La Tecnica del Divide et Impera – Parte II

Algoritmi 2 – Lezione 14

A. Monti
Sapienza Università di Roma

Corso: Algoritmi 2

Indice

1. Coppia di punti a distanza minima
   - Approccio esaustivo
   - Caso 1D: punti su retta
   - Divide et Impera
   - Ottimizzazione della fase Combina
   - Dimostrazione (Principio dei Cassetti)
   - Implementazione Python
   - Analisi della Complessità
2. Conteggio delle Inversioni
   - Introduzione al problema
   - Approccio esaustivo
   - Divide et Impera
   - Ricerca binaria ($O(n \log^2 n)$)
   - Fondi e Conta($O(n \log n)$)
   - Implementazione Python

La Coppia di Punti a Distanza Minima

- **Problema:** Dato un insieme di n punti nel piano, trovare la coppia con distanza euclidea minima.

- **Approccio Esaustivo:**
  - Calcola la distanza tra ogni possibile coppia di punti.

- **Un'ottimizzazione:** Il calcolo della distanza euclidea

  $$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

  è più costoso di una moltiplicazione. Poiché minimizzare d è equivalente a minimizzare d^2, possiamo evitare di calcolare la radice quadrata e confrontare direttamente le distanze al quadrato. Questo rende l'algoritmo più veloce in pratica.

- Complessità: $\Theta(n^2)$.

Il caso monodimensionale: punti su una retta

Un algoritmo di complessità $O(n \log n)$:

1. Ordiniamo i punti in base alla loro unica coordinata
2. Scandiamo l'array ordinato e calcoliamo la distanza tra ogni punto e il suo successore immediato.

In un array ordinato, la distanza minima si trova tra due elementi adiacenti.

- Obiettivo: Raggiungere $O(n \log n)$ anche nel caso bidimensionale.

La Soluzione Divide et Impera nel Piano

L'idea generale è dividere l'insieme di punti e combinare i risultati.

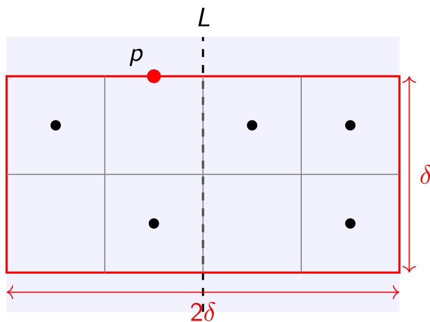
- **Pre-elaborazione**: Creare due liste di punti, una ordinata per coordinata X (P_x) e una per coordinata Y (P_y).
- **Dividi**: Trovare una linea verticale L che divide P_x in due metà, P_L e P_R.
- **Impera**: Risolvere ricorsivamente il problema su P_L e P_R, ottenendo le distanze minime δ_L e δ_R.
- **Combina**: Il passo più critico. Si definisce $\delta = \min(\delta_L, \delta_R)$. La distanza minima globale è δ, salvo l'esistenza di una coppia "a cavallo" (un punto in P_L e uno in P_R) con distanza inferiore.

La Fase di "Combina" in $O(n)$

- L'unico modo per cui una coppia a cavallo possa avere una distanza inferiore a δ è se i punti si trovano in una "striscia" di larghezza 2δ centrata sulla linea L.
- Non è necessario confrontare ogni punto della striscia con tutti gli altri.
- Per ogni punto p nella striscia, è sufficiente confrontarlo solo con un numero costante di punti successivi, poiché gli altri punti sono garantiti essere più lontani di δ.
- Si può dimostrare che ogni punto della striscia ha al massimo 7 candidati a cui confrontarsi.

Dimostrazione del Limite Costante (Principio dei Cassetti)

- Consideriamo un rettangolo di ricerca di dimensioni $2\delta \times \delta$ attorno a un punto p.
- Dividiamo questo rettangolo in 8 quadrati di lato $\delta/2 \times \delta/2$.
- La distanza massima tra due punti all'interno di un singolo quadrato è la lunghezza della sua diagonale, $d_{diag} = \sqrt{(\delta/2)^2 + (\delta/2)^2} = \delta/\sqrt{2} \approx 0.707\delta$.
- Poiché $d_{diag} < \delta$, ogni quadrato può contenere al massimo un punto.
- Per il Principio dei Cassetti, il rettangolo di ricerca può contenere al massimo 8 punti (incluso p stesso).
- Pertanto, per ogni punto della striscia, è sufficiente confrontare la sua distanza con i 7 punti successivi ordinati per la coordinata Y. Questo porta la fase di "Combina" ad avere una complessità $O(n)$.



Implementazione: funzione principale

Funzione principale di complessità $O(nlogn)$ che invoca l'algoritmo ricorsivo *coppia_minima_R* e poi restituisce la coppia di punti a distanza minima.

```
def coppia_minima(punti):
    """
    Restituisce  la coppia di punti a distanza minima.
    """
    n = len(punti)
    if n < 2:
        return None
    # Pre-ordinamento dei punti per coordinata x e y
    Px = sorted(punti, key=lambda p: p[0])
    Py = sorted(punti, key=lambda p: p[1])
    # La funzione ricorsiva restituisce
    # (coppia, distanza_quadrata)
    coppia_migliore, _ = coppia_minima_R(Px, Py)
    return coppia_migliore
```

Implementazione ricorsiva

```
def coppia_minima_R(Px, Py):
"""
Restituisce (coppia, distanza_quadrata).
"""
n = len(Px)
if n <= 3:
    # invoca una procedura esaustiva che restituisce
    # la coppia minima e la loro distanza quadrata
    return coppia_minima_esaustivo(Px)
mid = n // 2
punto_mediano = Px[mid]
Py_sinistra, Py_destra = [], []
for p in Py:
    if p[0] < punto_mediano[0]:
        Py_sinistra.append(p)
    else:
        Py_destra.append(p)
coppia_s, dist2_s = coppia_minima_R(Px[:mid], Py_sinistra)
coppia_d, dist2_d = coppia_minima_R(Px[mid:], Py_destra)
# Determina la distanza minima (al quadrato) trovata finora
if dist2_s < dist2_d:
    dist2_min = dist2_s; coppia_migliore = coppia_s
else:
    dist2_min = dist2_d; coppia_migliore = coppia_d
# Crea una "striscia" di punti vicini alla linea mediana
Striscia = [ ]
for p in Py:
    if (p[0] - punto_mediano[0])**2 < dist2_min:
        Striscia.append(p)
# Cerca una coppia a distanza minima nella striscia
for i in range(len(Striscia)):
    p1 = Striscia[i]
    for j in range(i + 1, len(Striscia)):
        p2 = Striscia[j]
        # interrompi se la sola distanza y e' gia' eccessiva
        if (p2[1] - p1[1])**2 >= dist2_min:
            break
        dist2 = (p1[0] - p2[0])**2 + (p1[1] - p2[1])**2
        if dist2 < dist2_min:
            dist2_min = dist2; coppia_migliore = (p1, p2)
return coppia_migliore, dist2_min
```

Relazione di Ricorrenza e Complessità

- L'algoritmo Divide et Impera per il problema della coppia di punti a distanza minima ha la seguente relazione di ricorrenza:

$$T(n) = 2T(n/2) + \Theta(n)$$

- La soluzione di questa ricorrenza è $\Theta(n \log n)$.
- Questo risulta efficiente quanto il caso monodimensionale e migliora il costo $\Theta(n^2)$ dell'algoritmo esaustivo.

Conteggio delle Inversioni

- **Problema:** Dato un vettore V di n interi, contare le inversioni, cioè le coppie di indici (i, j) tali che $i < j$ e $V[i] > V[j]$.

Ad esempio, per `V = [8, 4, 2, 9, 5]` la soluzione è 5. Le coppie invertite sono:

- $(8, 4)$, $(8, 2)$, $(8, 5)$
- $(4, 2)$
- $(9, 5)$

Conteggio delle Inversioni: Approccio Esaustivo

- L'approccio più semplice e diretto consiste nell'esaminare ogni possibile coppia di indici (i, j) con $i < j$ e verificare se $V[i] > V[j]$.

- La complessità di questo algoritmo è $\Theta(n^2)$ che diventa proibitiva per vettori di grandi dimensioni.

Conteggio delle Inversioni: Approccio Divide et Impera

- L'algoritmo segue lo schema tipico del Divide et Impera:
  - Si divide il vettore a metà.
  - Si contano ricorsivamente le inversioni nelle due metà.
  - Nella fase di unione (merge), si contano le "inversioni a cavallo", cioè quelle tra un elemento della prima metà e un elemento della seconda.

Un Primo Tentativo per contare le coppie "a cavallo": la Ricerca Binaria ($\Theta(n \log^2 n)$)

1. Ordiniamo il sotto-vettore destro, R. Questo richiede $\Theta(n \log n)$.
2. Per ogni elemento x nel sotto-vettore sinistro L, usiamo la ricerca binaria su R (ora ordinato) per contare quanti elementi in R sono più piccoli di x. Questo ci dà esattamente il numero di inversioni a cavallo che coinvolgono x.
3. La ricerca binaria per un elemento richiede $O(\log n)$. Ripetendola per tutti gli $|L| = n/2$ elementi, questo passo richiede $O(n \log n)$.

- Il costo totale della fase di **Combina** con ricerca binaria è quindi $\Theta(n \log n)$.
- La relazione di ricorrenza per questo algoritmo è

$$T(n) = 2T(n/2) + \Theta(n \log n)$$

che risolta dà $\boldsymbol{\Theta(n \log^2 n)}$

È un miglioramento, rispetto all'algoritmo $O(n^2)$, ma si può fare di meglio.

Approccio Ottimale: Fondi e conta ($O(n \log n)$)

L'idea geniale è contare le inversioni a cavallo *durante* la fusione dei due sotto-vettori, proprio come avviene nell'algoritmo **Mergesort**.

Supponiamo che L e R ci vengano restituiti ordinati dalle chiamate ricorsive. Quando fondiamo L e R per creare un nuovo vettore ordinato, confrontiamo gli elementi correnti:

- Se l'elemento di L è più piccolo, lo copiamo nel vettore fuso. Nessuna inversione viene contata.
- Se l'elemento di R è più piccolo, lo copiamo nel vettore fuso. In tal caso, l'elemento di R è minore di tutti quelli rimasti in L: contiamo tante inversioni quanti sono questi elementi.

- Il costo totale della fase di **Combina** in **Fondi e conta** è quindi $O(n)$.
- La relazione di ricorrenza per questo algoritmo è

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

  che risolta dà $\boldsymbol{O(n \log n)}$.

Implementazione

```
 def conta_inversioni(lista):
    """ Funzione principale che avvia il conteggio delle inversioni.  """
    return inversioniR(lista,  0, len(lista))

def inversioniR(lista, i, j):
    inversioni = 0
    # la ricorsione si ferma se il sotto-array ha < 2 elementi
    if j - i > 1:
        mid = (i + j) // 2
        # Calcola  le inversioni a sinistra e a destra
        inv1 = inversioniR(lista,  i, mid)
        inv2 = inversioniR(lista, mid, j)
        inv3 = _fondi_e_conta(lista, i, mid, j)   # Conta le inversioni "a cavallo" e fonde
        inversioni = inv1 + inv2 + inv3
    return inversioni

def _fondi_e_conta(lista, a, mid, b):
    """ Fonde i  sotto-array V[start:mid] e V[mid:end] e   conta le inversioni "a cavallo"  """
    lista_fusa = []
    i = a      # Puntatore per la metà sinistra
    j = mid        # Puntatore per la metà destra
    inversioni = 0
    # Scorre entrambe le metà e popola la lista_fusa
    while i < mid and j < b:
        if lista[i] <= lista[j]:
            lista_fusa.append(lista[i])
            i += 1
        else:
            # Conteggio delle inversioni
            lista_fusa.append(lista[j])
            j += 1
            inversioni += (mid - i)
    while i < mid:
        lista_fusa.append(lista[i])
        i += 1
    while j < b:
        lista_fusa.append(lista[j])
        j += 1
    # Riversa il contenuto della lista temporanea in lista
    for i in range(len(lista_fusa)):
        lista[a + i]= lista_fusa[i]
    return inversioni
```