assegnato a un valore, tale definizione viene effettuata con un ambito specifico. Le assegnazioni di livello superiore vengono effettuate in quello che è noto come ambito globale, mentre le assegnazioni effettuate all'interno del corpo di una funzione in genere hanno un ambito locale per quella chiamata di funzione
- Ogni ambito distinto in Python è rappresentato utilizzando un'astrazione nota come spazio dei nomi, che gestisce tutti gli identificatori attualmente definiti in un dato ambito

Moduli

La distribuzione standard di Python include valori, funzioni e classi organizzate in librerie aggiuntive note come moduli che possono essere importate dall'interno di un programma

L'istruzione `import` carica le definizioni da un modulo nello spazio dei nomi corrente. Un modulo utilizza la seguente sintassi:

dall'importazione matematica pi greco , sqrt

L'effetto di questa istruzione è di importare `pi` e `sqrt` nello spazio dei nomi corrente, consentendo l'uso diretto dell'identificatore `pi` o una chiamata della funzione `sqrt(x)`

Moduli

Se sono presenti molte definizioni dello stesso modulo, è possibile utilizzare un asterisco come carattere jolly:

dall'importazione matematica *

In alternativa è possibile importare il modulo stesso, utilizzando la seguente sintassi:

importare la matematica

Formalmente, questo aggiunge la matematica dell'identificatore allo spazio dei nomi corrente, con il modulo come valore. Una volta importate, è possibile accedere alle singole definizioni del modulo utilizzando un nome completo, come `math.pi` o `math.sqrt(x)`

Moduli

Creazione di un nuovo modulo

- Per creare un nuovo modulo è necessario inserire le relative definizioni in un file denominato con il suffisso .py.
- Tali definizioni possono essere importate da qualsiasi altro file .py all'interno della stessa directory del progetto Se, ad esempio, si deve chiamare la funzione count, che è contenuto in un file denominato myfunctions.py, la funzione può essere importata utilizzando la sintassi

dal conteggio delle importazioni di myfunctions

Moduli

Moduli esistenti

Nome del modulo	Descrizione
array	Fornisce l'archiviazione di array compatti per i tipi primitivi
raccolte	Definisce strutture dati aggiuntive e basi astratte se classi che coinvolgono raccolte di oggetti
copia	Definisce le funzioni generali per eseguire copie di oggetti progetti
heapq	Fornisce funzioni della coda di priorità basate su heap
matematica	Definisce le costanti e le funzioni matematiche comuni zioni
(continua)	

Moduli

Moduli esistenti

(continua)

Nome del modulo	Descrizione
<code>os</code>	Fornisce il supporto per le interazioni con il sistema operativo
<code>random</code>	Fornisce la generazione di numeri casuali
<code>re</code>	Fornisce il supporto per l'elaborazione delle espressioni regolari
<code>sys</code>	Fornisce un ulteriore livello di interazione con l'interprete Python
<code>time</code>	Fornisce il supporto per misurare il tempo o ritardare un programma

Analisi degli algoritmi

Operazioni primitive

Il tempo di esecuzione di un algoritmo (senza eseguire esperimenti) può essere analizzato eseguendo una descrizione di alto livello dell'algoritmo. A questo scopo, un insieme di operazioni primitive può essere definito come segue:

- Assegnazione di un identificatore a un oggetto
- Determinazione dell'oggetto associato a un identificatore
- Esecuzione di un'operazione aritmetica
- Confronto tra due numeri
- Accesso a un singolo elemento di un elenco Python tramite indice
- Chiamare una funzione
- Ritorno da una funzione

Analisi degli algoritmi

Operazioni primitive

- Un'operazione primitiva corrisponde a un'istruzione di basso livello con un tempo di esecuzione costante
- Invece di provare a determinare il tempo di esecuzione specifico di ciascuna operazione primitiva, verrà contato il numero di operazioni primitive e questo numero verrà utilizzato come misura del tempo di esecuzione dell'algoritmo
- Per catturare l'ordine di crescita del tempo di esecuzione di un algoritmo, verrà associata una funzione $f(n)$, che caratterizza il numero di operazioni primitive che vengono eseguite in funzione della dimensione dell'input n

Analisi degli algoritmi

Analisi asintotica

- Nell'analisi degli algoritmi, è necessario concentrarsi sul tasso di crescita del tempo di esecuzione in funzione della dimensione dell'input n
- I tempi di esecuzione degli algoritmi sono caratterizzati utilizzando funzioni che mappano la dimensione dell'input n su valori che corrispondono al fattore principale che determina il tasso di crescita in termini di n
- In altre parole, è possibile eseguire l'analisi di un algoritmo stimando il numero di operazioni primitive eseguite fino a un fattore costante, invece di impegnarsi in un'analisi specifica del linguaggio o dell'hardware del numero esatto di operazioni.

Analisi degli algoritmi

Notazione \tilde{y}

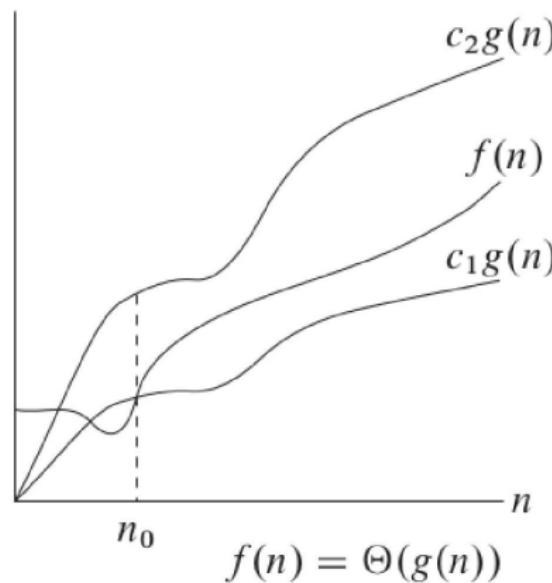
Per una data funzione $g(n)$, indichiamo con $\tilde{y}(g(n))$ l'insieme delle funzioni

$$\begin{aligned}\tilde{y}(g(n)) = \{f(n) : & \text{ costanti positive } c_1, c_2 \text{ e } n_0 \\ & \text{ tali che } 0 \leq c_1 g(n) \\ & \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0\}\end{aligned}$$

Una funzione $f(n)$ appartiene all'insieme $\tilde{y}(g(n))$ se esistono costanti positive c_1 e c_2 tali da poter essere “inserita” tra $c_1 g(n)$ e $c_2 g(n)$, per n sufficientemente grandi

Analisi degli algoritmi

Notazione $\ddot{\gamma}$



Analisi degli algoritmi

Notazione \tilde{y}

Dalla definizione di $\tilde{y}(g(n))$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0$$

segue che ogni $f(n) = \tilde{y}(g(n))$ deve essere asintoticamente non negativo, cioè $f(n)$ essere non negativo ogni volta che n è sufficientemente grande.

Di conseguenza, la funzione $g(n)$ stessa deve essere asintoticamente non negativa, altrimenti l'insieme $\tilde{y}(g(n))$ è vuoto.

Analisi degli algoritmi

Notazione Ѽ

Ad esempio, considera qualsiasi funzione quadratica

$$f(n) = an^2 + bn + c$$

dove a , b , c sono costanti e $a > 0$.

I termini di ordine inferiore di una funzione asintoticamente positiva possono essere ignorati nel determinare limiti asintoticamente stretti perché sono insignificanti per n grandi.

Analisi degli algoritmi

Notazione \tilde{y}

Ignorando i termini di ordine inferiore e ignorando le costanti si ottiene $f(n) = \tilde{y}(n^2)$

Più in generale, per qualsiasi polinomio

$$p(n) = \sum_{i=0}^D a_i n^i$$

dove gli a_i sono costanti e ad > 0, abbiamo $p(n) = \tilde{y}(n^D)$

Poiché ogni costante è un polinomio di grado 0, possiamo esprimere qualsiasi funzione costante come $\tilde{y}(1)$

Analisi degli algoritmi

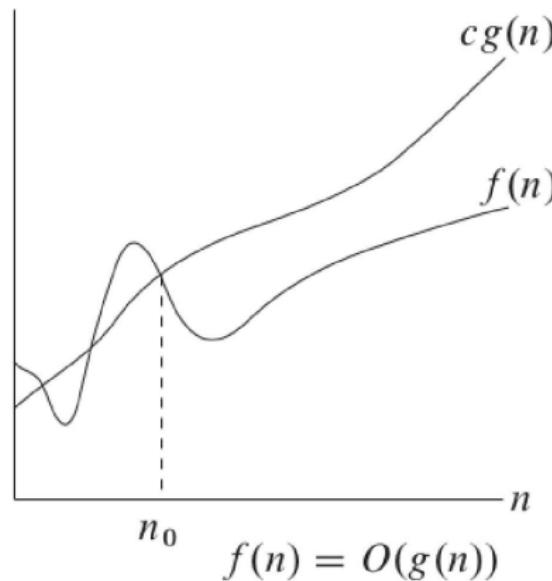
Notazione O

- La notazione \tilde{y} delimita asintoticamente una funzione dall'alto e dal basso
- Quando abbiamo solo un limite superiore asintotico, usiamo Notazione O
- Per una data funzione $g(n)$ indichiamo con $O(g(n))$ l'insieme delle funzioni

$$O(g(n)) = \{f(n) : \exists \text{ costanti positive } c \text{ e } n_0 \text{ tali che } 0 \leq f(n) \leq cg(n) \forall n \geq n_0\}$$

Analisi degli algoritmi

Notazione O



Analisi degli algoritmi

Notazione O

- Scriviamo $f(n) = O(g(n))$ per indicare che una funzione $f(n)$ è membro dell'insieme $O(g(n))$
- Si noti che $f(n) = \tilde{O}(g(n))$ implica $f(n) = O(g(n))$, poiché la notazione \tilde{O} è una nozione più forte della notazione O
- Poiché la notazione O descrive un limite superiore, quando la usiamo per limitare il tempo di esecuzione nel caso peggiore di un algoritmo, abbiamo un limite al tempo di esecuzione dell'algoritmo su ogni input

Analisi degli algoritmi

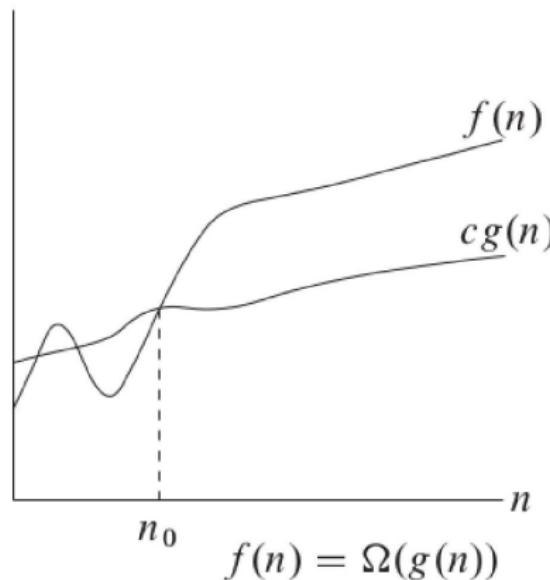
Notazione \tilde{y}

- Proprio come la notazione O fornisce un limite superiore asintotico su una funzione, la notazione \tilde{y} fornisce un limite inferiore asintotico
- Per una data funzione $g(n)$ indichiamo con $\tilde{y}(g(n))$ l'insieme delle funzioni

$$\tilde{y}(g(n)) = \{f(n) : \exists \text{ costanti positive } c \text{ e } n_0 \text{ tali che } 0 \leq cg(n) \leq f(n) \forall n \geq n_0\}$$

Analisi degli algoritmi

Notazione \tilde{y}



Analisi degli algoritmi

Notazione \ddot{y}

Dalla definizione delle notazioni asintotiche segue questo importante teorema

Teorema

Per due funzioni qualsiasi $f(n)$ e $g(n)$, abbiamo $f(n) = \ddot{y}(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \ddot{\Omega}(g(n))$.

Analisi degli algoritmi

Un esempio: medie dei prefissi

Data una successione S composta da n numeri, vogliamo calcolare una successione A tale che $A[j]$ sia la media degli elementi $S[0], S[1], \dots, S[j]$ per $n = 0, 1, \dots, n \leq 1$, cioè

$$A[j] = \frac{\sum_{i=0}^j S[i]}{j + 1}$$

Analisi degli algoritmi

Un esempio: medie dei prefissi (prima implementazione)

```
def prefisso_media1(S): n = len(S)

A = [0] * n
for j
    for i in range(n):
        totale = 0
        for i in range(j+1):
            totale += S[i]
        A[j] = totale / (j + 1)
    return A
```

Analisi degli algoritmi

Un esempio: medie dei prefissi (prima implementazione)

- L'istruzione **n = len(S)** viene eseguita in tempo costante (la classe **list** mantiene una variabile di istanza che registra la lunghezza corrente della lista)
- L'istruzione **A = [0] * n** provoca la creazione e l'inizializzazione di una lista Python di lunghezza n e con tutte le voci uguali a zero. Questo utilizza un numero costante di operazioni primitive per elemento e quindi viene eseguito in tempo $O(n)$. Il corpo del ciclo esterno, controllato da j , viene eseguito n volte. Pertanto le istruzioni **total = 0** e **A[j] = total / (j + 1)** vengono eseguite n volte ciascuna. Queste due affermazioni contribuiscono con un numero di operazioni primitive proporzionali al tempo n , ovvero al tempo $O(n)$.

Analisi degli algoritmi

Un esempio: medie dei prefissi (prima implementazione)

- Il corpo del ciclo interno, controllato da i , viene eseguito $j + 1$ volte, a seconda del valore corrente del contatore del ciclo esterno j . Pertanto, l'istruzione **total += S[i]** viene eseguita $1 + 2 + 3 + \dots + n = n(n + 1)/2$ volte, il che implica che l'istruzione contribuisce $O(n^2)$ tempo

Il tempo di esecuzione di questa implementazione è dato dalla somma di tre termini: il primo e il secondo termine sono $O(n)$, e il terzo il termine è $O(n^2)$. Pertanto il tempo di esecuzione è $O(n^2)$.

Analisi degli algoritmi

Un esempio: medie dei prefissi (seconda implementazione)

```
def prefisso_media2(S): n =
    len(S)
    A = [0] * n per j
    nell'intervallo(n):
        A[j] = somma(S[0:j+1]) / (j + 1) restituisce
    A
```

Analisi degli algoritmi

Un esempio: medie dei prefissi (seconda implementazione)

- L'espressione **sum(S[0:j+1])** è una chiamata di funzione e la sua valutazione richiede tempo $O(j + 1)$
- Anche il calcolo della sezione **S[0:j+1]** utilizza il tempo $O(j + 1)$, poiché costruisce una nuova istanza di lista per l'archiviazione

Il tempo di esecuzione di questa implementazione è dominato da una serie di passaggi che richiedono un tempo

proporzionale a $1 + 2 + 3 + \dots + n = n(n + 1)/2$, e quindi $\Theta(n^2)$

Analisi degli algoritmi

Un esempio: medie dei prefissi (terza implementazione)

```
def prefisso_media3(S): n = len(S)

A = [0] * n
totale
= 0

per j nell'intervallo(n):
    totale += S[j]
    A[j] = totale / (j + 1)
ritorno A
```

Analisi degli algoritmi

Un esempio: medie dei prefissi (terza implementazione)

- L'inizializzazione delle variabili n e $total$ utilizza il tempo $O(1)$
- L'inizializzazione della lista A utilizza il tempo $O(n)$
- $O(n)$ C'è un singolo ciclo, controllato da j . Ciò contribuisce per un totale di $O(n)$
- tempo. Il corpo del ciclo viene eseguito n volte, per $j = 0, 1, \dots, n - 1$. Pertanto, le istruzioni **$total += S[j]$** e **$A[j] = total / (j + 1)$** vengono eseguite n volte ciascuna: il loro contributo è $O(n)$

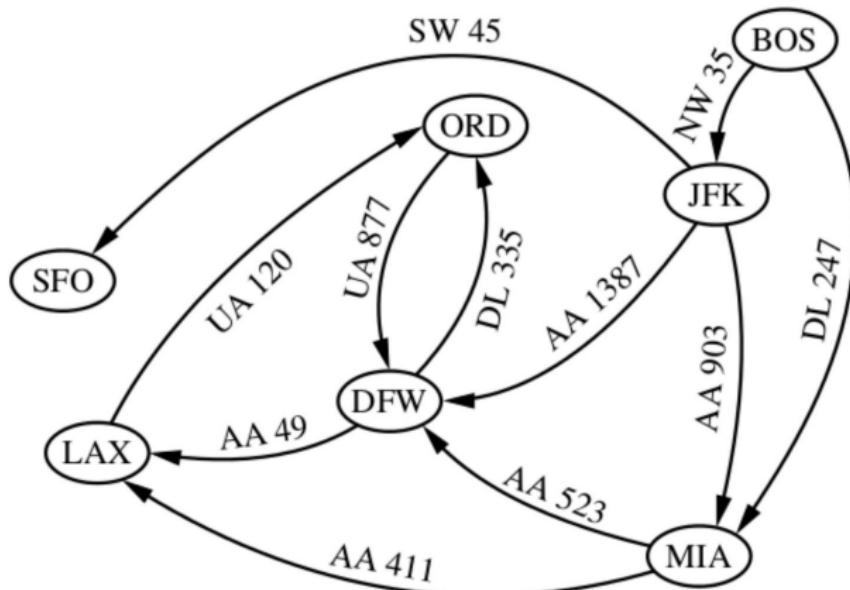
Il tempo di esecuzione di questa implementazione è dato dalla somma dei quattro termini: il primo è $O(1)$ e i restanti sono $O(n)$. Pertanto, il tempo di esecuzione è $O(n)$.

Grafici

introduzione

- Un grafico è un modo per rappresentare le relazioni che esistono tra coppie di oggetti
- In altre parole, un grafo è un insieme di oggetti, chiamati vertici (o nodi), insieme ad un insieme di connessioni a coppie tra loro, chiamate spigoli.
- Un grafo G è un insieme V di vertici e una raccolta E di archi
- Gli archi in un grafico sono diretti o non orientati
 - Un arco (u, v) si dice diretto da u a v se la coppia (u, v) è ordinata, con u che precede v
 - Un arco (u, v) si dice non orientato se la coppia (u, v) non è ordinata

Grafici



Grafici

Alcune definizioni

- Se tutti gli archi di un grafico non sono orientati, il grafico è un grafico non orientato
 - Allo stesso modo, un grafo diretto (o digrafo) è un grafo i cui bordi sono tutti diretti
 - Un grafo che ha sia archi diretti che non orientati è detto grafo misto
-
- I due vertici uniti da un bordo sono chiamati vertici finali (o punti finali) del bordo
 - Se un bordo è diretto, il suo primo punto finale è chiamato origine e l'altro è la destinazione del bordo
 - Due archi si dicono adiacenti se esiste un arco che li unisce

Grafici

Alcune definizioni

- Uno spigolo si dice incidente rispetto a un vertice se il vertice è uno degli estremi dello spigolo
- Gli spigoli uscenti di un vertice sono gli spigoli diretti la cui origine è nel vertice
- Gli archi entranti di un vertice sono gli archi diretti la cui destinazione è nel vertice
- Il grado di un vertice v , indicato con $\deg(v)$, è il numero di archi incidenti di v
- Il grado interno e quello esterno di un vertice v sono il numero dei bordi entrante e uscente di v e sono indicati rispettivamente $\text{indeg}(v)$ e $\text{outdeg}(v)$

Grafici

Alcune definizioni

- Secondo la definizione, l'insieme degli archi di un grafo è detto collezione (non insieme), consentendo così a due archi non orientati di avere gli stessi vertici finali, e a due archi orientati di avere la stessa origine e la stessa destinazione
- Tali bordi sono chiamati bordi paralleli o bordi multipli.
- Un altro tipo speciale di bordo se collega un vertice a se stesso. In altre parole, un arco è un self-loop se i suoi due punti finali coincidono. I grafici che non hanno archi paralleli o self-loop sono detti semplici

Grafici

Alcune definizioni

- Un percorso è una sequenza di vertici e spigoli alternati che inizia in un vertice e termina in un vertice tale che ogni spigolo è incidente rispetto al vertice predecessore e successore
- Se i vertici iniziale e finale coincidono il percorso si dice ciclabil
- Un ciclo deve includere almeno un vertice
- Un cammino si dice semplice se ogni vertice del cammino è distinto
- Un percorso diretto è un percorso tale che tutti gli spigoli diretti vengono attraversati lungo la loro direzione
- Un grafo diretto è aciclico se non ha cicli diretti

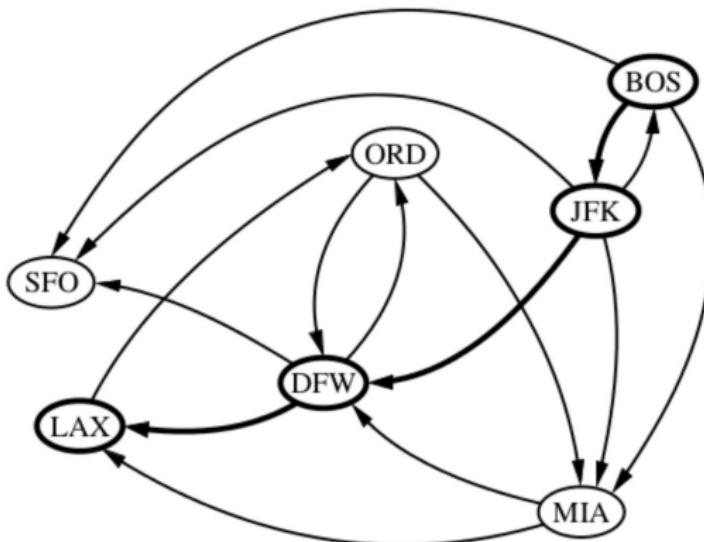
Grafici

Alcune definizioni

- Dati i vertici u e v di un grafo (diretto) G , diciamo che u raggiunge v se G ha un cammino (diretto) da u a v
- In un grafo non orientato la nozione di raggiungibilità è simmetrica
- Un grafo è connesso se, per due vertici qualsiasi, esiste un percorso tra di loro
- Un grafo diretto è fortemente connesso se per due vertici qualsiasi u e v , u raggiunge v e v raggiunge u

Grafici

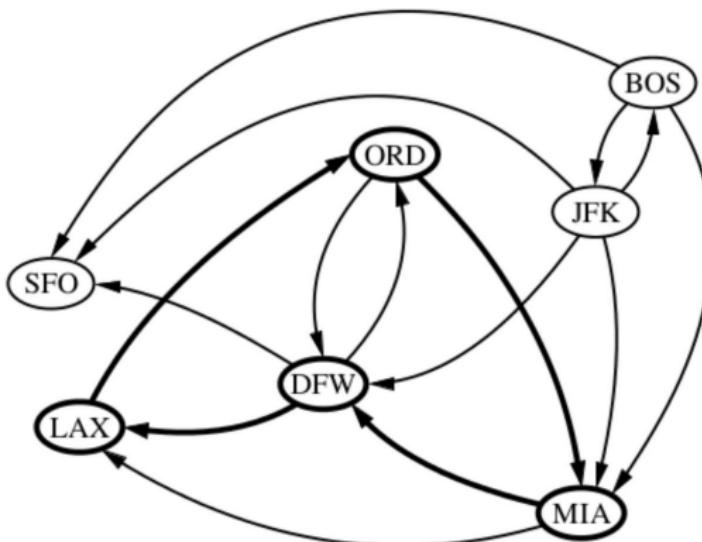
Alcune definizioni



Un percorso diretto da BOS a LAX

Grafici

Alcune definizioni



Un ciclo diretto (ORD; MIA; DFW; LAX; SFO; ORD)

Grafici

Alcune definizioni

- Un sottografo di un grafo G è un grafo H i cui vertici e archi sono sottoinsiemi dei vertici e degli archi di G
- Un sottografo di G è un sottografo di G che contiene tutti i vertici del grafo G
- Se un grafo G non è connesso, i suoi sottografi massimo connessi sono detti componenti connesse di G

Grafici

Alcune proprietà importanti

Proposizione 1

Se G è un grafo con m spigoli e insieme di vertici V , allora

$$\sum_{v \in V} \text{gradi}(v) = 2m$$

Giustificazione: un arco (u, v) viene contato due volte nella somma di cui sopra; una volta per il suo punto finale u e una volta per il suo punto finale v . Pertanto, il contributo totale degli spigoli ai gradi dei vertici è il doppio del numero di spigoli.

Grafici

Alcune proprietà importanti

Proposizione 2

Se G è un grafo diretto con m spigoli e insieme di vertici V , allora

$$\sum_{v \in V} \text{ind}_i(v) = \sum_{v \in V} \text{est}_e(v) = m$$

Giustificazione: In un grafo orientato, un arco (u, v) contribuisce con un'unità al grado esterno della sua origine u e con un'unità al grado interno della sua destinazione v . Pertanto, il contributo totale degli archi al grado esterno -grado (in gradi) dei vertici è uguale al numero di spigoli.

Grafici

Alcune proprietà importanti

Proposizione

3 Sia G un grafo semplice con n vertici e m spigoli. Se G non è orientato, allora

$$m \leq n(n - 1)/2$$

e se G è diretto, allora

$$m \leq n(n - 1)$$

Grafici

Alcune proprietà importanti

Giustificazione: Supponiamo che G non sia orientato. Poiché due archi non possono avere gli stessi punti finali e non ci sono auto-loop, il grado massimo di un vertice in G è $n \geq 1$ in questo caso. Quindi, per la Proposizione 1, $2m \leq n(n - 1)$.

Supponiamo ora che G sia diretto. Poiché non esistono due archi che possono avere la stessa origine e destinazione, e non ci sono auto-loop, il massimo grado interno di un vertice in G è $n - 1$ in questo caso. Quindi, per la Proposizione 2, $m \leq n(n - 1)$.

Tipo di dati astratti dei grafici (ADT)

Un grafico è una raccolta di vertici e archi. Può essere modellato come una combinazione di tre tipi di dati: vertice, bordo e grafico

- Un vertice è un oggetto che memorizza un elemento arbitrario fornito dall'utente e supporta un metodo, element()
- Un Edge memorizza un oggetto associato, recuperato con il metodo element()
- Supporta anche i seguenti metodi:
 - [endpoints\(\)](#), che restituisce una tupla (u, v) tale che il vertice u è l'origine del bordo e il vertice v è la destinazione . Per un grafico non orientato, l'orientamento è
 - arbitrariamente [opposto\(v\)](#), che restituisce l'altro punto finale se il vertice v è un punto finale del bordo

Tipo di dati astratti dei grafici (ADT)

Un grafo può essere orientato o non orientato. Un grafo misto può essere rappresentato come un grafo diretto, modellando l'arco $\{u, v\}$ come una coppia di archi diretti (u, v) e (v, u) . L'ADT grafico include i seguenti metodi:

`conteggio_vertice()`

Restituisce il numero di vertici del grafico

`vertici()`

Restituisce un'iterazione di tutti i vertici del grafico

`edge_count()`

Restituisce il numero di bordi del grafico

`bordi()`

Restituisce un'iterazione di tutti i bordi del grafico

Tipo di dati astratti dei grafici (ADT)

`get_bordo(u,v)`

Restituisce il bordo dal vertice u al vertice v, se ne esiste uno, altrimenti restituisce None

`grado(v, out=Vero)`

Per un grafo non orientato, restituire il numero di archi incidenti al vertice v. Per un grafico diretto, restituire il numero di archi uscenti (entranti) incidenti al vertice v

`incident_edges(v, out=True)`

Restituisce un'iterazione di tutti gli spigoli incidenti al vertice v. Nel caso di un grafico diretto, riporta gli spigoli in uscita per impostazione predefinita

Tipo di dati astratti dei grafici (ADT)

`insert_vertice(x=Nessuno)`

Crea e restituisce un nuovo elemento di memorizzazione Vertex x

`insert_edge(u, v, x=Nessuno)`

Crea e restituisce un nuovo Edge dal vertice u al vertice v, memorizzando l'elemento x

`rimuovi_vertice(v)`

Rimuovi il vertice v e tutti i suoi archi incidenti dal grafico

`rimuovi_bordo(e)`

Rimuovi il bordo e dal grafico

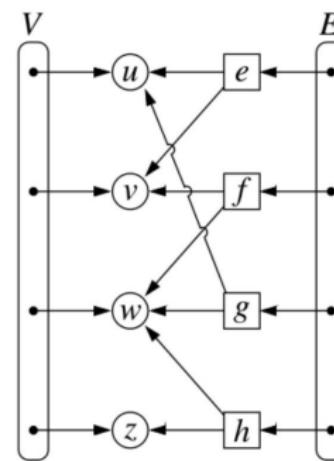
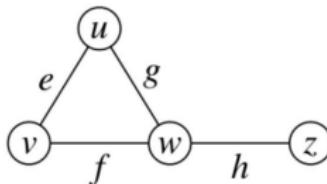
Strutture dati per grafici

- Elenco dei bordi
- Elenco delle adiacenze
- Mappa delle adiacenze
- Matrice di adiacenza

Strutture dati per grafici

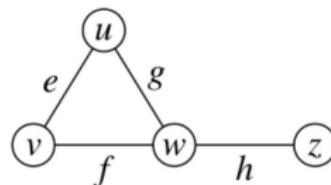
Elenco dei bordi

Elenco non ordinato di tutti i bordi. Questo è il minimo, poiché non esiste un modo efficiente per individuare un particolare spigolo (u, v) , o l'insieme di tutti gli spigoli incidenti su un vertice v



Strutture dati per grafici

Elenco dei bordi



In modo più semplice:

e ==> u - v

f ==> v - w

g ==> w - uh ==>

w - z

Strutture dati per grafici

Elenco dei bordi

Operazione	Tempo di esecuzione
conteggio_vertice()	O(1)
edge_count()	O(1)
vertici()	SU)
bordi()	O(m)
get_bordo(u,v)	O(m)
grado(v)	O(m)
bordi_incidente(v)	O(m)
inserisci_vertice(v)	O(1)
rimuovi_vertice(v)	O(m)
insert_edge(u,v,x)	O(1)
rimuovi_bordo(e)	O(1)

Strutture dati per grafici

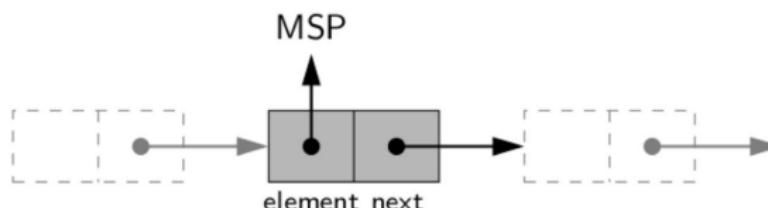
Edge List: performance

- La struttura della lista di bordi funziona bene in termini di reporting del numero di vertici o bordi, o nel produrre un'iterazione di quei vertici o bordi. Le
- limitazioni più significative di una struttura di lista di bordi sono i tempi di esecuzione $O(m)$ dei metodi `get_edge(u, v)`, `grado(v)` e `bordi_incidente(v)`. Ciò è dovuto alla disposizione del grafo in una lista non ordinata, per cui l'unico modo per rispondere a queste domande è ispezionare tutti gli spigoli. L'aggiornamento
- del grafo (inserimento di un vertice, o di uno spigolo, rimozione di uno spigolo) richiede $O(1)$ tempo La
- rimozione di un vertice, invece, richiede un tempo $O(m)$, perché devono essere rimossi anche tutti gli spigoli incidenti al vertice v . Per individuare gli spigoli incidenti rispetto al vertice, è necessario esaminare tutti gli spigoli

Digressione: elenchi collegati

Elenchi collegati singolarmente

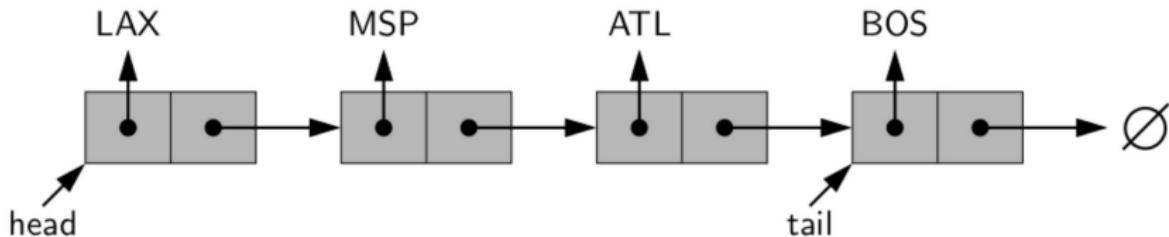
Una lista collegata singolarmente è una raccolta di nodi che collettivamente formano una sequenza lineare. Ogni nodo contiene un riferimento a un oggetto che è un elemento della sequenza nonché un riferimento al nodo successivo dell'elenco



Digressione: elenchi collegati

Elenchi collegati singolarmente

Il primo e il secondo nodo di una lista concatenata sono detti testa e coda della lista

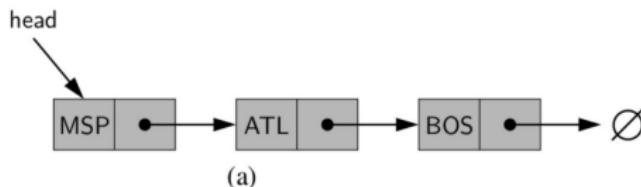


Una rappresentazione in memoria di una lista concatenata si basa sulla collaborazione di molti oggetti:

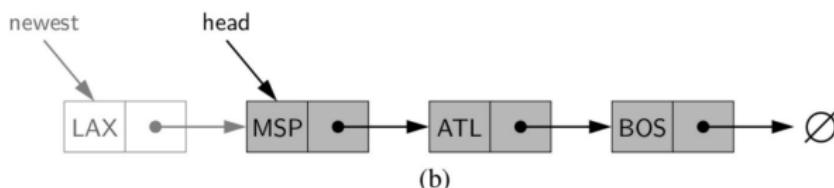
- Ogni nodo è rappresentato come un oggetto unico, con quell'istanza che memorizza un riferimento al suo elemento e un riferimento al nodo successivo (o Nessuno)

Digressione: elenchi collegati

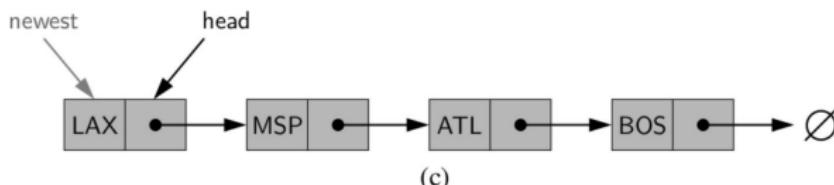
Inserimento di un elemento in testa ad una lista singolarmente concatenata



(a)



(b)

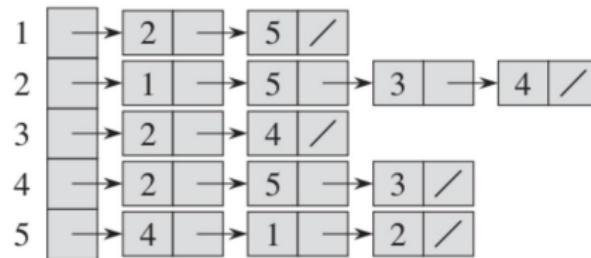
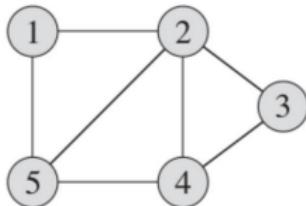


(c)

Strutture dati per grafici

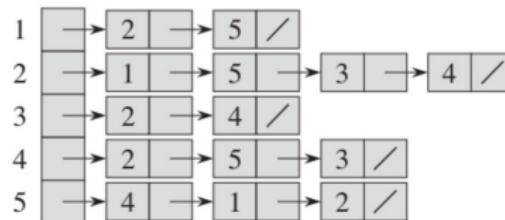
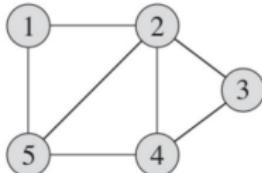
Elenco delle adiacenze

Per ciascun vertice viene mantenuto un elenco separato contenente gli spigoli incidenti al vertice. L'insieme completo degli spigoli può essere determinato prendendo l'unione degli insiemi più piccoli



Strutture dati per grafici

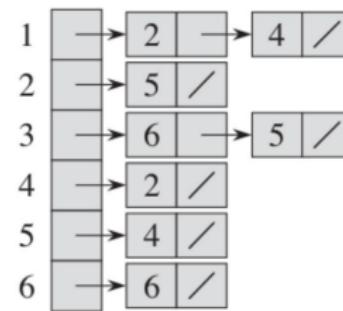
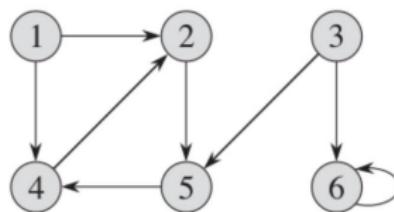
Elenco delle adiacenze



```
grafico = {1: set([2, 5]),
           2: imposta([1, 3, 4, 5]), 3:
           imposta([2, 4]), 4:
           imposta([2, 3, 5]), 5:
           imposta([1, 2, 4] )}
```

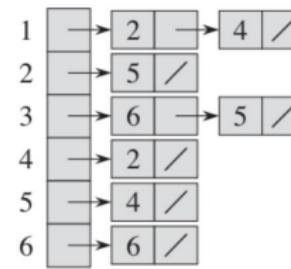
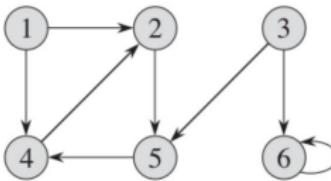
Strutture dati per grafici

Elenco delle adiacenze



Strutture dati per grafici

Elenco delle adiacenze



```
grafico = {1: set([2, 4]), 2: set([5]),  
          3: set([5, 6]), 4:  
              set([2]), 5: set([4]),  
          6: imposta([6])}
```

Strutture dati per grafici

Struttura dell'elenco di adiacenza

- La struttura dell'elenco di adiacenza raggruppa i bordi di un grafico memorizzandoli in contenitori secondari associati a ogni singolo vertice
- Per ogni vertice v , manteniamo una collezione $I(v)$, chiamata collezione di incidenza di v , le cui voci sono archi incidenti al vertice v .
- Nel caso di un grafo diretto, gli archi uscenti e entranti possono essere rispettivamente memorizzati in due raccolte separate, $I_{out}(v)$ e $I_{in}(v)$.

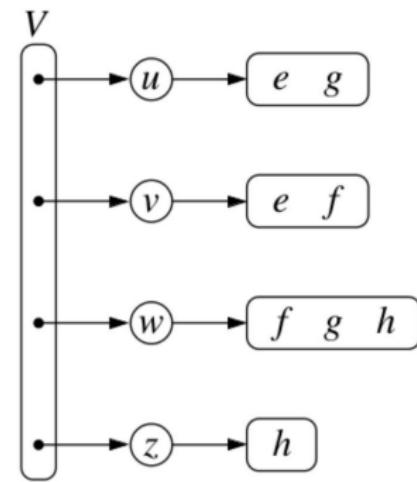
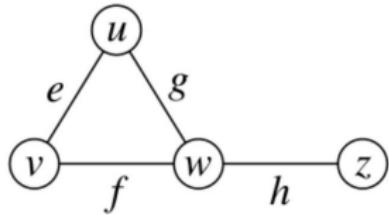
Strutture dati per grafici

Struttura dell'elenco di adiacenza

- La struttura primaria per una lista di adiacenza deve mantenere la raccolta V di vertici in modo tale da poter allocare la struttura secondaria $I(v)$ per un dato vertice v in tempo $O(1)$.
- Questo può essere fatto utilizzando una lista posizionale per rappresentare V , con ciascuna istanza di `Vertex` che mantiene un riferimento diretto alla sua
- raccolta di incidenza $I(v)$. Se i vertici possono essere numerati in modo univoco da 0 a $n - 1$, potremmo utilizzare una struttura primaria basata su array
- per accedere alle liste secondarie appropriate il vantaggio di una lista di adiacenza è che la collezione $I(v)$ contiene esattamente quegli spigoli che dovrebbero essere riportati dal metodo `incident_edges(v)`. Pertanto, possiamo implementare questo metodo iterando gli archi di $I(v)$ in $O(\deg(v))$, dove $\deg(v)$ è il grado del vertice

Strutture dati per grafici

Struttura dell'elenco di adiacenza



Strutture dati per grafici

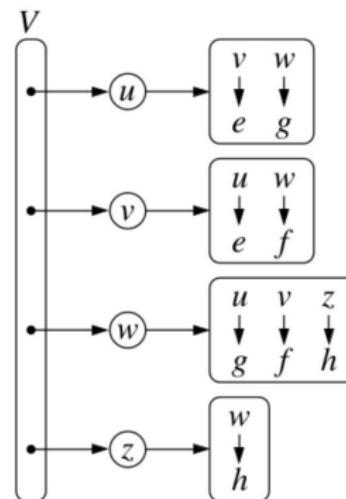
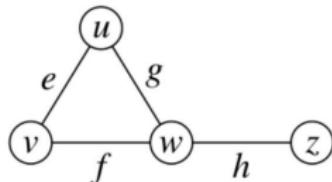
Elenco delle adiacenze

Operazione	Tempo di esecuzione
conteggio_vertice()	O(1)
edge_count()	O(1)
vertici()	SU
bordi()	O(m)
get_bordo(u,v)	O(min(du, dv))
grado(v)	O(1)
bordi_incidente(v)	O(dv)
inserisci_vertice(v)	O(1)
rimuovi_vertice(v)	O(dv)
insert_edge(u,v,x)	O(1)
rimuovi_bordo(e)	O(1)

Strutture dati per grafici

Mappa delle adiacenze

Molto simile ad una lista adiacente, ma il contenitore secondario di tutti i bordi incidenti ad un vertice è organizzato come una mappa, piuttosto che come una lista, con il vertice adiacente che funge da chiave. L'accesso ad uno specifico arco (u, v) viene effettuato in tempo $O(1)$.



Strutture dati per grafici

Mappa delle adiacenze

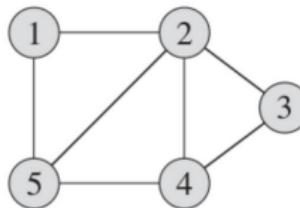
Operazione	Tempo di esecuzione
conteggio_vertice()	O(1)
edge_count()	O(1)
vertici()	SU)
bordi()	O(m)
get_bordo(u,v)	O(1) esp.
grado(v)	O(1)
bordi_incidente(v)	O(dv)
inserisci_vertice(v)	O(1)
rimuovi_vertice(v)	O(dv)
insert_edge(u,v,x)	O(1) esp.
rimuovi_bordo(e)	O(1) esp.

Strutture dati per grafici

Matrice di adiacenza

Fornisce l'accesso $O(1)$ nel caso peggiore a un bordo specifico (u, v) mantenendo una matrice $n \times n$ per un grafico con n vertici. Ciascuna voce è dedicata a memorizzare un riferimento al bordo (u, v) per una particolare coppia di vertici u e v . Se tale bordo non esiste, la voce sarà

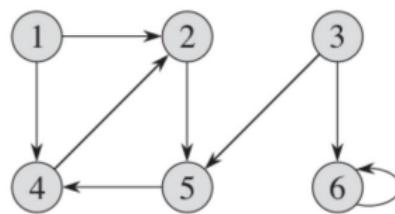
Nessuno



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

Strutture dati per grafici

Matrice di adiacenza



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Strutture dati per grafici

Matrice di adiacenza

Operazione	Tempo di esecuzione
conteggio_vertice()	O(1)
edge_count()	O(1)
vertici()	SU)
bordi()	O(m)
get_bordo(u,v)	O(1)
grado(v)	SU)
bordi_incidente(v)	O(n)
inserisci_vertice(v)	SU ²)
rimuovi_vertice(v)	SU ²)
insert_edge(u,v,x)	O(1)
rimuovi_bordo(e)	O(1)

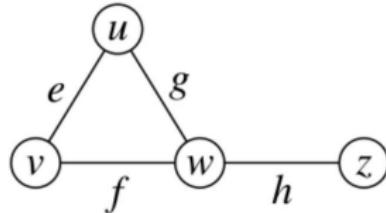
Strutture dati per grafici

Struttura della matrice di adiacenza

- La struttura della matrice di adiacenza per un grafo G utilizza una matrice A , che ci permette di localizzare un bordo tra una data coppia di vertici nel caso peggiore in tempo costante
- In questa rappresentazione, i vertici sono gli interi dell'insieme $\{0, 1, \dots, n - 1\}$ e gli archi sono coppie di tali interi Nello specifico, la cella $A[i, j]$ contiene un riferimento allo spigolo (u, v) se esiste, dove u è il vertice con indice i e v è il vertice con indice j . Se non esiste un tale arco, allora $A[i, j] = \text{Nessuno}$ L'array A è simmetrico se il grafico G non è
- orientato, poiché $A[i, j] = A[j, i]$ per tutte le coppie i e j

Strutture dati per grafici

Struttura della matrice di adiacenza



	0	1	2	3
$u \rightarrow$	0	e	g	
$v \rightarrow$	1	e	f	
$w \rightarrow$	2	g	f	h
$z \rightarrow$	3		h	

Strutture dati per grafici

Struttura della matrice di adiacenza

— Vantaggi

- Il vantaggio più significativo di una matrice di adiacenza è che è possibile accedere a qualsiasi arco (u, v) nel caso peggiore del tempo $O(1)$

— Svantaggi Per

- trovare archi incidenti al vertice v dobbiamo esaminare tutti gli n elementi nella riga associata a v . Una lista di adiacenza può individuare quegli archi in tempo $O(\deg(V))$ ottimale.
- Aggiungere o rimuovere vertici da un grafo è problematico, poiché la matrice deve essere ridimensionata
- Lo spazio 2) l'utilizzo dello spazio di una matrice di adiacenza è solitamente peggiore $O(n)$ rispetto allo spazio $O(n + m)$ richiesto per le liste di adiacenza, sebbene il numero di archi in un grafo denso sia proporzionale a n^2

Strutture dati per grafici

Operazione	Reg.	elenco bordi	Elenco agg.	Mappa agg.	Matrice
conteggio_vertice()	O(1)	O(1)	O(1)	O(1)	O(1)
edge_count()	O(1)	O(1)	O(1)	O(1)	O(1)
vertici()	SU)	SU)	SU)	SU)	SU)
bordi()	O(m)	O(m)	O(m)	O(m)	O(m)
get_bordo(u,v)	O(m)	O(min(du, dv))	O(1) esp. O(1)		
laurea(v)	O(m)	O(1)	O(1)	SU)	
bordi_incidente(v)	O(m)	O(dv)	O(dv)	SU)	
inserisci_vertice(v)	O(1)	O(1)	O(1)	SU ²)	
rimuovi_vertice(v)	O(m)	O(dv)	O(dv)	SU ²)	
insert_edge(u,v,x)	O(1)	O(1)	O(1) esp. O(1)		
rimuovi_bordo(e)	O(1)	O(1)	O(1) esp. O(1)		

Strutture dati per grafici

Implementazione

vertice **della classe :**

```
def __init__(self, x): self._elemento  
    = x
```

```
def elemento(x):  
    restituisce self._elemento
```

```
def __hash__(self):  
    restituisce hash(id(self))
```

Strutture dati per grafici

Implementazione

```
class Edge: def  
    __init__(self, u, v, x): self._origin = u  
        self._destination = v  
        self._element = x  
  
    def endpoint(self):  
        ritorno (self._origine, self._destinazione)  
  
    def opposto(sé, v):  
        restituisce self._destination se v è self._origin altrimenti self._origin  
  
    def elemento(self):  
        restituisce self._elemento  
  
    def __hash__(self):  
        return hash(self._origin, self._destination)
```

Strutture dati per grafici

Implementazione

grafico **della classe :**

```
def __init__(self, diretto=False):
    self._outgoing = {}
    self._incoming = {} se diretto else self._outgoing

def è_diretto(self):
    return self._incoming non è self._outgoing

def conteggio_vertice(self):
    return len(self._outgoing)

def vertici(self): return
    self._outgoing.keys()
```

Continua ...

Strutture dati per grafici

Implementazione

```
def edge_count(self):
    totale = sum(len(self._outgoing[v]) for v in self.
                 _outgoing)
    restituisce il totale if self.id_directed() else total // 2

def bordi(self): risultato
    = set() for
    secondaria_map in self._outgoing.values():
        risultato.update(secondaria_map.values())
    risultato restituito

def get_edge(self, u, v): return
    self._outgoing[u].get(v)
```

Continua ...

Strutture dati per grafici

Implementazione

```
def grado(self,v, in uscita=Vero):
    adj = self._in uscita if in uscita else self._in entrata return len(adj[v])
```

```
def incidenti_edges(self, v, outgoing=True):
    adj = self._outgoing se in uscita else self._incoming for edge in
    adj[v].values(): yield edge
```

Continua ...

Strutture dati per grafici

Implementazione

```
def insert_vertice(self, x=Nessuno):
    v = self.Vertice(x)
    self._outgoing[v] = {} if
    self.is_directed():
        self._in arrivo[v] = {}
    ritorno

def insert_edge(self, u, v, x=None): e = self.Edge(u, v,
    x) self._outgoing[u][v] = e
    self._incoming[v][u] = e
```

Non continua :-)

Attraversamenti del grafico

Una traversata è una procedura sistematica per esplorare un grafo esaminando tutti i suoi vertici e i suoi bordi. Una traversata è efficiente se visita tutti i vertici e gli spigoli in un tempo proporzionale al loro numero, cioè in tempo lineare

Attraversamenti del grafico

Problemi interessanti che riguardano la raggiungibilità in un grafo G non orientato includono quanto segue:

- Calcolare un percorso dal vertice u al vertice v o segnalare che tale percorso non esiste
- Dato un vertice iniziale s di G , calcolare per ogni vertice v di G un percorso con il numero minimo di archi tra s e v , o segnalare che tale percorso non esiste
- Verifica se G è连通的
- Calcolando uno spanning tree di G , se G è连通的
- Calcolare un ciclo in G o riportare che G non ha cicli

Attraversamenti del grafico

Problemi interessanti che riguardano la raggiungibilità in un grafo orientato G includono quanto segue:

- Calcolare un percorso diretto dal vertice u al vertice v o segnalare che tale percorso non esiste
- Trovare tutti i vertici di G raggiungibili da un dato vertice v
- Determina se G è aciclico
- Determina se G è fortemente connesso

Ricerca in profondità

La ricerca in profondità (DFS) in un grafo G è utile per testare una serie di proprietà dei grafi, incluso se esiste un percorso da un vertice a un altro e se un grafo è connesso o meno

- Iniziamo da uno specifico vertice iniziale $s \in G$
- Supponiamo che in un dato momento il nostro vertice attuale sia u
- Attraversiamo quindi G considerando un bordo arbitrario (u, v) incidente al vertice corrente u
- Se lo spigolo (u, v) conduce a un vertice v già visitato, quel spigolo viene ignorato
- Se invece (u, v) porta ad un vertice non visitato v , il prossimo vertice corrente sarà v
- Il vertice v è contrassegnato come “visitato” ed è considerato come il vertice attuale

Ricerca in profondità

- Eventualmente c'è la possibilità di arrivare ad un "vicolo cieco", cioè ad un vertice attuale v tale che tutti gli archi incidenti a v conducano a vertici già visitati
- Per uscire da questa impasse ripercorriamo a ritroso il bordo che portava a v , riportandoci ad un vertice precedentemente visitato u
- Ora u è il vertice corrente e il calcolo viene ripetuto per tutti gli archi incidenti a u che non sono stati ancora considerati
- Se tutti gli archi incidenti a u sono stati visitati, è necessario tornare indietro fino al vertice che ha portato a u , e ripetere la procedura in quel vertice
- Il processo termina quando il backtrack riconduce al vertice iniziale s , e non ci sono più bordi inesplorati incidenti su s

Ricerca in profondità

Algoritmo DFS(G, u):

Input: un grafico G e un vertice u di G

Output: una raccolta di vertici raggiungibili da u , con i relativi bordi di scoperta

per ogni arco uscente $e = (u,v)$ di u fai se il vertice v non è stato visitato allora

Segna il vertice v come visitato (tramite il bordo e)

Chiama ricorsivamente $\text{DFS}(G, v)$

Ricerca in profondità

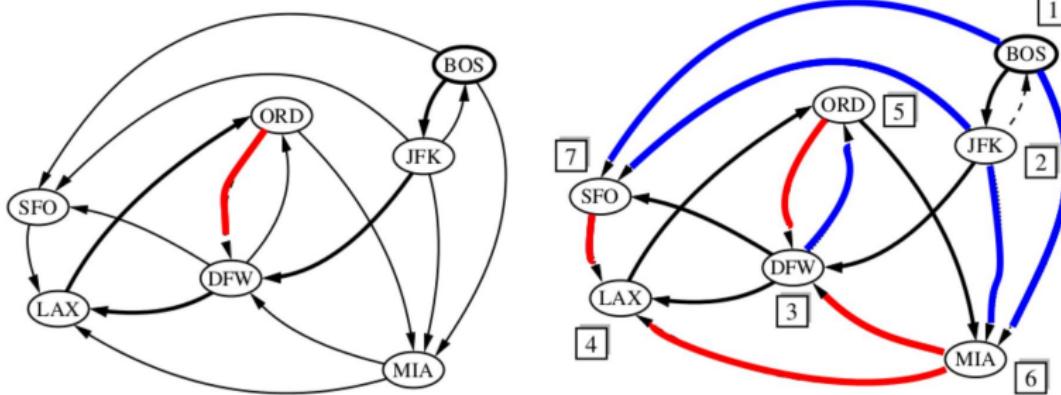
- Un'esecuzione di DFS può essere utilizzata per analizzare la struttura di un grafico
- Il processo DFS identifica naturalmente l'albero di ricerca in profondità radicato in un vertice iniziale s
- Ogni volta che un bordo $e = (u, v)$ viene utilizzato per scoprire un nuovo vertice v durante l'algoritmo DFS, quel bordo è noto come bordo di scoperta o bordo dell'albero
- Tutti gli altri archi considerati durante l'esecuzione di DFS sono noti come archi non ad albero e ci portano ad un vertice precedentemente esplorato
- Nel caso di un grafo non orientato, tutti gli archi non ad albero esplorati collegano il vertice corrente a quello che è un suo antenato nell'albero DFS (albero posteriore)

Ricerca in profondità

Quando si esegue una DFS su un grafico orientato, esistono tre possibili tipi di bordi non ad albero:

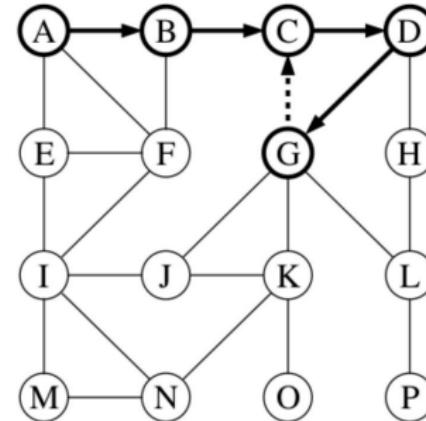
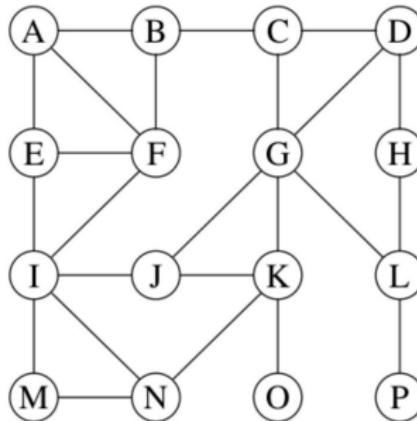
- bordi posteriori, che collegano un vertice al suo antenato nel Albero DFS
- bordi anteriori, che collegano un vertice al suo discendente nell'albero DFS
- bordi incrociati, che collegano un vertice a un vertice che non è né il suo antenato né il suo discendente

Ricerca in profondità



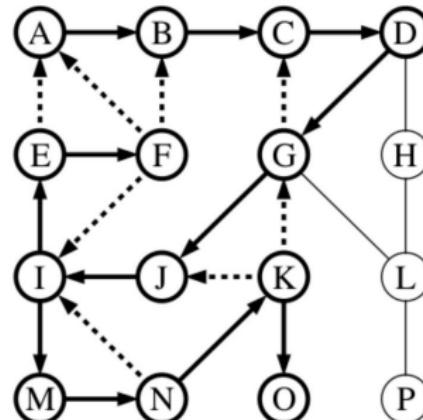
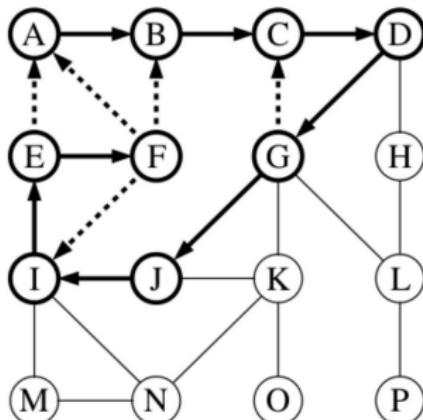
- bordi degli alberi, linee
- spesse bordi posteriori,
- linee rosse bordi anteriori e bordi trasversali, linee blu

Ricerca in profondità



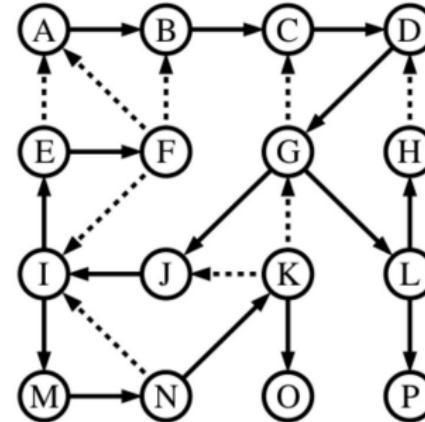
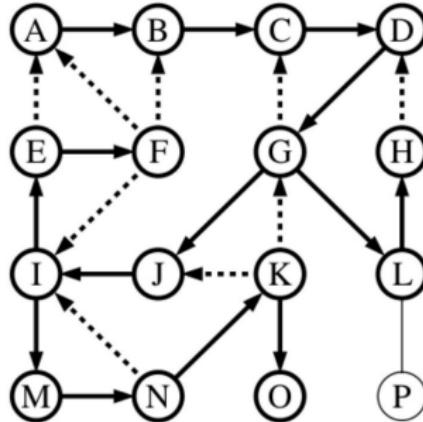
- I vertici visitati e gli spigoli esplorati vengono evidenziati
- I bordi diversi dagli alberi vengono disegnati come linee tratteggiate
- A sinistra: grafico di input
- A destra: attraversamento DFS che inizia dal vertice A e procede in ordine alfabetico (il bordo GC è un bordo non ad albero)

Ricerca in profondità



- A sinistra: il vertice F è un vicolo cieco
- A destra: DFS viene riportato indietro fino al vertice I. Il vertice O è un altro vicolo cieco

Ricerca in profondità



- I vertici visitati e gli spigoli esplorati vengono evidenziati
- I bordi diversi dagli alberi vengono disegnati come linee tratteggiate
- A sinistra: dopo essere tornati indietro verso G, DFS continua con il bordo (G, L) e raggiunge un altro vicolo cieco in H
- A destra: risultato finale

Ricerca in profondità

Proprietà di un DFS

Proposizione

Sia G un grafo non orientato sul quale è stata eseguita una traversata DFS a partire da un vertice s . Quindi l'attraversamento visita tutti i vertici nella componente连通的 di s , e gli archi di scoperta formano uno spanning tree della componente连通的 di s .

Motivazione

Supponiamo che ci sia almeno un vertice w non visitato nella componente连通的 di s , e sia v il primo vertice non visitato su qualche percorso da s a w .

Poiché v è il primo vertice non visitato su questo percorso, ha un vicino u che è stato visitato. Ma quando u è stato visitato, il bordo (u, v) deve essere stato considerato, quindi non può essere corretto che v non sia visitato. Pertanto non ci sono vertici non visitati nella componente连通的 di s .

Ricerca in profondità

Proprietà di un DFS

Proposizione

Sia $\ddot{y}G$ un grafo orientato. Un DFS su $\ddot{y}G$ che parte da un vertice s visita tutti i vertici di $\ddot{y}G$ raggiungibili da s . Inoltre, l'albero DFS contiene percorsi diretti da s a ogni vertice raggiungibile da s .

Motivazione (prima parte)

Sia V_s il sottoinsieme dei vertici di $\ddot{y}G$ visitati da DFS a partire dal vertice s . Vogliamo mostrare che V_s contiene s , e ogni vertice raggiungibile da s appartiene a V_s .

Supponiamo ora che esista un vertice w raggiungibile da s che non è in V_s . Consideriamo un percorso diretto da s a w , e sia (u, v) il primo arco su tale percorso che ci porta fuori da V_s , ovvero $u \in V_s$ ma $v \notin V_s$.

Ricerca in profondità

Proprietà di un DFS

Proposizione

Sia \tilde{G} un grafo orientato. Un DFS su \tilde{G} che parte da un vertice s visita tutti i vertici di \tilde{G} raggiungibili da s . Inoltre, l'albero DFS contiene percorsi diretti da s a ogni vertice raggiungibile da s .

Continua

Quando DFS raggiunge u , esplora tutti gli archi uscenti di u , e quindi deve raggiungere anche il vertice v tramite il bordo (u, v) . Quindi v dovrebbe essere presente e abbiamo ottenuto una contraddizione. Pertanto V_s deve $\forall s$, contenere ogni vertice raggiungibile da s .

Ricerca in profondità

Proprietà di un DFS

Proposizione

Sia \tilde{G} un grafo orientato. Un DFS su \tilde{G} che parte da un vertice s visita tutti i vertici di \tilde{G} raggiungibili da s . Inoltre, l'albero DFS contiene percorsi diretti da s a ogni vertice raggiungibile da s .

Motivazione (seconda parte)

La seconda parte si dimostra per induzione sui passi dell'algoritmo. Ogni volta che viene identificato un arco di rilevamento (u, v) , esiste un percorso diretto da s a v nell'albero DFS. Poiché u deve essere stato scoperto in precedenza, esiste un percorso da s a u , quindi aggiungendo lo spigolo (u, v) a quel percorso, abbiamo un percorso diretto da s a v .

Ricerca in profondità

Tempo di esecuzione di DFS

- DFS è un metodo efficiente per attraversare un grafo DFS
- viene chiamato al massimo una volta su ciascun vertice (poiché viene contrassegnato come visitato), e quindi ogni bordo viene esaminato al massimo due volte per un grafo non orientato (una volta per ciascuno dei suoi vertici), e al massimo una volta per un grafo orientato (dal vertice di origine)
- Se lasciamo che $ns \leq n$ sia il numero di vertici raggiungibili da un vertice s , e $ms \leq m$ sia il numero di archi incidenti verso quei vertici, un DFS che inizia da s viene eseguito in tempo $O(ns + ms)$, a condizione che

Ricerca in profondità

Tempo di esecuzione di DFS

...

- Il grafico è rappresentato da una struttura dati tale che la creazione e l'iterazione attraverso `incident_edges(v)` richiede tempo $O(\deg(v))$ e il metodo `e.opposite(v)` richiede tempo $O(1)$. La lista di adiacenza è una di queste strutture (non la matrice di adiacenza)
- Esiste un modo per contrassegnare un vertice o uno spigolo come esplorato e per verificare se un vertice o uno spigolo è stato esplorato in tempo $O(1)$

Date queste premesse è possibile risolvere una serie di problemi

Ricerca in profondità

Proprietà di un DFS

Proposizione

Sia G un grafo non orientato con n vertici e m archi. Un attraversamento DFS di G può essere eseguito in tempo $O(n + m)$ e può essere utilizzato per risolvere i seguenti problemi in tempo $O(n + m)$:

- Calcolare un percorso tra due dati vertici di G , se ne esiste uno
- Verifica se G è connesso
- Calcolare uno spanning tree di G , se G è connesso
- Calcolo delle componenti connesse di G
- Calcolare un ciclo in G o riportare che G non ha cicli

Ricerca in profondità

Proprietà di un DFS

Proposizione

Sia \tilde{G} un grafo orientato. Può essere una traversata DFS all'avvio \tilde{G} eseguita in tempo $O(n + m)$ e può essere utilizzata per risolvere la seguenti problemi in tempo $O(n+m)$:

- Calcolare un percorso diretto tra due vertici dati di \tilde{G} , Se ne esiste uno
- Calcolare l'insieme dei vertici di \tilde{G} raggiungibili da a dato il vertice s
- Verificare se \tilde{G} è fortemente connesso
- Calcolare un ciclo diretto in \tilde{G} o riportare che \tilde{G} lo è aciclico
- Calcolando la chiusura transitiva di \tilde{G}

Ricerca in profondità

Implementazione

```
def DFS(g, u, scoperto):
    """ G           nodo
        tu          iniziale
        scoperto: dizionario
    """
    for e in g.incident_edges(u):
        v = e.opposite(u)
        if v non è stato scoperto:
            scoperto[v] = e
            DFS(g, v, scoperto)
```

Ricerca in profondità

Ricostruire un percorso da u a v

Per ricostruire il percorso che collega u a v:

- ① Vai alla fine del percorso (vale a dire, considera il vertice v)
- ② Nome vertice v cammino
- ③ Esaminare il dizionario per determinare a quale bordo è stato utilizzato
raggiungere a piedi
- ④ Determinare l'altro punto finale del bordo, diciamo il vertice w
- ⑤ Nome vertice w genitore
- ⑥ Aggiungi il vertice genitore al percorso
- ⑦ Dai un nome alla passeggiata del genitore
- ⑧ Ripetere i passaggi 3-7 finché parent = u

Ricerca in profondità

Ricostruire un percorso da u a v

```
def percorso_costrutto(u, v, scoperto):
    percorso =
    [] se v in scoperto:
        percorso.append(v)
        cammina = v
        mentre camminare non sei tu:
            e = scoperto[cammina]
            genitore = e.opposto(cammina)
            percorso.append(genitore)
            cammina = genitore
        path.reverse()
    restituisce il percorso
```

Ricerca in profondità

Test di connettività

- DFS può essere utilizzato per determinare se un grafico è connesso
- Per un grafo non orientato, dopo una DFS dobbiamo verificare che $\text{len}(\text{discovered}) == n$
- Se il grafo è connesso tutti i vertici saranno stati scoperti
- In caso contrario, deve esserci almeno un vertice v che non è raggiungibile da v

Ricerca in profondità

Test di connettività

- Per i grafi orientati, DFS permette di verificare se è fortemente connesso, cioè se per ogni coppia di vertici u e v , sia u raggiunge v che v raggiunge u
- In linea di principio il test dovrebbe essere eseguito per ogni vertice, e quindi il costo computazionale sarebbe $O(n(m + n))$
- Fortunatamente è sufficiente eseguire solo due DFS, la seconda con l'orientamento di tutti i bordi invertito

Ricerca in profondità

Calcolo di tutti i componenti connessi

Quando un grafo non è连通的, è spesso utile identificare tutte le componenti connesse di un grafo non orientato, o le componenti fortemente connesse di un grafo orientato

```
def DFS_complete(g): foresta
    = {} for u in
        g.vertices(): se non sei nella
            foresta:
                foresta[u] = Nessuno
                DFS(g, u, foresta)
            foresta di ritorno
```

Ricerca in profondità

Calcolo di tutti i componenti connessi

- Sebbene la funzione DFS_complete effettui più chiamate a DFS, il tempo totale impiegato da una chiamata è $O(n + m)$
- In effetti, una singola chiamata a DFS che inizia da vertex s viene eseguita nel tempo $O(ns + ms)$, dove ns è il numero di vertici raggiungibili da s, e ms è il numero di archi incidenti a quei vertici
- Ogni chiamata a DFS esplora un componente diverso, la somma di ns + ms è n + m
- Si noti che $O(n + m)$ si applica anche ai grafi orientati, anche se gli insiemi dei vertici raggiungibili non sono necessariamente disgiunti

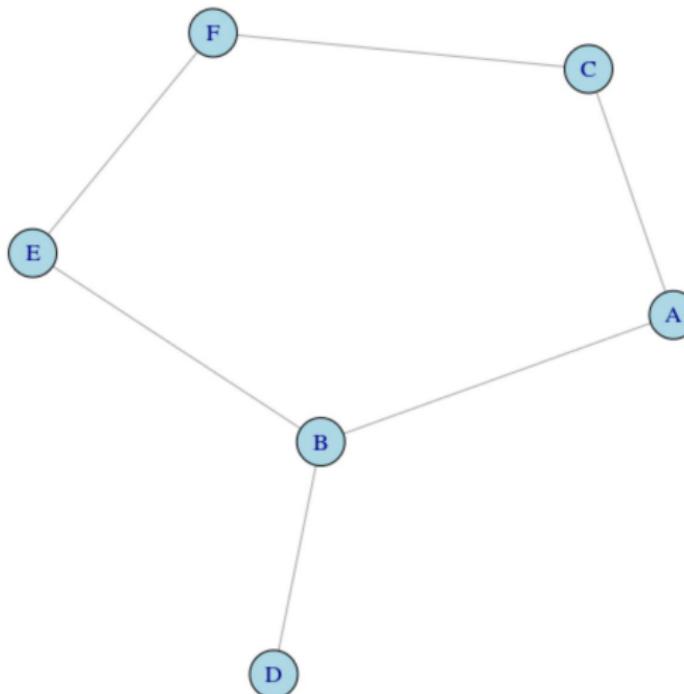
Ricerca in profondità

Rilevamento dei cicli

- Sia per i grafi diretti che per quelli non orientati, esiste un ciclo se e solo se esiste un bordo posteriore relativo all'attraversamento DFS di quel grafo
- Se esiste un bordo posteriore, esiste un ciclo portando il bordo posteriore dal discendente al suo antenato e poi seguendo i bordi dell'albero fino al discendente
- Al contrario, se nel grafico esiste un ciclo, deve esserci un bordo posteriore relativo a un DFS
- In un grafo non orientato, tutti gli archi sono alberi o archi posteriori, quindi trovare un ciclo è relativamente facile
- In un grafo orientato, quando l'esplorazione porta ad un vertice precedentemente visitato, è necessario riconoscere se quel vertice è un antenato del vertice attuale

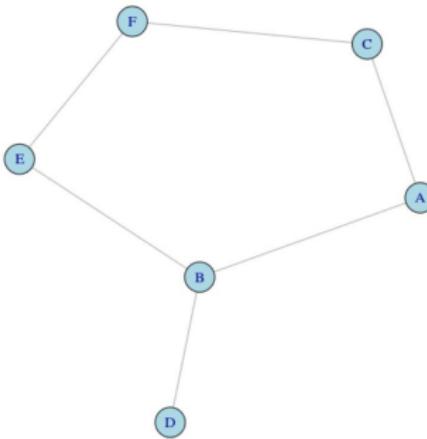
Ricerca in profondità

Implementazioni



Ricerca in profondità

Implementazioni



```
grafico = {'A': set(['B', 'C']),  
          'B': set(['A', 'D', 'E']),  
          'C': set(['A', 'F']),  
          'D': set(['B']),  
          'E': set(['B', 'F']),  
          'F': set(['C', 'E'])}
```

Ricerca in profondità

Implementazioni

```
def dfs(grafico, inizio):
    visitato, stack = set(), [start] while stack:

        vertice = stack.pop() se il
        vertice non è visitato:
            visitato.add(vertice)
            stack.extend(grafico[vertice] - visitato)
    ritorno visitato
```

Ricerca in profondità

Implementazioni

Iteration 0

```
Found node A connected to {'C', 'B'}
Already visited nodes {'A'}
Added nodes {'C', 'B'}
Stack contains ['C', 'B']
```

Iteration 1

```
Found node B connected to {'A', 'E', 'D'}
Already visited nodes {'A', 'B'}
Added nodes {'E', 'D'}
Stack contains ['C', 'E', 'D']
```

Iteration 2

```
Found node D connected to {'B'}
Already visited nodes {'A', 'B', 'D'}
Added nodes set()
Stack contains ['C', 'E']
```

Iteration 3

```
Found node E connected to {'B', 'F'}
Already visited nodes {'A', 'B', 'E', 'D'}
Added nodes {'F'}
Stack contains ['C', 'F']
```

Iteration 4

```
Found node F connected to {'C', 'E'}
Already visited nodes {'A', 'B', 'E', 'D', 'F'}
Added nodes {'C'}
Stack contains ['C', 'C']
```

Iteration 5

```
Found node C connected to {'A', 'F'}
Already visited nodes {'A', 'C', 'B', 'E', 'D', 'F'}
Added nodes set()
Stack contains ['C']
```

Iteration 6

```
Found a vertex already visited: C
```

Ricerca in profondità

Realizzazione: percorso

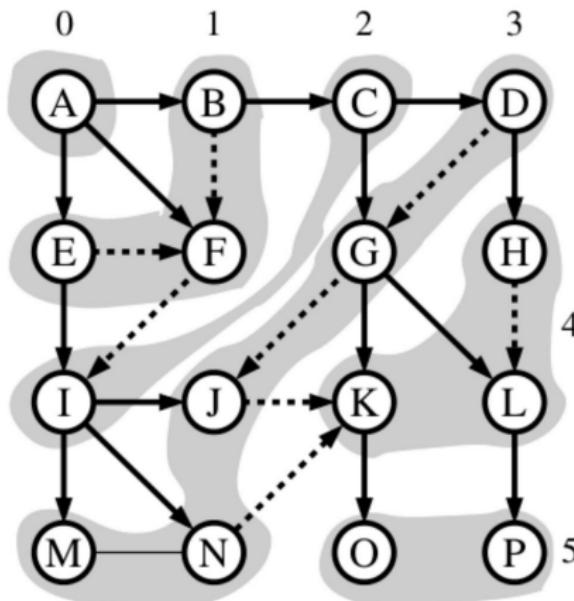
```
def dfs_paths(grafico, inizio, obiettivo): stack =  
[(inizio, [inizio])] while stack:  
  
    (vertice, percorso) = stack.pop() for  
    next in graph[vertice] - set(path): if next == goal: yield  
        path + [next]  
  
    altro:  
        stack.append((successivo, percorso + [successivo]))
```

Ricerca in profondità

Realizzazione: percorso

```
giacomo@giacomo-N76VB2:~/Dropbox/Teach/AACM/Lectures-20152016/examples$ python dfs_path
Stack contains:  [(['C', ['A', 'C']])
['A', 'C']
Stack contains:  [(['C', ['A', 'C']], ('B', ['A', 'B']))
['A', 'B']
Stack contains:  [(['C', ['A', 'C']], ('E', ['A', 'B', 'E']))
['A', 'B', 'E']
Stack contains:  [(['C', ['A', 'C']], ('E', ['A', 'B', 'E']), ('D', ['A', 'B', 'D']))
['A', 'B', 'D']
['A', 'B', 'E', 'F']
['A', 'C', 'F']
```

Ricerca in ampiezza



Ricerca in ampiezza

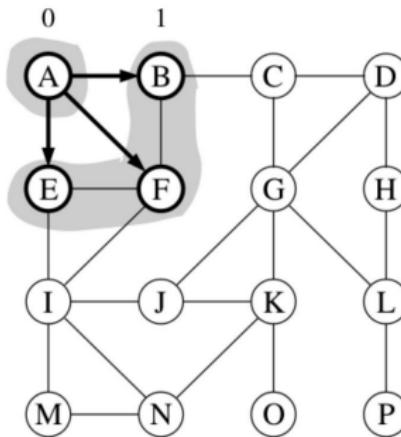
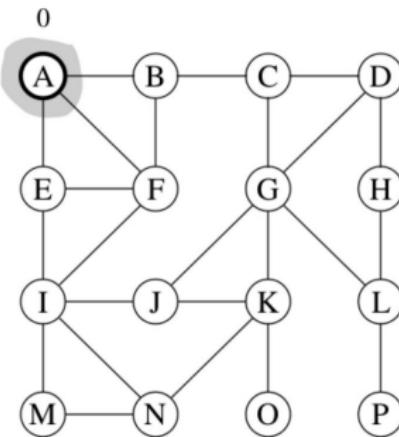
- DFS definisce un attraversamento che potrebbe essere tracciato fisicamente da una singola persona
- che esplora il grafico. Breadth-First Search (BFS) può essere descritto come l'invio di molti esploratori che esplorano collettivamente il grafico (niente a che fare con le implementazioni parallele)
- Un BFS procede per giri e divide i vertici in livelli. Nel primo giro,
- vengono contrassegnati come “visitati” tutti i vertici adiacenti al vertice iniziale s (questi vertici sono ad un passo dall'inizio e sono posizionati nel livello 1)
- Nel secondo turno, le esplorazioni riguardano i vertici a due passi (bordi) di distanza dal vertice iniziale. I nuovi vertici sono adiacenti ai vertici del livello 1 e vengono quindi posizionati nel livello 2 Il processo continua in modo simile terminando quando non vengono trovati nuovi vertici in un livello

Ricerca in ampiezza

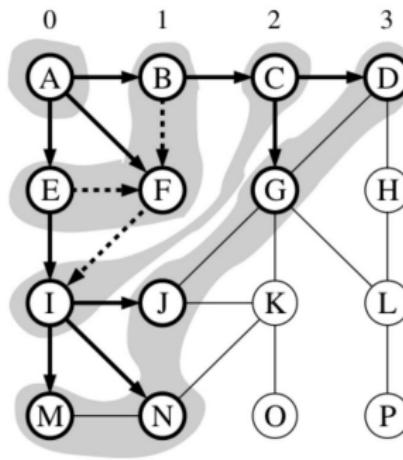
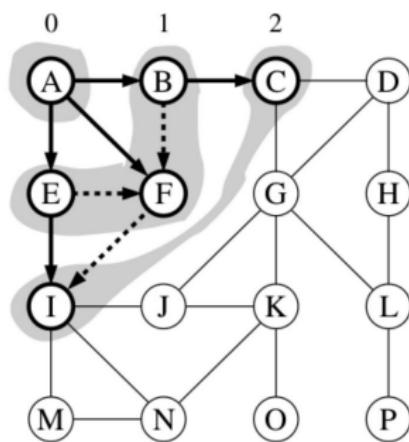
Implementazione

```
def BFS(g, s, scoperto): : grafico :  
    """  
        G           nodo  
        s           iniziale  
        scoperto: dizionario  
    """  
  
    level = [s] while  
    len(level) > 0: next_level = []  
        for u in level:  
  
            for e in g.incident_edges(u): v =  
                e.opposite(u) se v non è  
                scoperto:  
                    scoperto[v] = e  
                    next_level.append(v)  
livello = livello_successivo
```

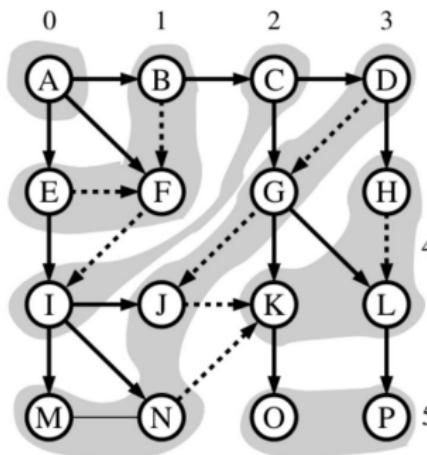
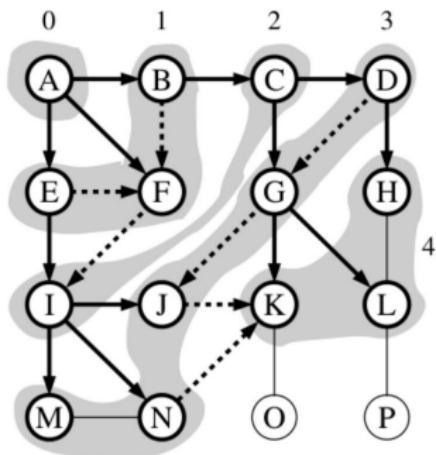
Ricerca in ampiezza



Ricerca in ampiezza



Ricerca in ampiezza



Ricerca in ampiezza

Glossario

Tipo di bordo Collega...

bordo un vertice a uno dei suoi antenati

posteriore bordo un vertice a uno se discendenti

anteriore bordo incrociato un vertice ad un altro vertice (né antenato né discendente)

Per BFS:

- sui grafi non orientati, tutti gli archi non ad albero sono archi incrociati
- sui grafici diretti, tutti gli spigoli non ad albero sono spigoli posteriori appassiti o bordi incrociati

Ricerca in ampiezza

Proprietà

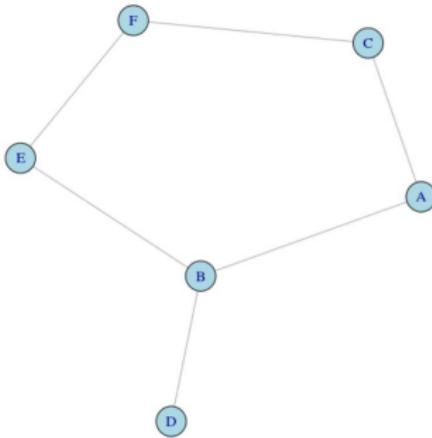
Proposizione

Sia G un grafo non orientato o orientato su cui è stata eseguita una traversata BFS a partire dal vertice s . Poi

- La traversata visita tutti i vertici di G raggiungibili da s
- Per ogni vertice v al livello i , il percorso dell'albero BFS T tra s e v ha i bordi, e qualsiasi altro percorso di G da s a v ha almeno i bordi
- Se (u, v) è un arco che non è nell'albero BFS, allora il numero di livello di v può essere al massimo 1 maggiore del numero di livello di u

Ricerca in ampiezza

Un'altra implementazione



```
grafico = {'A': set(['B', 'C']),
           'B': set(['A', 'D', 'E']),
           'C': set(['A', 'F']),
           'D': set(['B']),
           'E': set(['B', 'F']),
           'F': set(['C', 'E'])}
```

Ricerca in ampiezza

Un'altra implementazione

```
def bfs(grafico, inizio):
    visitato, coda = set(), [inizio] while coda:
        vertice =
            coda.pop(0) se vertice non in
            visitato:
                visitato.add(vertice)
                coda.extend(grafico[vertice] - visitato)
    ritorno visitato
```

Ricerca in ampiezza

Un'altra implementazione

```
Iteration 0
Found node A connected to {'C', 'B'}
Already visited nodes {'A'}
Added nodes {'C', 'B'}
Stack contains ['C', 'B']
```

```
Iteration 1
Found node C connected to {'A', 'F'}
Already visited nodes {'A', 'C'}
Added nodes {'F'}
Stack contains ['B', 'F']
```

```
Iteration 2
Found node B connected to {'A', 'E', 'D'}
Already visited nodes {'A', 'C', 'B'}
Added nodes {'E', 'D'}
Stack contains ['F', 'E', 'D']
```

```
Iteration 3
Found node F connected to {'C', 'E'}
Already visited nodes {'A', 'C', 'B', 'F'}
Added nodes {'E'}
Stack contains ['E', 'D', 'E']
```

```
Iteration 4
Found node E connected to {'B', 'F'}
Already visited nodes {'A', 'C', 'B', 'E', 'F'}
Added nodes set()
Stack contains ['D', 'E']
```

```
Iteration 5
Found node D connected to {'B'}
Already visited nodes {'A', 'C', 'B', 'E', 'D', 'F'}
Added nodes set()
Stack contains ['E']
```

```
Iteration 6
Found a vertex already visited: E
```

Chiusura transitiva

La definizione matematica

- La chiusura transitiva di una relazione binaria R su un insieme X è la relazione transitiva $R \dagger$ sull'insieme X tale che $R \dagger$ contiene R e $R \dagger$ è minimo
- Se la relazione stessa è transitiva, allora la chiusura transitiva è la stessa relazione binaria
- Altrimenti, la chiusura transitiva è una relazione diversa

Se X è un insieme di aeroporti e xRy significa "c'è un volo diretto dall'aeroporto x all'aeroporto y ", allora la chiusura transitiva di R su X è la relazione $R \dagger$: "è possibile volare da x a y in uno o più voli."

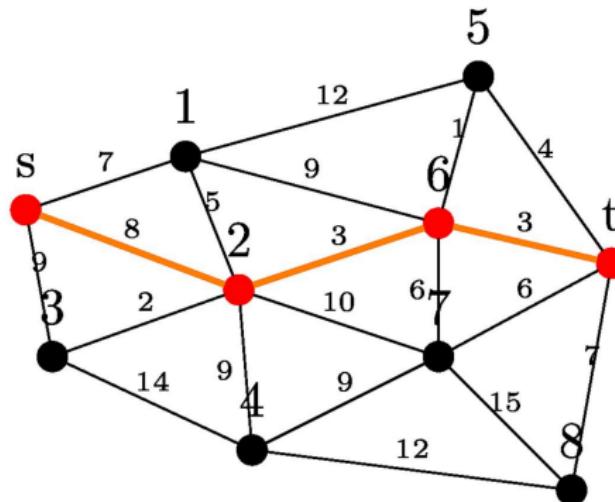
Chiusura transitiva

- Gli attraversamenti del grafico possono essere utilizzati per rispondere a domande di base sulla raggiungibilità in un grafico diretto
- In particolare, DFS o BFS possono essere utilizzati per sapere se esiste un percorso dal vertice u al vertice v
- Se il grafo è rappresentato con una lista di adiacenze, la questione della raggiungibilità può essere calcolata in tempo $O(n + m)$
- In alcune applicazioni, la raggiungibilità può essere calcolata in modo più efficiente, ad esempio precalcolando una rappresentazione più conveniente di un grafico
- Una possibile soluzione è rappresentata dalla chiusura transitiva di un grafo orientato $\ddot{y}yG$

Chiusura transitiva

- La chiusura transitiva di un grafo diretto \tilde{G} è un grafo diretto \tilde{G} tale che i vertici di \tilde{G} e \tilde{G} hanno gli stessi di, un arco (u, v) ogni volta che \tilde{G} ha vertici di \tilde{G} percorso diretto da u a v
- Se il grafo è rappresentato come una lista di adiacenze, la sua chiusura transitiva può essere calcolata in tempo $O(n + m)$ utilizzando n attraversamenti del grafo, uno per ciascun vertice iniziale

Il percorso più breve



L'algoritmo di Floyd-Warshall

- L'algoritmo di Floyd-Warshall (FW) è una soluzione al problema dei cammini minimi di tutte le coppie su un grafo orientato $G = (V, E)$
- Funziona in $O(V^3)$ tempo
- L'algoritmo FW considera i vertici intermedi di un cammino minimo, dove un vertice intermedio di un cammino semplice $p = (v_1, v_2, \dots, v_l)$ è qualsiasi vertice di p diverso da v_1 o v_l , cioè qualsiasi vertice in $\{v_2, v_3, \dots, v_l\}$

L'algoritmo di Floyd-Warshall

- Supponiamo che i vertici di G siano $V = \{1, 2, \dots, N\}$
- Consideriamo un sottoinsieme $\{1, 2, \dots, k\}$ di vertici per alcuni k . Sia p un percorso di peso minimo tra loro.
- L'algoritmo FW sfrutta una relazione tra il percorso per i cammini minimi da i a j con tutti i vertici intermedi nell'insieme $\{1, 2, \dots, k - 1\}$.
- La relazione dipende dal fatto che k sia o meno un vertice intermedio del percorso p .

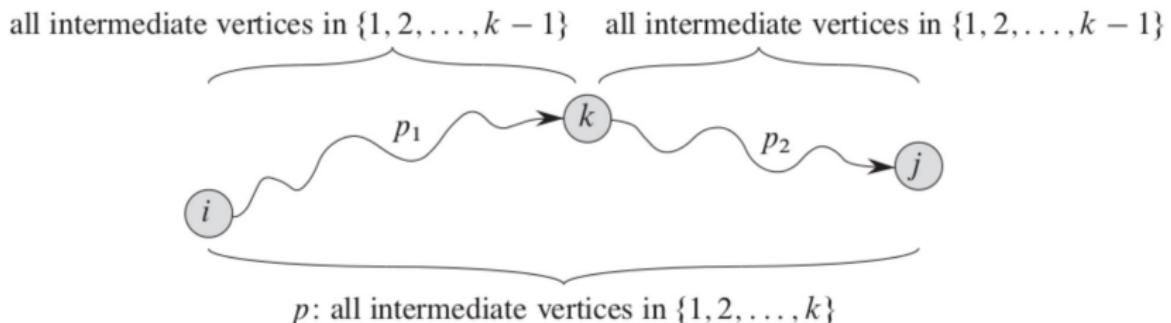
L'algoritmo di Floyd-Warshall

- Se k non è un vertice intermedio del cammino p , allora tutti i vertici intermedi del cammino p appartengono all'insieme $\{1, 2, \dots, k - 1\}$
- Pertanto, un percorso minimo dal vertice i al vertice j con tutti i vertici intermedi nell'insieme $\{1, 2, \dots, k - 1\}$ è anche un cammino minimo da i a j con tutti i vertici intermedi nell'insieme $\{1, 2, \dots, K\}$

L'algoritmo di Floyd-Warshall

- Se k è un vertice intermedio del cammino p , allora il cammino p può essere scomposto in due cammini p_1 (da i a k) e p_2 (da k a j)
 p_1 è
 - un cammino minimo da i a k con tutti i vertici intermedi nel impostare $\{1, 2, \dots, k - 1\}$
 - Allo stesso modo, p_2 è un cammino minimo da k a j con tutti i vertici intermedi nell'insieme $\{1, 2, \dots, k - 1\}$

L'algoritmo di Floyd-Warshall



L'algoritmo di Floyd-Warshall

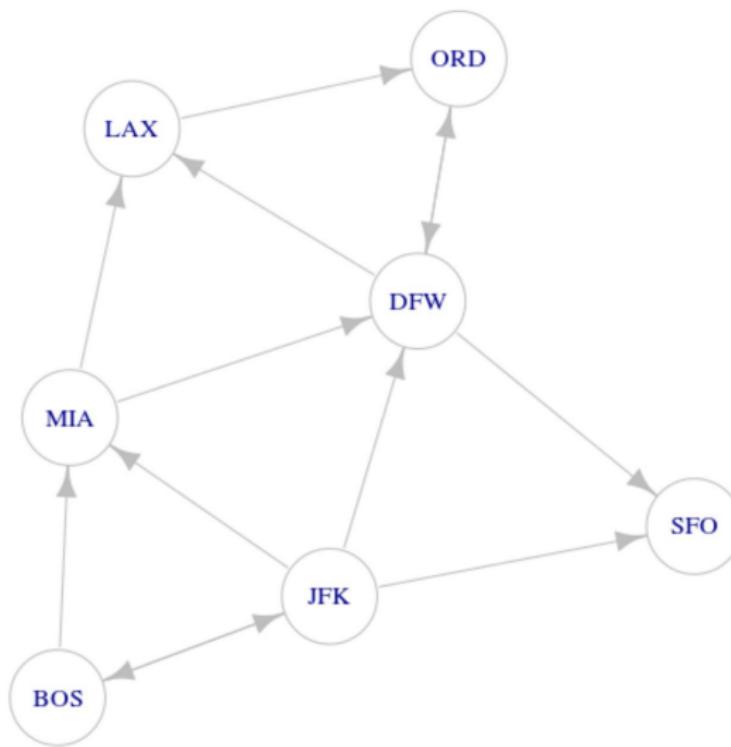
Implementazione

```
def floyd_marshall(g):
    chiusura = deepcopy(g) verts
    = list(closure.vertices()) n = len(verts) for k in
    range(n):

        per i nell'intervallo(n):
            if i != k e chiusura.get_edge(verts[i], verts[k])
                non è Nessuno:
                    per j in range(n): if i != j !=
                        k and closing.get_edge(verts[k],
                        verts[j]) non è Nessuno:
                            se closing.get_edge(verts[i], verts[j]) lo è
                                Nessuno:
                                    chiusura.insert_edge(verts[i], verts[j])
                chiusura del reso
```

L'algoritmo di Floyd-Warshall

Un esempio



L'algoritmo di Floyd-Warshall

Un esempio

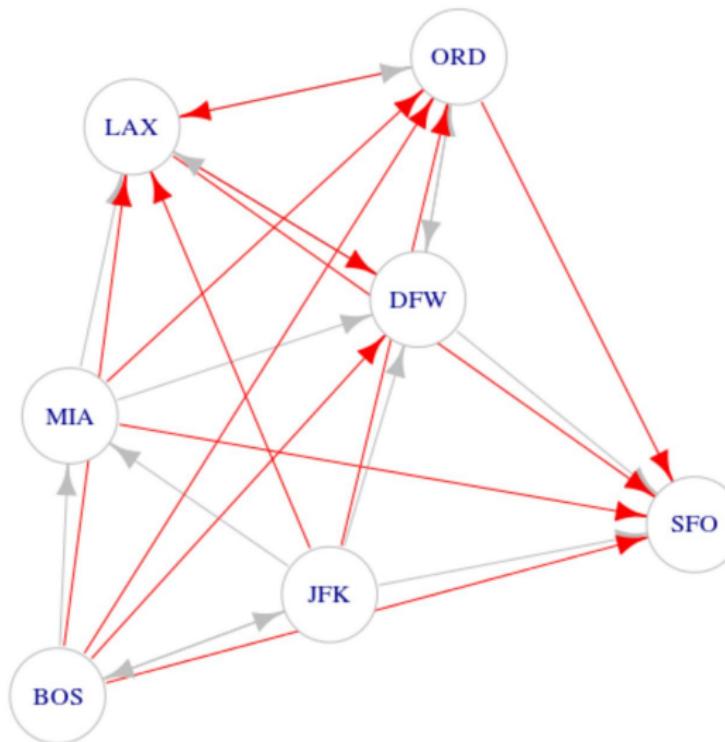


Grafico aciclico diretto

I grafi diretti senza cicli diretti sono abbastanza comuni. Per esempio:

- Prerequisiti tra insegnamenti di un corso di laurea
- Ereditarietà tra classi di un programma orientato agli oggetti
- Ereditarietà tra directory in un file system

Grafico aciclico diretto

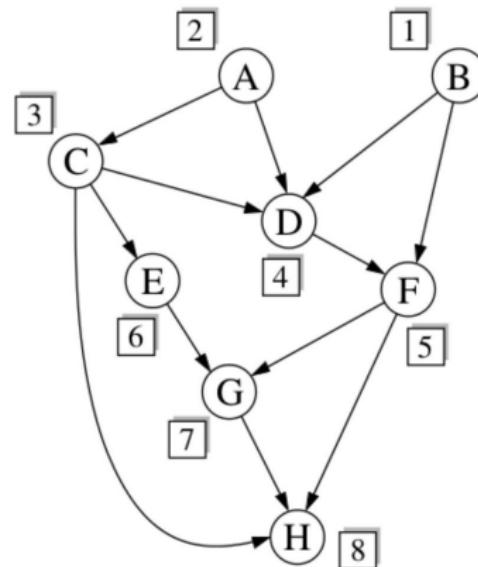
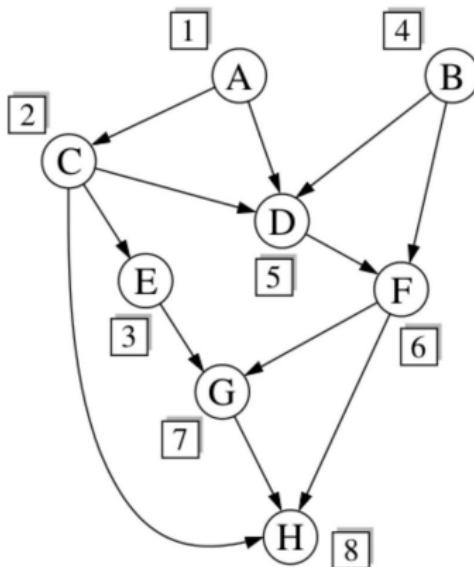


Grafico aciclico diretto

Ordinamento topologico

Sia $\ddot{y}G$ un grafo diretto con n vertici. Un ordinamento topologico di $\ddot{y}G$ è un ordinamento v_1, \dots, v_n dei vertici di $\ddot{y}G$ tale che per ogni spigolo (v_i, v_j) di $\ddot{y}G$, è il caso che $i < j$.

Un ordinamento topologico è un ordinamento tale che qualsiasi percorso diretto in $\ddot{y}G$ attraversa i vertici in ordine crescente (un ordinamento topologico non è necessariamente unico)

Proposizione

$\ddot{y}G$ ha un ordinamento topologico se e solo se è aciclico.

Grafico aciclico diretto

Ordinamento topologico

L'ultima Proposizione suggerisce un algoritmo per calcolare un ordinamento topologico di un grafo orientato, chiamato ordinamento topologico

L'implementazione utilizza un dizionario di conteggio per mappare ciascun vertice v su un contatore che rappresenta il numero corrente di archi in entrata su v , escludendo quelli provenienti da vertici che sono stati precedentemente aggiunti all'ordine topologico.

Grafo aciclico diretto - Ordinamento topologico

```
def ordinamento_topologico(g):
    topo = []
    ready = []
    incount = {} for
    u in g.vertices():
        incount[u] = g.degree(u, False) if incount[u]
        == 0: ready.append(u)

    while len(ready) > 0: u =
        ready.pop()
        topo.append(u) for
        e in g.incident_edges(u): v = e.opposite(u)
        incount[v] -= 1 if
        incount[v ] == 0:
            ready.append(v) restituisce
                la topografia
```

Grafico aciclico diretto

Ordinamento topologico

Proposizione

Sia \mathcal{G} un grafo diretto con n vertici e m spigoli, utilizzando una rappresentazione con lista di adiacenza. L'algoritmo di ordinamento topologico viene eseguito in tempo $O(n + m)$ utilizzando lo spazio ausiliario $O(n)$ e calcola un ordinamento topologico di \mathcal{G} oppure non riesce a includere alcuni vertici, il che indica che \mathcal{G} ha un ciclo diretto.

Grafico aciclico diretto

Ordinamento topologico

```
def ordinamento_topologico(g):
    topo = []
    ready = []
    incount = {} for
    u in g.vertices():
        incount[u] = g.degree(u,
            False) if
        incount[u] == 0:
            ready.append(u)
    while len(ready) > 0: u =
        ready.pop()
        topo.append(u) for
        e nel g:
            bordi_incidente(u):
                v = e.opposite(u)
                incount[v] -= 1 if
                incount[v] == 0:
                    ready.append(v)
    restituire la tabella
```

- La registrazione iniziale degli n gradi utilizza il tempo $O(n)$ in base al metodo dei gradi
- Un vertice u viene visitato dall'algoritmo di ordinamento topologico quando u viene rimosso dalla ready list
- Un vertice u può essere visitato solo quando $\text{incount}(u)$ è zero, cioè tutti i suoi predecessori (vertici che escono in u) sono stati precedentemente visitati
- Tutti i bordi uscenti di ciascun vertice visitato vengono attraversati una volta, quindi il suo tempo di esecuzione è proporzionale al numero di bordi uscenti dei vertici visitati

