Relazione ASD 2022/2023

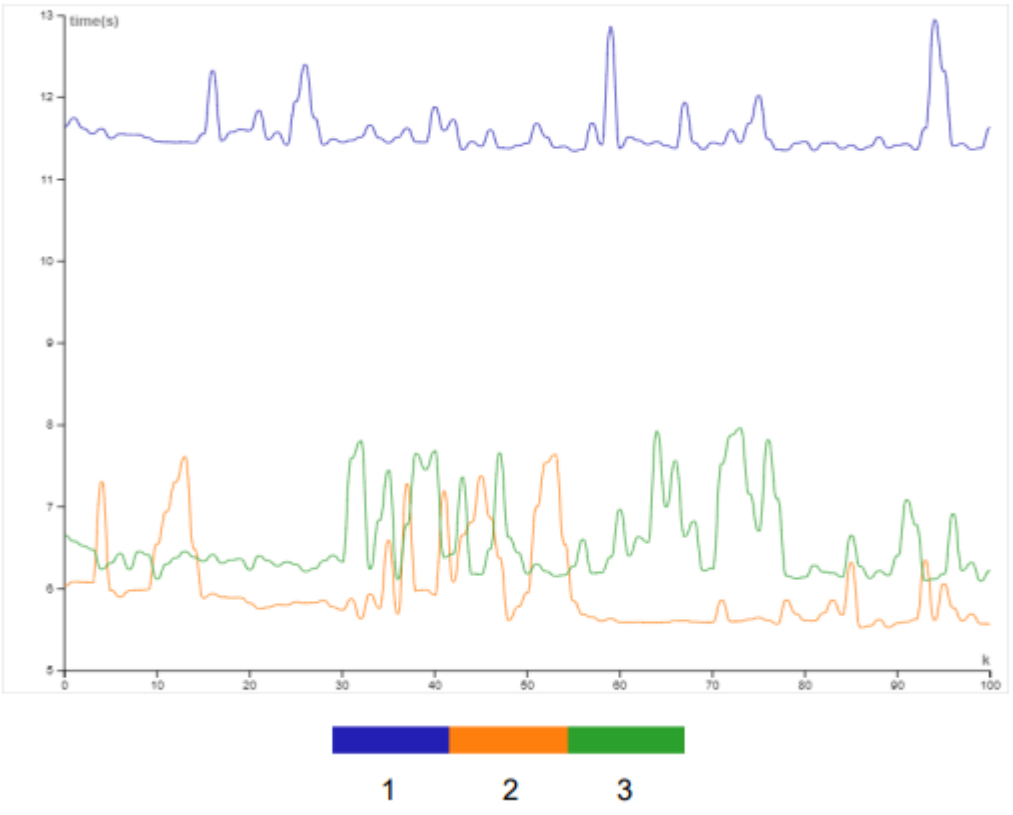
Simone Tantillo, Giuseppe Raso, Jacopo Malpesi

1. Merge_Binary_Insertion_Sort

La libreria MergeBinaryinsertionSort fornisce le funzioni generiche:

```
void merge_binary_insertion_sort(void *, size_t , size_t , size_t , int (*compar)(const void *, const void*));
void insertionSort(void*,size_t,size_t, int (*compar)(const void*, const void*));
size_t binarySearch(void*, void*, size_t , size_t , int (*compar)(const void *, const void*));
void merge(void*,size_t, size_t,size_t,  int (*compar)(const void*, const void*));
```

merge_binary_insertion_sort approfitta del fatto che, nonostante la complessità nel caso medio e peggiore sia più alta, può ordinare più velocemente di merge_sort quando la sottolista è piccola.



field 1: stringe
field 2: interi
field 3: double

I seguenti test svolti sul file Record.csv fornito con diversi valori del parametro k mostrano, per ogni tipo di dato confrontato, quale è il valore limite per il quale i costi dovuti dall'overhead creato da mergesort diventano superiori ai benefici. Per sotto liste di dimensioni inferiori a questi numeri quindi il binary_insertion_sort diventa l'algoritmo migliore.

I valori di k ottimali sembrano essere influenzati anche dalla macchina e dalle ottimizzazioni in compilazione sulla quale vengono eseguiti i test e, in questo caso sono circa:

- tra i 60 e 70 per valori interi e stringhe
- tra i 20 e 30 per i double

1.1 Utilizzo della libreria

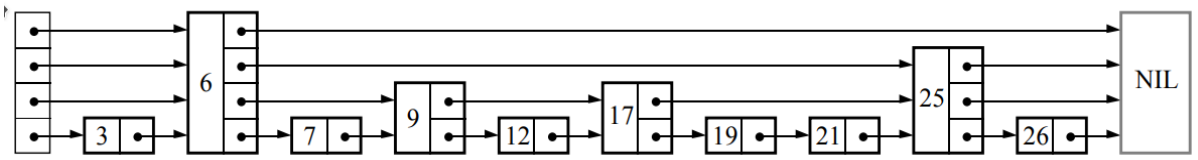
Il makefile fornito compila tutti i file con la formula make all e offre le formule make test e make run per l'esecuzione rispettivamente degli unit test e del file main.c. Per quanto riguarda la configurazione predefinita di make run è:

```
./build/main_ex1 data/records.csv data/sorted.csv data/records.csv data/sorted.csv 40 2
```

ma si può cambiare scrivendo make run ARG0= ARG1=. Una compilazione completa potrebbe essere la seguente:

```
make run ARG0="input.csv output.csv" ARG1="15 3"
```

2. SkipList



La libreria SkipList.h implementa la struttura dati SkipList. La SkipList è un tipo di lista concatenata che memorizza una lista ordinata di elementi, al contrario delle liste concatenate classiche, la SkipList è una struttura dati probabilistica che permette di realizzare l'operazione di ricerca con complessità O(log n) in termini di tempo. Anche le operazioni di inserimento e cancellazione di elementi possono essere realizzate in tempo O(log n). Ogni nodo di una lista concatenata contiene un puntatore all'elemento successivo nella lista. Bisogna quindi scorrere la lista sequenzialmente per trovare un elemento nella lista. La SkipList velocizza l'operazione di ricerca creando delle "vie espresse" che permettono di saltare parte della lista durante l'operazione di ricerca. Questo è possibile perché ogni nodo della SkipList contiene non solo un singolo puntatore al prossimo elemento della lista, ma un array di puntatori che permettono di saltare a diversi punti seguenti nella lista. Le funzioni fornite dalla libreria sono le seguenti:

```
void new_skiplist(struct SkipList **list, size_t max_height, int (*compar)(const void*, const void*));

void clear_skiplist(struct SkipList **list);

void insert_skiplist(struct SkipList *list, void *item);

const void* search_skiplist(struct SkipList *list, void *item);
```

L'altezza della SkipList è un aspetto cruciale che influisce direttamente sulle prestazioni complessive della struttura dati. L'altezza è determinata dal numero di livelli presenti nella Skiplist. Un valore ottimale per l'altezza massima della SkipList dipende dalla dimensione dell'insieme di dati e dalla probabilità di avere puntatori tra i livelli consecutivi. Per scegliere il valore migliore possibile per l'altezza della SkipList, è importante trovare un equilibrio tra la complessità delle operazioni e l'efficienza delle ricerche. In questa implementazione l'altezza della SkipList è generata con una distribuzione geometrica e, dai test eseguiti sul file dictionary.txt, il valore migliore si trova intorno al logaritmo in base 2 della dimensione dei dati che si vogliono inserire(in questo caso circa 19), ma bisogna tenere presente che la SkipList è una struttura dati probabilistica e i risultati potrebbero non essere sempre gli stessi.

2.1 Utilizzo della libreria

Il makefile fornito compila tutti i file con la formula make all e propone le formule make test e make run per l'esecuzione rispettivamente degli unit test e del file main.c. Per quanto riguarda la configurazione predefinita di make run è:

./build/main_ex2 data/dictionary.txt data/correctme.txt 20

ma si può cambiare scrivendo make run ARG0=<path dei file di input e output tra virgolette
separati da spazio>
ARG1=<max height della SkipList>.
Una compilazione completa potrebbe essere la seguente:

make run ARG0="input.csv output.csv" ARG1="35"

3. Priority Queue, grafi sparsi e MSF

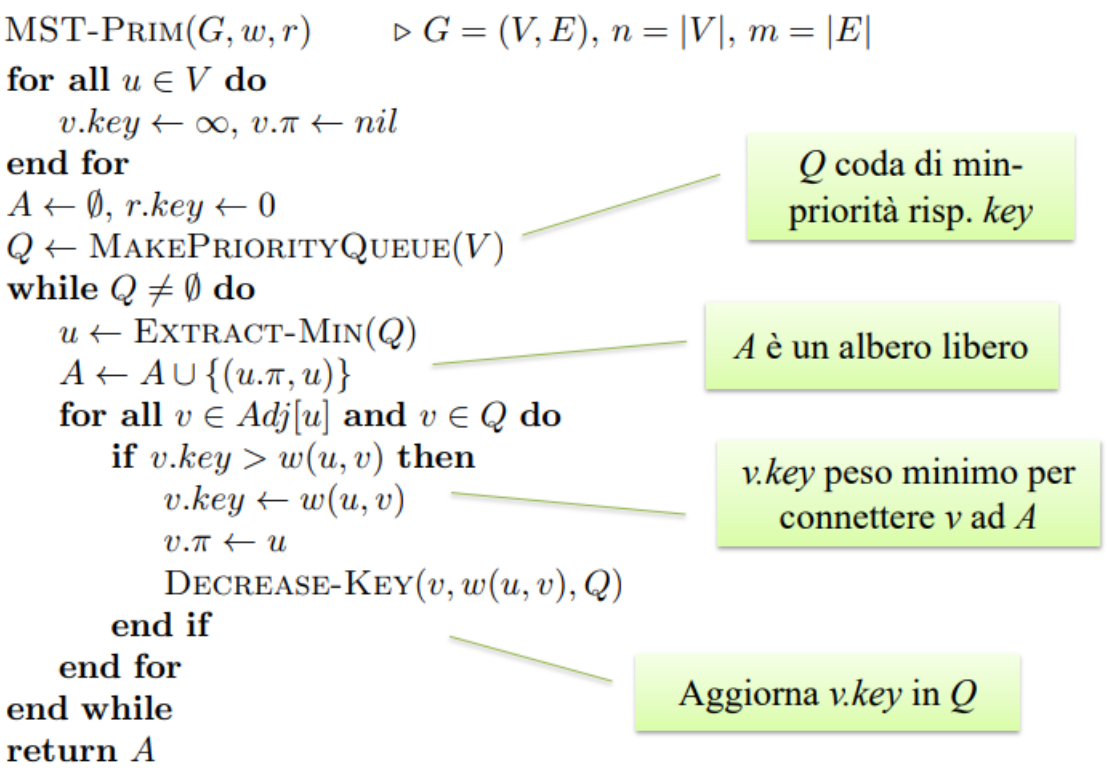
La libreria "**PriorityQueue**" implementa una struttura dati "Priority Queue" utilizzando un heap binario. Una PriorityQueue è una struttura dati che mantiene una coda di elementi, ognuno dei quali è associato a una priorità. Gli elementi con una priorità più alta vengono estratti prima dagli elementi con una priorità più bassa. L'implementazione della PriorityQueue si basa sulla classe "Binary Heap", che rappresenta l'heap binario. L'heap binario è una struttura dati che mantiene gli elementi in un array, organizzandoli in modo che l'elemento principale(con la priorità più alta) sia posizionato all'indice 0 dell'array. Inoltre, ogni elemento dell'heap ha un padre, un figlio sinistro e un figlio destro secondo una particolare relazione di ordinamento definita da un comparatore. Le funzioni della fornite dalla libreria sono:

```
public interface AbstractQueue<E> {
    public boolean empty(); // controlla se la coda è vuota -- O(1)
    public boolean push(E e); // aggiunge un elemento alla coda -- O(logN)
    public boolean contains(E e); // controlla se un elemento è in coda -- O(1)
    public E top(); // accede all'elemento in cima alla coda -- O(1)
    public void pop(); // rimuove l'elemento in cima alla coda -- O(logN)
    public boolean remove(E e); // rimuove un elemento se presente in coda -- O(logN)
};
```

Notare che per poter cercare un elemento in un heap con complessità O(1) è necessaria una hashmap per tenere traccia di tutti gli elementi presenti. Questa aggiunta permette quindi la risoluzione della funzione contains in O(1) e conseguentemente la remove in O(logN).

La classe "**Graph**" implementa la struttura di dati del grafo utilizzando una HashMap di adiacenza, che offre un modo efficiente per gestire i nodi e gli archi del grafo. Le operazioni di aggiunta, rimozione e ricerca dei nodi e degli archi sono eseguite in tempo costante o lineare rispetto alla dimensione del grafo, garantendo prestazioni efficienti per la gestione dei grafi di dimensioni variabili.

La classe "**Prim**" fornisce metodi per, dato un grafo passato come file di estensione csv, trovare la foresta di copertura minima del grafo. Il metodo minimumSpanningForest implementa l'algoritmo di Prim per trovare la foresta di copertura minima di un grafo. Prende in input un grafo e restituisce una collezione di archi che rappresentano l'albero di copertura minimo. Il metodo utilizza una coda di priorità per selezionare gli archi di peso minimo da aggiungere all'albero e la classe node per estendere ogni nodo con informazioni utili all'algoritmo.



L'algoritmo di prim esegue passo passo:

Per ogni nodo u nel grafo G:

- u.key viene impostato a infinito, indicando che il nodo non è ancora stato incluso nella MSF.
- u.parent viene impostato a null, indicando che il nodo non ha ancora un genitore nella msf.

2. Viene creata una coda Q contenente tutti i nodi nel grafo.
3. Fino a quando Q non è vuota.
4. Viene estratto il nodo con la chiave minima da Q e assegnato a u. Questo nodo viene aggiunto alla foresta di copertura minima.
5. Per ogni nodo v adiacente a u nel grafo G:

- Se v è presente in Q (ancora non incluso nella MSF) e il peso dell'arco tra u e v è minore della chiave corrente di v, allora:
  - v.parent viene impostato a u, indicando che u è il genitore di v nell'MSF.
  - v.key viene aggiornato con il nuovo peso
  - La priorità di v viene ridotta nella coda Q per riflettere il cambiamento della chiave.