Relazione progetto Algoritmi

- Esercizio 1

Introduzione

L'esercizio 1 richiede di implementare una libreria che offre una versione ibrida del Merge Sort e del Binary Insertion Sort, chiamata Merge-Binary Insertion Sort. Questo algoritmo sfrutta l'efficienza del Binary Insertion Sort per sottoliste di lunghezza ridotta (specificata dal parametro k), ed utilizza il Merge Sort per liste più lunghe.

Struttura e Funzionalità

La libreria comprende varie funzioni:

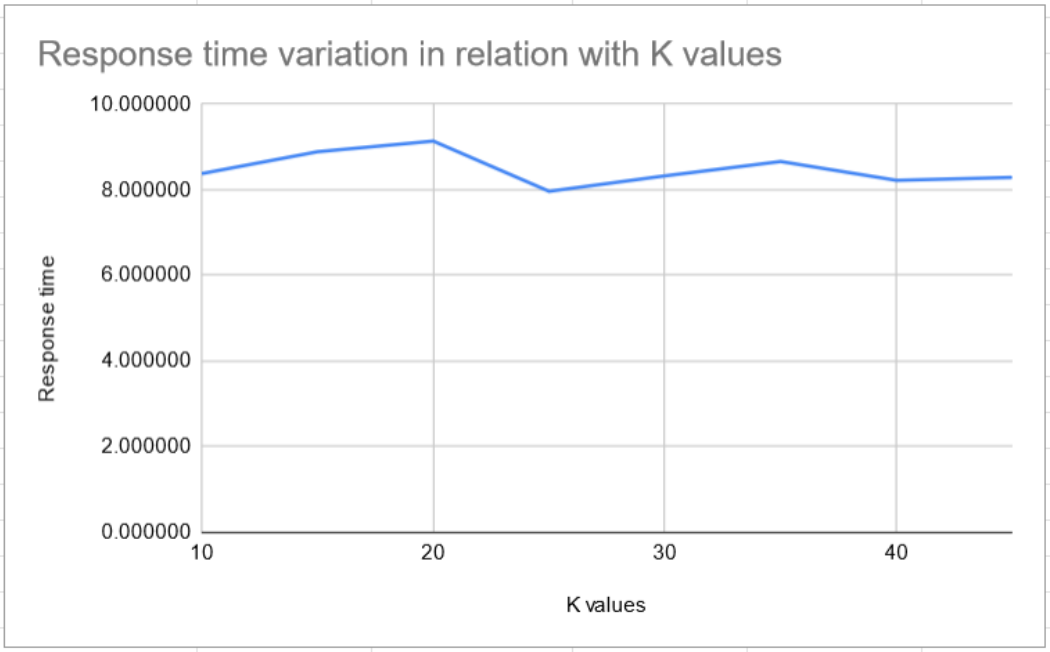
- `merge_binary_insertion_sort:` Implementa un algoritmo di ordinamento ibrido che utilizza merge sort per array di dimensioni superiori a k e binary insertion sort per array più piccoli.
- `merge_sort:` Implementa l'algoritmo di merge sort ricorsivo, suddividendo l'array in due parti e ordinandole separatamente per poi unirle.
- `binary_insertion_sort:` Implementa l'algoritmo di ordinamento per inserzione binaria, che inserisce ogni elemento nella sua posizione corretta nel sottoarray già ordinato.
- `binary_search:` Esegue una ricerca binaria all'interno di un array ordinato per trovare la posizione di inserimento corretta per un dato elemento.
- `merge:` Esegue la fusione di due sottoarray ordinati nel contesto del merge sort, combinando gli elementi in un unico array ordinato.

Scelta del valore: `k`

Abbiamo quindi dovuto scegliere il valore **k** in modo opportuno. Creando un nuovo file C, nel quale poter testare più esecuzioni, ed aggiungendo due clock, abbiamo potuto monitorare i tempi di risposta alla variazione del k stesso.

Risultati ottenuti(media risultati dei 3 field):

K values	Response time
10	8.377926
15	8.888336
20	9.137207
25	7.964554
30	8.324989
35	8.660393
40	8.220894
45	8.288872



Dai risultati ottenuti, notiamo che il **k** dovrebbe essere circa 25. I risultati riscontrati sono più o meno quelli che ci aspettavamo, sapendo che Binary Insertion Sort funziona in modo ottimale con insiemi piccoli.

- Esercizio 2

Introduzione

Nell'ambito di questa implementazione, la SkipList è utilizzata per ottimizzare la ricerca di parole in un testo rispetto a un dato dizionario. In questo caso ci si concentra sulle operazioni fondamentali di una SkipList: inserimento, ricerca e pulizia della memoria.

Struttura e Funzionalità

Il codice è diviso in due file principali: `skiplist.h`, contenente le dichiarazioni, e `skiplist.c` contenente le varie implementazioni.

- La funzione `find_errors` nel file principale utilizza la SkipList per identificare le parole nel testo da correggere che non sono presenti nel dizionario fornito.
- La funzione `normalize_word` assicura che solo i caratteri alfabetici siano considerati, trasformandoli in minuscolo. Questo standardizza le parole per il confronto e la ricerca.
- Le funzioni `insert_skiplist` e `search_skiplist` sono implementate per l'inserimento e la ricerca. Utilizzano il concetto di livelli per accelerare entrambe le operazioni.
- La funzione `clear_skiplist` si occupa della corretta deallocazione della memoria.

Scelta del valore: `max_height`

La scelta di `max_height` nella SkipList è fondamentale per l'equilibrio tra uso della memoria e velocità di esecuzione. Un `max_height` elevato può portare a un aumento del consumo di memoria e potenzialmente rallentare le operazioni di inserimento, a causa del maggior numero di puntatori da gestire. D'altra parte, un `max_height` basso potrebbe ridurre l'efficienza della ricerca, rendendo la struttura più simile a una lista concatenata normale.

E’ quindi opportuno scegliere un `max_height` che si aggira intorno al logaritmo della dimensione del dizionario. Questo offre un compromesso tra velocità di ricerca e uso della memoria.

Nel nostro caso, utilizzando il dizionario presente su gitlab, avremo:
$ln(660000) = 13.4$

In questo modo, i test effettuati hanno mostrato che il sistema è in grado di identificare con successo le parole non presenti nel dizionario con una significativa velocità di esecuzione.

- Esercizio 4

Scelte Implementative

1. **Strutture dati:**
    - *HashMap:* L'utilizzo di HashMap per rappresentare i nodi, i vicini e gli archi, è stato scelto per garantire un accesso efficiente ai dati. La complessità di accesso è O(1), fornendo prestazioni ottimali per le operazioni di grafo.
    - *HashSet:* I collegamenti tra nodi sono memorizzati in HashSet per garantire l'efficienza delle operazioni di aggiunta, rimozione e verifica di presenza.
    - *ArrayList:* Per tenere traccia dei vicini di un nodo, è stata utilizzata un’ ArrayList, poiché fornisce una struttura dati dinamica e flessibile.
2. **Rappresentazione degli archi:**
    - Gli archi sono rappresentati dalla classe `Edge`, che contiene informazioni su due nodi collegati e una possibile etichetta. Questo approccio facilita l'aggiunta e la rimozione di archi, nonché la gestione delle etichette.
    - Per ottimizzare la ricerca degli archi, è stato utilizzato un HashMap (`edgeHashMap`) basato sull'hash dei nodi collegati.

3. **Gestione dei nodi e degli archi:**
    - L'aggiunta di un nodo o di un arco è gestita in modo efficiente verificando prima l'esistenza del nodo o dell'arco mediante l'uso di `containsNode` e `containsEdge`.
    - Per migliorare la flessibilità, la classe supporta grafi diretti e indiretti attraverso la variabile `directed`.
    - In caso di grafi indiretti, l'aggiunta di un arco comporta anche l'aggiunta del suo corrispondente arco inverso per garantire la coerenza delle relazioni di vicinato.
4. **Gestione della rimozione:**
    - La rimozione di un nodo comporta la rimozione di tutti gli archi associati, gestita tramite `removeNode`.
    - La rimozione di un arco aggiorna accuratamente tutte le strutture dati coinvolte per mantenere l'integrità del grafo.