Amitrano Simone

Relazione Esercizio 1

Confronto tra le prestazioni dei due algoritmi usati:

MERGE SORT:

Il merge sort per l'ordinamento dei record presenti nel csv ha mostrato tempi di esecuzione stabili e simili per tutti e tre i campi (stringhe, interi e float). Questo è coerente con la natura stabile di Merge Sort, che ha una complessità temporale di $O(nlogn)$ e non dipende dalla distribuzione dei dati. Merge Sort è un algoritmo stabile che non è influenzato negativamente dalla struttura dei dati. Pertanto, i risultati stabili su tutti i campi sono attesi. La stabilità e la prevedibilità dei tempi di esecuzione confermano che Merge Sort è particolarmente adatto a ordinare grandi quantità di dati indipendentemente dalla natura dei campi.

QUICK SORT:

- *Field 1 (Stringhe):* L'ordinamento su stringhe ha impiegato un tempo elevato, suggerendo che l'algoritmo non è ottimale per questo tipo di dati e portando a un fallimento dell'esperimento.

- *Field 2 (Interi):* L'algoritmo ha funzionato correttamente, ma il tempo di esecuzione è stato maggiore rispetto a Merge Sort.

- *Field 3 (Float):* Anche qui, Quick Sort ha impiegato più tempo rispetto a Merge Sort, ma comunque accettabile.

L'ordinamento delle stringhe con Quick Sort ha causato un ritardo significativo. Questo potrebbe essere dovuto alla natura meno prevedibile delle stringhe rispetto ai numeri interi o float, causando il degrado delle prestazioni in presenza di pivot non ottimali portando a una complessità temporale vicina a $O(n^2)$ nel caso peggiore e lasciando intendere un'errore nell'esecuzione dell'algoritmo a causa dei tempi di esecuzione elevati. Probabilmente un tempo così elevato è dovuto dal fatto che l'algoritmo deve scorrere l'array di record ogni volta e rieffettuare un confronto tra le stringhe presenti nei vari **Field1**
L'aumento dei tempi di esecuzione per **Field 2** e **Field 3** è coerente con la sensibilità di Quick Sort alla selezione dei pivot. Anche se le prestazioni sono rimaste entro limiti accettabili per i numeri interi e float, l'algoritmo non ha superato Merge Sort in termini di velocità, come ci si potrebbe aspettare nei migliori casi.

I risultati suggeriscono che **Merge Sort** è più affidabile per dataset grandi e diversi come il file record.csv utilizzato,, con prestazioni costanti su tutti i tipi di dati. Al contrario, **Quick Sort** mostra variabilità in base alla natura dei dati, con performance scadenti nel caso di stringhe. Ciò indica che Merge Sort potrebbe essere preferibile per applicazioni che richiedono robustezza e tempi di esecuzione prevedibili.

Relazione Esercizio 2

EDIT DISTANCE:
L'esercizio implementa un programma che corregge le parole contenute in un file chiamato *"correctme.txt"* confrontandole con quelle di un dizionario contenuto in un altro file chiamato *"dictionary.txt"*. La correzione si basa sul calcolo della distanza di edit, ovvero il numero minimo di operazioni (inserzioni, cancellazioni o sostituzioni di caratteri) necessarie per trasformare una stringa in un'altra.

FUNZIONE UTILIZZATA:
Per l'esecuzione di questo esercizio sono state implementate due versioni per il calcolo della distanza di edit, uno ricorsivo semplice e uno invece che sfrutta la programmazione dinamica. Quest'ultimo è stato il metodo poi utilizzato per la correzione delle parole. L'approccio dinamico per il calcolo della distanza di edit è più efficiente rispetto a quello ricorsivo semplice, poiché utilizza una tecnica chiamata **memoization**. In questo metodo, il programma costruisce una matrice per memorizzare i risultati parziali del confronto tra le due stringhe. La matrice è di dimensioni **`(len1 + 1) x (len2 + 1)`**, dove **`len1`** e **`len2`** sono le lunghezze delle stringhe da confrontare. Questo approccio evita di ripetere i calcoli già eseguiti, riducendo drasticamente il tempo di esecuzione, specialmente quando si lavora con stringhe lunghe o quando si confrontano molte parole come in questo caso.

Il programma legge le parole dal file `correctme.txt` e, per ognuna di esse, confronta tutte le parole del dizionario per trovare quelle con la minima distanza di edit. Per fare questo, ogni volta che si analizza una parola da correggere, il file del dizionario viene percorso nuovamente all'inizio in modo che tutte le parole del dizionario possano essere valutate.

Nel file di input, ogni parola è **tokenizzata**, cioè viene letta e separata una alla volta. Per esempio, se una parola nel file da correggere è *"cinqve"*, il programma cercherà tra le parole del dizionario quella che ha la distanza di edit minore. In questo caso, il programma restituisce "cinque" e "cive", perché entrambe hanno una distanza di edit di 2 da "cinqve". La distanza di edit 2 indica che occorrono due operazioni di modifica (inserzioni, cancellazioni o sostituzioni) per trasformare "cinqve" in una parola del dizionario. Per ogni parola corretta, il

programma stampa il risultato, mostrando la parola originaria, la distanza di edit minima e le possibili correzioni.

Prendiamo un altro esempio: "perpeteva" viene confrontata con le parole del dizionario, e tra quelle con la distanza di edit minima troviamo "erpete", "permetteva" e altre parole con una distanza di 3. Questo vuol dire che, partendo da "perpeteva", sono necessarie tre modifiche per ottenere queste parole.

Un aspetto interessante è che, se la parola letta è corretta e presente nel dizionario (come "avevo"), la distanza di edit sarà zero, e il programma la restituirà senza modifiche, come avviene nell'output: "avevo (ED: 0): avevo".

L'output fornisce, per ogni parola da correggere, la distanza di edit e tutte le parole del dizionario che corrispondono a quella distanza minima. Se la parola esatta è trovata, il programma smette di cercare ulteriori corrispondenze e stampa il risultato, rendendo il processo più efficiente.

Ad esempio, nel caso di "squola", il programma trova "suola" con una distanza di edit di 1, suggerendo una piccola correzione per ottenere la parola più simile nel dizionario.

ESEMPIO DI OUTPUT:

Quando (ED: 2): ando, quando, usando

avevo (ED: 0): avevo

cinqve (ED: 2): cinque, cive

anni, (ED: 1): anni

mia (ED: 0): mia

made (ED: 0): made

mi (ED: 0): mi

perpeteva (ED: 3): erpete, permetteva, perpendeva, perpetra, perpetrava, perpetrera, perpetua, perpetuava, perpetue, perpetuera, repeteva, ripeteva

sempre (ED: 0): sempre

che (ED: 0): che

Execution time: 30.598000 seconds

La parte finale del programma misura il tempo di esecuzione complessivo, che nel caso mostrato è stato di circa 30 secondi, indicativo del carico computazionale nel confronto di molte parole. In sintesi, il programma prende una serie di parole, le confronta con un dizionario, e utilizza un approccio dinamico per trovare la migliore correzione possibile.