Complessità e algoritmi

In questa lezione:

- Algoritmo di Euclide per il MCD
- Ricerca sequenziale
- Ricerca binaria
- Introduzione ai grafi
- Introduzione agli alberi e loro rappresentazione
- Algoritmi di visita di alberi

Algoritmo di Euclide per il MCD

Uno dei più antichi algoritmi a noi pervenuti è l'algoritmo per trovare il massimo comune divisore (MCD) tra due numeri interi, presente negli Elementi, libro del 300 a.C. di Euclide di Alessandria.

L'algoritmo, nella versione originale, si basa sull'osservazione che il MCD deve dividere anche la differenza dei due numeri.

Infatti, posto d il MCD tra a e b, allora se $d|a$ (leggi: d divide a) e $d|b$ si ha che $\frac{a}{d} - \frac{b}{d}$ è un numero intero. Ma $\frac{a}{d} - \frac{b}{d} = \frac{a-b}{d}$, quindi anche $\frac{a-b}{d}$ è intero, cioè d divide $a - b$. Segue che si può cercare d tra $a - b$ e b invece di cercarlo tra a e b.
Inoltre si può riapplicare l'osservazione ricorsivamente a $a - b$ e b e fermarsi quando si trovano due numeri uguali: il loro valore è d.

Algoritmo di Euclide per il MCD

Nella versione moderna: *dati due numeri naturali a e b, si controlla se b è zero. Se lo è, a è il MCD. Se non lo è, si divide a per b e si assegna ad r il resto della divisione. Se r = 0 allora si può terminare affermando che b è il MCD cercato, altrimenti si assegna a = b e b = r e si ripete nuovamente la divisione.*

La versione moderna si basa sull'osservazione che il MCD divide il resto r della divisione di a per b. Infatti, sia c il quoziente della divisione e d il MCD cercato.
Allora $a = c \cdot b + r$. Quindi $\frac{a}{d} = \frac{c \cdot b + r}{d}$, cioè $\frac{a}{d} = c \cdot \frac{b}{d} + \frac{r}{d}$. Ma d divide sia a che b quindi $\frac{a}{d}$ e $\frac{b}{d}$ sono interi. Ne segue che $\frac{r}{d}$ è intero, cioè d divide r.

Poiché i resti delle divisioni sono sempre più piccoli, allora si arriverà a $r = 0$ e l'algoritmo termina.

Complessità temporale dell'Algoritmo di Euclide

Si può dimostrare che nel caso peggiore il numero delle divisioni è proporzionale al numero di cifre n di a (assumendo $a > b$). Da notare che $2n$ è la misura dell'input dell'algoritmo (cioè quanti dati al massimo dobbiamo specificare).

Poiché n è proporzionale a $\log(a)$ possiamo dire la complessità temporale dell'algoritmo di Euclide è $O(n) = O(\log a)$.

Confrontiamo questo tempo con il tempo dell'algoritmo ingenuo di trovare tutti i divisori di a e di b per prova, cioè dividendo a (e b) per tutti i numeri minori di a (e di b), e poi vedendo il massimo in comune. Le divisioni da provare sono $O(a) + O(b) = O(a) + O(a) = O(2a) = O(a)$.

Ma se n è circa $\log(a)$ allora a è circa 10^n. Quindi l'algoritmo ingenuo ha complessità $O(a) = O(10^n)$, cioè è esponenziale nella misura dell'input! Per numeri grandi non va bene: molto meglio l'algoritmo di Euclide.

Implementazione in Python dell'Algoritmo di Euclide

L'algoritmo di Euclide ha una implementazione molto elegante in Python:

```
def MCD(a,b):
    while b != 0:
        a, b = b, a % b
    return a
```

Si confronti questo codice con i codici scritti in altri linguaggi reperibili in rete (ad esempio alla voce "Algoritmo di Euclide" su Wikipedia).

E' interessante notare che l'algoritmo di Euclide non richiede la fattorizzazione dei due interi.
Per quest'ultimo problema (trovare un fattore di un numero dato) ad oggi non esiste nessun algoritmo efficiente, cioè la cui complessità temporale sia limitata da un polinomio di grado k delle n cifre del numero dato, in breve $O(n^k)$.

Algoritmi di ricerca

Supponiamo di dover cercare una parola in un vocabolario. Anche se il vocabolario include decine di migliaia di parole, la ricerca sarà veloce.

Il motivo è che le parole sono ordinate seguendo l'ordine alfabetico.

Supponiamo ora di avere un ipotetico vocabolario in cui le parole sono inserite senza un particolare ordine. Come trovare una parola data?

Non ci resta che scorrere il vocabolario parola per parola e confrontarla con quella data, impiegando moltissimo tempo, ma non possiamo far altro.

Questa ricerca si chiama ricerca sequenziale o ricerca lineare e si applica necessariamente quando abbiamo una lista di dati non ordinati all'interno della quale cercare un elemento.

Ricerca sequenziale

Affrontiamo il problema specifico di trovare la posizione di un numero dato una lista di numeri naturali. Se il numero è presente nella lista, l'algoritmo deve ritornare l'indice della posizione del numero altrimenti -1.
Poiché gli elementi della lista non sono ordinati, qualsiasi algoritmo impiegherà un tempo $\Omega(n)$ perché, nel caso peggiore, il numero cercato è l'ultimo elemento a cui si accede nella lista, oppure il numero cercato non è presente nella lista: in questi casi bisogna fare almeno n accessi alla lista.
Diamo il codice del programma Python con complessità temporale $O(n)$.

```
## Finds a value in a list, using the linear search algorithm.
#  @param values the list to search.
#  @param target the value to find
#  @return the index where the target occurs, or -1 if not in the list

def linearSearch(values, target) :
   for i in range(len(values)) :
      if values[i] == target :
         return i

   return -1
```


Ricerca sequenziale

```
##
#  This program demonstrates the linear search algorithm.
#

from random import randint
from linearsearch import linearSearch

# Construct random list.
n = 20
values = []
for i in range(n) :
   values.append(randint(1, 100))
print(values)

done = False
while not done :
   target = int(input("Enter number to search for, -1 to quit: "))
   if target == -1 :
      done = True
   else :
      pos = linearSearch(values, target)
      if pos == -1 :
         print("Not found")
      else :
         print("Found in position", pos)
```

Ricerca binaria

Cosa possiamo fare nel caso in cui la lista è ordinata?
Possiamo sempre fare la ricerca sequenziale, ma possiamo fare meglio!
Supponiamo di chiederci se 15 è nella lista:

1	4	5	8	9	12	17	20	24	31

Restingiamo la ricerca chiedendoci se 15 è nella prima o seconda metà della lista. Poiché il valore più alto della prima metà è 9, cerchiamo nella seconda metà:

1	4	5	8	9	12	17	20	24	31

Poiché l'elemento centrale della seconda metà è 20, l'elemento deve essere cercato nella sequenza evidenziata:

1	4	5	8	9	12	17	20	24	31

Ricerca binaria

Poiché in

1	4	5	8	9	12	17	20	24	31

la prima metà ha valore più alto 12, il valore è da cercare in:

1	4	5	8	9	12	17	20	24	31

Ma adesso la lista è di un solo elemento ed è banale dire che 15 non è nella lista.

Ricerca binaria

Il processo di ricerca illustrato si chiama ricerca binaria o ricerca dicotomica perché dividiamo sempre per due la zona dove cercare.
Ecco il codice in Python che implementa l'algoritmo:

```
## Finds a value in a range of a sorted list, using the binary search.
#  @param values the list in which to search
#  @param low the low index of the range
#  @param high the high index of the range
#  @param target the value to find
#  @return the index where the target occurs, or -1 if not in the list
#
def binarySearch(values, low, high, target) :
   if low <= high :
      mid = (low + high) // 2

      if values[mid] == target :
         return mid
      elif values[mid] < target :
         return binarySearch(values, mid + 1, high, target)
      else :
         return binarySearch(values, low, mid - 1, target)

   else :
      return -1
```


Ricerca binaria

Analisi di complessità

Per fare un'analisi di complessità dell'algoritmo, contiamo il numero di accessi alla lista. Per ipotesi assumiamo che il numero di elementi n sia una potenza di due, cioè: $n = 2^m$. Per ogni chiamata dell'algoritmo, si accede solo all'elemento centrale e poi l'algoritmo richiama se stesso su una metà della lista. Quindi:

$$T(n) = T(n/2) + 1$$

Ma allora, utilizzando la stessa equazione, abbiamo:

$$T(n/2) = T(n/4) + 1$$

e quindi:

$$T(n) = T(n/4) + 2$$

Generalizzando, si ottiene:

$$T(n) = T(n/2^k) + k$$

Ricerca binaria

Analisi di complessità

Poichè per la lista di lunghezza 1 si fa un solo accesso, allora $T(1) = 1$
Quindi, quando $k = m$, osservando che $m = log_2 n$, si ottiene:

$$\begin{aligned} T(n) &= T(n/2^k) + k \\ &= T(2^m/2^m) + m \\ &= T(1) + \log_2 n \\ &= 1 + \log_2 n \end{aligned}$$

Di conseguenza, l'algoritmo di ricerca dicotomica ha complessità $O(\log n)$.

I grafi

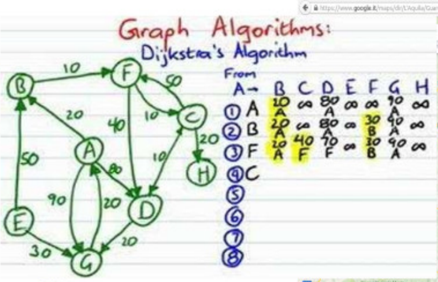
In molte situazioni è necessario rappresentare un insieme di entità e le relazioni che intercorrono tra esse.
Ecco alcuni esempi:

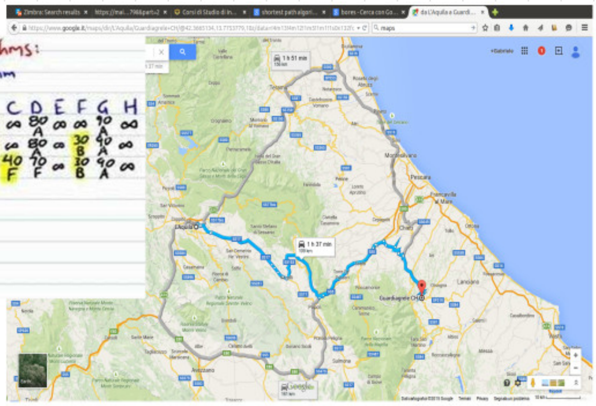
- in una rete sociale le entità sono le persone e la relazione tra due di loro può essere la reciproca conoscenza.
- In Internet e nelle reti di comunicazione in generale, le entità sono dispositivi in relazione tra loro se direttamente connessi con un cavo o via radio.
- In una rete stradale le entità sono località e le strade che le connettono le relazioni.
- In un albero genealogico le entità sono persone e la relazione è "essere genitore di".

Queste situazioni possono essere modellate attraverso strutture matematiche chiamate grafi, su cui operare mediante algoritmi.

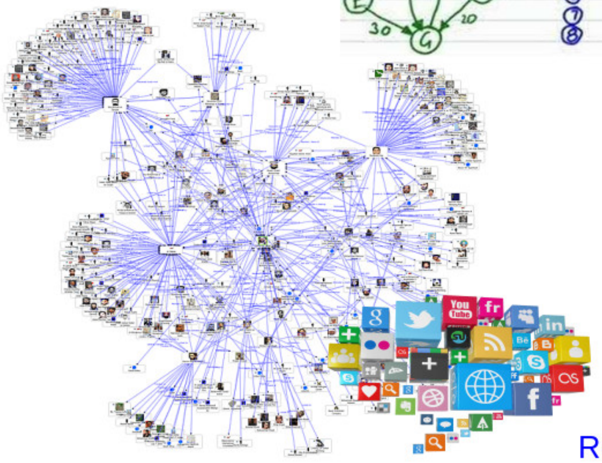
I grafi

L'Algoritmo di Dijkstra: dato un nodo di una rete trova i cammini minimi da quel nodo verso ogni altro nodo della rete efficientemente.





Percorsi su mappe stradali.



Routing di pacchetti di dati su Internet.

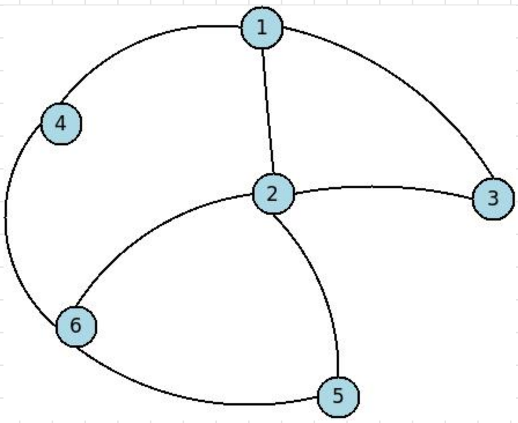
Social Networks Analisys.

I grafi

Formalmente un grafo $G = (V, E)$ è dato da una coppia di insiemi:

- V: insieme dei vertici (o nodi) del grafo
- E: insieme degli archi (o lati): un arco è una coppia di elementi di V.

$V = \{1, 2, 3, 4, 5, 6\}$ e $E = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 6), (2, 5), (4, 6), (5, 6)\}$:



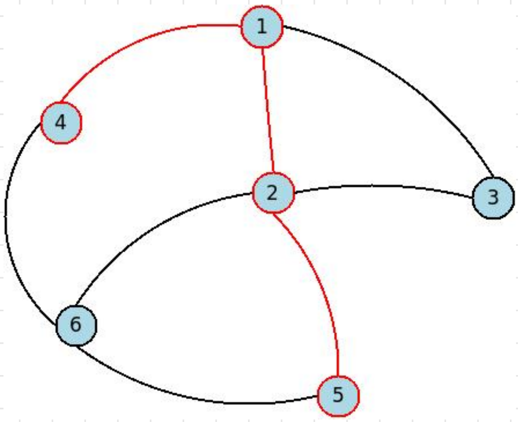
Rappresentazione del grafo $G = (V, E)$.

I nodi collegati da un arco (cioè presenti in una coppia) si dicono adiacenti.

I grafi

Cammmini e cicli

Una sequenza di nodi distinti si dice cammino se esiste un arco tra ogni coppia di nodi consecutivi.

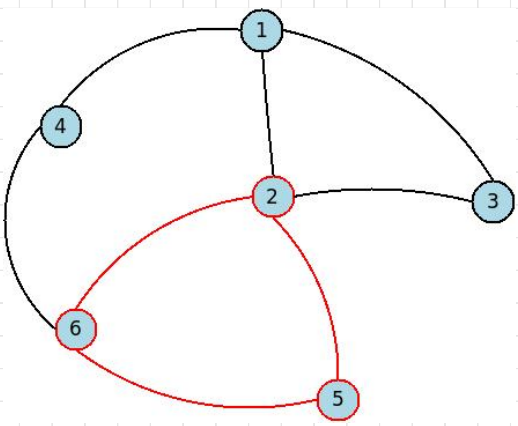


Cammino $[4, 1, 2, 5]$ tra i nodi 4 e 5 nel grafo G

I grafi

Cammmini e cicli

Una sequenza di nodi si dice ciclo se esiste un arco tra ogni coppia di nodi consecutivi e i nodi sono tutti distinti tranne il primo e l'ultimo che coincidono.

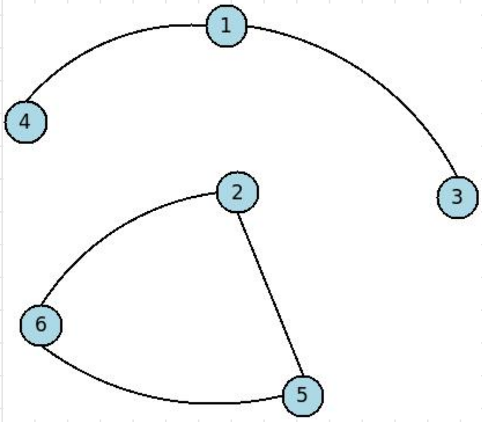


Ciclo $[2, 5, 6, 2]$ nel grafo G

I grafi

Connettività

Un grafo è connesso se per ogni coppia di nodi u e v esiste un cammino tra u e v. Un grafo si dice sconnesso in caso contrario.

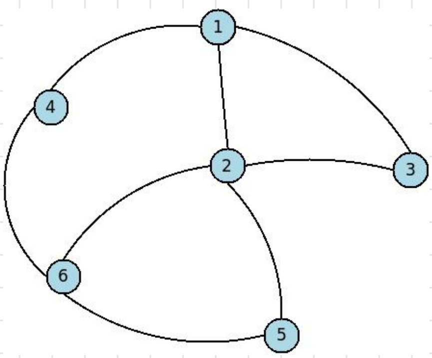


Esempio di grafo sconnesso.

I grafi

Rappresentazione in Python

Esistono molti modi per rappresentare un grafo con una struttura dati. Una di queste è la rappresentazione con liste di adiacenza: ad ogni nodo si associa la lista dei nodi adiacenti. Questa struttura può essere realizzata in Python con un dizionario.




```
Grafo = { #in forma estesa
          1:[2,3,4],
          2:[1,3,5,6],
          3:[1,2],
          4:[1,6],
          5:[2,6],
          6:[2,4,5]
        }
```

```
Grafo = { #in forma compatta
          1:[2,3,4],
          2:[3,5,6],
          3:[],
          4:[6],
          5:[6],
          6:[]
        }
```

I grafi

Inserimento di nodi e archi

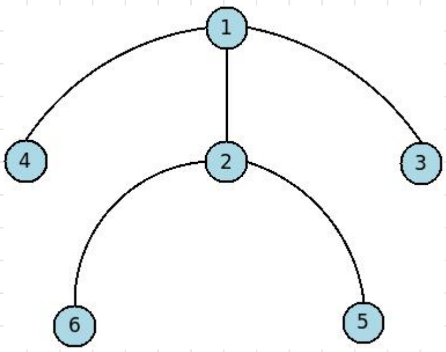
Ecco due procedure in Python per l'inserimento di nodo (se non presente) e di un arco (se non presente) in un grafo G.

```
#inserimento di un nodo
#
def inserisci_nodo(G, nodo) :
    if nodo not in G:
       G[nodo] = [] #inserisce il nodo in G con lista vicini vuota



#inserimento di un arco
#
def inserisci_arco(G, nodo1, nodo2):
    inserisci_nodo(G, nodo1)   #inserisce il nodo1 se non presente
    inserisci_nodo(G, nodo2)   #inserisce il nodo2 se non presente
    if nodo2 not in G[nodo1]:  #se nodo2 non è già vicino di nodo1
            G[nodo1].append(nodo2) #lo pone tra i vicini di nodo1
            #aggiungere t[nodo2].append(nodo1) se in forma estesa
```


Gli alberi

Definizione

Un albero è un grafo connesso senza cicli.



Esempio di albero

Costruzione in Python:

```
Tree = {}  #albero vuoto

inserisci_arco(Tree,1,4)
inserisci_arco(Tree,1,2)
inserisci_arco(Tree,1,3)
inserisci_arco(Tree,2,5)
inserisci_arco(Tree,2,6)

print(Tree) # Tree è:
            # { 1: [4, 2, 3],
            #   2: [5, 6],
            #   3: [],
            #   4: [],
            #   5: [],
            #   6: []}
```

Alberi con radice

Definizione

Un albero con radice è un albero in cui si individua un nodo particolare detto radice.

Negli alberi con radice, per ogni altro nodo x :

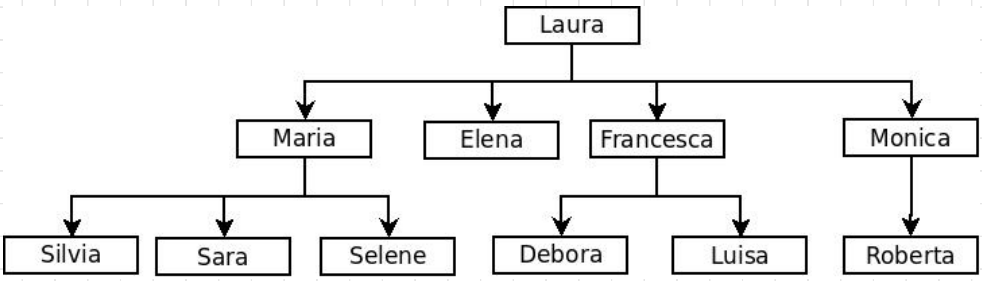
- si dice padre di x il nodo che precede x nel cammino dalla radice a x
- si dicono figli di x tutti gli altri nodi adiacenti a x
- si dice che x è una foglia dell'albero se non ha figli

Gli alberi con radice in genere si usano per rappresentare gerarchie, discendenze, organigrammi, etc.

Alberi con radice

Esempio

Nel seguente albero di discendenti:



- il nodo radice è "Laura"
- i nodi foglia sono "Silvia", "Sara", "Selene", "Debora", "Luisa" e "Roberta"
- il nodo padre di "Roberta" è "Monica"
- i nodi figli di "Francesca" sono "Debora" e "Luisa"
- il nodo padre di "Francesca" è "Laura"

Algoritmi di visita di alberi

Come possiamo visitare i nodi di un albero in modo tale che un figlio e tutti i sui discendenti siano visitati prima di un altro figlio e di tutti i sui discendenti?
Operazioni di questo genere si chiamano "visite di un albero" e possono essere implementate da un algoritmo ricorsivo.

Eccone un esempio in Python:

```
 def visita(t,nodo) :
    #inserire azione da fare sul nodo: in questo caso una stampa
    print(nodo)

    figli = t[nodo]          # t[nodo] contiene la lista dei figli
    for figlio in figli:     # ciclo per visitare tutti i figli
            visita(t, figlio)

visita(Tree,1)   #stampa i nodi nell'ordine 1 4 2 5 6 3
```


Algoritmi di visita in pre-ordine e post-ordine

L'algoritmo precedente effettua una visita in pre-ordine perché compie un'azione su un nodo prima di visitare i suoi figli.

Se l'azione viene compiuta dopo la visita dei figli, si dice che la visita è in post-ordine.

Vediamo come si presenta un algoritmo di visita in postordine:

```
def visita_postordine(t,nodo) :
    figli = t[nodo]          # t[nodo] contiene la lista dei figli
    for figlio in figli:     # ciclo per visitare tutti i figli
            visita_postordine(t, figlio)

    #inserire azione da fare sul nodo: in questo caso una stampa
    print(nodo)


visita_postordine(Tree,1)   #stampa i nodi nell'ordine 4 5 6 2 3 1
```

Algoritmi di visita

Basandoci sull'idea di visita ricorsiva, possiamo calcolare proprietà dell'albero, come, ad esempio, la sua profondità.

La profondità di un albero con radice x è data dalla massima profondità dei sottoalberi dei figli di x più 1.
Un albero composto da un nodo senza figli ha profondità 0

Ecco come si può calcolare questo valore:

```
def profondita(t,nodo) :
    prof_figli = -1         #-1 perché una foglia ha profondità 0
    figli = t[nodo]         # t[nodo] contiene la lista dei figli
    for figlio in figli:    # ciclo per visitare tutti i figli
            prof_figli= max(prof_figli, profondita(t, figlio))

    return prof_figli + 1


print(profondita(Tree,1))  #stampa 2
```