A.D. 1308
unipg
DIPARTIMENTO
DI INGEGNERIA

Tesina Finale di

Algoritmi e Strutture di Dati

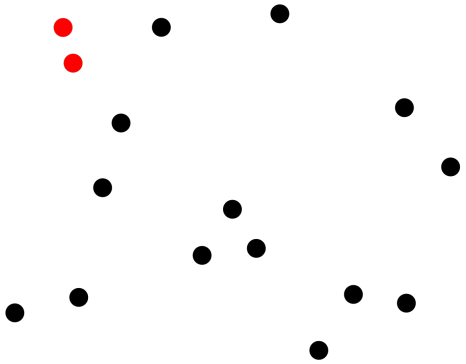
Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2022-2023

Dipartimento di Ingegneria

docente

Prof. Emilio Di Giacomo

Implementazione di un algoritmo per calcolare la coppia di punti più vicina in un insieme di punti dati



studenti

330594 **Martin** **Arsoski** martin.arsoski@studenti.unipg.it

329683 **Michele** **Mariotti** michele.mariotti@studenti.unipg.it

0. Indice

1 Descrizione del Problema **2**

2 Algoritmi Implementati **4**
2.1 Algoritmo di Forza Bruta 4
2.2 Algoritmo Divide et Impera 4
2.3 Altre classi utilizzate nell'implementazione dell'algoritmo 7
2.4 Interfaccia grafica 7

3 Analisi della complessità **9**
3.1 Complessità dell'algoritmo di Forza Bruta 9
3.2 Complessità dell'algoritmo di Divide et Impera 9

4 Dati Sperimentali **11**

1. Descrizione del Problema

Il progetto sviluppato si pone l'obiettivo di analizzare e realizzare l'implementazione di un algoritmo per calcolare la coppia di punti più vicini in un insieme di punti dati nel piano.

Il problema della coppia di punti più vicina in un piano è un classico problema della geometria computazionale. La risoluzione di questo problema ha diverse applicazioni pratiche in vari campi, tra cui:

1. Ricerca di punti simili in grandi set di dati: per riconoscere pattern o identificare oggetti simili in immagini, come il riconoscimento di volti o la classificazione di oggetti;

2. Problemi di ottimizzazione: per trovare il percorso più breve tra diverse destinazioni in un sistema di navigazione;

3. Geolocalizzazione: per determinare la posizione più vicina a un dato punto di riferimento;

4. Rilevamento di anomalie: per trovare i punti che sono significativamente diversi dagli altri in una collezione di dati.

La distanza tra due punti viene calcolata utilizzando la formula della distanza euclidea:

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

Il problema può essere risolto attraverso due algoritmi, uno che utilizza l'approccio di forza bruta mentre l'altro la tecnica del divide et impera.

L'idea di base dell'approccio "forza bruta" consiste nel confrontare ogni coppia di punti all'interno dell'insieme e calcolare la distanza tra di essi. La coppia di punti che presenta la distanza minima è quindi identificata come la coppia di punti più vicina.

L'approccio "divide et impera" si basa sulla suddivisione dell'insieme di punti in due sottoinsiemi, affrontando i relativi sottoproblemi tramite un approccio ricorsivo, per poi combinare i risultati ottenuti. Ciò porta a una soluzione più efficiente in termini di tempo di esecuzione.

2. Algoritmi Implementati

Per risolvere il problema, vengono implementati entrambi gli algoritmi: l'algoritmo di forza bruta e l'algoritmo divide et impera. La classe che si occupa delle implementazioni è ClosestPairOfPoints.

2.1 Algoritmo di Forza Bruta

L'implementazione dell'algoritmo prende in input l'array di punti P ed attraverso due cicli for annidati confronta tutte le coppie di punti possibili e restituisce la coppia di punti più vicina e la relativa distanza.

```
BRUTE_FORCE(P)
1  d_min = infinito
2  coppia_min = NIL
3  for i=1 to P.length
4          for j=i+1 to P.length
5                  calcola la distanza d tra Pi e Pj
6                  if d < d_min
7                          d_min = d
8                          coppia_min = (Pi, Pj)
9  return (coppia_min, d_min)
```

2.2 Algoritmo Divide et Impera

L'altra soluzione adottata è l'implementazione dell'algoritmo divide et impera. L'algoritmo riceve in input i due array di punti pX e pY ordinati rispettivamente in base alla coordinata x ed y.

La prima parte si occupa del "divide": l'insieme viene diviso in due parti uguali attraverso una retta verticale coincidente alla coordinata x del punto in posizione

pX[m], con m = $\lceil \frac{n}{2} \rceil$; poi abbiamo la parte "impera": viene calcolata la distanza minima tra le coppie di punti in ciascun sottoinsieme, utilizzando la stessa procedura ricorsiva.

Una volta ottenute le due distanze minime e i relativi punti, viene selezionato il valore minimo tra le due distanze. A questo punto abbiamo la parte "combina": è necessario considerare le coppie di punti che si trovano a cavallo della linea di separazione (un punto a sinistra e l'altro a destra). Questi punti possono essere a una distanza minore rispetto a quanto già calcolato.

Infine, si restituisce la coppia di punti più vicina tra la coppia trovata precedentemente e quelle trovate attraverso la scansione a cavallo della linea di separazione.

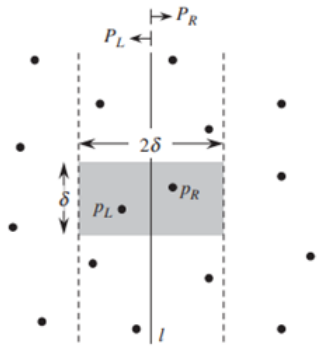
```
CLOSEST_PAIR_OF_POINTS_RIC(pX, pY)
1   n = pX.length
2   If n <= 3
3           return BRUTE_FORCE (pX)
4   m = ⌈n/2⌉
5   if pX[m] == pX[m+1]
6           d_min = 0
7           coppia_min = (pX[m], pX[m+1])
8           return (coppia_min, d_min)
9   pL = array che contiene i punti da pX[1] a pX[m]
10  pR = array che contiene i punti da pX[m+1] a pX[n]
11  yL = nuovo array di dimensione uguale a pL
12  yR = nuovo array di dimensione uguale a pR
13  indexL = 0
14  indexR = 0
15  for i=1 to n
16          if pY[i].x < pL[m].x
17                  yL[indexL] = pY[i]
18                  indexL++
19          else if pY[i].x == pL[m].x and pY[i].y <= pL[m].y
20                  yL[indexL] = pY[i]
21                  indexL++
22          else
23                  yR[indexR] = pY[i]
24                  indexR++
25  risultatoL = CLOSEST_PAIR_OF_POINTS_RIC (pL, yL)
26  risultatoR = CLOSEST_PAIR_OF_POINTS_RIC (pR, yR)
27  risultato = risultato con distanza minima, tra risultatoL e risultatoR
28  sp = nuovo array list di punti vuoto
29  for ogni p ∈ pY
30          if |p.x – pX[m].x|< risultato.d_min
31                  sp.add(p)
32  for i = 1 to sp.size
33          for j = i+1 to sp.size
34                  a =  GET_ELEMENT_AT(sp, i)
35                  b = GET_ELEMENT_AT(sp, j)
36                  if b.y – a.y < risultato.d_min
37                          d = distanza tra a e b
38                          if d < risultato.d_min
39                                  risultato.d_min = d
40                                  risultato.coppia_min = (a, b)
41  return (coppia_min, d_min)
```

Righe 2-3: se il numero di punti è minore o uguale a 3, siamo nel caso base della ricorsione, quindi viene invocato il metodo di forza bruta descritto sopra.

Righe 5-8 e 13-24: permettono la creazione dei due array yL e yR. Questi due array, che contengono gli elementi rispettivamente a sinistra e a destra della retta verticale di divisione ordinati in base alla coordinata y, potrebbero essere ottenuti ordinando ogni volta gli array pL e pR, ma questo metodo aumenterebbe la complessità dell'algoritmo rendendolo più inefficiente.
Per questo essi vengono ottenuti dividendo i punti del già ordinato pY in base alla coordinata x.
La combinazione delle righe 19-21 e con l'ordinamento iniziale di pX che ordina i punti in base alle coordinate y nei casi in cui le coordinate x coincidono, permette di ottenere sempre i giusti array yL e yR.
Senza questa combinazione di accorgimenti nel caso in cui avessimo ad esempio un insieme di punti P = (0,y0), (1,y1), (1,y2), (2,y3) , con y0,. . . ,y3 valori qualsiasi, avremmo che pL = (0,y0), (1,y1) e pR = (1,y2), (2,y3) mentre yL = (0,y0), (1,y1), (1,y2) e yR = (2,y3), quindi avremmo un errore.
Nel caso estremo in cui abbiamo punti con le stesse coordinate ciò non basterebbe, per questo si verifica nelle righe 5-8 se pX[m] = pX[m+1], ovvero se hanno le stesse coordinate: ciò implica che la distanza tra i due punti è 0 quindi si salva questa coppia come risultato.

Righe 25-27: vengono calcolate in modo ricorsivo le distanze minime nella parte sinistra e destra e viene salvata la distanza minima δ tra le due chiamate ricorsive con la relativa coppia di punti.

Righe 28-41: si verifica se è presente una coppia di punti a cavallo alla retta di separazione (con un punto nella parte sinistra e uno nella parte destra) con distanza minore a δ. Per fare ciò, a partire da pY, viene creata una lista contenente tutti i punti che si trovano a una distanza minore a δ dalla retta (righe 29-31). La lista contiene punti con coordinate y non decrescenti, quindi la banda viene scansionata dall'alto verso il basso, considerando solo i punti che hanno una distanza (in coordinate y) minore di d dal predecessore; viene calcolata la loro distanza e se è minore di δ, essa viene aggiornata.



CLOSEST_PAIR_OF_POINTS(P)	
1	pX = P ordinato in base a x (in caso di coordinate x uguali, ordinato anche in base a y)
2	pY = P ordinato in base a y
3	return CLOSEST_PAIR_OF_POINTS_RIC(pX, pY)

Per calcolare la coppia di punti più vicina nell'array di punti P, viene invocato CLOSEST_PAIR_OF_POINTS(P), il quale crea due array di punti pX e pY, sempre della stessa dimensione.
L'array pX viene ordinato in base alla coordinata x utilizzando il metodo compare della classe PointsXCoordComparator. Esso è realizzato in modo tale che in caso di stessa coordinata x, viene confrontata anche la coordinata y per stabilire l'ordine. Per quanto riguarda l'array pY è sufficiente ordinarlo solo in base alla coordinata y, senza considerare la coordinata x.
Infine viene invocato il metodo ricorsivo CLOSEST_PAIR_OF_POINTS_RIC(pX, pY) passando in input i due array pX e pY.

2.3 Altre classi utilizzate nell'implementazione dell'algoritmo

La classe Point viene utilizzata al fine di rappresentare un punto nel piano con le sue coordinate x ed y. All'interno di questa classe ci sono dei metodi che restituiscono le coordinate del punto e consentono di verificare se due punti sono uguali.

Per tenere traccia delle distanze tra i vari punti, viene creata la classe CPOPResult la quale permette di memorizzare i due punti e la loro distanza euclidea.

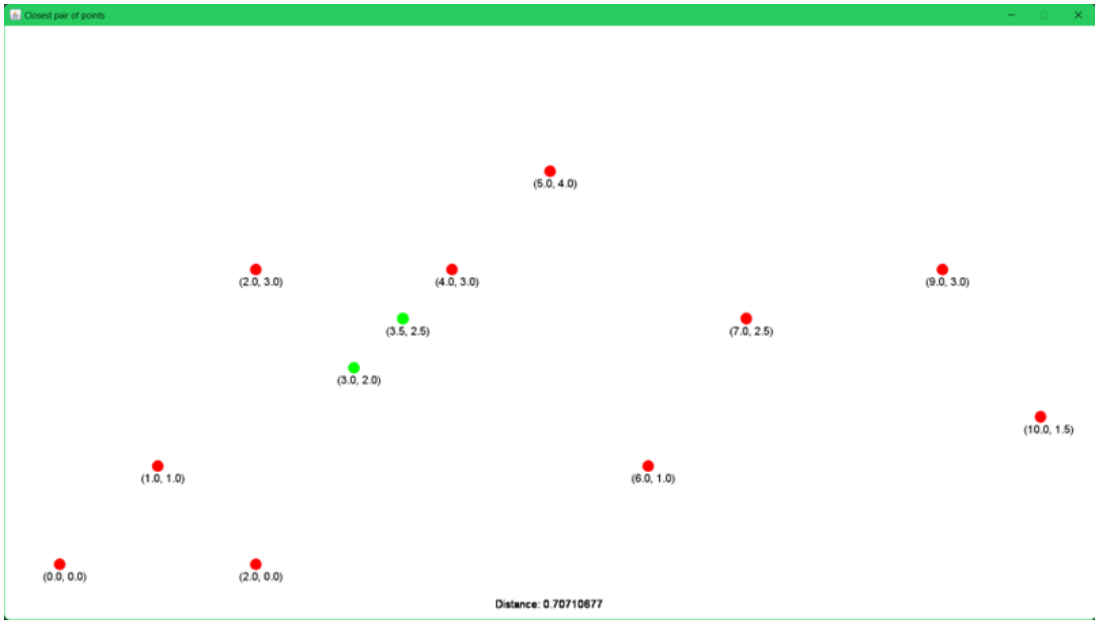
La classe PointsXCoordComparator, che implementa l'interface Comparator contiene il metodo compare che viene invocato quando si ordina l'array di punti in base alla coordinata x. Esso è realizzato in modo tale che in caso di stessa coordinata x, si prende in considerazione anche la coordinata y per stabilire l'ordine.

2.4 Interfaccia grafica

L'interfaccia grafica che permette all'utente di testare l'algoritmo è implementata dalle classi TestGUI, TestPanel e SetupPanel, che estendono classi del framework Java Swing.

La classe TestGUI estende la classe JFrame, ovvero è la finestra dell'applicazione; essa contiene il metodo main().

La classe SetupPanel, che estende la classe JPanel, permette all'utente di inserire i punti da dare in input all'algoritmo. Una volta inseriti tutti i punti premendo il JButton "end" si cambia il pannello mostrato in finestra, rimuovendo l'istanza di SetupPanel ed aggiungendo quella della classe TestPanel, anch'essa estensione di JPanel, che mostra graficamente i punti inseriti e le relative coordinate, ed inoltre colora di verde i due punti più vicini, indicando in basso la distanza che li separa.



3. Analisi della complessità

Al fine di evidenziare maggiormente le prestazioni di ciascun algoritmo, è possibile effettuare un'analisi della complessità asintotica, in particolare andando ad analizzare gli pseudo-codici degli algoritmi riportati precedentemente.

Entrambe le procedure di forza bruta e divide et impera lavorano sull'input rappresentato dall'insieme di punti. Le prestazioni dei due algoritmi variano in funzione della dimensione dell'insieme; è quindi interessante analizzare il costo computazionale in funzione di tale grandezza.

3.1 Complessità dell'algoritmo di Forza Bruta

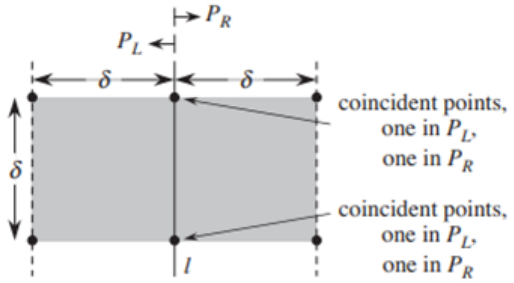
Per esaminare tutte le possibili coppie di punti, è necessario eseguire due cicli annidati. Il primo ciclo scorre tutti i punti una volta, mentre il secondo ciclo scorre tutti i punti rimanenti per ogni punto nel primo ciclo. Il numero totale di coppie generate è dato da $\binom{n}{2}$, che è dell'ordine di n^2. In questo caso non esiste un caso peggiore e un caso migliore. L'algoritmo della coppia di punti più vicina implementato tramite forza bruta ha una complessità temporale quindi di $\theta(n^2)$, dove n è il numero di punti nel piano.

3.2 Complessità dell'algoritmo di Divide et Impera

Inizialmente, i punti vengono ordinati in due nuovi insiemi in base alle coordinate x e y, impiegando un tempo di ordine O(n log n), dove n rappresenta il numero di punti. Questo preordinamento ci permette di evitare di dover riordinare gli insiemi ad ogni chiamata ricorsiva.

L'analisi della complessità temporale di questo algoritmo coinvolge le tre fasi principali della ricorsione: la divisione, la conquista e la combinazione.

Nella fase di divisione, l'insieme di punti viene suddiviso in due sottoinsiemi tramite una linea verticale che separa gli elementi a metà. Questa operazione può essere eseguita in un tempo lineare di O(n).
Nella fase di combinazione, viene controllata anche la distanza minima tra le coppie di punti che sono a cavallo della linea di divisione. Questa operazione richiede un tempo lineare di O(n), poiché è necessario scorrere i punti nella fascia a cavallo della retta di separazione e confrontare ogni punto con al massimo i 7 punti successivi. Non possono esserci più di 8 punti nel rettangolo δ x 2δ come mostrato nella figura sottostante, dato che altrimenti la distanza minima trovata fino a questo momento non sarebbe δ, quindi significa che il numero massimo di confronti richiesti è 7n, ovvero O(n).



Nella fase di conquista, l'algoritmo viene richiamato ricorsivamente per risolvere il problema separatamente per i due sottoinsiemi di punti. Quindi abbiamo che il costo complessivo è quindi T(n) = 2T(n/2) + O(n) per n>3 e O(1) per n<=3.

A questo va sommato il costo dell'ordinamento iniziale, quindi avremo che la complessità totale dell'algoritmo è T(n)' = T(n) + O(n log n).
Utilizzando il teorema del master: ci troviamo nel caso 2, a=2 e b=2, $n^{\log_b(a)}$ = n = O(n), quindi si ottiene T(n) = O(n log n), da cui segue che T(n)' = O(n log n).

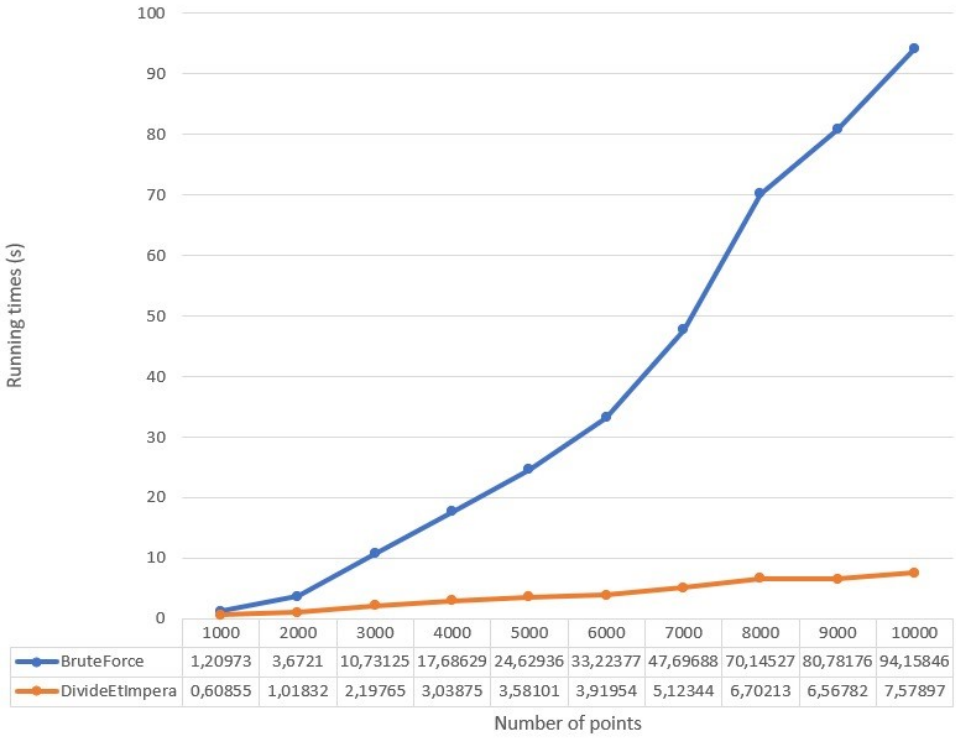
Il tempo richiesto per l'ordinamento dei punti è dominante rispetto agli altri termini. La complessità temporale dell'algoritmo, utilizzando la strategia divide et impera, è di O(n log n).

4. Dati Sperimentali

I due algoritmi implementati come soluzione al problema in esame presentano differenze significative nelle loro prestazioni, soprattutto quando vengono eseguiti su insiemi contenenti un grande numero di punti. Al fine di evidenziare le diverse velocità di esecuzione delle due procedure sviluppate, sono stati condotti dei test per misurare il tempo impiegato nel trovare la coppia di punti più vicina in insiemi di dimensioni crescenti.

Per effettuare queste misurazioni, è stata utilizzata la classe Experiment, che ha permesso di ripetere l'esecuzione di ciascun algoritmo più volte. La prima misurazione è stata effettuata su un insieme di 1000 elementi e successivamente la dimensione è stata aumentata di 1000 elementi ad ogni iterazione fino ad arrivare a 10000 elementi. Per ogni dimensione, sono state eseguite dieci misurazioni per ciascun algoritmo, al fine di calcolare una media dei tempi di esecuzione.
I dati raccolti sono stati scritti in un file di testo in cui viene riportato, per ogni dimensione, il tempo di esecuzione corrispondente ad ogni algoritmo.

Come evidenziato precedentemente durante l'analisi della complessità, l'algoritmo di forza bruta si rivela meno efficiente rispetto all'algoritmo divide et impera, soprattutto quando le dimensioni dell'insieme tendono ad aumentare.



	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
BruteForce	1,20973	3,6721	10,73125	17,68629	24,62936	33,22377	47,69688	70,14527	80,78176	94,15846
DivideEtImpera	0,60855	1,01832	2,19765	3,03875	3,58101	3,91954	5,12344	6,70213	6,56782	7,57897

In conclusione, sia l'algoritmo di forza bruta che quello di divide et impera offrono la possibilità di calcolare la coppia di punti più vicina in un insieme di punti dato nel piano. Tuttavia, è importante notare che le prestazioni dell'algoritmo di forza bruta calano visibilmente all'aumentare delle dimensioni dell'insieme, mentre l'algoritmo divide et impera rimane efficiente anche per input di grandi dimensioni. Pertanto, se si lavora con insiemi di punti di grandi dimensioni, l'utilizzo dell'algoritmo divide et impera può risultare molto vantaggioso dal punto di vista delle prestazioni.