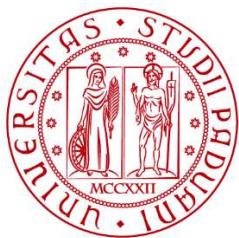


Università degli studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Triennale in Ingegneria dell'Informazione



UNIVERSITÀ
DEGLI STUDI
DI PADOVA



**Studio di algoritmi efficienti per il campionamento ed applicazione a
dataset di sequenze di DNA**

Relatore:

Prof. Fabio Vandin

Laureando:

Federico Berton

ANNO ACCADEMICO 2020-2021

Data di laurea 23/07/2021

Sommario

1	Introduzione	4
2	Descrizione dell'algoritmo sviluppato nelle sue varianti	5
2.1	Scopo dell'analisi dell'algoritmo	5
2.2	Versioni dell'algoritmo.....	6
2.3	Descrizione e analisi della struttura dell'algoritmo.....	7
2.3.1	Parametri di ingresso e uscita	7
2.3.2	Struttura dell'algoritmo.....	9
2.3.3	Differenze strutturali tra <i>Massive R-W</i> e <i>Iterate R-W</i> version	10
2.3.4	Differenze strutturali tra <i>Full version</i> e <i>Partial version</i>	11
2.4	Applicazione a supporto dell'algoritmo.....	13
2.4.1	Versione da riga di comando.....	14
2.4.2	Versione con interfaccia grafica	15
2.4.3	Applicazioni di supporto per test	16
2.5	Analisi della complessità computazionale	19
2.5.1	Calcolo della complessità computazionale asintotica al caso pessimo.....	19
3	Risultati sperimentali.....	23
3.1	Raccolta dei dati.....	24
3.1.1	Tempi di esecuzione	24
3.1.2	Record letti dal dataset di input	26
3.2	Selezione dei dati utili all'analisi.....	27
3.2.1	Tempo totale e tempo di CPU	27
3.2.2	Versioni GUI e versioni CLI.....	29
3.2.3	Massive R-W e Iterate R-W version.....	30
3.3	Analisi dei dati raccolti in termini di tempo di esecuzione.....	32
3.3.1	Dipendenza del tempo di esecuzione dal parametro n	32

3.3.2	Dipendenza del tempo di esecuzione dal parametro k.....	35
3.3.3	Dipendenza del tempo di esecuzione dal tipo di versione (full o partial).....	37
3.4	Analisi dei dati raccolti in termini di correttezza	41
3.4.1	Analisi di tutto il dataset in input	41
3.4.2	Omogeneità di scrittura ad ogni iterazione.....	42
3.4.3	Contenimento della scrittura multipla dello stesso record.....	49
4	Conclusioni	53
5	Appendice.....	56
5.1	Appendice 1: codice sorgente delle applicazioni.....	56
5.1.1	Codice sorgente delle classi comuni.....	60
5.1.2	Codice sorgente delle classi singolari GUI	72
5.1.3	Codice sorgente delle classi singolari CLI	77

1 Introduzione

Questo elaborato si ripropone di sviluppare e studiare un algoritmo efficiente di campionamento uniforme con reinserimento per dataset con elevate quantità di dati; questo algoritmo verrà inserito all'interno della ricerca *SPRISS*, chi si occupa di implementare un algoritmo efficiente più complesso che, dato un dataset di sequenze di RNA, identifichi la presenza di k-mer (sottosequenze dei record che compongono il dataset) frequenti, indicati con (*FKs*), contenuti nel dataset. L'utilità di questa ricerca è data dal fatto che, dato un insieme di materiale genetico, per numerose applicazioni non è utile tanto considerare tutte le sue sottosequenze, quanto prendere in analisi solo quelle che compaiono più frequentemente, e che quindi sono presenti in maniera preponderante all'interno della popolazione. Esempi di applicazioni in cui è fondamentale questo aspetto sono la caratterizzazione di variazioni all'interno delle sequenze di RNA, l'analisi dei cambiamenti strutturali nel genoma, lo sviluppo di metodi per la quantificazione del genoma, la correzione di errori nell'assemblaggio del materiale genetico e molto altro.

L'algoritmo *SPRISS*, quindi, si inserisce tra le tecnologie *NGS (Next Generation Sequencing)*, che si ripropongono di trovare strumenti validi per il sequenziamento di grandi genomi in un tempo ristretto. L'algoritmo che viene sviluppato in questo elaborato sarà inserito nella sezione preliminare dell'algoritmo *SPRISS*, in quanto permette di poter lavorare con un dataset rappresentativo di dimensioni ridotte (a scelta) invece di considerare il dataset totale, mantenendo comunque le stesse caratteristiche informative del genoma di partenza. La difficoltà nell'individuazione di k-mer frequenti, infatti, risiede nelle elevatissime risorse utilizzate (in termini di tempi di esecuzione e di memoria) dall'algoritmo di ricerca; per un dataset di 729 Gbases, ad esempio, viene richiesto un tempo di esecuzione di circa 2.5 ore ed un consumo di memoria pari a 34 GB e 500 GB di spazio nel disco. Riducendo le dimensioni del dataset di analisi e utilizzando quest'ultimo per la ricerca degli *FKs*, anche l'utilizzo di queste risorse scenderà significativamente: per lo stesso dataset da poco citato, grazie all'algoritmo *SPRISS* il tempo di esecuzione si riduce a 30 minuti, con 300 GB di memoria utilizzati e 97 GB di spazio nel disco (1).

L'algoritmo in questione verrà sviluppato in più versioni e verrà studiato sia con un approccio teorico che con uno più pratico, in modo da poter ricavare la correlazione tra tempi di esecuzione e parametri di ingresso, e poter valutare quali siano le migliori combinazioni dei parametri in input per ottenere realizzazioni più efficienti.

2 Descrizione dell'algoritmo sviluppato nelle sue varianti

L'algoritmo sviluppato, da inserire all'interno del progetto SPRISS, ha il compito, dato un dataset di partenza con grandi quantità di dati, di crearne uno alternativo con dimensioni fissate molto più ridotte, ma che allo stesso tempo contenga le stesse caratteristiche informative di quello originale. Per concretizzare quest'ultima caratteristica, ovvero per far in modo che le informazioni genetiche dei due dataset si equivalgano, viene utilizzato un meccanismo di campionamento randomico uniforme con reinserimento.

L'algoritmo verrà sviluppato in più versioni, ognuna delle quali porterà presumibilmente a risultati differenti; inoltre, per poter eseguire dei test in maniera più funzionale ed efficace, verrà sviluppata un'applicazione (anch'essa in più versioni) che permetterà di eseguire l'algoritmo consecutivamente in modo semplice e quindi di raccogliere i dati più agevolmente.

2.1 Scopo dell'analisi dell'algoritmo

L'analisi contenuta in questo elaborato si ripropone di raccogliere e analizzare i risultati dell'algoritmo in termini di tempo di esecuzione e correttezza, esaminandone gli esiti con configurazioni diverse dei parametri di ingresso. I tempi di esecuzione esaminati, in particolare, saranno due:

- *Wall time*: tempo totale impiegato dall'elaboratore per l'esecuzione dell'algoritmo;
- *CPU time*: tempo speso dalla CPU per lo svolgimento dell'algoritmo, escludendo il tempo impiegato per le operazioni di input-output e per l'esecuzione parallela di altri programmi.

L'analisi quindi si ripropone di esaminare i tempi di esecuzione al variare dei parametri di input, per riuscire ad individuare una relazione tra input e tempo impiegato per lo svolgimento della procedura.

Avendo sviluppato versioni diverse dell'algoritmo, e anche dell'applicazione che ne permette l'esecuzione, verranno inoltre realizzati dei confronti tra i risultati ottenuti nelle molteplici varianti.

Scopo ultimo dell'analisi sarà quello di identificare le migliori conformazioni di parametri di ingresso per rendere l'algoritmo più efficiente.

2.2 Versioni dell'algoritmo

Le versioni dell'algoritmo si differenzieranno in base a:

- *Porzione del record scritta in output:* come si vedrà meglio in seguito, ogni record del dataset in input contiene una sola riga significativa delle quattro di cui è composto; si avranno quindi due versioni:
 - a. *Full version:* in output compariranno tutte e quattro le righe per ogni record;
 - b. *Partial version:* in output riporterà la sola riga significativa nel dataset d'uscita.
- *Modalità di lettura e scrittura dei file di ingresso e uscita:* per leggere i record dal dataset di partenza e scriverli in quello di output è possibile seguire due vie, alle quali corrispondono due differenti versioni dell'algoritmo:
 - a. *Massive reading-writing version:* scrive in memoria (in una struttura dati di supporto) tutti i record del dataset di partenza, ne esegue il campionamento scrivendo in memoria i record selezionati (in una seconda struttura dati), e solo successivamente li scrive massivamente nel file di output;
 - b. *Iterated reading-writing version:* legge il record dal dataset iniziale solo nel momento in cui l'algoritmo lo necessita (durante la procedura di campionamento), e nello stesso momento scrive il record nel file di uscita, senza quindi mantenere in memoria tutti i record di input e quelli di output.

2.3 Descrizione e analisi della struttura dell'algoritmo

2.3.1 Parametri di ingresso e uscita

Entrando nell'analisi più specifica dell'algoritmo, i parametri di ingresso che riceverà saranno:

- Un dataset D contenente m record (ovvero con cardinalità $|D| = m$); ognuno degli m record è composto da $l = 4$ righe, delle quali solo la terza è significativa; immagineremo quindi ogni record con la seguente struttura:



Figura 2.1 - struttura dei record del dataset

- Il numero di record desiderati nel dataset di output $n \in [1, m]$;
- Un parametro $k \in [1, m]$.

In output invece verrà prodotto un solo output, ovvero il dataset D' , con $|D'| = n$ e con le caratteristiche citate precedentemente. In base alla versione scelta (*full* o *partial*) i record avranno struttura differente: nel primo caso la struttura sarà la stessa mostrata in **Errore**. **L'origine riferimento non è stata trovata.**, mentre nel secondo caso ogni record sarà composto da una sola riga:



Figura 2.2 - struttura dei record di output per partial version

Lo schema a blocchi di input-output dell'algoritmo sarà quindi il seguente:



Figura 2.3 - Schema input-output dell'algoritmo

2.3.2 Struttura dell'algoritmo

Il diagramma a blocchi dell'algoritmo è rappresentabile come segue



da cui si apprende che il fulcro dell'algoritmo, su cui quindi si concentreranno le analisi, è il secondo blocco, ovvero il campionamento dei record letti in entrata.

La struttura e il funzionamento del campionamento può essere appreso analizzandone il relativo pseudocodice riportato di seguito:

```

1   while ( $n > 0$ ) do
2        $k = \min(k, m)$ ;
3        $c \leftarrow$  campione random ottenuto dalla variabile aleatoria  $X \sim \text{Binomial}(n, k/m)$ ;
4        $S \leftarrow$  vettore con  $|S| = c$  ottenuto con campionamento uniforme
5           con reinserimento dall'intervallo  $[1, k]$ ;
6       forall  $j \in [1, k]$  do
7            $r \leftarrow$  carica un nuovo record da  $D$ ;
8            $r_c \leftarrow$  numero di occorrenze dell'indice  $j$  in  $S$ ;
9            $\text{output} \leftarrow r_c$  copie del record  $r$ ;
10      end
11       $n \leftarrow n - c$ ;
12       $m \leftarrow m - k$ ;
13  end
  
```

Figura 2.4 - pseudocodice generale dell'algoritmo

L'algoritmo presenta un ciclo *while* più esterno, che termina quando il parametro n diventa negativo; ad ogni iterazione del ciclo:

- Al parametro k viene assegnato il minimo tra sé stesso e il numero di record richiesti in output;
- Viene preso un campione c casuale da una variabile aleatoria binomiale di parametri n (prove effettuate) e k/m (probabilità di successo della singola prova);
- Viene creato un vettore S di c interi presi dal campionamento uniforme con reinserimento degli interi contenuti nell'intervallo $[1, m]$;
- Viene eseguito un ciclo *for* con indice j che parte da 1 e termina al valore k ; ad ogni iterazione del ciclo:

- i. Viene caricato un record r in maniera sequenziale dal dataset di partenza;
 - ii. Viene calcolato il numero r_c di occorrenze dell'indice j all'interno del vettore S ;
 - iii. Viene copiato il record r esattamente r_c volte nel dataset di output;
- e) Viene aggiornato il parametro n , sottraendo ad esso il valore c , ovvero il numero di record scritti nella corrente iterazione del ciclo *while*;
- f) Viene aggiornato il parametro m , sottraendo ad esso il parametro k .

2.3.3 Differenze strutturali tra *Massive R-W* e *Iterate R-W version*

Come già esplicato, la differenza tra queste due versioni dell'algoritmo sta nella modalità di lettura e scrittura dei dataset di input e output; questa differenza può essere tradotta a livello di pseudocodice per comprendere meglio, nei due differenti casi, in quali istanti l'algoritmo effettui le operazioni di input-output:

1. *Massive R-W version*: nello pseudocodice seguente verranno indicate con I la struttura dati che contiene i record del dataset in input, mentre con O quella che contiene i record da scrivere nel dataset in output:

```

1  forall (record contenuti in inputFile) do
2      I  $\leftarrow$  leggi e inserisci record corrente;
3  end
4  while ( $n > 0$ ) do
5       $k = \min(k, m)$ ;
6       $c \leftarrow$  campione random ottenuto dalla variabile aleatoria  $X \sim \text{Binomial}(n, k/m)$ ;
7       $S \leftarrow$  vettore con  $|S| = c$  ottenuto con campionamento
           uniforme con reinserimento dall'intervallo  $[1, k]$ ;
8      forall  $j \in [1, k]$  do
9           $r \leftarrow$  carica un nuovo record da  $I$ ;
10          $r_c \leftarrow$  numero di occorrenze dell'indice  $j$  in  $S$ ;
11          $O \leftarrow$  inserisci  $r_c$  copie del record  $r$  in apposita struttura dati;
12     end
13      $n \leftarrow n - c$ ;
14      $m \leftarrow m - k$ ;
15   end
16   forall (record contenuti in  $O$ ) do
17       outputFile  $\leftarrow$  scrivi record corrente;
18   end

```

Figura 2.5 - pseudocodice per *Massive R-W version*

2. Iterate R-W version:

```

1   while ( $n > 0$ ) do
2        $k = \min(k, m)$ ;
3        $c \leftarrow$  campione random ottenuto dalla variabile aleatoria  $X \sim \text{Binomial}(n, k/m)$ ;
4        $S \leftarrow$  vettore con  $|S| = c$  ottenuto con campionamento uniforme
5           con reinserimento dall'intervallo  $[1, k]$ ;
6       forall  $j \in [1, k]$  do
7            $r \leftarrow$  legge un nuovo record da inputFile;
8            $r_c \leftarrow$  numero di occorrenze dell'indice  $j$  in  $S$ ;
9           outputFile  $\leftarrow$  scrive  $r_c$  copie del record  $r$ ;
10      end
11       $n \leftarrow n - c$ ;
12       $m \leftarrow m - k$ ;
13  end

```

Figura 2.6 - pseudocodice per Iterate R-W version

Nei due pseudocodici sono state evidenziate le linee di codice in cui vengono effettuate le operazioni sui file di lettura e scrittura vera e propria.

È immediato verificare come, oltre al punto dell'algoritmo in cui si trovano le operazioni sul disco di lettura e scrittura, anche il numero di queste operazioni varierà: infatti l'algoritmo non per forza necessita tutti i record del dataset di input per il suo funzionamento, in quanto il parametro n potrebbe diventare negativo (e quindi causare la terminazione del ciclo *while*) prima che l'ultimo record del dataset originario venga caricato in memoria. Si analizzerà con più attenzione quest'aspetto nel paragrafo 3.2.2.

La differente modalità di lettura e scrittura, e il diverso numero di operazioni di input-output di queste due versioni, porterà presumibilmente a risultati ineguali.

2.3.4 Differenze strutturali tra *Full version* e *Partial version*

A differenza della trattazione svolta per il confronto tra le versioni *Massive R-W* e *Iterated R-W*, per queste due versioni è possibile far riferimento ad un unico schema di pseudocodice (quello generale mostrato in Figura 2.4), con l'unica differenza che le righe 7 e 9 (ovvero le righe che operano sui record in lettura/scrittura) possono essere espansse in modi differenti:

1. *Full version*: indicando con r_0, r_1, r_2, r_3 le righe che compongono il record r

7 $\left\{ \begin{array}{l} r_0 \leftarrow \text{carica un nuova riga da } D; \\ r_1 \leftarrow \text{carica un nuova riga da } D; \\ r_2 \leftarrow \text{carica un nuova riga da } D; \\ r_3 \leftarrow \text{carica un nuova riga da } D; \end{array} \right.$

9 $\left\{ \begin{array}{l} \textbf{for } i \in [0, r_c] \textbf{ do} \\ \quad \text{output} \leftarrow \text{riga } r_0 \text{ di } r; \\ \quad \text{output} \leftarrow \text{riga } r_1 \text{ di } r; \\ \quad \text{output} \leftarrow \text{riga } r_2 \text{ di } r; \\ \quad \text{output} \leftarrow \text{riga } r_3 \text{ di } r; \\ \textbf{end} \end{array} \right.$

2. *Partial version*:

7 $\left\{ \begin{array}{l} \text{carica un nuova riga da } D \text{ senza salvarla;} \\ \text{carica un nuova riga da } D \text{ senza salvarla;} \\ r \leftarrow \text{carica un nuova riga da } D; \\ \text{carica un nuova riga da } D \text{ senza salvarla;} \end{array} \right.$

9 $\left\{ \begin{array}{l} \textbf{for } i \in [0, r_c] \textbf{ do} \\ \quad \text{output} \leftarrow \text{riga } r_2 \text{ di } r; \\ \textbf{end} \end{array} \right.$

Si vedrà nel seguito come questa differenza porterà a differenze sostanziali anche nei tempi di esecuzione delle due versioni.

2.4 Applicazione a supporto dell'algoritmo

Con lo scopo di agevolare l'utilizzo dell'algoritmo, è stata sviluppata un'applicazione in varie versioni differenti:

1. *Versione da riga di comando*: permette l'inserimento dei dati di input e la visualizzazione dei risultati di output (tempi di esecuzione) direttamente dalla consolle;
2. *Versione con interfaccia grafica*: presenta una semplice GUI che permette l'impostazione dei dati in input e la valutazione dei tempi ottenuti direttamente dalla finestra dell'applicazione.

Le applicazioni sviluppate, scritte in linguaggio C++ tramite l'ambiente di sviluppo integrato *Qt Creator*, hanno lo scopo di semplificare la raccolta dei dati, in particolare di:

- Eseguire l'algoritmo più volte consecutivamente senza interrompere l'esecuzione dell'applicazione;
- Impostare i parametri di ingresso, ovvero:
 - Percorso del file (*input-file-path*) che costituisce il dataset di partenza;
 - Percorso della cartella (*output-directory-path*) che conterrà i risultati in output;
 - Parametro *n*;
 - Parametro *k*.
- Scegliere che versione dell'algoritmo eseguire (*full* o *partial*); in aggiunta l'applicazione offre la possibilità di eseguire una versione *comparing*, ovvero di eseguire entrambe le versioni sopracitate e stampare in output i risultati di entrambe le versioni per poterli confrontare;
- Ottenere in uscita, oltre al dataset di output, un file di log contenente:
 - Un compendio dei parametri di ingresso;
 - Il nome del file che costituisce il dataset di output;
 - Il numero di record scritti in tale dataset;
 - *Wall-time* speso dall'algoritmo;
 - *CPU time* speso dall'algoritmo.
- Visualizzare nella finestra dell'applicazione gli stessi risultati stampati nel file di log

```

----- Outcome log - full version -----
Input dataset path: C:\Users\fberton\Desktop\Tesi\dataset.fastq
Parameter k: 1
Parameter n: 100705
Output dataset name: outData_full_n100705_k1_202143194936.fastq
Written records to output-dataset: 100705
Wall time spended by the algorithm: 6.049257 seconds
CPU time spended by the algorithm: 6.046875 seconds

```

Figura 2.7 – esempio di struttura del file di log

2.4.1 Versione da riga di comando

La versione più semplice, ovvero quella da riga di comando, una volta lanciata apre un’istanza del prompt dei comandi che:

- Informa l’utente sul funzionamento dell’applicazione;
- Richiede in input i parametri necessari;
- Richiede la versione desiderata per eseguire l’algoritmo;
- Esegue l’algoritmo;
- Dà la possibilità, una volta terminata l’esecuzione dell’algoritmo, di chiudere l’applicazione, rieseguire l’algoritmo con la stessa configurazione di parametri, oppure settare nuovamente i parametri di ingresso.

```

Input:
- dataset D with |D| = m reads of 4 lines (only 3^ is significant)
- sample size n
- parameter k in [1, m]
Output:
- file with n reads of D chosen at random uniformly with replacement
- log file with execution times
Version:
- Full : in the output dataset are written all four lines
- Partial : in the output dataset is written only significant line
- Comparing : runs both versions and compares them

Insert the desired input-file path:
C:\dataset.fastq

Insert the desired output-file path
C:\Output

Insert the desired sample size (parameter n):
755288

Insert the desired parameter k
10

```

```

Insert the desired version:
  0 -> Full version
  1 -> Partial version
  2 -> Comparing version
1

----- Outcome log - partial version -----
Input dataset path: C:\dataset.fastq
Parameter k: 10
Parameter n: 755288
Output dataset name: outData_partial_n755288_k10_2021415191644.fastq
Written records to output-dataset: 755288
Wall time spended by the algorithm: 10.071 seconds
CPU time spended by the algorithm: 9.875 seconds

Press:
  0 -> Close application
  1 -> Re-run algorithm with the same configuration
  2 -> Set input parameters and re-run algorithm

```

Figura 2.8 - esempio di esecuzione dell'applicazione da riga di comando

2.4.2 Versione con interfaccia grafica

La versione da riga di comando fornisce le stesse funzionalità della precedente, ma in aggiunta permette di interfacciarsi con l'applicazione tramite una *Graphical User Interface* di cui di seguito si mostra la struttura e un esempio della sua esecuzione:

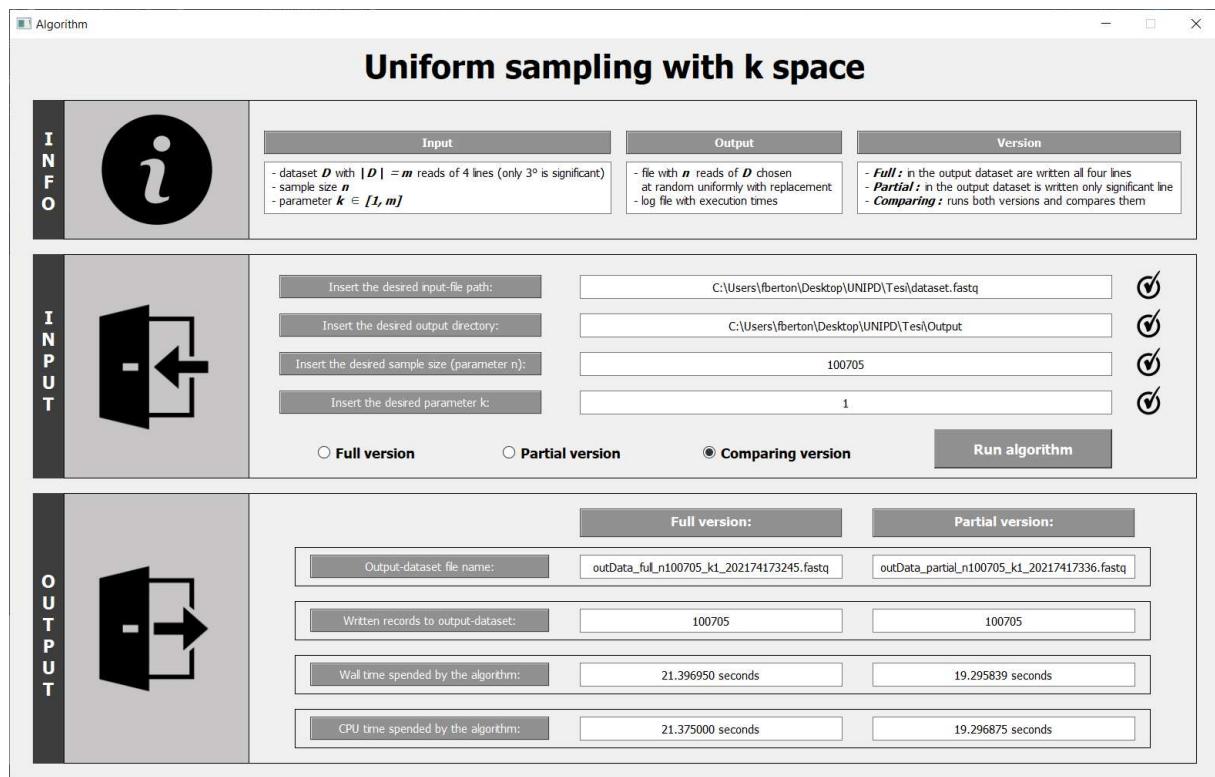


Figura 2.9 - esempio di esecuzione dell'applicazione con GUI

2.4.3 Applicazioni di supporto per test

Visto l'elevato numero di test da effettuare necessari per una corretta analisi dell'algoritmo, e considerato che per uno studio completo è necessario raccogliere altri dati oltre ai tempi di esecuzione, sono state sviluppate altre due applicazioni, una da riga di comando (*CLI-TOT*) e l'altro con GUI (*GUI-TOT*), che permettono di:

- Inserire più valori per i parametri di ingresso n e k , in modo da poter effettuare più test in un'unica esecuzione senza dover reinserire i dati di input;
- Per tutte le combinazioni dei dati inseriti, eseguire tutte le versioni sviluppate per l'algoritmo: *full iterate version*, *full massive version*, *partial iterate version*, *partial massive version*;
- Per l'applicazione con GUI, visualizzare in tempo reale quanti test sono stati effettuati per ogni tipo di algoritmo;
- Visualizzare nei file “executionTimesCPU.csv” e “executionTimesWall.csv” tutti i risultati in termini di tempi di esecuzione in forma tabellare;

Iterate Full Version				Iterate Partial Version			
	$n = 100705$	$n = 100705$	$n = 251763$		$n = 100705$	$n = 100705$	$n = 251763$
$k = 1$	6,3125	8,203125	13,484375	$k = 1$	3,296875	3,828125	5,4375
$k = 10$	8,109375	6,796875	12,609375	$k = 10$	3,75	3,609375	5,171875
Massive Full Version				Massive Partial Version			
	$n = 100705$	$n = 100705$	$n = 251763$		$n = 100705$	$n = 100705$	$n = 251763$
$k = 1$	6,46875	7,046875	13,75	$k = 1$	3,5625	3,609375	5,109375
$k = 10$	7,203125	6,90625	12,625	$k = 10$	3,515625	3,484375	4,921875

Figura 2.10 - esempio di file .csv di output per tempi di esecuzione

- Visualizzare nel file “parameterChanges.csv” l’evoluzione dei parametri n , m , c e dei record duplicati ad ogni iterazione per tutti i possibili valori di k . Essendo questi dati indipendenti dal tipo di algoritmo utilizzato (*full iterate version*, *full massive version*, *partial iterate version*, *partial massive version*) per queste informazioni è stato inserito un solo valore per ogni n e k ;

n = 100705, k = 1				
Iterations:	m decrease:	n decrease:	c values:	Duplicated r
1	1007050	100705	0	0
2	1007049	100705	0	0
3	1007048	100705	0	0
4	1007047	100705	0	0
5	1007046	100705	0	0
6	1007045	100705	0	0
7	1007044	100705	0	0
8	1007043	100705	0	0
9	1007042	100705	0	0
10	1007041	100705	0	0
11	1007040	100705	0	0
12	1007039	100705	0	0
13	1007038	100705	0	0
14	1007037	100705	0	0
15	1007036	100705	0	0
16	1007035	100705	0	0
17	1007034	100705	0	0
18	1007033	100705	0	0
19	1007032	100705	0	0
20	1007031	100705	0	0
21	1007030	100705	0	0
22	1007029	100705	0	0
23	1007028	100705	0	0
24	1007027	100705	0	0
25	1007026	100705	0	0

n = 251763, k = 1				
Iterations:	m decrease:	n decrease:	c values:	Duplicated r
1	1007050	251763	1	0
2	1007049	251762	1	0
3	1007048	251761	1	0
4	1007047	251760	1	0
5	1007046	251759	1	0
6	1007045	251758	1	0
7	1007044	251757	1	0
8	1007043	251756	1	0
9	1007042	251755	1	0
10	1007041	251754	1	0
11	1007040	251753	1	0
12	1007039	251752	1	0
13	1007038	251751	1	0
14	1007037	251750	1	0
15	1007036	251749	1	0
16	1007035	251748	1	0
17	1007034	251747	1	0
18	1007033	251746	1	0
19	1007032	251745	1	0
20	1007031	251744	1	0
21	1007030	251743	1	0
22	1007029	251742	1	0
23	1007028	251741	1	0
24	1007027	251740	1	0
25	1007026	251739	1	0

n = 503525, k = 1				
Iterations:	m decrease:	n decrease:	c values:	Duplicated r
1	1007050	503525	1	0
2	1007049	503524	1	0
3	1007048	503523	1	0
4	1007047	503522	1	0
5	1007046	503521	1	0
6	1007045	503520	1	0
7	1007044	503519	1	0
8	1007043	503518	1	0
9	1007042	503517	1	0
10	1007041	503516	1	0
11	1007040	503515	1	0
12	1007039	503514	1	0
13	1007038	503513	1	0
14	1007037	503512	1	0
15	1007036	503511	1	0
16	1007035	503510	1	0
17	1007034	503509	1	0
18	1007033	503508	1	0
19	1007032	503507	1	0
20	1007031	503506	1	0
21	1007030	503505	1	0
22	1007029	503504	1	0
23	1007028	503503	1	0
24	1007027	503502	1	0
25	1007026	503501	1	0

Figura 2.11 - esempio di file .csv di output per l’evoluzione dei parametri ad ogni iterazione

- Visualizzare nel file “inputRecordSelected.csv” i record di input letti in forma tabellare per ogni combinazione di parametri in ingresso e per ogni versione di algoritmo utilizzata; anche questi dati risultano indipendenti dalla tipologia di algoritmo utilizzata.

Input record selected				
	n = 100705	n = 251763	n = 503525	n = 755288
k = 1	1007044	1007044	1007044	1007044
k = 10	1006990	1006990	1007050	1007050
k = 100	1007050	1007050	1007050	1007050
k = 1000	1007050	1007050	1007050	1007050
k = 10000	1007050	1007050	1007050	1007050
k = 50000	1007050	1007050	1007050	1007050
k = 100000	1007050	1007050	1007050	1007050
k = 500000	1007050	1007050	1007050	1007050
k = 1000000	1007050	1007050	1007050	1007050

Figura 2.12 - esempio di file .csv per il numero di record di input selezionati

La modalità di introduzione dei dati rimane la stessa, con l’unica differenza dell’inserimento dei parametri k e n : per questi parametri può essere inserito un numero a piacere di valori, separati da un carattere di punto e virgola.

L’output per la versione da riga di comando rimane lo stesso (ovvero il log dell’esecuzione di ogni test man mano che vengono effettuati), mentre per la versione con GUI vengono mostrati i nomi dei file .csv di output e il numero di test effettuati con una progress-bar che ne indica l’avanzamento. Di seguito ne viene mostrato un esempio di esecuzione:

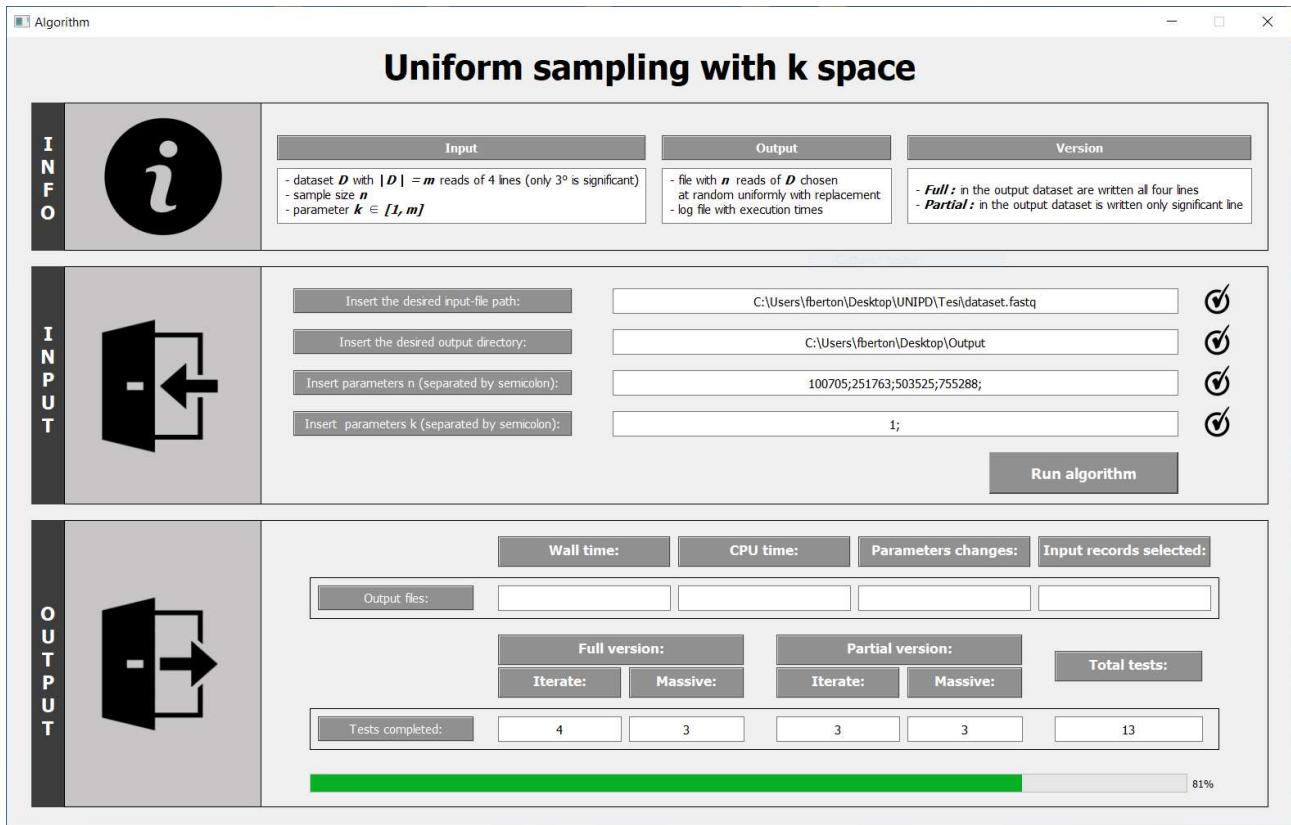


Figura 2.13 - esempio di esecuzione dell'applicazione per test con GUI

La struttura di directory creata per contenere gli output dell'algoritmo sarà la seguente:

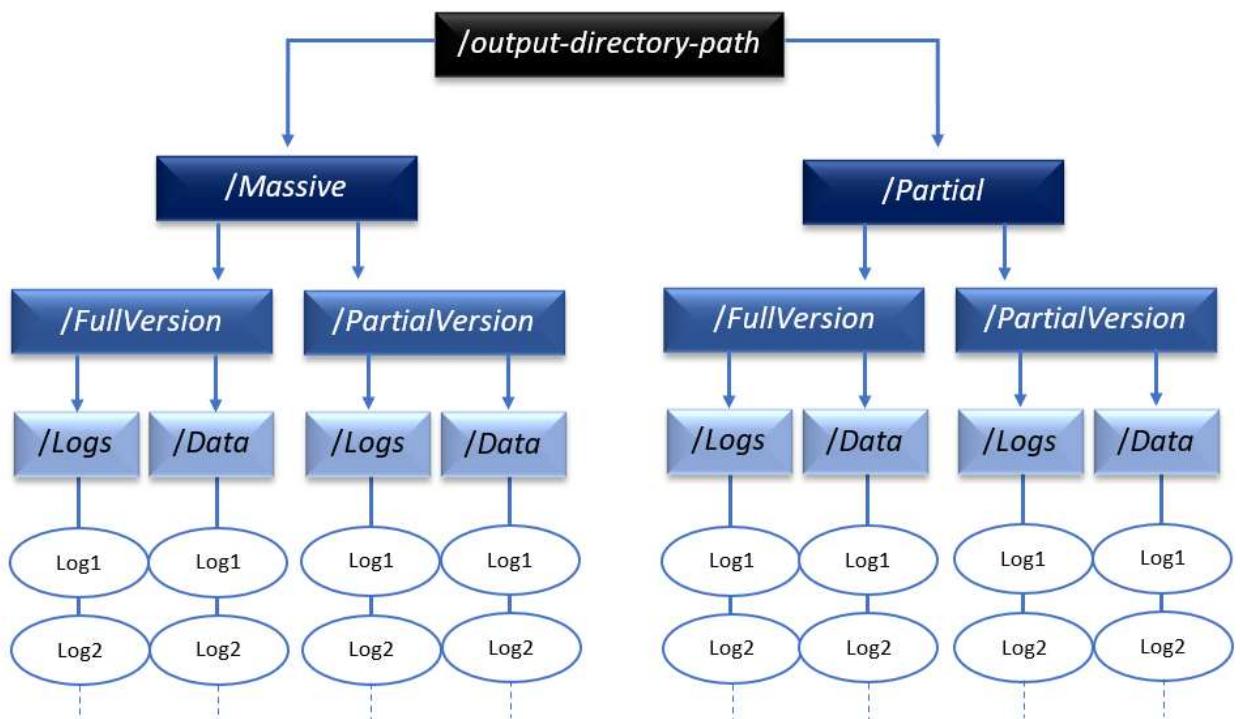


Figura 2.14 - struttura di directory in output

2.5 Analisi della complessità computazionale

Per analizzare l'efficienza dell'algoritmo in termini di complessità computazionale, utilizzeremo un'analisi asintotica in funzione della taglia e dei parametri d'ingresso; l'obiettivo, quindi, sarà quello di individuare una stima (limite superiore) del numero di *passi base* (ovvero operazioni con costo unitario, la cui esecuzione non dipende dai valori e dai tipi di variabili) svolti dall'algoritmo, facendo tendere la sua taglia a infinito.

Per quest'analisi ipotizzeremo inoltre che l'algoritmo esegua una sola sequenza di operazioni per ogni data istanza (algoritmo deterministico e non probabilistico), anche se ciò in realtà non accade in quanto nell'algoritmo vengono utilizzati valori aleatori creati tramite un generatore di numeri random e una variabile aleatoria binomiale (righe 3 e 4 dello pseudocodice mostrato in Figura 2.4).

Lo scopo di questa sezione sarà quindi quello di individuare, a partire dalla taglia m del dataset in input e dagli altri parametri in ingresso (m, n, k), una funzione $g(m, n, k)$ che sia limite superiore della complessità dell'algoritmo in funzione degli stessi parametri, che d'ora in poi indicheremo con $f(m, n, k)$. In termini matematici si tratta di individuare una funzione $g(m, n, k)$ tale che

$$f(m, n, k) \in O(g(m, n, k))$$

e dimostrare quindi che

$$\exists q > 0 \text{ e } \exists m_0 \geq 1 \text{ tali che } f(m, n, k) \leq q * g(m, n, k), \forall m \geq m_0$$

In termini più pratici, dovremo quindi verificare che esiste un m_0 abbastanza grande tale per cui per ciascuna istanza di taglia m_0 l'algoritmo esegue un numero minore o uguale di $q * g(m_0, n, k)$ operazioni.

D'ora in poi, per semplicità di scrittura, l'insieme dei tre parametri di ingresso m, n, k lo indicheremo col vettore \vec{t} .

2.5.1 Calcolo della complessità computazionale asintotica

Facendo riferimento allo pseudocodice in Figura 2.4, è possibile fare le seguenti osservazioni:

- Il numero di iterazioni del ciclo *while* può assumere al massimo il valore

$$i_{\text{while}}^{\max} = \left\lfloor \frac{m}{k} \right\rfloor + (1 - u(p)) (m \bmod k) \in O\left(\frac{m}{k}\right) \quad \text{Equazione 2.1}$$

Il ciclo *while*, infatti, si interrompe all'azzerarsi del parametro n ; quest'ultimo, se non si azzerà nelle iterazioni precedenti, diventa pari a zero quando m diventa minore di k , portando il secondo parametro della binomiale ad 1 e quindi assegnando il valore n a c .

Tutto ciò accade all'iterazione $\frac{m}{k}$ -esima se k è divisore di m , a quella successiva in caso contrario;

2. Tenendo in considerazione quanto appena detto, il tempo di esecuzione del ciclo *while* (e quindi dell'algoritmo) di conseguenza si può esprimere come:

$$f(\vec{t}) = T_{\text{while}}(\vec{t}) = \sum_{i=0}^{i_{\text{while}}^{\max}+1} T_{\text{condizione}_i}(\vec{t}) + \sum_{i=0}^{i_{\text{while}}^{\max}} T_{\text{while}_i}(\vec{t}) \\ \in O\left(\frac{m}{k} * \left(\max_i T_{\text{condizione}_i}(\vec{t}) + \max_i T_{\text{while}_i}(\vec{t})\right)\right)$$
Equazione 2.2

3. Poiché la valutazione della condizione del ciclo può essere considerata di costo unitario (verifica solamente la positività di n), l'espressione finale del tempo dell'algoritmo si può scrivere come

$$f(\vec{t}) \in O\left(\frac{m}{k} * \max_i T_{\text{while}_i}(\vec{t})\right)$$
Equazione 2.3

4. La complessità della singola iterazione del ciclo *while* può essere espressa come:

$$T_{\text{while}_i}(\vec{t}) = T_{r_2}(\vec{t}) + T_{r_3}(\vec{t}) + T_{\text{for}}(\vec{t}) + T_{r_{11}}(\vec{t}) + T_{r_{12}}(\vec{t})$$
Equazione 2.4

Ogni iterazione del ciclo *while* svolge:

- a. Alle righe 2, 11 e 12 delle operazioni di costo unitario $O(1)$, in quanto si tratta di semplici operazioni aritmetiche (11 e 12) o di assegnazioni condizionali (2).
- b. La riga 3, seppur non svolgendo operazioni elementari, effettua un numero di passi base indipendente dalla taglia dell'algoritmo o di altri parametri in ingresso; si può dire che anche questa riga di codice venga eseguita in un tempo che è $O(1)$.
- c. La riga 4 ha una complessità che dipende dalla grandezza dell'array da costruire; infatti, potrebbe essere espansa nel ciclo *for* seguente:

4 $\left\{ \begin{array}{l} \textbf{for } i \in [0, c] \textbf{ do} \\ \quad S_i \leftarrow \text{numero casuale nell'intervallo } [1, k]; \\ \textbf{end} \end{array} \right.$

Figura 2.15 - espansione riga 4 dello pseudocodice in figura 3.4

Ipotizzando che ottenere un numero random in un dato intervallo abbia complessità unitaria, la complessità di questa riga di codice sarà quindi

$$T_{r_4} = \sum_{i=0}^{c-1} T_{for_i}(\vec{t}) \in O\left(c * \max_i T_{for_i}(\vec{t})\right) \in O(c) \quad \text{Equazione 2.5}$$

La complessità $T_{while_i}(m)$ quindi è esprimibile come:

$$T_{while_i}(\vec{t}) \in O(1) + O(c) + O\left(T_{for}(\vec{t})\right) + O(1) + O(1) \quad \text{Equazione 2.6}$$

Escludendo le operazioni di costo unitario, si ottiene quindi

$$T_{while_i}(m) \in O(c) + O\left(T_{for}(\vec{t})\right) \quad \text{Equazione 2.7}$$

5. Il ciclo *for* più interno esegue esattamente k iterazioni, e la sua complessità è esprimibile come

$$T_{for}(\vec{t}) = \sum_{i=0}^{k-1} T_{for_i}(\vec{t}) \in O\left(k * \max_i T_{for_i}(\vec{t})\right) \quad \text{Equazione 2.8}$$

Per valutare $T_{for_i}(\vec{t})$ si può osservare che le operazioni svolte nelle righe 7 e 9 dello pseudocodice richiedono un numero di passi base indipendente dalla taglia dell'istanza in input, quindi esprimibili come $O(1)$, e di conseguenza

$$T_{for_i}(m) = O\left(T_{riga_8}(m)\right) \quad \text{Equazione 2.9}$$

6. La riga 8 è espandibile in un ciclo *for* che itera su tutti gli elementi dell'array S :

8 $\left\{ \begin{array}{l} r_c \leftarrow 0; \\ \textbf{for } i \in [0, c] \textbf{ do} \\ \quad \textbf{if } (S[i] = j) \textbf{ do} \\ \quad \quad r_c \leftarrow r_c + 1; \\ \quad \textbf{end} \\ \textbf{end} \end{array} \right.$

Figura 2.16 - espansione riga 8 dello pseudocodice in figura 3.4

Essendo che le istruzioni eseguite all'interno di questo ciclo *for* consistono nella valutazione di una condizione di uguaglianza o nell'assegnazione di un valore ad una variabile, si può scrivere

$$T_{riga_8}(\vec{t}) = \sum_{i=0}^{c-1} T_{iter_i}(\vec{t}) \in O\left(c * \max_i T_{iter_i}(\vec{t})\right) \in O(c) \quad \text{Equazione 2.10}$$

7. Unendo i risultati delle equazioni [Equazione 2.8](#), [Equazione 2.9](#) e [Equazione 2.10](#) si ottiene la formulazione finale della complessità del ciclo *for* interno, ovvero

$$T_{for}(\vec{t}) = O(c * k) \quad \text{Equazione 2.11}$$

8. Unendo le equazioni [Equazione 2.7](#) e [Equazione 2.11](#) si ottiene quindi

$$T_{while_i}(\vec{t}) \in O(c) + O(c * k) = O(c * k) \quad \text{Equazione 2.12}$$

e di conseguenza, ricordando l'[Equazione 2.3](#), il risultato finale sarà

$$f(\vec{t}) \in O\left(\frac{m}{k} * \max_i T_{while_i}(\vec{t})\right) = O\left(\frac{m}{k} * c * k\right) \quad \text{Equazione 2.13}$$

9. I parametri m e k sono parametri di ingresso, quindi è accettabile che compaiano nell'espressione finale della complessità. Il parametro c , al contrario, non rientra tra i parametri di input, e quindi va stimato secondo i parametri appartenenti a \vec{t} . Essendo c un valore random preso dalla distribuzione $Binomial(n, k/m)$, sotto l'ipotesi che il valore atteso rimanga all'incirca costante ad ogni iterazione (questa ipotesi verrà meglio analizzata in seguito), si avrà:

$$f(m, n, k) \in O(n * k) \quad \text{Equazione 2.14}$$

Prendendo in considerazione l'[Equazione 2.14](#), ci si aspetterà quindi che, fissando due dei tre parametri di ingresso, la complessità sia di tipo lineare rispetto al terzo parametro.

3 Risultati sperimentali

Questa sezione si occuperà di illustrare i risultati sperimentali, in termini di correttezza e tempi di esecuzione, delle prove effettuate con i vari tipi di algoritmo e le diverse versioni dell'applicazione di supporto.

Per ogni tipo di applicazione (da riga di comando e con GUI), verranno quindi riportati i tempi di esecuzione (sia *CPU time* che *Wall time*) e tutti gli altri dati di output combinando le due modalità di lettura-scrittura (*Massive R-W* e *Iterate R-W*) e le due tipologie di selezione dei dati (*Full version* e *Partial version*).

La scelta dei parametri per i test è la seguente:

- Il dataset di ingresso utilizzato ha cardinalità $|D| = 1007050$;
- Per il parametro n i dati di ingresso utilizzati sono, per $p = 1, 2, 3, 4$
 $n_p = g_p * m$ con $g_1 = 0.1, g_2 = 0.25, g_3 = 0.5, g_4 = 0.75$
- Il parametro k assumerà i valori

$$\begin{aligned} k_1 &= 1, & k_2 &= 10, & k_3 &= 100 \\ k_4 &= 1000, & k_5 &= 10000, & k_6 &= 1000000 \end{aligned}$$

I risultati sono stati ottenuti da diverse prove consecutive, conseguendo più valori per ogni coppia di parametri k e n in input; infine, di questi valori ne è stata fatta una media, ottenendo le qualità che sono riportate nelle seguenti tabelle.

3.1 Raccolta dei dati

3.1.1 Tempi di esecuzione

3.1.1.1 Applicazione con GUI

Iterate Full Version - wall time				
Parameter k	Parameter n			
	100705	251763	503525	755288
	1	6,233	12,299	22,230
	10	6,878	12,534	22,209
	100	6,727	12,443	21,844
	1000	6,971	12,789	22,859
	10000	7,580	17,517	26,156
	50000	9,287	21,389	39,397
	100000	13,469	37,364	66,487
	500000	43,308	119,957	233,828
	1000000	113,517	249,643	785,428
				1201,180

Iterate Full Version - CPU time				
Parameter k	Parameter n			
	100705	251763	503525	755288
	1	6,141	12,297	22,219
	10	6,828	12,516	22,203
	100	6,719	12,406	21,828
	1000	6,969	12,719	22,844
	10000	7,563	17,469	26,109
	50000	9,266	21,375	39,375
	100000	13,469	37,313	66,484
	500000	43,281	119,906	233,656
	1000000	113,406	249,438	785,063
				1200,453

Tabella 3.1 – tempi di esecuzione per GUI Full Version + Iterate R-W

Iterate Partial Version - wall time				
Parameter k	Parameter n			
	100705	251763	503525	755288
	1	3,320	5,235	7,796
	10	3,649	5,225	9,087
	100	3,652	5,206	7,674
	1000	3,675	5,548	8,214
	10000	4,419	6,956	11,488
	50000	6,774	13,771	25,189
	100000	10,380	29,866	60,748
	500000	41,301	110,296	221,674
	1000000	106,002	241,138	762,198
				1164,652

Iterate Partial Version - CPU time				
Parameter k	Parameter n			
	100705	251763	503525	755288
	1	3,313	5,234	7,781
	10	3,641	5,219	9,078
	100	3,641	5,203	7,656
	1000	3,672	5,531	8,203
	10000	4,391	6,953	11,438
	50000	6,766	13,766	25,172
	100000	10,328	29,844	60,703
	500000	40,000	110,234	221,500
	1000000	105,953	239,984	761,859
				1163,938

Tabella 3.2 - tempi di esecuzione per GUI Partial Version + Iterate R-W

Massive Full Version - wall time				
Parameter k	Parameter n			
	100705	251763	503525	755288
	1	6,284	12,617	22,271
	10	6,890	12,699	23,996
	100	6,997	12,617	22,185
	1000	7,144	13,480	24,191
	10000	7,727	15,639	25,933
	50000	9,701	20,595	39,082
	100000	15,161	44,878	67,110
	500000	41,311	115,803	231,831
	1000000	101,551	246,677	778,137
				1182,323

Massive Full Version - CPU time				
Parameter k	Parameter n			
	100705	251763	503525	755288
	1	6,266	12,609	22,219
	10	6,875	12,656	23,969
	100	6,969	12,594	22,141
	1000	7,047	13,469	24,141
	10000	7,719	15,578	25,891
	50000	9,688	20,578	39,063
	100000	15,109	44,828	67,016
	500000	41,266	115,734	231,672
	1000000	101,438	246,377	777,828
				1181,703

Tabella 3.3 - tempi di esecuzione per GUI Full Version + Massive R-W

Massive Partial Version - wall time					
Parameter k	Parameter n				
	100705	251763	503525	755288	
	1	3,364	4,971	7,509	9,876
	10	3,555	4,988	7,986	9,862
	100	3,580	5,128	7,397	11,380
	1000	3,987	5,506	7,893	10,524
	10000	4,521	7,362	11,163	15,174
	50000	6,759	12,951	24,486	35,340
	100000	9,879	35,246	51,427	85,767
	500000	37,801	107,760	216,307	559,127
	1000000	106,898	239,138	768,203	1169,904

Massive Partial Version - CPU time					
Parameter k	Parameter n				
	100705	251763	503525	755288	
	1	3,359	4,969	7,500	9,875
	10	3,516	4,953	7,969	9,859
	100	3,578	5,031	7,391	11,328
	1000	3,984	5,469	7,875	10,500
	10000	4,469	7,344	11,156	15,141
	50000	6,750	12,938	24,438	35,313
	100000	9,875	35,219	51,344	85,641
	500000	37,797	107,719	216,266	558,656
	1000000	106,859	238,984	767,656	1169,156

Tabella 3.4 - tempi di esecuzione per GUI Partial Version + Massive R-W

3.1.1.2 Applicazione con CLI

Iterate Full Version - wall time					
Parameter k	Parameter n				
	100705	251763	503525	755288	
	1	6,649	12,869	22,864	32,870
	10	6,862	13,220	22,894	32,439
	100	6,902	12,840	22,673	33,137
	1000	6,934	13,584	23,213	34,273
	10000	7,651	14,773	26,635	38,398
	50000	9,439	21,507	40,420	60,196
	100000	13,598	43,613	74,293	118,613
	500000	42,244	120,292	241,909	585,700
	1000000	112,630	256,924	793,431	1198,650

Iterate Full Version - CPU time					
Parameter k	Parameter n				
	100705	251763	503525	755288	
	1	6,531	12,828	22,844	32,813
	10	6,844	13,156	22,875	32,422
	100	6,859	12,828	22,625	33,078
	1000	6,934	13,564	23,203	34,203
	10000	7,625	14,750	26,578	38,344
	50000	9,375	21,469	40,391	60,141
	100000	13,578	43,547	74,281	118,547
	500000	42,188	120,141	241,750	585,203
	1000000	112,563	256,703	792,797	1197,766

Tabella 3.5 - tempi di esecuzione per CLI Full Version + Iterate R-W

Iterate Partial Version - wall time					
Parameter k	Parameter n				
	100705	251763	503525	755288	
	1	3,348	5,414	7,990	10,874
	10	3,738	5,331	7,895	10,479
	100	3,779	5,321	7,942	10,487
	1000	3,787	5,440	8,278	11,183
	10000	4,391	6,988	11,738	15,982
	50000	6,709	13,896	25,475	39,375
	100000	10,464	37,501	51,177	91,460
	500000	38,979	110,765	218,536	553,982
	1000000	105,599	243,152	770,691	1165,029

Iterate Partial Version - CPU time					
Parameter k	Parameter n				
	100705	251763	503525	755288	
	1	3,344	5,391	7,953	10,844
	10	3,719	5,328	7,891	10,453
	100	3,750	5,297	7,922	10,469
	1000	3,766	5,438	8,219	11,172
	10000	4,359	6,984	11,700	15,969
	50000	6,703	13,706	25,469	39,250
	100000	10,453	37,484	51,078	91,453
	500000	38,969	110,719	218,281	553,500
	1000000	105,531	242,906	770,094	1164,203

Tabella 3.6 - tempi di esecuzione per CLI Partial Version + Iterate R-W

Massive Full Version - wall time					Massive Full Version - CPU time					
Parameter k	Parameter n				Parameter n					
		100705	251763	503525	755288		100705	251763	503525	755288
	1	7,360	13,043	26,304	32,584		7,328	13,016	26,281	32,531
	10	7,406	14,112	23,153	32,758		7,375	14,078	23,109	32,719
	100	7,151	13,163	24,015	33,767		7,125	13,141	23,969	33,719
	1000	7,291	14,484	24,332	39,098		7,266	14,469	24,313	39,000
	10000	7,913	14,576	27,937	38,405		7,859	14,568	27,891	38,344
	50000	9,826	21,049	40,247	59,426		9,818	21,016	40,172	59,344
	100000	15,006	43,953	69,193	106,681		14,969	43,922	69,078	106,641
	500000	42,726	118,193	237,834	572,929		42,672	118,141	237,641	572,578
1000000	100,190	241,835	782,447	1186,366		100,047	241,547	781,859	1185,375	

Tabella 3.7 - tempi di esecuzione per CLI Full Version + Massive R-W

Massive Partial - wall time					Massive Partial - CPU time					
Parameter k	Parameter n				Parameter n					
		100705	251763	503525	755288		100705	251763	503525	755288
	1	3,741	5,154	7,785	10,130		3,730	5,056	7,750	10,125
	10	3,567	5,079	7,582	10,080		3,563	5,047	7,547	10,047
	100	3,702	5,134	7,606	10,192		3,688	5,094	7,578	10,188
	1000	3,682	5,297	8,071	11,538		3,672	5,266	8,063	11,484
	10000	4,195	6,797	10,878	15,264		4,188	6,781	10,875	15,172
	50000	6,710	13,960	25,048	36,052		6,688	12,969	25,016	36,031
	100000	9,882	32,046	52,011	84,437		9,844	32,016	51,953	84,359
	500000	38,079	107,170	219,143	552,550		38,047	107,109	219,078	552,266
1000000	96,740	232,316	775,735	1161,954		96,688	232,188	775,109	1161,203	

Tabella 3.8 - tempi di esecuzione per CLI Partial Version + Massive R-W

3.1.2 Record letti dal dataset di input

Essendo il numero di record selezionati un dato indipendente dalla modalità di lettura scrittura (*Massive* o *Iterate*), dal numero di righe del record che si vogliono scrivere in output (*Full* o *Partial*) e dal tipo di applicazione utilizzata (con GUI o CLI), questi dati sono riportati in una sola tabella globale che ha validità per tutte le versioni di algoritmo o applicazione; in verde sono stati evidenziate le combinazioni di parametri che permettono una lettura totale del dataset di ingresso, mentre in giallo quelli che escludono alcuni record finali di D dalla lettura.

Record di input selezionati					
Parameter k	Parameter n				
		100705	251763	503525	755288
	1	1007044	1007044	1007044	1007044
	10	1006990	1006990	1007050	1007050
	100	1007050	1007050	1007050	1007050
	1000	1007050	1007050	1007050	1007050
	10000	1007050	1007050	1007050	1007050
	50000	1007050	1007050	1007050	1007050
	100000	1007050	1007050	1007050	1007050
	500000	1007050	1007050	1007050	1007050
1000000	1007050	1007050	1007050	1007050	

Tabella 3.9 – numero di record di input letti per ogni combinazione di parametri n e k

3.2 Selezione dei dati utili all’analisi

Osservando i dati raccolti si possono fare le seguenti osservazioni preliminari che permettono di limitare l’attenzione su alcuni dati escludendo altri dati che risulterebbero ridondanti o con bassa capacità informativa. Tutto ciò viene fatto per potersi concentrare nella sezione fondamentale, ovvero quella in cui si studieranno le relazioni tra k, n e tempi di esecuzione, solo su dati utili e non ridondanti.

3.2.1 Tempo totale e tempo di CPU

Il tempo di CPU è sempre di poco inferiore (o uguale a causa dell’approssimazione a tre cifre decimali) del tempo totale di esecuzione; lo si può osservare dalle seguenti tabelle che riportano il valore

$$t_{diff} = t_{wall} - t_{CPU}$$

GUI Iterate Full Version				
Parameter k	Parameter n			
	100705	251763	503525	755288
1	0,092	0,002	0,011	0,074
10	0,050	0,018	0,006	0,044
100	0,009	0,037	0,016	0,032
1000	0,002	0,070	0,015	0,078
10000	0,017	0,048	0,047	0,058
50000	0,021	0,014	0,022	0,047
100000	0,000	0,051	0,003	0,062
500000	0,026	0,051	0,172	0,341
1000000	0,111	0,205	0,365	0,727

GUI Iterate Partial Version				
Parameter k	Parameter n			
	100705	251763	503525	755288
1	0,007	0,001	0,015	0,005
10	0,008	0,006	0,009	0,011
100	0,011	0,002	0,018	0,017
1000	0,003	0,017	0,011	0,015
10000	0,028	0,003	0,050	0,019
50000	0,008	0,006	0,017	0,072
100000	0,052	0,023	0,044	0,081
500000	1,301	0,062	0,174	0,375
1000000	0,049	1,153	0,338	0,714

GUI Massive Full Version				
Parameter k	Parameter n			
	100705	251763	503525	755288
1	0,018	0,008	0,052	0,057
10	0,015	0,043	0,028	0,085
100	0,029	0,023	0,044	0,063
1000	0,097	0,011	0,051	0,012
10000	0,008	0,061	0,042	0,113
50000	0,013	0,017	0,020	0,093
100000	0,052	0,050	0,094	0,140
500000	0,045	0,069	0,159	0,523
1000000	0,114	0,300	0,309	0,620

GUI Massive Partial Version				
Parameter k	Parameter n			
	100705	251763	503525	755288
1	0,004	0,002	0,009	0,001
10	0,040	0,035	0,017	0,002
100	0,002	0,096	0,006	0,051
1000	0,003	0,037	0,018	0,024
10000	0,052	0,018	0,007	0,034
50000	0,009	0,013	0,048	0,027
100000	0,004	0,027	0,083	0,126
500000	0,005	0,041	0,041	0,471
1000000	0,038	0,153	0,547	0,748

GUI Iterate Full Version					GUI Iterate Partial Version				
Parameter k	Parameter n				Parameter k	Parameter n			
		100705	251763	503525			100705	251763	503525
	1	0,118	0,041	0,020		0,058	0,004	0,023	0,037
	10	0,018	0,064	0,019		0,017	0,020	0,003	0,005
	100	0,043	0,012	0,048		0,059	0,029	0,024	0,020
	1000	0,000	0,020	0,010		0,070	0,022	0,003	0,059
	10000	0,026	0,023	0,057		0,055	0,031	0,004	0,038
	50000	0,064	0,038	0,029		0,055	0,005	0,190	0,007
	100000	0,020	0,066	0,011		0,066	0,011	0,017	0,099
	500000	0,056	0,151	0,159		0,497	0,011	0,047	0,255
1000000	0,068	0,221	0,634	0,884		0,068	0,246	0,597	0,826

GUI Massive Full Version					GUI Massive Partial Version				
Parameter k	Parameter n				Parameter k	Parameter n			
		100705	251763	503525			100705	251763	503525
	1	0,032	0,027	0,022		0,053	0,011	0,098	0,035
	10	0,031	0,034	0,044		0,040	0,005	0,032	0,035
	100	0,026	0,023	0,046		0,048	0,015	0,040	0,027
	1000	0,025	0,015	0,019		0,098	0,010	0,032	0,009
	10000	0,054	0,008	0,046		0,061	0,008	0,016	0,003
	50000	0,008	0,033	0,075		0,083	0,022	0,991	0,032
	100000	0,037	0,031	0,115		0,041	0,039	0,030	0,058
	500000	0,054	0,052	0,193		0,351	0,032	0,060	0,064
1000000	0,143	0,288	0,588	0,991		0,052	0,128	0,626	0,750

Tabella 3.10 - differenza tra tempi totali e tempi di CPU

La differenza $t_{diff} = t_{wall} - t_{CPU}$ non risulta sempre costante in quanto dipende dall'occupazione della CPU nell'intervallo di tempo di esecuzione dell'algoritmo da parte di altre attività non propriamente richieste dall'applicazione in uso, ma si può notare che assume sempre un valore non negativo.

Quanto appena osservato verifica un risultato che già poteva essere previsto; infatti, il tempo di CPU non include il tempo in cui il processore è impegnato in altre attività al di fuori dell'esecuzione della nostra applicazione, e quindi deve per forza risultare sempre inferiore (o uguale, nel caso limite in cui il processore sia occupato unicamente dall'applicazione in questione) rispetto al tempo totale.

È quindi ragionevole tenere in considerazione per le analisi future il solo tempo di CPU, più affidabile di quello totale. Nelle analisi che verranno fatte d'ora in poi, quindi, eviteremo lo studio ridondante sia di t_{wall} che di t_{CPU} , e ci limiteremo a quello di t_{CPU} .

3.2.2 Versioni GUI e versioni CLI

Per quanto riguarda la distinzione tra applicazioni da riga di comando e con GUI, si può dimostrare che le differenze minime in termini di tempi di esecuzione (del solo algoritmo, ovvero t_{CPU}) non sono riconducibili ad una regola sistematica secondo cui una versione impiega più tempo dell'altra; infatti, dai grafici mostrati di seguito che mostrano il valore

$$t_{diff} = t_{CLI} - t_{GUI}$$

si può osservare come questa differenza sia casuale e assuma valori positivi o negativi senza alcuna ricorrenza e secondo nessuna regola precisa; sono stati evidenziati in verde i valori positivi e in giallo quelli negativi, per dare risalto alla casualità della distribuzione di valori maggiori o minori di zero:

Iterate Full Version					Iterate Partial Version					
Parameter k	Parameter n				Parameter k	Parameter n				
	100705	251763	503525	755288		100705	251763	503525	755288	
	1	-0,391	-0,531	-0,625	-0,984	1	-0,031	-0,156	-0,172	-0,547
	10	-0,016	-0,641	-0,672	-0,141	10	-0,078	-0,109	1,188	-0,297
	100	-0,141	-0,422	-0,797	-1,063	100	-0,109	-0,094	-0,266	-0,359
	1000	0,035	-0,845	-0,359	-1,453	1000	-0,094	0,094	-0,016	-0,531
	10000	-0,063	2,719	-0,469	-0,594	10000	0,031	-0,031	-0,262	-0,063
	50000	-0,109	-0,094	-1,016	2,328	50000	0,063	0,059	-0,297	-1,453
	100000	-0,109	-6,234	-7,797	1,281	100000	-0,125	-7,641	9,625	1,641
	500000	1,094	-0,234	-8,094	2,703	500000	1,031	-0,484	3,219	8,875
1000000	0,844	-7,266	-7,734	2,688	1000000	0,422	-2,922	-8,234	-0,266	

Massive Full Version					Massive Partial Version					
Parameter k	Parameter n				Parameter k	Parameter n				
	100705	251763	503525	755288		100705	251763	503525	755288	
	1	-1,063	-0,406	-4,063	-0,828	1	-0,371	-0,088	-0,250	-0,250
	10	-0,500	-1,422	0,859	-1,094	10	-0,047	-0,094	0,422	-0,188
	100	-0,156	-0,547	-1,828	-1,750	100	-0,109	-0,063	-0,188	1,141
	1000	-0,219	-1,000	-0,172	-5,516	1000	0,313	0,203	-0,188	-0,984
	10000	-0,141	1,010	-2,000	-0,969	10000	0,281	0,563	0,281	-0,031
	50000	-0,131	-0,438	-1,109	-2,203	50000	0,063	-0,031	-0,578	-0,719
	100000	0,141	0,906	-2,063	1,422	100000	0,031	3,203	-0,609	1,281
	500000	-1,406	-2,406	-5,969	7,797	500000	-0,250	0,609	-2,813	6,391
1000000	1,391	4,830	-4,031	-3,672	1000000	10,172	6,797	-7,453	7,953	

Tabella 3.11 - differenza tra tempi di esecuzione dell'applicazione con GUI e CLI

Essendo queste differenze molto piccole e quindi trascurabili, d'ora in poi verranno considerati per le analisi future i tempi raccolti dalla sola applicazione con GUI, escludendo quelli dell'applicazione di test da riga di comando.

3.2.3 Massive R-W e Iterate R-W version

In questa sezione verranno analizzate e comprese le differenze tra le due versioni dell'algoritmo che si differenziano in base alla lettura e alla scrittura dei record del dataset. In particolare, le differenze che presentano le due versioni sono:

- Il numero di record di input letti: mentre la versione *Massive R-W* legge tutti i record indistintamente, senza sapere quanti effettivamente verranno utilizzati dall'algoritmo, la versione *Iterate R-W* invece legge solo quelli strettamente necessari;
- La modalità di lettura e scrittura dei record: se per la versione Massive i record di input vengono letti e salvati in una lista, e quelli di output salvati in una lista e poi scritti, per la versione Iterate R-W invece tutti i record sono letti di volta in volta quando necessario, e se l'algoritmo decreta che debbano essere scritti in output, questi verranno scritti immediatamente senza la necessità di liste ausiliarie.

La versione *Iterate R-W* ha quindi il vantaggio di evitare di leggere i record non necessari all'esecuzione dell'algoritmo, ma allo stesso tempo presenta lo svantaggio di avere un'efficienza in termini di tempo di poco inferiore: scrivere tutti le linee di un file di testo in una volta, infatti, risulta leggermente più veloce di scrivere una linea alla volta, intermezzando la scrittura con altre istruzioni che non interessano la stampa del file di output.

Osserviamo ora le seguenti tabelle che indicano il valore

$$t_{diff} = t_{Massive} - t_{Iterate}$$

evidenziato in verde in caso sia positivo, arancione se negativo.

Full Version					Partial Version					
Parameter k	Parameter n				Parameter k	Parameter n				
	100705	251763	503525	755288		100705	251763	503525	755288	
	1	0,125	0,313	0,000	-0,125	1	0,047	-0,266	-0,281	-0,422
	10	0,047	0,141	1,766	-0,656	10	-0,125	-0,266	-1,109	-0,297
	100	0,250	0,188	0,313	-0,047	100	-0,063	-0,172	-0,266	1,219
	1000	0,078	0,750	1,297	0,734	1000	0,313	-0,063	-0,328	-0,141
	10000	0,156	-1,891	-0,219	-0,375	10000	0,078	0,391	-0,281	-0,766
	50000	0,422	-0,797	-0,313	-5,328	50000	-0,016	-0,828	-0,734	-2,484
	100000	1,641	7,516	0,531	-11,766	100000	-0,453	5,375	-9,359	-7,453
	500000	-2,016	-4,172	-1,984	-7,531	500000	-2,203	-2,516	-5,234	-3,719
1000000	-11,969	-3,061	-7,234	-18,750	1000000	0,906	-1,000	5,797	5,219	

Tabella 3.12 – differenza tra il tempo di esecuzione della versione Massive R-W e Iterate R-W

Nelle seguenti tabelle invece viene indicato di quanto la Iterate R-W supera l'antagonista in valore percentuale, ovvero il valore

$$r_{M-I} = \frac{t_{Massive} - t_{Iterate}}{t_{Iterate}} * 100$$

In particolare, sono stati evidenziati in arancione quei valori che superano il 5% in negativo, in verde quelli che lo superano in positivo.

<i>Full Version</i>					<i>Partial Version</i>					
Parameter <i>k</i>	Parameter <i>n</i>				Parameter <i>k</i>	Parameter <i>n</i>				
	100705	251763	503525	755288		100705	251763	503525	755288	
	1	2,036	2,541	0,000	-0,393	1	1,415	-5,075	-3,614	-4,097
	10	0,686	1,124	7,952	-2,033	10	-3,433	-5,090	-12,220	-2,923
	100	3,721	1,511	1,432	-0,146	100	-1,717	-3,303	-3,469	12,056
	1000	1,121	5,897	5,677	2,242	1000	8,511	-1,130	-4,000	-1,322
	10000	2,066	-10,823	-0,838	-0,993	10000	1,779	5,618	-2,459	-4,813
	50000	4,553	-3,728	-0,794	-8,529	50000	-0,231	-6,016	-2,917	-6,573
	100000	12,181	20,142	0,799	-9,819	100000	-4,387	18,010	-15,418	-8,006
	500000	-4,657	-3,479	-0,849	-1,281	500000	-5,508	-2,282	-2,363	-0,661
1000000	-10,554	-1,227	-0,922	-1,562	1000000	0,855	-0,417	0,761	0,448	

Tabella 3.13 - differenza tra il tempo di esecuzione della versione Massive R-W e Iterate R-W

Per quanto riguarda la *full version*, possiamo osservare che:

- La versione *Iterate R-W* risulta meno efficiente per valori di *n* grandi, essendo le prime due colonne prevalentemente verdi e le ultime arancioni: questo è giustificato dal fatto che all'aumentare di *n* aumenta il numero di record da scrivere in output, e quindi la maggior efficienza della scrittura massiva già precedentemente citata fa sì che la versione *Massive R-W* sia leggermente più veloce;
- La versione *Iterate R-W* risulta più efficiente per *k* bassi, essendo le prime righe in maggioranza verdi e le ultime arancioni: questo si può spiegare ricordando i risultati del paragrafo 3.1.2, che mostravano come, per *k* piccoli, non venisse letta la totalità dei record di input, ma solo una sua parte; la versione *Iterate R-W* in questo caso vince in quanto evita la lettura inutile di qualche record non utilizzato.

Riguardo la versione *partial*, invece, notiamo che quasi la totalità dei valori sono negativi, determinando una maggior efficienza della versione *Massive R-W* rispetto all'altra. Infatti, se per la versione *full Iterate* lo svantaggio di scrivere un record alla volta si ammortizzava sulle quattro righe di testo da scrivere, per la versione *partial Iterate* invece la scrittura diventa più inefficiente, dovendo scrivere una sola riga alla volta.

Per le considerazioni fatte riguardo alla lettura di record in eccesso per la *Massive R-W*, è possibile considerare per le analisi future soltanto i risultati ottenuti dalla *Iterate R-W*.

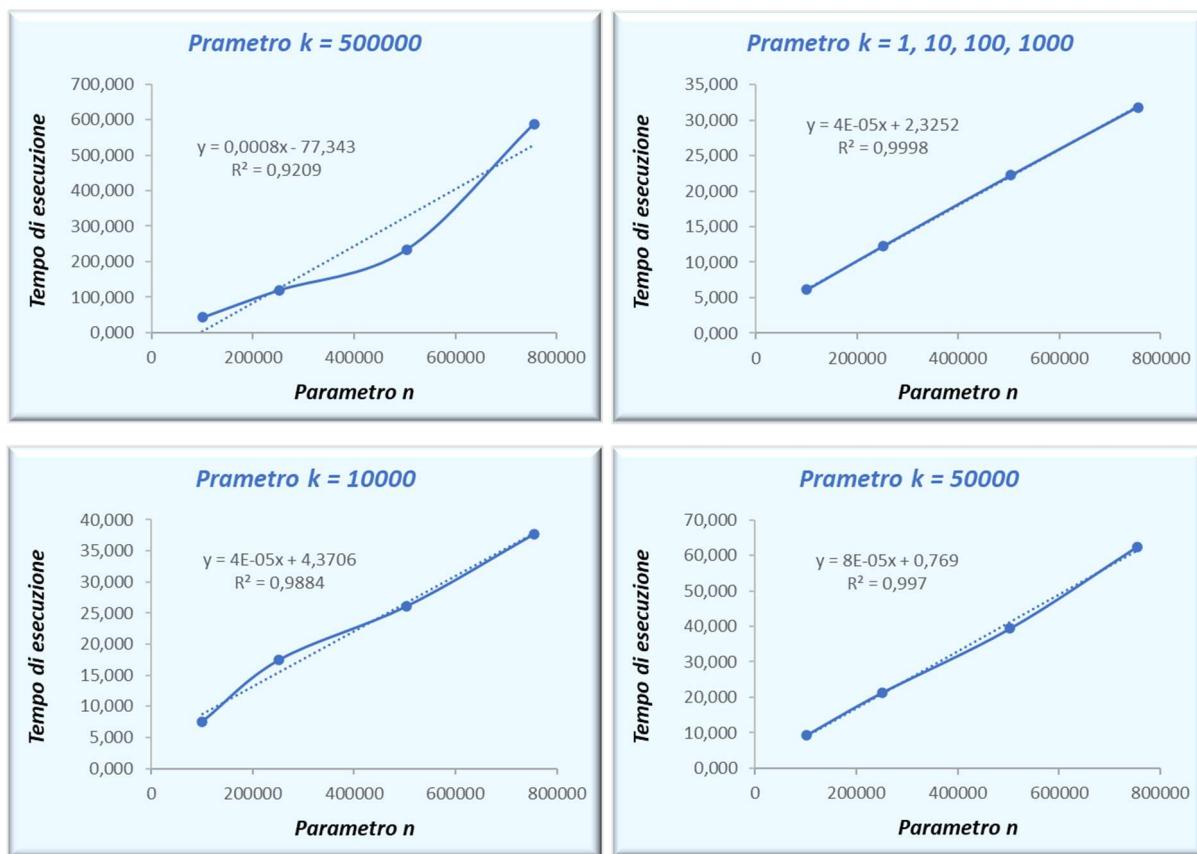
3.3 Analisi dei dati raccolti in termini di tempo di esecuzione

3.3.1 Dipendenza del tempo di esecuzione dal parametro n

Verrà analizzata ora la dipendenza del tempo di esecuzione dal parametro n , che secondo l'analisi computazionale sviluppata precedentemente dovrebbe essere di tipo lineare: infatti, ricordando l'[Equazione 2.14](#), abbiamo che $f(m, n, k) \in O(n * k)$.

Si osservino quindi i grafici seguenti, che mostrano, per ogni k , come varia il tempo di esecuzione in funzione del parametro n , e inoltre indicano per ogni curva la sua regressione lineare e la rispettiva espressione analitica (con coefficiente di determinazione).

Verranno mostrati prima i grafici relativi alla versione *full* dell'algoritmo, e successivamente quelli della versione *partial*. I grafici relativi ai parametri con $k \in \{1, 10, 100, 1000\}$ sono stati unificati nello stesso grafico in quanto presentano gli stessi tempi di esecuzione.



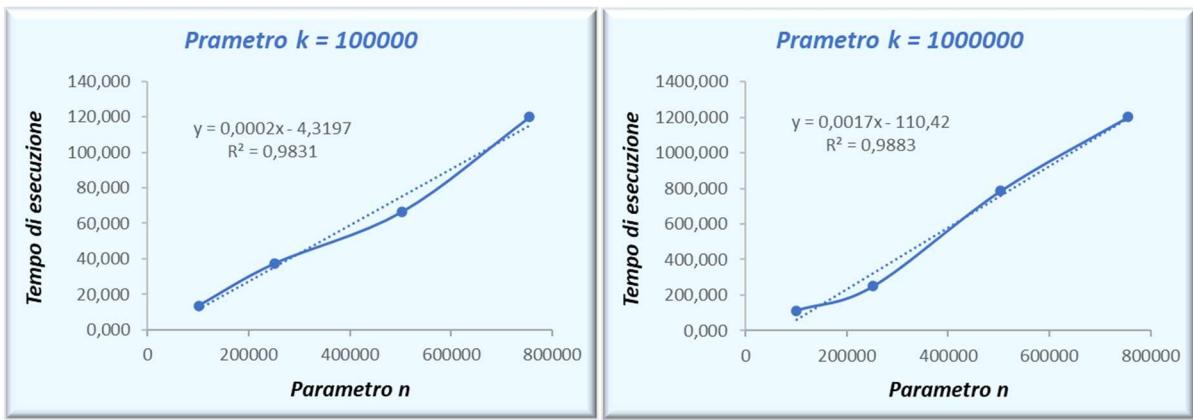


Grafico 3.1 - dipendenza del tempo di esecuzione (full version) dal parametro n e sua regressione lineare

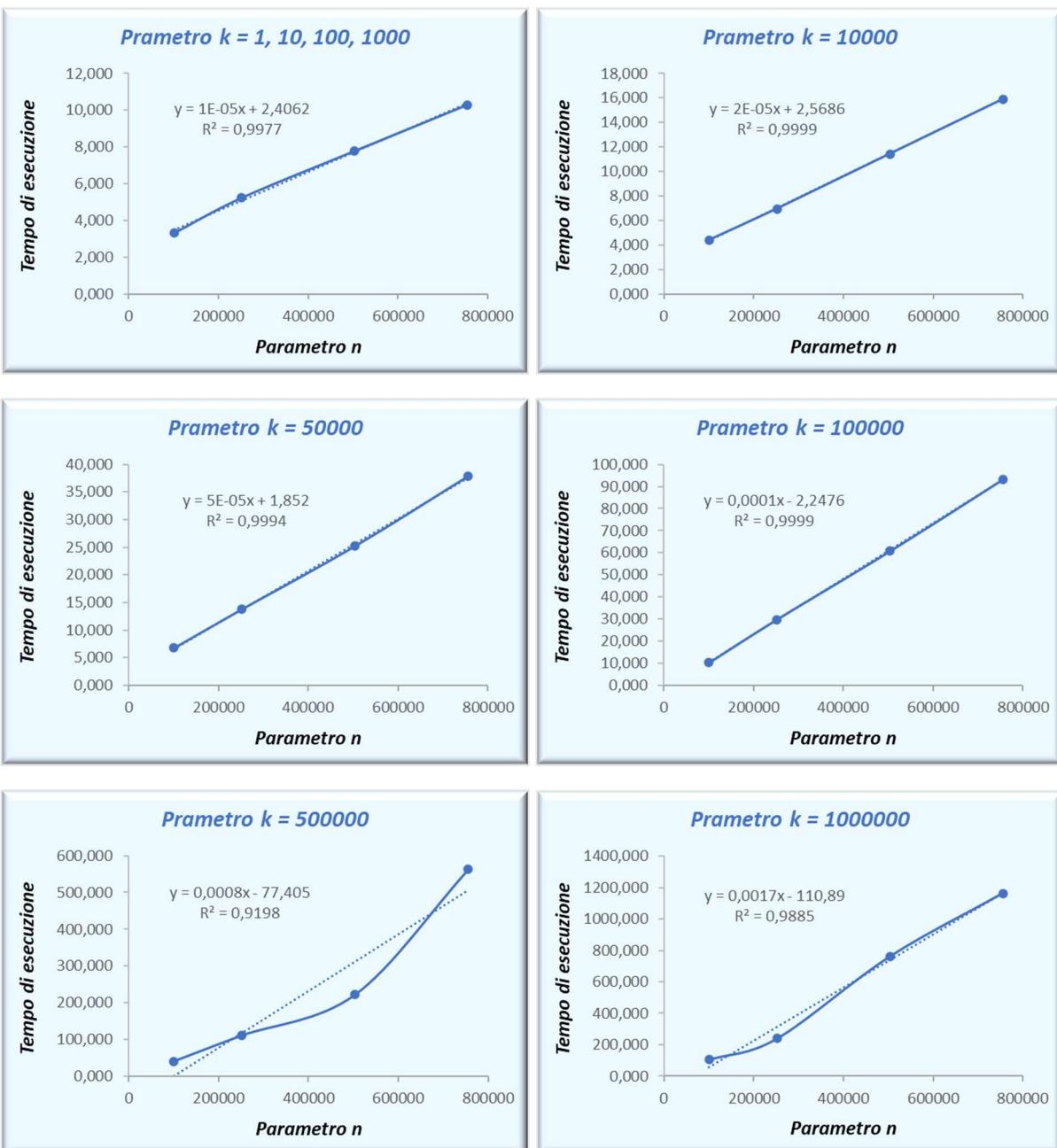


Grafico 3.2 - dipendenza del tempo di esecuzione (partial version) dal parametro n e sua regressione lineare

Come si può notare, sia per la *full version* che per la *partial version* le regressioni lineari per i valori $k \in \{1, 10, 100, 1000, 10000, 50000, 100000\}$ approssimano in maniera quasi perfetta la curva descritta dai valori raccolti; infatti, il coefficiente di determinazione è molto prossimo ad 1, il che significa che la regressione lineare predice in maniera corretta l'andamento dei valori al variare di n .

Anche per $k \in \{500000, 1000000\}$ il coefficiente di correlazione risulta vicino all'unità, anche se si può notare con facilità come in questi due casi la regressione lineare sia meno precisa nell'approssimare la curva rispetto ai valori di k inferiori; infatti, abbiamo già avuto modo di osservare come, per valori di k molto alti, l'algoritmo abbia un funzionamento anomalo, risolvendosi in pochissime iterazioni del ciclo *while* e trasferendo il maggior sovraccarico di occupazione della CPU nel ciclo *for*, precisamente nella ricerca delle occorrenze all'interno del vettore S (riga 8 della Figura 2.4). Possiamo notare come per questi due parametri i tempi della *full version* e della *partial version* risultino molto simili, tanto da avere regressione lineare coincidente; questo poiché il tempo impiegato per stampare in output i record in questo caso passa in secondo piano, essendo preponderante il tempo richiesto per scorrere tutto il vettore S (che avrà dimensione dell'ordine di grandezza del parametro n) per $n_{it} * k$ volte (indicando con n_{it} il numero di iterazioni del ciclo *while*); il metodo che è stato implementato per calcolare il numero di occorrenze nel vettore S , infatti, è un semplice metodo di ricerca lineare, che scorre fino alla fine il vettore ogni volta che viene chiamato.

Per attenuare questo comportamento, di poca rilevanza per parametri k bassi ma molto penalizzante per k più alti, si può ricorrere ad un'altra forma del metodo di ricerca delle occorrenze, che invece di contare solamente le ricorrenze dell'indice all'interno del vettore, rimuoverà ogni elemento trovato nel vettore con contenuto pari a quello dell'indice ricercato; in questo modo, ad ogni iterazione del ciclo *for* si avrà una diminuzione della cardinalità di S , e quindi della complessità, senza comunque intaccare il funzionamento dell'algoritmo. Con questo nuovo approccio, i tempi di esecuzione per k bassi rimangono pressoché invariati; tuttavia, quelli per k con valori superiori a mille risultano diminuiti, soprattutto per valori di n grandi. Di seguito sono riportati questi risultati in forma tabellare:

Full Version				
Parameter k	Parameter n			
	100705	251763	503525	755288
	1	6,047	12,859	23,094
	10	6,828	16,203	22,391
	100	6,813	12,766	23,203
	1000	6,766	12,781	23,672
	10000	8,266	13,313	23,859
	50000	7,547	14,609	26,422
	100000	7,469	14,813	27,141
	500000	6,766	17,531	38,266
	1000000	8,313	20,438	51,688

Partial Version				
Parameter k	Parameter n			
	100705	251763	503525	755288
	1	3,297	5,344	7,906
	10	3,563	6,406	7,766
	100	3,563	5,188	8,156
	1000	3,688	5,234	7,969
	10000	3,875	5,703	8,875
	50000	4,266	6,953	11,328
	100000	4,391	8,016	11,906
	500000	4,688	9,422	23,234
	1000000	4,828	13,422	36,719

Tabella 3.14 - tempi di esecuzione con nuova implementazione di ricerca delle occorrenze

Vengono riportati anche i risultati in forma grafica per il valore $k = 500000$, ovvero quello che precedentemente presentava un coefficiente di correlazione leggermente più basso rispetto agli altri:

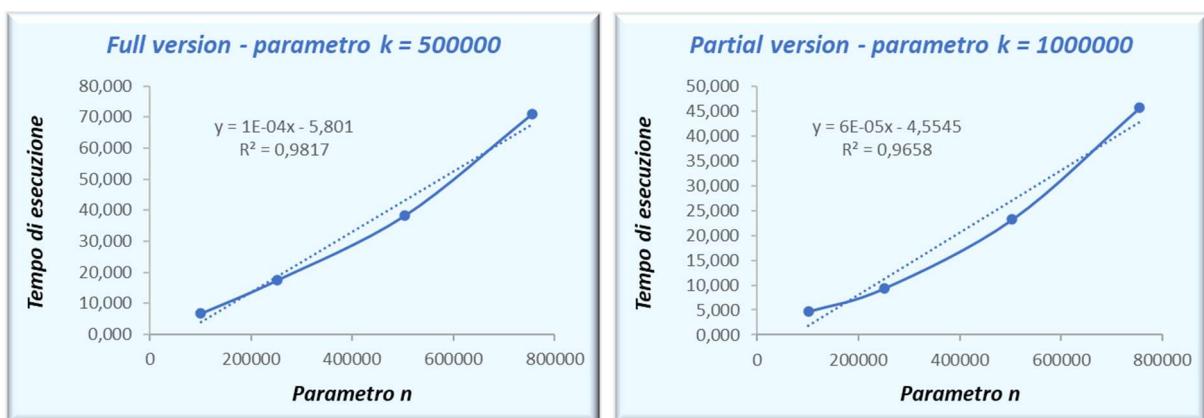
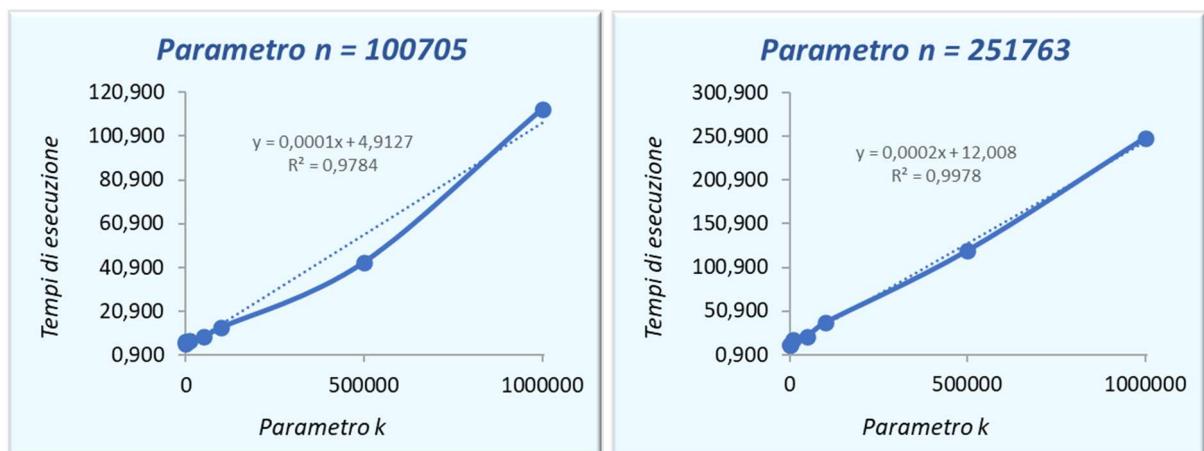


Grafico 3.3 - dipendenza del tempo di esecuzione dal parametro n con nuova implementazione di ricerca delle occorrenze

3.3.2 Dipendenza del tempo di esecuzione dal parametro k

Il parametro k , fissato n , fa variare sensibilmente i tempi di esecuzione; questo lo si poteva già apprezzare osservando i dati in forma tabellare al paragrafo 3.1.1, ma si osserva meglio dai grafici seguenti che mostrano la variazione di t_{CPU} al crescere di k :



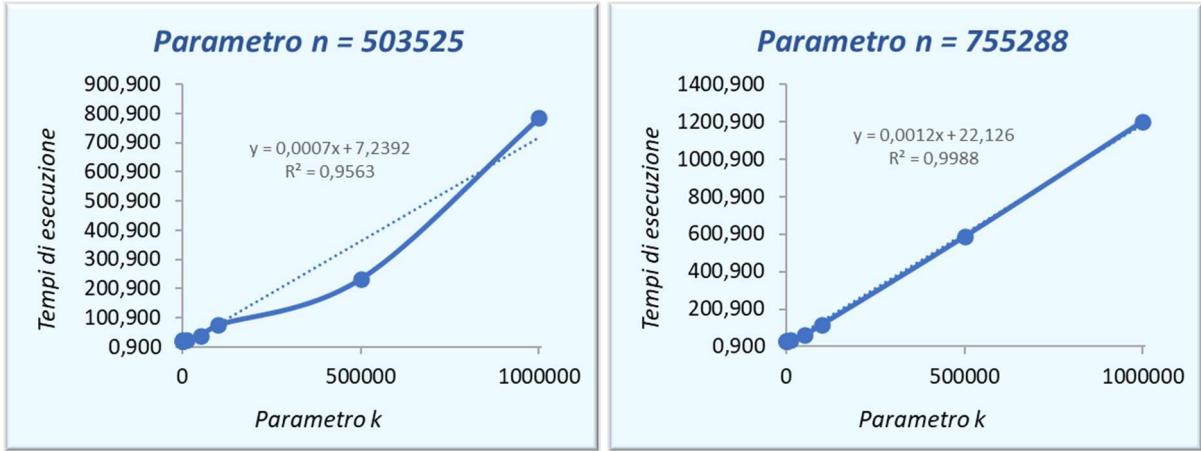


Grafico 3.4 – andamento del tempo di esecuzione al variare del parametro k per full version

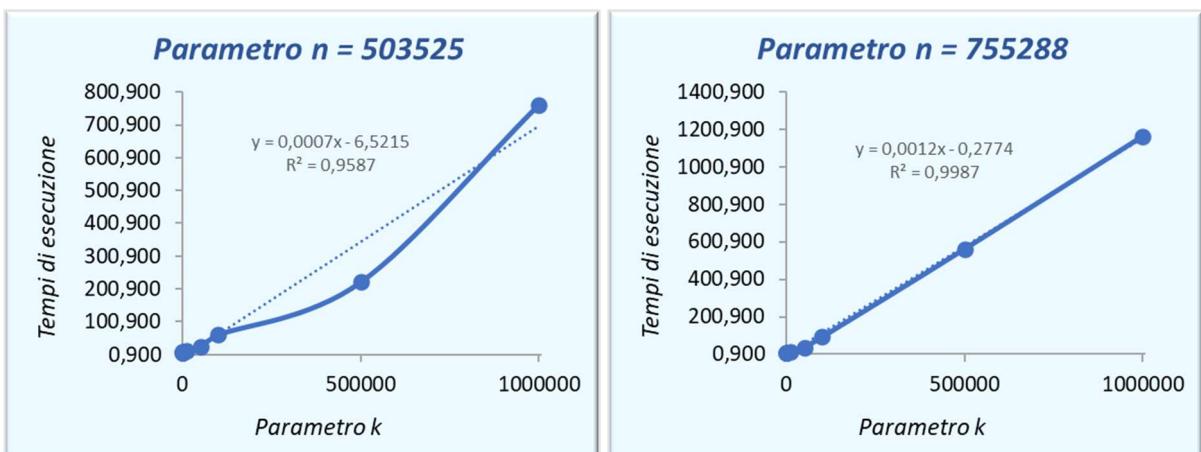
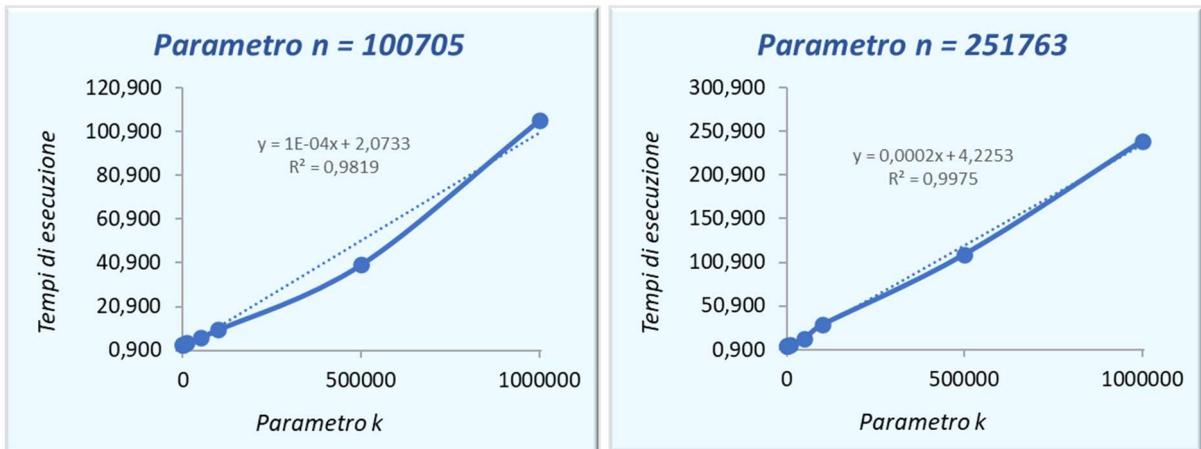


Grafico 3.5 – andamento del tempo di esecuzione al variare del parametro k per partial version

Osserviamo che tutte le regressioni lineari stimate per le varie curve hanno un coefficiente di correlazione prossimo all'unità, pur essendoci valori di k per cui la regressione lineare è meno precisa, e la curva si accosta meno ad una retta. Possiamo comunque affermare che la stima della complessità computazionale eseguita durante l'analisi dell'algoritmo viene confermata, in quanto si può verificare una correlazione pseudo-lineare tra t_{CPU} e k .

3.3.3 Dipendenza del tempo di esecuzione dal tipo di versione (full o partial)

L'esecuzione dell'algoritmo per *full version* e *partial version* è identica in tutto e per tutto nella fase di lettura, in quanto non è possibile, utilizzando il linguaggio C++, posizionarsi su una precisa riga partendo dal suo indice: è necessario iterare quindi tutto il file tramite il metodo *getline(...)* fino ad ottenere la riga voluta.

Per quanto riguarda invece la stampa dei record in output, la differenza tra i due algoritmi è significativa: nella prima versione per ogni record devono essere stampate quattro linee di testo, mentre per la seconda soltanto una; questa è l'unica dissomiglianza tra le due versioni, che tuttavia comporta risultati significativamente diversi in termini di tempi di esecuzione.

È scontato affermare, in base alle osservazioni fatte sulle differenze strutturali delle due versioni, che la versione *partial* dovrà per forza portare a tempi di esecuzione inferiori.

Analizziamo orai seguenti grafici, che riportano l'andamento del rapporto tra i tempi di esecuzione delle due versioni al variare del parametro k , per n fissato. Indicando con t_{full} e t_{part} i due tempi di esecuzione, verrà mostrato quindi il valore

$$r_t = \frac{t_{full}}{t_{part}}$$

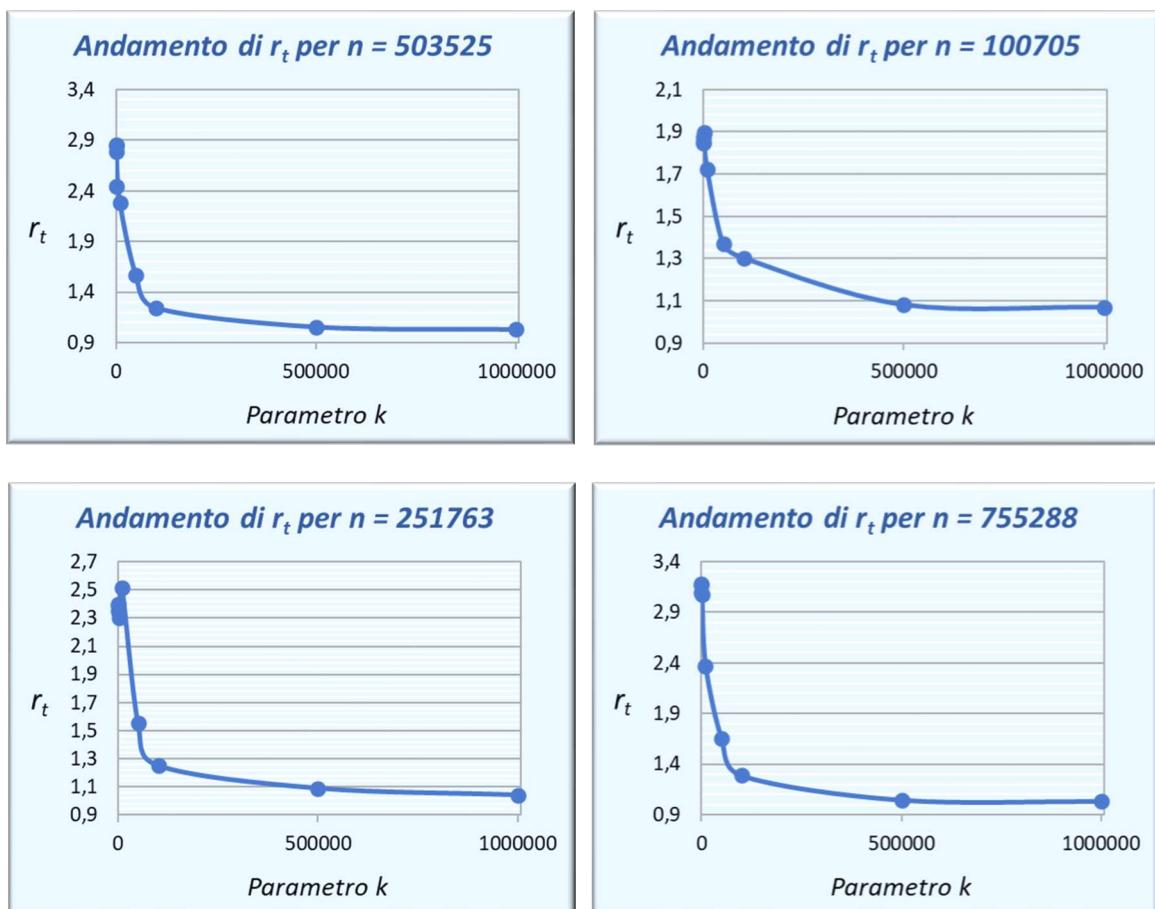


Grafico 3.6 – andamento del rapporto r_t in funzione del parametro k per versione con ricerca delle occorrenze standard

È facile osservare come i valori di r_t siano sempre superiori all'unità, confermando il fatto che la versione parziale è sempre più efficiente; notiamo però come, all'aumentare di k , il rapporto tra i tempi di esecuzione diminuisca: come già spiegato nella sezione precedente, infatti, con k alti ciò che influisce maggiormente sul tempo di esecuzione non è tanto la lettura/scrittura dei record (che è l'unico punto di distacco tra le due versioni), ma è invece molto più determinante il tempo impiegato all'interno del ciclo *for* dal metodo di ricerca delle occorrenze; ripetendo gli stessi grafici per i dati ottenuti dalla versione dell'algoritmo con ricerca delle occorrenze ottimizzata, si ottengono i seguenti risultati:

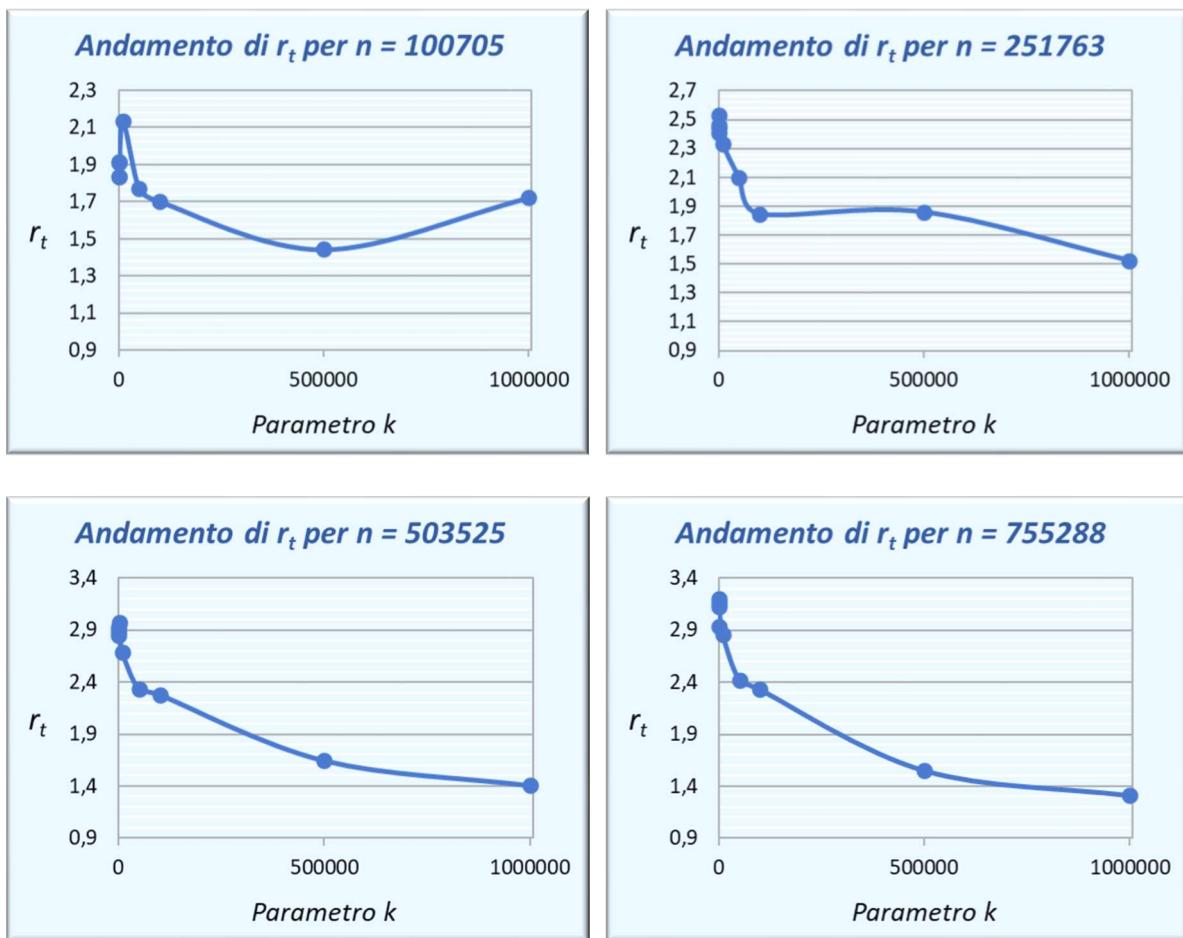


Grafico 3.7 - andamento del rapporto r_t in funzione del parametro k per versione con ricerca delle occorrenze ottimizzata

Notiamo, quindi, come l'effetto precedentemente descritto per k grandi sia di molto attenuato, ma comunque presente anche con l'utilizzo della versione ottimizzata.

Possiamo affermare, in conclusione, che la versione *partial*, che offre in output meno informazioni ma che risulta più efficiente rispetto alla sua antagonista, perde in efficienza all'aumentare di k , passando da una diminuzione del tempo di esecuzione di quasi tre volte, ad una diminuzione di all'incirca una volta e mezza.

Passiamo ora a valutare la dipendenza di questo rapporto r_t dal parametro n , tenendo questa volta fissato il parametro k . Per lo stesso motivo spiegato poco sopra, considereremo i tempi ottenuti tramite l'algoritmo con ricerca ottimizzata delle occorrenze in S .

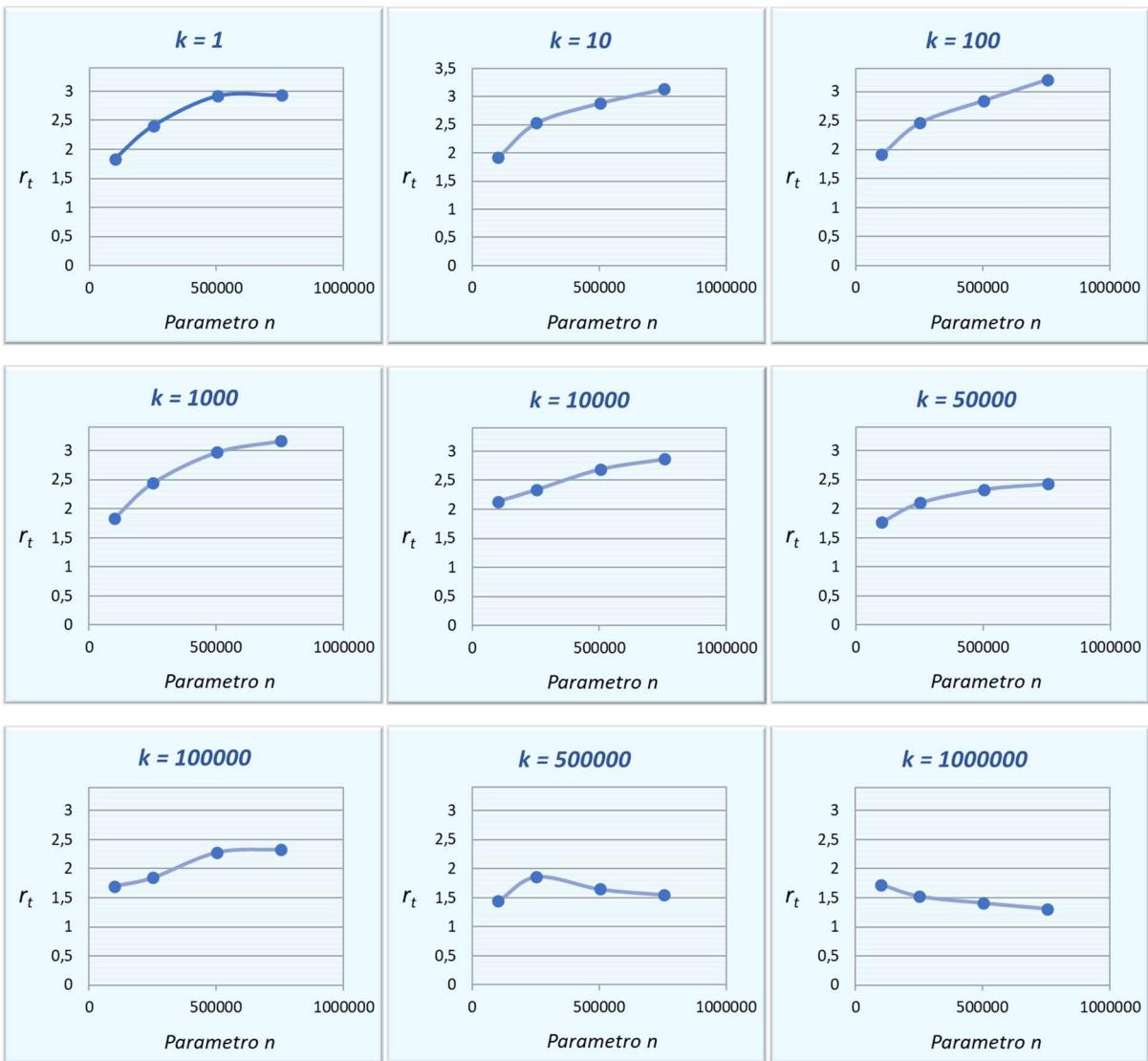


Grafico 3.8 - andamento del rapporto r_t in funzione del parametro n

Si nota ancora una volta come tendenzialmente il rapporto aumenti all'aumentare della variabile indipendente, implicando quindi un risparmio di tempo maggiore per valori di n alti; questo asseconda l'intuizione, in quanto è ragionevole pensare che, aumentando n , aumenti anche il numero di record da scrivere in output, e tanti più sono i record da stampare, tanto più tempo è possibile risparmiare con la versione *partial*.

Tuttavia, anche in questo caso si verifica un'inversione di tendenza per valori di k molto alti (da 10000 in su), per lo stesso motivo esposto poco sopra.

Da quest'analisi si deduce, quindi, che per valori di k ragionevoli e non eccessivamente alti la versione parziale apporti considerevoli benefici all'algoritmo in termini di efficienza temporale, e che quindi, se non si necessita delle restanti righe informative che compongono il record, questa versione sia la più appropriata per effettuare il campionamento.

3.4 Analisi dei dati raccolti in termini di correttezza

In questa sezione ci si occuperà di analizzare l'influenza dei parametri di ingresso sulla correttezza dell'esecuzione dell'algoritmo, per trarne delle regole su cui poter improntare la scelta del parametro k più adeguato.

Come già considerato nell'analisi della complessità computazionale, il parametro k determina in maniera sostanziale il numero di iterazioni del ciclo *while* e, come si vedrà in questa sezione, ha un impatto fondamentale sulla correttezza dell'algoritmo.

Perché l'algoritmo effettui un campionamento omogeneo, è necessario che le seguenti condizioni siano rispettate:

1. Il dataset di input andrebbe analizzato quasi completamente, per evitare che alcuni record di input vengano esclusi dalla lettura, e che quindi abbiano probabilità inferiore (in quanto nulla) rispetto agli altri di essere scritti in output;
2. I record da scrivere in output dovrebbero essere presi omogeneamente nel corso dell'esecuzione dell'algoritmo, per evitare che nella prima parte di esecuzione (e quindi nelle prime iterazioni del ciclo *while*) vengano scritti eccessivi record in output rispetto alla seconda parte (ultime iterazioni del ciclo *while*), o viceversa;
3. Il fatto che l'algoritmo di campionamento preveda il “reinserimento” dei record già letti, a differenza degli algoritmi di campionamento senza reinserimento, ha lo svantaggio di prevedere la stampa multipla dello stesso record nel dataset di output (2). Tuttavia, la probabilità di scrivere un record di input più volte nel dataset di output andrebbe limitata al minimo, poiché, essendo il dataset di output di cardinalità inferiore, sarebbe ragionevole che ogni record di input venisse scritto zero o una volta in quello di output, e non di più.

Ci si concentrerà ora su queste tre condizioni separatamente.

3.4.1 Analisi di tutto il dataset in input

Per analizzare quest'aspetto è sufficiente osservare la Tabella 3.9, che mostra proprio il numero di record analizzati del dataset in input; è immediato osservare come alcuni record siano esclusi dall'analisi solo in due casi:

- $k = 1$;
- $k = 10$ e $n \in \{1007050, 251763\}$;

In queste due circostanze un numero irrisorio di record rispetto al totale (6 nel primo caso, 60 nel secondo) viene scartato dall'algoritmo; in tutti gli altri casi, le combinazioni di n e k permettono una lettura totale del dataset di partenza.

Una volta assicurata la lettura totale del dataset di input, è necessario assicurarsi che ad ogni iterazione del ciclo *while* venga scritto un numero uniforme di record e che questo venga fatto selezionando omogeneamente i record dai k disponibili; proprio questi due obiettivi saranno gli oggetti dell'analisi dei prossimi due paragrafi.

3.4.2 Omogeneità di scrittura ad ogni iterazione

È possibile effettuare le seguenti considerazioni preliminari:

- a) Il parametro k non può assumere valori superiori alla cardinalità del dataset di input (al caso limite, si può avere $k = m$);
- b) Il numero di iterazioni del ciclo *while* può assumere al massimo il valore

$$i_{\text{while}}^{\max} = \left\lfloor \frac{m}{k} \right\rfloor + (1 - u(p)) (m \bmod k) \quad \text{Equazione 3.1}$$

Come già osservato nel paragrafo 2.5.1 dedicato all'analisi della complessità computazionale;

- c) Se il ciclo *while* compie il massimo di iterazioni possibili, verranno analizzati per forza tutti i record del dataset di partenza; allo stesso modo, se nessun record di input viene escluso dall'analisi, allora il numero di iterazioni del ciclo *while* sarà massimo; secondo l'analisi paragrafo precedente (3.4.1), quindi, in tutti i casi di test, se non per $k = 1$ e $k = 10$ e $n \in \{1007050, 251763\}$, il ciclo *while* eseguirà esattamente i_{while}^{\max} iterazioni;
- d) Ad ogni iterazione del ciclo *while* vengono scritti esattamente c record nel dataset di output. Questi record potranno essere tutti differenti tra loro, oppure alcuni di essi potrebbero ripetersi; questo tema sarà trattato nel paragrafo successivo, relativo al contenimento della scrittura multipla dello stesso record.
- e) Ad ogni iterazione del ciclo *while* vengono analizzati esattamente k record del dataset di partenza, ad eccezione dell'ultima iterazione in cui, essendo $k < m$, ne verranno analizzati m , cioè quanti ancora ne rimangono da leggere.
- f) A determinare se un fissato record letto da input debba essere scritto o no, intervengono:

- a. L'aleatorietà del valore di c ;
- b. L'aleatorietà dei valori inseriti nel vettore S che, una volta determinato c , stabilirà quali record dovranno essere stampati in output, e quante volte.

Tuttavia, se c è nullo, il vettore S avrà cardinalità anch'essa a zero, e di conseguenza non verrà scritto nessuno dei k record di input nel dataset di output.

Se ne deduce quindi che la correttezza dell'algoritmo si basa sul parametro c , che dovrebbe assumere un valore più o meno costante ad ogni iterazione. Questo valore ideale, considerando quanto detto nei punti b) e c) precedenti, dovrebbe assumere il valore

$$\hat{c} = \frac{n}{i_{\text{while}}^{\max}} \approx n * \frac{k}{m} \quad \text{Equazione 3.2}$$

che è proprio il valore atteso della variabile aleatoria binomiale alla prima iterazione ($\mathbb{E}(X) = n * \frac{k}{m}$).

Tuttavia, la distribuzione binomiale che genera i valori di c cambia i suoi parametri ad ogni iterazione, e di conseguenza anche il suo valore atteso presumibilmente varierà; infatti, se all'iterazione i -esima si ha una distribuzione

$$X_1 \sim \text{Binomial}\left(n_1, \frac{k}{m_1}\right) \quad \text{Equazione 3.3}$$

all'iterazione successiva si avrà

$$X_2 \sim \text{Binomial}\left(n_1 - c_1, \frac{k}{m_1 - k}\right) \quad \text{Equazione 3.4}$$

Si osserva dall'[Equazione 3.4](#) che:

- ad ogni nuova iterazione il primo parametro della binomiale decresce di c , contribuendo alla diminuzione quindi il valor medio di X .
- Nella stessa iterazione il secondo parametro aumenta (poiché diminuisce il denominatore di k), contribuendo all'aumento del valor medio di X .

Questi due comportamenti si contrastano l'uno con l'altro, mantenendo un valore atteso tutto sommato costante; infatti, se per X_1 si ha

$$\mathbb{E}(X_1) = n_1 * \frac{k}{m_1} \quad \text{Equazione 3.5}$$

per X_2 otterremo

$$\mathbb{E}(X_2) = (n_1 - c_1) \frac{k}{m_1 - k}$$

Equazione 3.6

che, se sostituendo c_1 col suo valore atteso, risulta

$$\mathbb{E}(X_2) = n_1 \left(1 - \frac{k}{m_1}\right) \frac{k}{m_1 - k} = n_1 * \frac{k}{m_1} = \mathbb{E}(X_1)$$

Equazione 3.7

Si è giunti, quindi, alla conclusione che ad ogni iterazione la binomiale X varia sempre i suoi parametri, ma mantiene costante il suo valore atteso, permettendo una stabilità del parametro c che viene generato a partire da essa.

Questa stabilità, tuttavia, non si osserva nel caso in cui k sia eccessivamente basso rispetto a m , ed n sia anch'esso limitato: il prodotto tra il primo parametro (n) e il secondo (k/m) della distribuzione binomiale è talmente basso da rendere il parametro c sempre nullo per le prime h iterazioni, causando quindi l'automatica esclusione dei primi $k * h$ record letti da input; la conseguenza sarà che i primi record letti avranno probabilità molto più bassa rispetto a quelli finali di essere scritti in output, causando una prevalenza, nel dataset di output, di record che si trovano alla fine del dataset di partenza rispetto a quelli che si posizionano alla fine.

Un altro caso in cui il numero di record scritti in output non sarà all'incirca \hat{c} , ma sarà invece inferiore, è l'ultima iterazione del ciclo *while*, in cui k assumerà il valore m , quindi $X \sim Binomial(n, 1)$ e di conseguenza c sarà pari ad n . Una stima del valore di n (e quindi di c) in quest'ultima iterazione può essere calcolato ricordando che, se in tutte le iterazioni precedenti il valore di c è costante, allora avremo che il residuo di n dopo i decrementi di ogni iterazione sarà

$$n_{res} = n \bmod \hat{c}$$

Equazione 3.8

ovvero un valore sicuramente inferiore a \hat{c} .

Ciò che è appena stato descritto si può verificare nei grafici di seguito, che mostrano la decrescita del parametro n ad ogni iterazione del ciclo *while*, per tutti i k considerati:

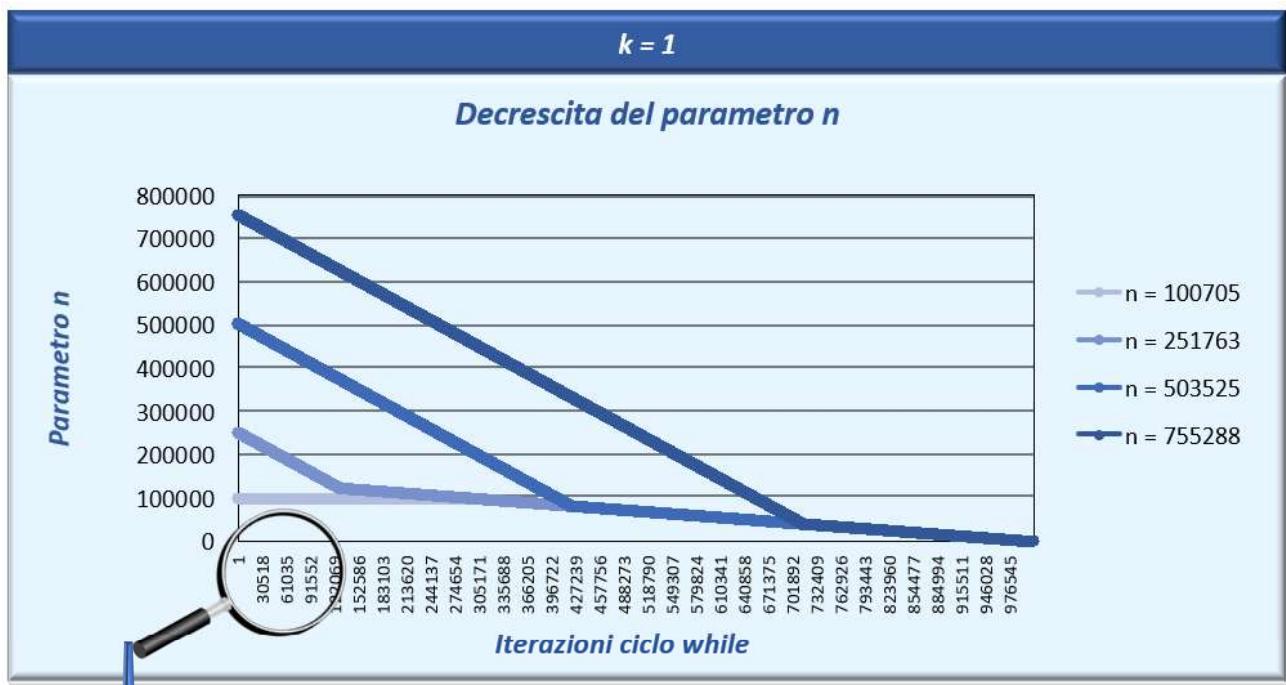


Grafico 3.9 - decrescita del parametro n per $k = 1$

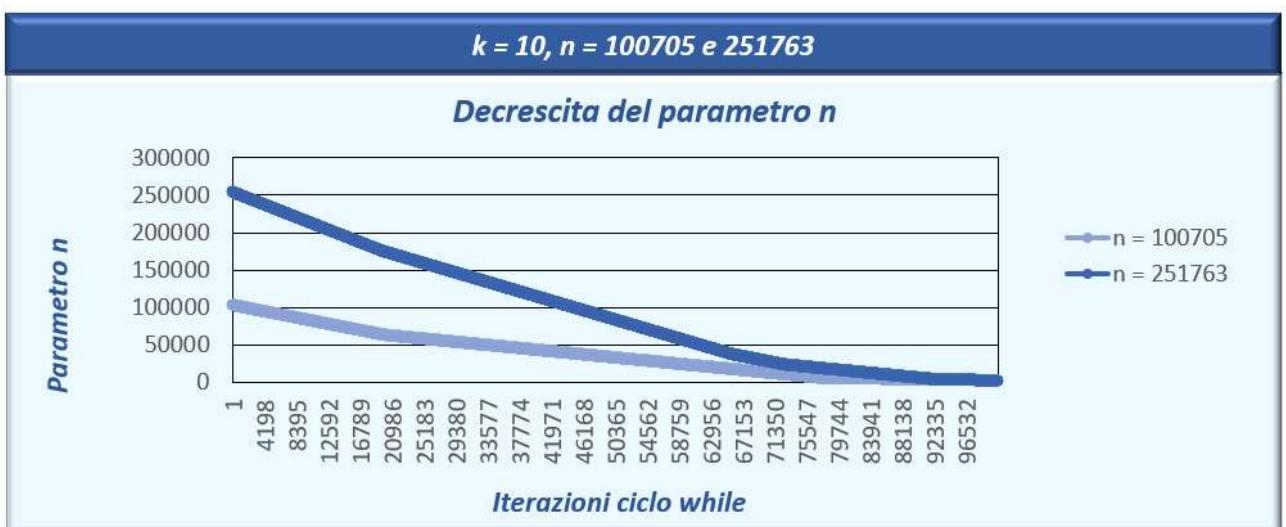


Grafico 3.10 - decrescita del parametro n per $k = 10$ e $n = 100705 \text{ e } 251763$

$$k = 10, n = 503525 \text{ e } 755288$$

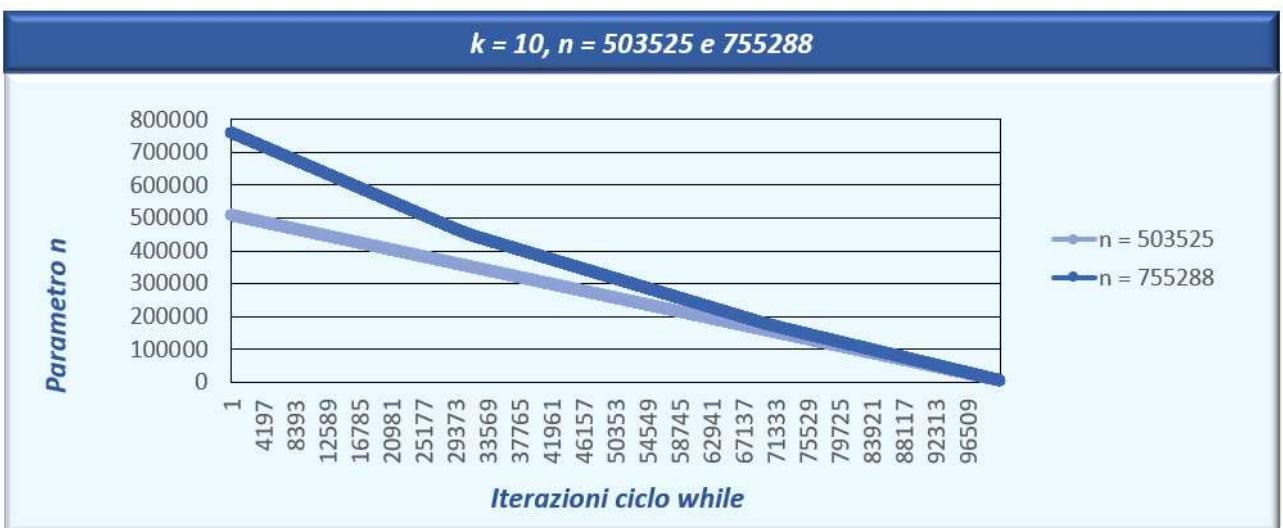


Grafico 3.11 - decrescita del parametro n per $k = 10$ e $n = 503525$ e 755288

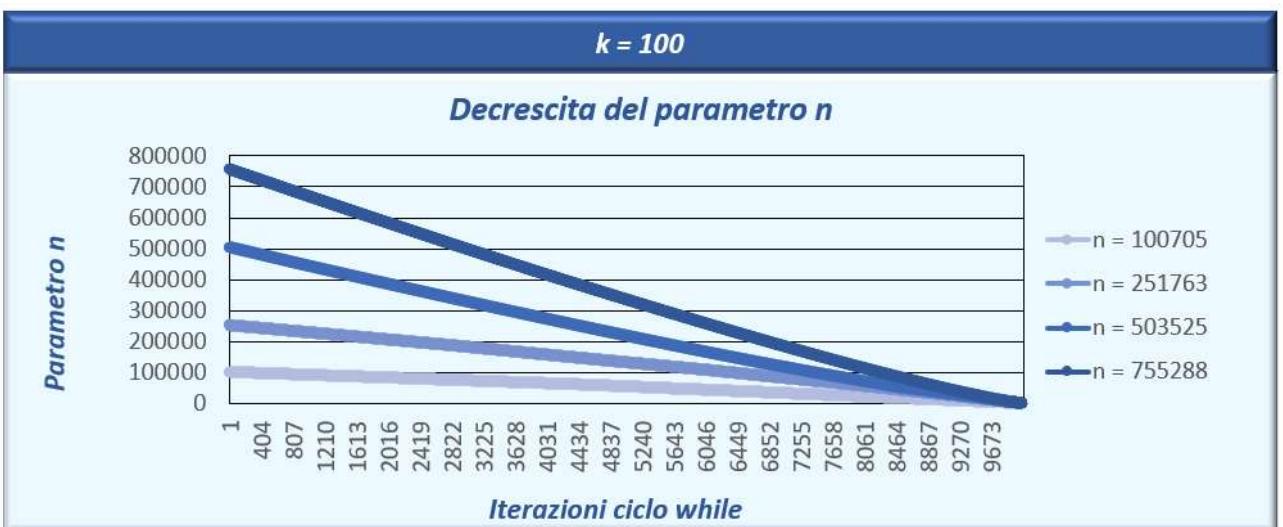


Grafico 3.12 - decrescita del parametro n per $k = 100$

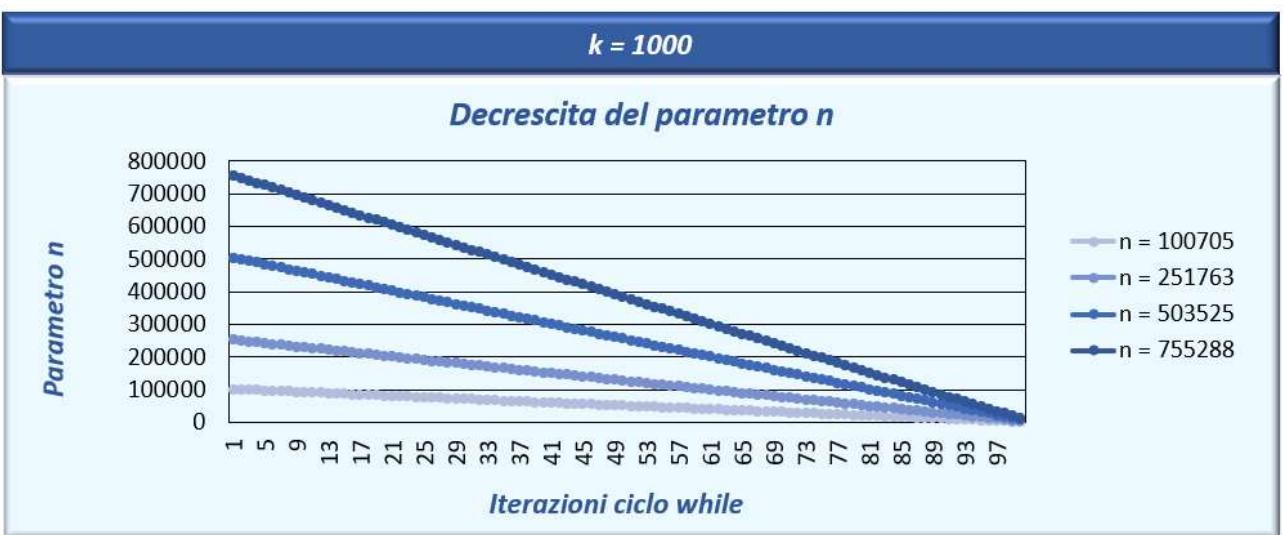


Grafico 3.13 - decrescita del parametro n per $k = 1000$

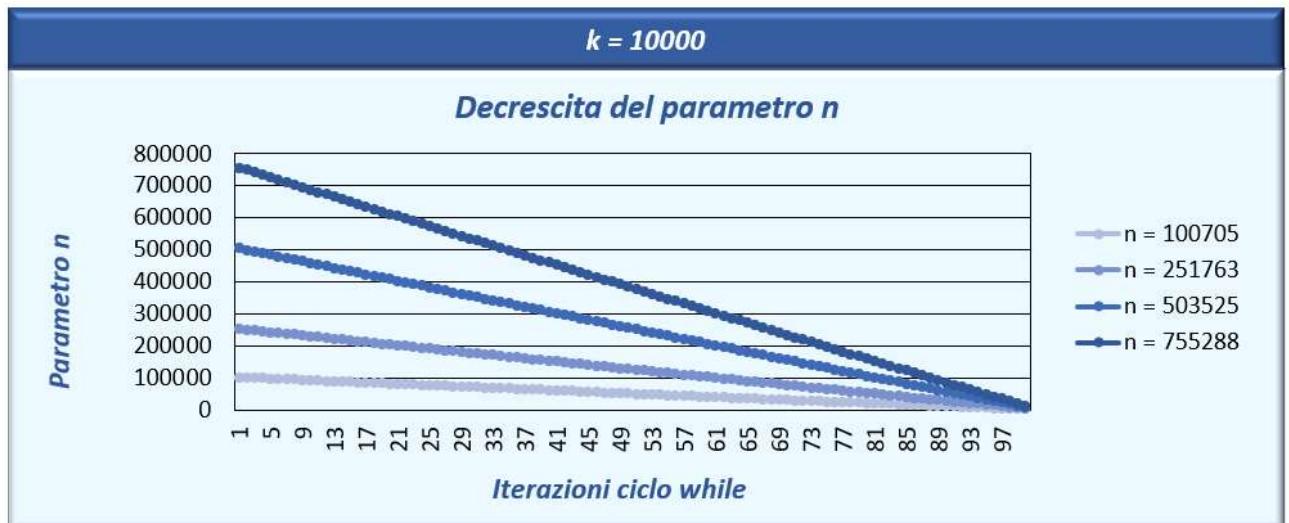


Grafico 3.14 - decrescita del parametro *n* per *k* = 10000

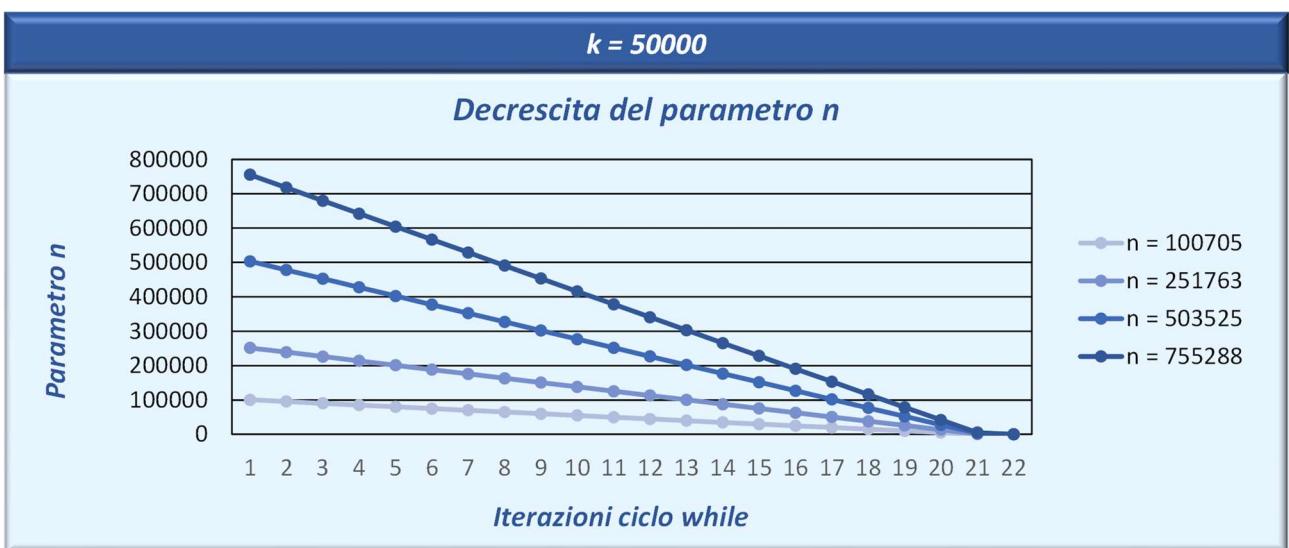


Grafico 3.15 - decrescita del parametro *n* per *k* = 10000

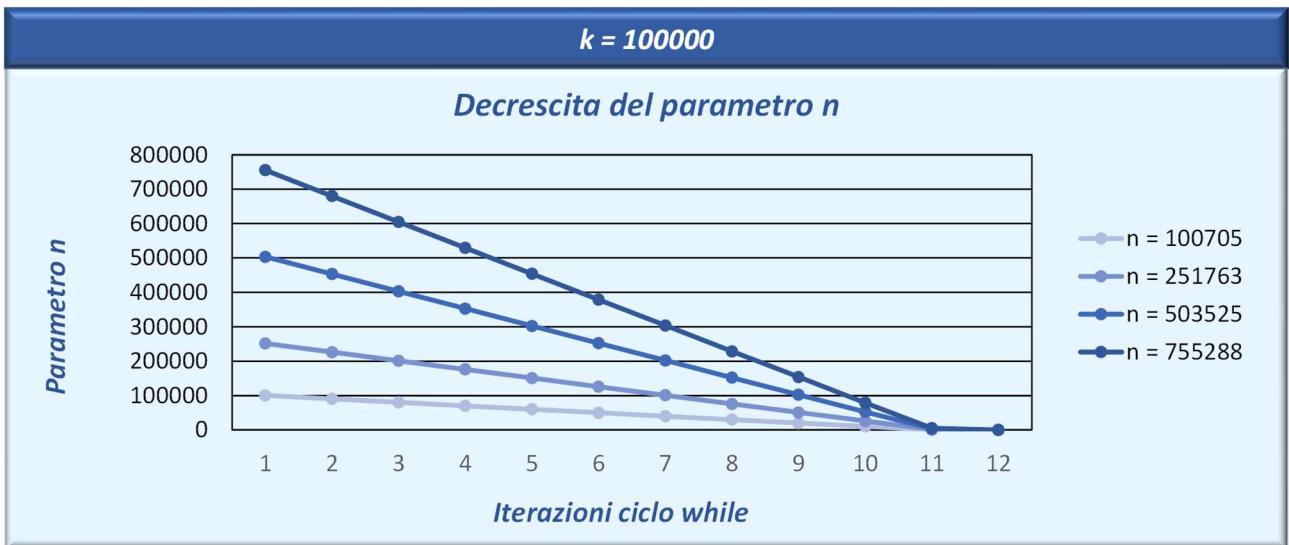


Grafico 3.16 - decrescita del parametro *n* per *k* = 100000

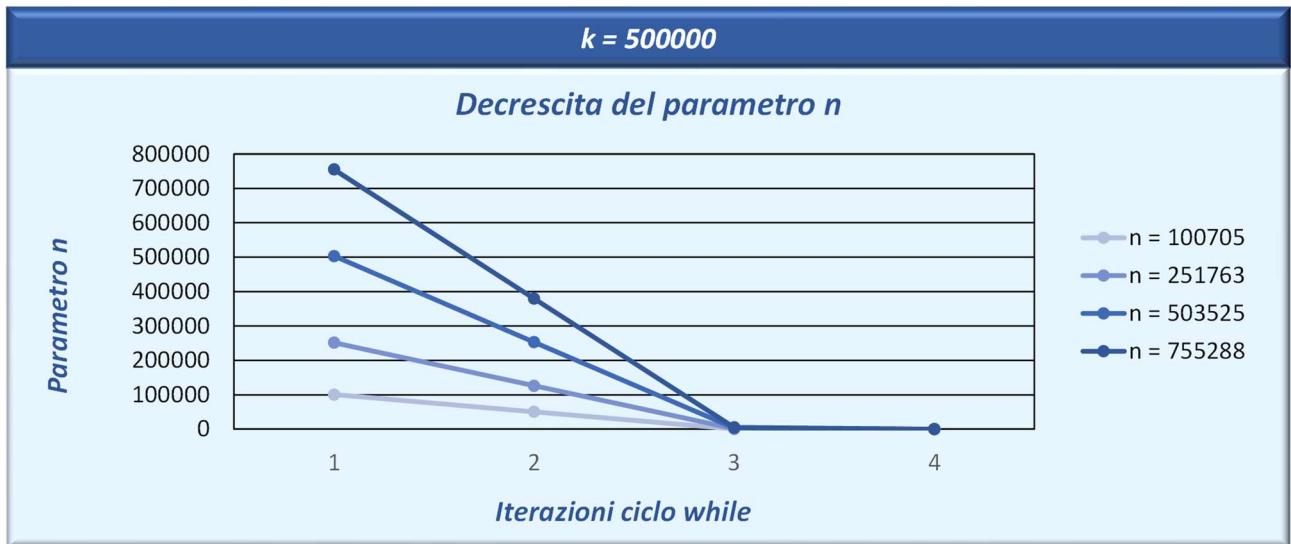


Grafico 3.17 - decrescita del parametro n per $k = 500000$

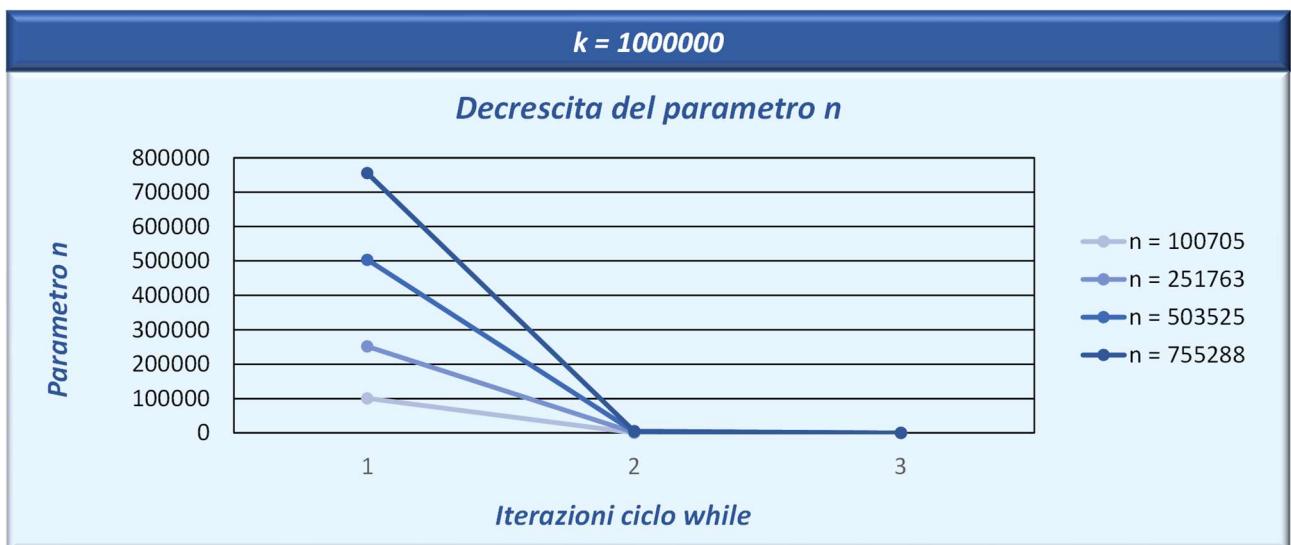


Grafico 3.18 - decrescita del parametro n per $k = 1000000$

Si può notare, quindi, come il parametro k influisca sull’omogenea decrescenza del parametro n :

- Per $k = 1$ e $n = 100705$ si ha un primo tratto della curva con pendenza nulla, il che significa che in tutte le iterazioni di quel tratto non viene scritto nemmeno un record in output; questo verifica quanto ipotizzato nella parte introduttiva riguardo al prodotto $k * m$ eccessivamente basso;
- Per $k = 1$ e $n \neq 100705$ e si ha una netta variazione di inclinazione (in punti diversi, a seconda del parametro n), che sta a indicare una densità di record scritti in output più alta per i record iniziali rispetto a quelli finali;

- Per $k = 10$ si distinguono tre tratti differenti per ogni curva, ognuno con inclinazione diversa che decresce all'aumentare del numero di iterazioni; questo testimonia ancora una volta quanto spiegato nel punto precedente, anche se l'effetto in questo caso è quasi impercettibile, tanto da non notarsi graficamente nel diagramma;
- Per $k \in \{100, 1000, 10000\}$ il grafico presenta una retta pressoché perfetta, che indica un'omogeneità di scrittura ad ogni iterazione;
- Per $k \in \{50000, 100000, 500000, 1000000\}$ si nota una curva costantemente rettilinea, che solo nell'ultimo tratto (corrispondente all'ultima iterazione) cambia di inclinazione, più o meno drasticamente a seconda del valore del parametro n .

I valori di k accettabili secondo queste osservazioni, quindi, sono $k \in \{100, 1000, 10000\}$, che garantiscono una scrittura uniforme ad ogni iterazione. Anche i valori $k \in \{50000, 100000\}$ possono essere considerati validi (soprattutto per valori di n bassi), tenendo a mente però che, nell'ultima iterazione del ciclo, il numero di record scritti sarà inferiore a quello di tutte le iterazioni precedenti.

3.4.3 Contenimento della scrittura multipla dello stesso record

Una volta assicurata anche l'omogeneità di scrittura ad ogni iterazione, l'ultimo passo per garantire la correttezza dell'algoritmo è limitare il più possibile, tramite una scelta oculata del parametro k , la ripetizione dello stesso record in output.

Studiamo ora la correlazione tra numero di record duplicati e parametri di input; questo dato è riportato nella tabella seguente, ottenuta dal file “parameterChanges.csv” di output dell'applicazione per test:

Parameter k	Parameter n			
	100705	251763	503525	755288
1	0	0	0	0
10	6	3589	52887	174811
100	996	24387	104406	221456
1000	4558	28786	106908	223728
10000	4958	29150	107736	224807
50000	7245	42246	150282	303271
100000	13748	75036	243463	455561
500000	48699	185863	434421	684366
1000000	68742	217191	467144	1007050

Tabella 3.15 – numero di record ripetuti nel dataset di output

Il numero di volte in cui un determinato record di input viene scritto in output è determinato dal vettore aleatorio S , originato da una funzione generatrice di numeri random nell'intervallo $[1, k]$: avere ripetizioni dello stesso intero all'interno di S implica automaticamente la scrittura multipla di un determinato record di input nel dataset di output. La cardinalità di S , ad ogni iterazione, assumerà sempre il valore c . Si possono presentare quindi i due seguenti casi:

- $c > k$: in questo caso ci saranno certamente record duplicati nel dataset di output, in quanto il numero di record disponibili per essere scritti (k) è inferiore al numero di record che dovranno essere scritti (c); ciò, tuttavia, non si verifica mai nei test svolti.
- $c \leq k$: in questo caso non è assicurato che vengano stampati record ripetuti, ma la probabilità che questo accada è tanto più alta quanto è alto il valore di c ; lo si può osservare dal grafico seguente, che mostra l'evoluzione del parametro c e il numero di record duplicati per ogni iterazione del ciclo *while* nel caso di $k = 50000$ e $n = 100705$:

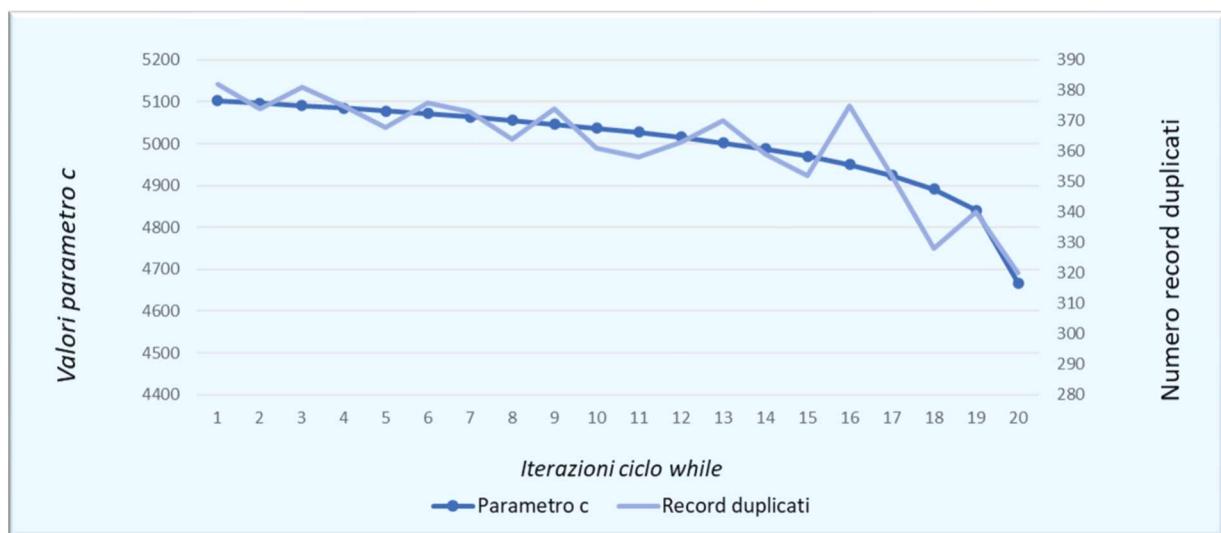


Grafico 3.19 - evoluzione del parametro c e dei record duplicati per $k = 50000$ e $n = 100705$

Essendo il numero di record duplicati un valore determinato dall'aleatorietà del vettore S , esso assumerà, come apprezzabile dal grafico, valori molto variabili; tuttavia, come è facile notare, la sua tendenza segue quella di c .

Il parametro c a sua volta può aumentare il suo valore:

- all'aumentare del valore di n : incrementandosi n , infatti, aumenta il primo parametro della binomiale, e questo fa sì che i valori di c ottenuti da essa siamo maggiori; il seguente grafico esemplificativo indica l'aumento dei record duplicati al variare di n per $k = 50000$:



Grafico 3.20 - andamento del numero di record duplicati rispetto ad n per n = 50000

- all'aumentare del valore di k : lo si osserva dai grafici riportati di seguito che indicano l'evoluzione del numero di record duplicati al variare del parametro k :

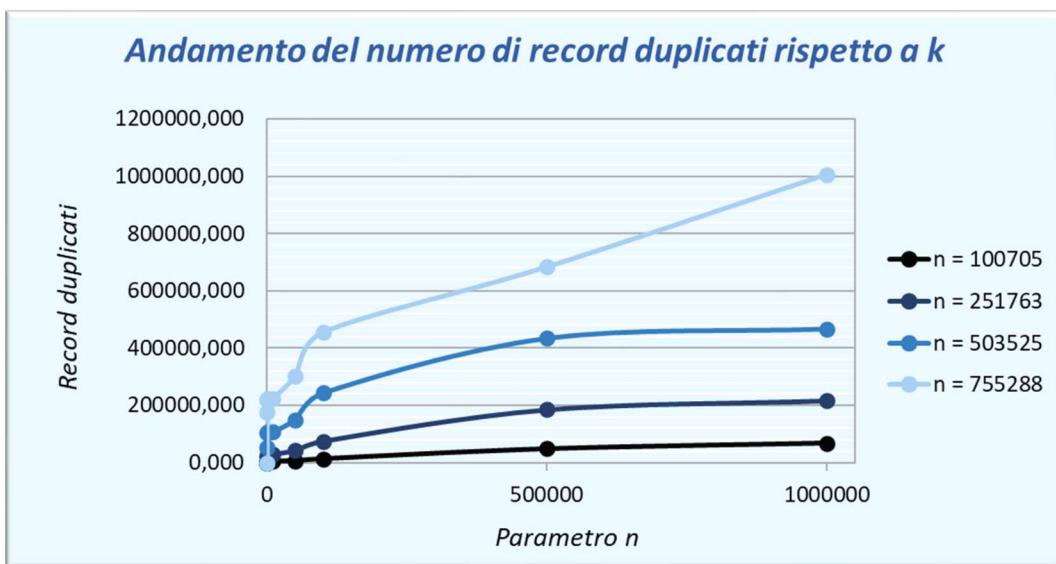


Grafico 3.21 - andamento del numero di record duplicati rispetto ad n per n = 50000

Questi due comportamenti si possono comprendere facilmente eseguendo un'analisi probabilistica del numero di record duplicati in output: prendendo in considerazione lo pseudocodice di Figura 2.4, la probabilità che, all'interno del ciclo *for*, l'indice j sia trovato nella posizione t -esima del vettore S è pari a

$$p_{S, j, t} = P(j = S[t]) = \frac{1}{k}$$

essendo il valore $S[t]$ assegnato tramite una funzione generatrice di valori random in $[1, k]$.

Poiché ogni valore del vettore S è assegnato indipendentemente dagli altri, si può affermare che la probabilità che l'indice j sia trovato esattamente d volte in S , e quindi la probabilità che il record in considerazione venga scritto d volte in output, è pari a

$$p_{rep, d} = d * P(j = S[t]) = d * p_{S, j, t} = d * \frac{1}{k}$$

Se ne deduce facilmente che la probabilità che un record sia stampato più di una volta, e quindi la probabilità che l'indice j abbia più di un'occorrenza in S , è pari a

$$p_{duplicati} = \sum_{i=2}^c p_{rep, i} = \sum_{i=2}^c i * \frac{1}{k} = \frac{c^2 + c - 2}{2k}$$

che è una funzione crescente all'aumentare di c . Poiché c (in valore atteso) cresce all'aumentare di k e n (il suo valore atteso è pari a $\mathbb{E}(X) = n * \frac{k}{m}$), possiamo dedurre che anche il numero di record duplicati avrà lo stesso andamento, confermando quanto osservato nei due grafici precedenti.

In conclusione, è facile dedurre come, specialmente per parametri n particolarmente alti, sia necessario utilizzare valori di k relativamente bassi per limitare l'eccessiva ripetizione dello stesso record nel dataset di output.

4 Conclusioni

Nella sezione dedicata all’analisi dei risultati sperimentali si ha avuto modo di valutare come influiscano i diversi tipi di applicazioni o algoritmo e i diversi valori assegnati ai parametri di input sui tempi di esecuzione e sulla correttezza dell’algoritmo.

Per quanto riguarda le varie versioni dell’applicazione di supporto, abbiamo si è potuto constatare che utilizzare l’una o l’altra non influisce sul tempo o sulla correttezza dell’esecuzione dell’algoritmo. In merito alla tipologia di algoritmo, invece, si è verificato come essa non infierisca sulla correttezza, ma tuttavia vada a determinare il tempo di esecuzione: il caso più evidente è quello di *full* e *partial version*, che presentano tempi di esecuzione molto differenti (paragrafo 3.3.3), ma una minima discrepanza si nota anche per *Iterate* e *Massive R-W version*, come analizzato nel paragrafo 3.2.3. A questo proposito possiamo affermare che, se si necessita della sola informazione genetica contenuta nella terza riga, sarà conveniente utilizzare la versione *partial*, molto più efficiente. Una volta fatta questa scelta, se la versione utilizzata sarà quella *partial* allora sarà sensato preferire la versione *Iterate*; al contrario, decidendo per la *full version*, la versione consigliata sarà l’*Iterate* per k ed n relativamente piccoli, l’altra altrimenti.

Discutendo invece dell’influenza dei parametri sul tempo di esecuzione, si è potuto confermare l’attendibilità della complessità computazionale stimata nel paragrafo 2.5.1, verificando che il tempo di esecuzione dell’algoritmo dipende linearmente dal parametro n , che, per un algoritmo di campionamento con cardinalità di output pari ad n , è il risultato migliore ottenibile. Allo stesso modo è stato possibile provare la dipendenza lineare del tempo di esecuzione dal secondo parametro di ingresso dell’algoritmo, ovvero k .

Oltre a ciò, il parametro k è stato protagonista della maggior parte delle analisi di questo elaborato in termini di correttezza, e in base al tema di analisi di ogni paragrafo, si è giunti a conclusioni differenti su quale valore fosse meglio assegnare a tale variabile:

- Nel paragrafo 3.3.3 si è giunti a concludere che, in termini di tempi di esecuzione, sia più conveniente assegnare a k valori bassi, anche se, per valori di k contenuti (inferiori di circa due ordini di grandezza rispetto a m) le variazioni in termini di tempo sono irrisonie;

- Nel paragrafo 3.4.2 è stato possibile osservare come i valori che permettono una scrittura omogenea dei record siano quelli intermedi, da quattro a due ordini di grandezza inferiori rispetto al parametro m ;
- Nel paragrafo 3.4.32.4.1 è stato possibile osservare come il numero di record ripetuti in output aumenti all'aumentare di k ; questo significa che, se ci si pone l'obiettivo di limitare il numero di ripetizioni in output, l'unica soluzione sia quella di limitare il valore di k (solo $k = 1$, tra i valori provati, è quello che garantisce l'assenza assoluta di ripetizioni).

È immediato osservare come i valori di k eccessivamente alti siano da evitare, sia in termini di tempi di esecuzione che in termini di correttezza. In base a quanto si vuole ottenere dall'algoritmo, si sceglieranno quindi valori di k bassi se si vuole garantire una limitata ripetizione dei record in output a discapito dell'omogeneità di scrittura ad ogni iterazione, intermedi in caso contrario.

Dall'analisi si evince, quindi, che scegliendo accuratamente i parametri di ingresso dell'algoritmo esso funzionerà in maniera efficiente e corretta, generando un dataset di output in tempistiche limitate e con contenuto informativo che rappresenta in maniera più completa possibile il dataset di partenza.

5 Bibliografia

1. **Vandin, Santoro, Pellegrina.** *SPRISS: Approximating Frequent k-mers by Sampling Reads.*,.
2. **Byung-Hoon Park, George Ostrouchov, Nagiza F. Samatova, Al Geist.** Reservoir-based Random Sampling with Replacement from Data Stream.
3. **Mitzenmacher, Michael.** *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis.* 2005.

(3)

6 Appendice

6.1 Appendice 1: codice sorgente delle applicazioni

In questa sezione verrà inserito il codice sorgente delle applicazioni sviluppate; dato l'elevato numero di versioni per queste applicazioni, nel seguito verrà mostrato il codice unicamente delle applicazioni per test (CLI-TOT e GUI-TOT) che in sé includono tutte le versioni dell'algoritmo, la prima con interfaccia da riga di comando e la seconda con Graphical User Interface.

Di seguito si mostra la gerarchia delle directory di ognuno dei due progetti, che presentano alcune parti comuni (quelle strettamente legate all'algoritmo) e altre leggermente differenti (quelle legate alla modalità di richiesta e ottenimento dei dati in input dall'utente).

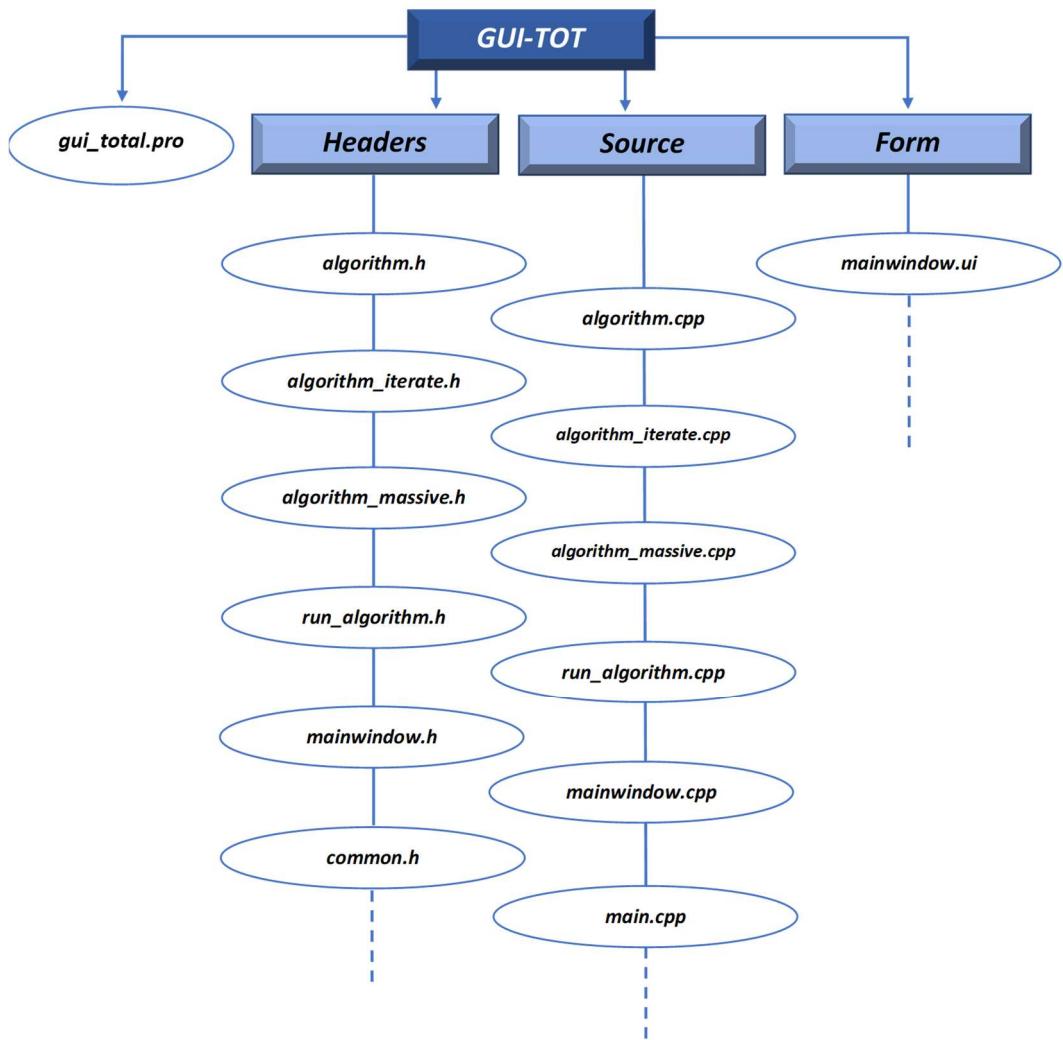


Figura 6.1 – gerarchia delle directory di GUI-TOT

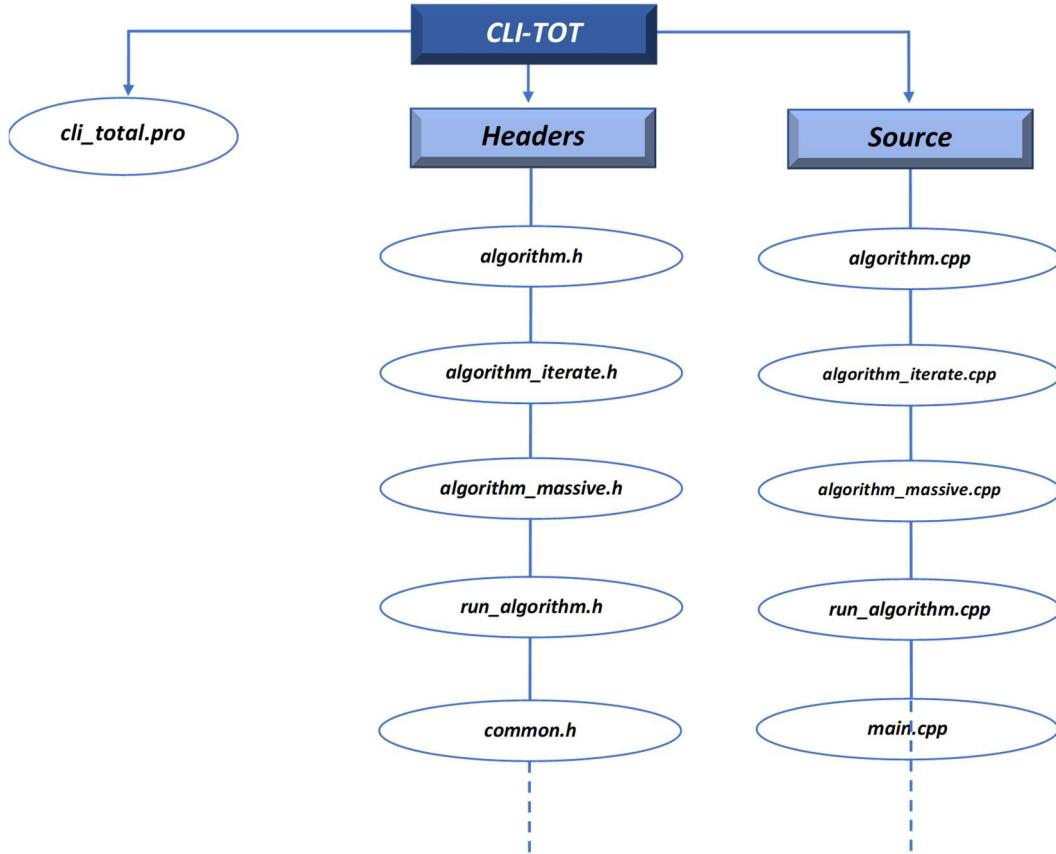


Figura 6.2 - gerarchia delle directory di CLI-TOT

- File *main.h/.cpp* (GUI): classe per avviare l’interfaccia grafica dell’applicazione;
- File *main.cpp* (CLI) e *mainwindow.h/.cpp* (GUI): classi per richiedere i dati in input ed elaborarli;
- File *run_algorithm.h/.cpp*: classe per avviare i diversi tipi di algoritmo e stampare in output i file di log e i file .csv necessari;
- File *algorithm_iterate.h/.cpp* e *algorithm_massive.h/.cpp*: classi che eseguono il vero e proprio algoritmo di campionamento;
- File *common.h*: file di intestazione contenente strutture e metodi utili comuni per più classi;
- File *mainwindow.ui* (GUI): file che definisce la struttura grafica dell’interfaccia dell’applicazione.

Come prima cosa verranno mostrate le classi comuni tra i due algoritmi (*run_algorithm.h/.cpp*, *algorithm_iterate.h/.cpp*, *algorithm_massive.h/.cpp*, *common.h*) per poi passare a quelli singolari per i due tipi di applicazione.

Sarà escluso dall’inserimento in quest’appendice il codice di *mainwindow.ui*, che risulterebbe eccessivamente lungo e poco funzionale.

6.1.1 Codice sorgente delle classi comuni

6.1.1.1 Codice sorgente di common.h

```
1  #ifndef COMMON_H
2  #define COMMON_H
3
4  #include <string>
5  #include <qmessagebox.h>
6  #include <QAbstractButton>
7  #include <time.h>
8  #include <cmath>
9  #include <ctgmath>
10 #include <QMainWindow>
11 #include <QMessageBox>
12 #include <iostream>
13 #include <fstream>
14 #include <sys/stat.h>
15 #include <ctgmath>
16 #include <QDir>
17 #include <regex>
18 #include <QObject>
19 #include <vector>
20 #include <random>
21
22 using namespace std;
23
24 struct inputValues {
25     string input_file_path = "";
26     string output_file_dir = "";
27     int output_dataset_size = -1;
28     int input_dataset_size = -1;
29     int parameter_k = -1;
30 };
31
32 struct outputValues {
33     int k_in;
34     int n_in;
35     int written;
36     double elapsed_wall;
37     double elapsed_CPU;
38     string outputFileName;
39     int inputRecordSelected = 0;
40     vector<int> n_decrease;
41     vector<int> m_decrease;
42     vector<int> c_values;
43     vector<int> duplicated_records;
44     int whileiterations = 0;
45 };
46
47 struct parametersChangesObject {
48     string n;
49     string k;
50     int whileIterations;
51     vector<string> n_dec;
52     vector<string> m_dec;
53     vector<string> c_val;
54     vector<string> duplicated;
55 };
56
57 struct record {
58     string line1;
59     string line2;
60     string line3;
61     string line4;
62 };
63
64 struct parameterSettingStatus {
65     int toStart = 0;
66     int setted_1 = 1;
67     int setted_2 = 2;
68     int setted_3 = 3;
69     int setted_4 = 4;
70 };
71
72 inline string getOutputFileName(string toAppend, inputValues input) {
```

```

74     string fileName;
75     time_t t = time(0);
76     struct tm * now = localtime(& t );
77     fileName = "outData_" + toAppend + "_" +
78         "n" + to_string(input.output_dataset_size) + "_" +
79         "k" + to_string(input.parameter_k) + "_" +
80         to_string(now->tm_year + 1900) +
81         to_string(now->tm_mon + 1) +
82         to_string(now->tm_mday) +
83         to_string(now->tm_hour) +
84         to_string(now->tm_min) +
85         to_string(now->tm_sec) +
86         ".fastq";
87     return fileName;
88 }
89
90 inline string getOutputFileLogName(string tag, inputValues input) {
91     string fileName;
92     time_t t = time(0);
93     struct tm * now = localtime(& t );
94     fileName = "logExecutionTimes" + tag + " " +
95         "n" + to_string(input.output_dataset_size) + "_" +
96         "k" + to_string(input.parameter_k) + "_" +
97         to_string(now->tm_year + 1900) +
98         to_string(now->tm_mon + 1) +
99         to_string(now->tm_mday) +
100        to_string(now->tm_hour) +
101        to_string(now->tm_min) +
102        to_string(now->tm_sec) +
103        ".log";
104    return fileName;
105 }
106
107 #endif // COMMON_H
108

```

6.1.1.2 Codice sorgente di algorithm.h

```

1  #ifndef ALGORITHM_H
2  #define ALGORITHM_H
3
4  #include "common.h"
5
6  using namespace std;
7
8  class Algorithm
9  {
10 public:
11     Algorithm(inputValues input, int version, string dirTag);
12     virtual outputValues run() = 0;
13
14 protected:
15     inputValues inputVal;
16     outputValues outputVal;
17     int version;
18     string directTag;
19
20     struct timespec begin_wall, begin_CPU, end_wall, end_CPU;
21
22     ifstream input_file;
23     ofstream output_file;
24
25     void uniformSampling();
26     void getRandomSet(vector<int> set, int max_num, int size);
27     int findOccurrences(vector<int> v, int element_to_find);
28     int getBinomialDistribution(int a, double b);
29     void startMeasuringTime();
30     void stopMeasuringTime();
31     virtual record getNextRecord(int index) = 0;
32     virtual void addRecordToOutputFile(record r) = 0;
33 };
34
35 #endif // ALGORITHM_H
36

```

6.1.1.3 Codice sorgente di algorithm_iterate.h

```
1 #ifndef ALGORITHM_ITERATE_H
2 #define ALGORITHM_ITERATE_H
3
4 #include "algorithm.h"
5
6 using namespace std;
7
8 class AlgorithmIterate : public Algorithm {
9 public:
10     AlgorithmIterate(inputValues input, int version, string dirTag) :
11         Algorithm(input, version, dirTag) {}
12     using Algorithm::run;
13     using Algorithm::getNextRecord;
14     using Algorithm::addRecordToOutputFile;
15     virtual outputValues run() override;
16 protected:
17     void openInputOutputFiles();
18     virtual record getNextRecord(int index) override;
19     virtual void addRecordToOutputFile(record my_record) override;
20     void closeInputOutputFiles();
21 };
22
23 #endif // ALGORITHM_ITERATE_H
24
```

6.1.1.4 Codice sorgente di algorithm_massive.h

```
1 #ifndef ALGORITHM_MASSIVE_H
2 #define ALGORITHM_MASSIVE_H
3
4 #include "algorithm.h"
5
6 using namespace std;
7
8 class AlgorithmMassive : public Algorithm {
9 public:
10     AlgorithmMassive(inputValues input, int version, string dirTag) :
11         Algorithm(input, version, dirTag) {}
12     using Algorithm::run;
13     using Algorithm::getNextRecord;
14     using Algorithm::addRecordToOutputFile;
15     virtual outputValues run() override;
16 protected:
17     void getRecordList();
18     virtual record getNextRecord(int index) override;
19     virtual void addRecordToOutputFile(record my_record) override;
20     void printOutputRecordListToFile();
21
22     vector<record> output_record_list;
23     vector<record> input_record_list;
24 };
25
26 #endif // ALGORITHM_MASSIVE_H
27
```

6.1.1.5 Codice sorgente di run_algorithm.h

```
1  ifndef ALGORITHM_H
2  define ALGORITHM_H
3
4  include "common.h"
5
6  using namespace std;
7
8  class Algorithm
9  {
10 public:
11     Algorithm(inputValues input, int version, string dirTag);
12     virtual outputValues run() = 0;
13
14 protected:
15     inputValues inputVal;
16     outputValues outputVal;
17     int version;
18     string directTag;
19
20     struct timespec begin_wall, begin_CPU, end_wall, end_CPU;
21
22     ifstream input_file;
23     ofstream output_file;
24
25     void uniformSampling();
26     void getRandomSet(vector<int> set, int max_num, int size);
27     int findOccurrences(vector<int> v, int element_to_find);
28     int getBinomialDistribution(int a, double b);
29     void startMeasuringTime();
30     void stopMeasuringTime();
31     virtual record getNextRecord(int index) = 0;
32     virtual void addRecordToOutputFile(record r) = 0;
33 };
34
35 #endif // ALGORITHM_H
36
```

6.1.1.6 Codice sorgente di algorithm.cpp

```

1 #include "algorithm.h"
2
3 Algorithm::Algorithm(inputValues input, int version, string dirTag)
4 {
5     this->version = version;
6     inputVal = input;
7     directTag = dirTag;
8     outputVal.k_in = inputVal.parameter_k;
9     outputVal.n_in = inputVal.output_dataset_size;
10 }
11
12 void Algorithm::startMeasuringTime() {
13     clock_gettime(CLOCK_REALTIME, &begin_wall);
14     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &begin_CPU);
15 }
16
17 void Algorithm::uniformSampling() {
18     int input_dataset_index = 0;
19     int n = inputVal.output_dataset_size;
20     int m = inputVal.input_dataset_size;
21     int k = inputVal.parameter_k;
22     outputVal.written = 0;
23     while (n > 0) {
24         outputVal.n_decrease.push_back(n);
25         outputVal.m_decrease.push_back(m);
26         outputVal.whileiterations++;
27         k = (k < m) ? k : m;
28         int c = getBinomialDistribution(n, k/((double)m));
29         outputVal.c_values.push_back(c);
30         std::vector<int> S;
31         for (int i = 0; i < c; i++) {
32             S.push_back(rand()%k+1);
33         }
34         int dupl_rec = 0;
35         for (int j = 1; j <= k; j++) {
36             record r = getNextRecord(input_dataset_index);
37             outputVal.inputRecordSelected++;
38             int r_c = findOccurrences(S, j);
39             dupl_rec += (r_c > 1) ? (r_c - 1) : 0;
40             for (int z = 0; z < r_c; z++) {
41                 addRecordToOutputFile(r);
42                 outputVal.written++;
43             }
44             input_dataset_index++;
45         }
46         outputVal.duplicated_records.push_back(dupl_rec);
47         n = n - c;
48         m = m - k;
49     }
50     outputVal.inputRecordSelected = input_dataset_index;
51     outputVal.n_decrease.push_back(n);
52     outputVal.m_decrease.push_back(m);
53 }
54
55 void Algorithm::stopMeasuringTime() {
56     clock_gettime(CLOCK_REALTIME, &end_wall);
57     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end_CPU);
58     long second_wall = end_wall.tv_sec - begin_wall.tv_sec;
59     long nanosec_wall = end_wall.tv_nsec - begin_wall.tv_nsec;
60     outputVal.elapsed_wall = second_wall + nanosec_wall*1e-9;
61     long second_CPU = end_CPU.tv_sec - begin_CPU.tv_sec;
62     long nanosec_CPU = end_CPU.tv_nsec - begin_CPU.tv_nsec;
63     outputVal.elapsed_CPU = second_CPU + nanosec_CPU*1e-9;
64 }
65
66 int Algorithm::getBinomialDistribution(int a, double b) {
67     std::default_random_engine generator;
68     b = (b < 1) ? b : 1;
69     std::binomial_distribution<int> distribution(a, b);
70     return distribution(generator);
71 }
72
73 void Algorithm::getRandomSet(std::vector<int> set, int max_num, int size) {

```

```
74     for (int i = 0; i < size; i++) {
75         set.push_back(rand()%max_num+1);
76     }
77 }
78
79 int Algorithm::findOccurrences(vector<int> *set, int element_to_find) {
80     int count = 0;
81     auto it = std::begin(*set);
82     while (it != std::end(*set)) {
83         if (*it == element_to_find) {
84             count++;
85             it = set->erase(it);
86         } else {
87             it++;
88         }
89     }
90     return count;
91 }
92
93
```

6.1.1.7 Codice sorgente di algorithm_iterate.cpp

```
1 #include "algorithm_iterate.h"
2
3 outputValues AlgorithmIterate::run() {
4     startMeasuringTime();
5     openInputOutputFiles();
6     uniformSampling();
7     closeInputOutputFiles();
8     stopMeasuringTime();
9     return outputVal;
10 }
11
12 void AlgorithmIterate::openInputOutputFiles() {
13     input_file.open(inputVal.input_file_path);
14
15     outputVal.outputFileName = (version == 0) ? getOutputFileName("full", inputVal)
16         : getOutputFileName("partial", inputVal);
17     string tagVersion = (version == 0) ? "FullVersion" : "PartialVersion";
18     string outputPath = inputVal.output_file_dir + "\\\" + directTag + "\\\" +
19     tagVersion + "\\OutputDataset\\\" + outputVal.outputFileName;
20     output_file.open(outputPath, ios::trunc);
21 }
22
23 record AlgorithmIterate::getNextRecord(int index) {
24     string line;
25     record my_record;
26     if (version == 0) {
27         getline (input_file, line);
28         my_record.line1 = line;
29         getline (input_file, line);
30         my_record.line2 = line;
31         getline (input_file, line);
32         my_record.line3 = line;
33         getline (input_file, line);
34         my_record.line4 = line;
35     } else {
36         getline (input_file, line);
37         getline (input_file, line);
38         getline (input_file, line);
39         my_record.line3 = line;
40         getline (input_file, line);
41     }
42     return my_record;
43 }
44
45 void AlgorithmIterate::addRecordToOutputFile(record my_record) {
46     if (version == 0) {
47         output_file << my_record.line1 << endl;
48         output_file << my_record.line2 << endl;
49         output_file << my_record.line3 << endl;
50         output_file << my_record.line4 << endl;
51     } else {
52         output_file << my_record.line3 << endl;
53     }
54 }
55 void AlgorithmIterate::closeInputOutputFiles() {
56     input_file.close();
57     output_file.close();
58 }
```

6.1.1.8 Codice sorgente di algorithm_massive.cpp

```
1 #include "algorithm_massive.h"
2
3 outputValues AlgorithmMassive::run () {
4     startMeasuringTime();
5     getRecordList();
6     uniformSampling();
7     printOutputRecordListToFile();
8     stopMeasuringTime();
9     return outputVal;
10 }
11
12 void AlgorithmMassive::getRecordList() {
13     string line;
14     record my_record;
15     ifstream input_file;
16     input_file.open (inputVal.input_file_path);
17     if (input_file.is_open()) {
18         while (getline(input_file, line)) {
19             if (version == 0) {
20                 my_record.line1 = line;
21                 getline (input_file, line);
22                 my_record.line2 = line;
23                 getline (input_file, line);
24                 my_record.line3 = line;
25                 getline (input_file, line);
26                 my_record.line4 = line;
27             } else {
28                 getline (input_file, line);
29                 getline (input_file, line);
30                 my_record.line3 = line;
31                 getline (input_file, line);
32             }
33             input_record_list.push_back(my_record);
34         }
35         input_file.close();
36     }
37 }
38
39 record AlgorithmMassive::getNextRecord(int index) {
40     return input_record_list[index];
41 }
42
43 void AlgorithmMassive::addRecordToOutputFile(record my_record) {
44     output_record_list.push_back(my_record);
45 }
46
47
48 void AlgorithmMassive::printOutputRecordListToFile() {
49     ofstream output_file;
50     outputVal.outputFileName = (version == 0) ? getOutputFileName("full", inputVal)
51         : getOutputFileName("partial", inputVal);
52     string tagVersion = (version == 0) ? "FullVersion" : "PartialVersion";
53     string outputPath = inputVal.output_file_dir + "\\" + directTag + "\\" +
54     tagVersion + "\\OutputDataset\\" + outputVal.outputFileName;
55     output_file.open(outputPath, ios::trunc);
56     if (output_file.is_open()) {
57         for (record& my_record : output_record_list) {
58             if (version == 0) {
59                 output_file << my_record.line1 << endl;
60                 output_file << my_record.line2 << endl;
61                 output_file << my_record.line3 << endl;
62                 output_file << my_record.line4 << endl;
63             } else {
64                 output_file << my_record.line3 << endl;
65             }
66         }
67     }
68 }
```

6.1.1.9 Codice sorgente di run_algorithm.cpp

```
1 #include "run_algorithm.h"
2
3 RunAlgorithm::RunAlgorithm(inputValues input, vector<int> n_values, vector<int>
4 k_values)
5 {
6     this->input = input;
7     this->n_values = n_values;
8     this->k_values = k_values;
9 }
10
11 void RunAlgorithm::run()
12 {
13     checkOutputDictionariesForAllVersions();
14     runAlgorithm();
15     openAndSetOutputFiles();
16     printToCsvFile();
17     closeOutputCsvFile();
18 }
19
20 void RunAlgorithm::checkOutputDictionariesForAllVersions() {
21     checkOrCreateOutputDirectories("Iterate\\FullVersion\\Logs");
22     checkOrCreateOutputDirectories("Iterate\\FullVersion\\Logs");
23     checkOrCreateOutputDirectories("Iterate\\FullVersion\\OutputDataset");
24     checkOrCreateOutputDirectories("Iterate\\PartialVersion\\Logs");
25     checkOrCreateOutputDirectories("Iterate\\PartialVersion\\OutputDataset");
26     checkOrCreateOutputDirectories("Massive\\FullVersion\\Logs");
27     checkOrCreateOutputDirectories("Massive\\FullVersion\\OutputDataset");
28     checkOrCreateOutputDirectories("Massive\\PartialVersion\\Logs");
29     checkOrCreateOutputDirectories("Massive\\PartialVersion\\OutputDataset");
30 }
31
32 void RunAlgorithm::checkOrCreateOutputDirectories(string tag) {
33     string output_path_str = input.output_file_dir + "\\\" + tag;
34     QString output_path = output_path_str.c_str();
35     QDir dir(output_path);
36     if (!dir.exists()) {
37         dir.mkpath(output_path);
38     }
39 }
40
41 void RunAlgorithm::runAlgorithm() {
42     emit progressBarValueChanged(0);
43     int i = 0;
44     for (int k : k_values) {
45         for (int n : n_values) {
46             input.parameter_k = k;
47             input.output_dataset_size = n;
48             AlgorithmIterate *fullIterate = new AlgorithmIterate(input, 0, "Iterate");
49             AlgorithmIterate *partialIterate = new AlgorithmIterate(input, 1,
50 "Iterate");
51             AlgorithmMassive *fullMassive = new AlgorithmMassive(input, 0, "Massive");
52             AlgorithmMassive *partialMassive = new AlgorithmMassive(input, 1,
53 "Massive");
54             out_full_iterate = fullIterate->run();
55             emit testValueChanged(true, true, ++testFullIterate);
56             printOutputLogFile(out_full_iterate, "Iterate", "FullVersion");
57             mapFullIterate[k].push_back(out_full_iterate);
58             emit progressBarValueChanged(++i);
59             out_partial_iterate = partialIterate->run();
60             emit testValueChanged(false, true, ++testPartialIterate);
61             printOutputLogFile(out_partial_iterate, "Iterate", "PartialVersion");
62             mapPartialIterate[k].push_back(out_partial_iterate);
63             emit progressBarValueChanged(++i);
64             out_full_massive = fullMassive->run();
65             emit testValueChanged(true, false, ++testFullMassive);
66             printOutputLogFile(out_full_massive, "Massive", "FullVersion");
67             mapFullMassive[k].push_back(out_full_massive);
68             emit progressBarValueChanged(++i);
69             out_partial_massive = partialMassive->run();
70             emit testValueChanged(false, false, ++testPartialMassive);
71             printOutputLogFile(out_partial_massive, "Massive", "PartialVersion");
72             mapPartialMassive[k].push_back(out_partial_massive);
73             emit progressBarValueChanged(++i);
```

```

71
72
73
74
75 void RunAlgorithm::printOutputLogFile(outputValues out, string dirTag, string
76 fullOrPartial) {
77     ofstream output_log;
78     string outputFileNameLog = getOutputFileLogName(fullOrPartial, input);
79     string outputPath = input.output_file_dir + "\\\" + dirTag + "\\\" +
80     fullOrPartial + "\\Logs\\\" + outputFileNameLog;
81     output_log.open(outputPath, ios::trunc);
82     if (output_log.is_open()) {
83         output_log << "----- Outcome log - " + fullOrPartial + " " + dirTag
84         + "-----" << endl;
85         output_log << "Input dataset path: " + input.input_file_path << endl;
86         output_log << "Parameter k: " + to_string(input.parameter_k) << endl;
87         output_log << "Parameter n: " + to_string(input.output_dataset_size) << endl;
88         output_log << "Output dataset name: " + out.outputFileName << endl;
89         output_log << "Written records to output-dataset: " +
90         to_string(out.written) << endl;
91         output_log << "Wall time spended by the algorithm: " +
92         to_string(out.elapsed_wall) + " seconds" << endl;
93         output_log << "CPU time spended by the algorithm: " +
94         to_string(out.elapsed_CPU) + " seconds" << endl;
95         output_log << endl;
96     }
97 }
98
99 void RunAlgorithm::openAndSetOutputFiles() {
100     string outputPathWall = input.output_file_dir + "\\executionTimesWall.csv";
101     string outputPathCPU = input.output_file_dir + "\\executionTimesCPU.csv";
102     string outputPathRecord = input.output_file_dir + "\\inputRecordSelected.csv";
103     string outputPathParametersChanges = input.output_file_dir +
104     "\\parametersChanges.csv";
105     output_csv_wall.open(outputPathWall, ios::trunc);
106     output_csv_CPU.open(outputPathCPU, ios::trunc);
107     output_csv_record_selected.open(outputPathRecord, ios::trunc);
108     output_csv_parameter_changes.open(outputPathParametersChanges, ios::trunc);
109     prepareOutputCsvFile(output_csv_wall);
110     prepareOutputCsvFile(output_csv_CPU);
111     prepareOutputCsvFile(output_csv_record_selected);
112 }
113
114 void RunAlgorithm::prepareOutputCsvFile(ofstream &csvOut) {
115     csvOut << "Iterate Full Version;";
116     writeEmptyCellsInCsv(n_values.size(), csvOut);
117     csvOut << "Iterate Partial Version;";
118     writeEmptyCellsInCsv(n_values.size(), csvOut);
119     csvOut << "Massive Full Version;";
120     writeEmptyCellsInCsv(n_values.size(), csvOut);
121     csvOut << "Massive Partial Version;";
122     writeEmptyCellsInCsv(n_values.size(), csvOut);
123     csvOut << endl;
124     for (int i = 0; i < 4; i++) {
125         csvOut << " ";
126         for (int n : n_values) {
127             csvOut << "n = " + std::to_string(n) + ",";
128         }
129         csvOut << " ";
130     }
131     csvOut << endl;
132 }
133
134 void RunAlgorithm::writeEmptyCellsInCsv(int number, ofstream &outFile) {
135     for (int i = 0; i <= number; i++) {
136         outFile << " ";
137     }
138 }
139
140 void RunAlgorithm::printToCsvFile() {
141     for (int k : k_values) {
142         printWallCsv(mapFullIterate[k], k);
143         printWallCsv(mapPartialIterate[k], k);
144     }
145 }
```

```

137     printWallCsv(mapFullMassive[k], k);
138     printWallCsv(mapPartialMassive[k], k);
139     printCpuCsv(mapFullIterate[k], k);
140     printCpuCsv(mapPartialIterate[k], k);
141     printCpuCsv(mapFullMassive[k], k);
142     printCpuCsv(mapPartialMassive[k], k);
143     printRecordsCsv(mapFullIterate[k], k);
144     printRecordsCsv(mapPartialIterate[k], k);
145     printRecordsCsv(mapFullMassive[k], k);
146     printRecordsCsv(mapPartialMassive[k], k);
147     getMediaVector(mapFullIterate[k], mapPartialIterate[k], mapFullMassive[k],
148     mapPartialMassive[k]);
149     output_csv_wall << endl;
150     output_csv_CPU << endl;
151     output_csv_record_selected << endl;
152   }
153   printWhileIterationsCsv(parametersChangesVector);
154 }
155 void RunAlgorithm::printWallCsv(vector<outputValues> lista, int k) {
156   output_csv_wall << "k = " + std::to_string(k) + ";";
157   for (const outputValues &out : lista) {
158     output_csv_wall << getFormattedValue(out.elapsed_wall) + ";";
159   }
160   output_csv_wall << " ";
161 }
162 void RunAlgorithm::printCpuCsv(vector<outputValues> lista, int k) {
163   output_csv_CPU << "k = " + std::to_string(k) + ";";
164   for (const outputValues &out : lista) {
165     output_csv_CPU << getFormattedValue(out.elapsed_CPU) + ";";
166   }
167   output_csv_CPU << " ";
168 }
169
170 string RunAlgorithm::getFormattedValue(double d) {
171   string s = std::to_string(d);
172   s.replace(s.find("."), 1, ",");
173   return s;
174 }
175
176 void RunAlgorithm::getMediaVector(vector<outputValues> listal,
177   vector<outputValues> lista2, vector<outputValues> lista3,
178   vector<outputValues> lista4) {
179   for (int p = 0; p < (int) listal.size(); p++) {
180     parametersChangesObject d;
181     d.k = std::to_string(listal[p].k_in);
182     d.n = std::to_string(listal[p].n_in);
183     d.whileIterations = listal[p].whileIterations;
184     for (int i = 1; i <= listal[p].whileIterations+1; i++) {
185       d.m_dec.push_back(getStringMedia(listal[p].m_decrease[i-1],
186         lista2[p].m_decrease[i-1], lista3[p].m_decrease[i-1],
187         lista4[p].m_decrease[i-1]));
188       d.n_dec.push_back(getStringMedia(listal[p].n_decrease[i-1],
189         lista2[p].n_decrease[i-1], lista3[p].n_decrease[i-1],
190         lista4[p].n_decrease[i-1]));
191       d.c_val.push_back(getStringMedia(listal[p].c_values[i-1],
192         lista2[p].c_values[i-1], lista3[p].c_values[i-1],
193         lista4[p].c_values[i-1]));
194       d.duplicated.push_back(getStringMedia(listal[p].duplicated_records[i-1],
195         lista2[p].duplicated_records[i-1], lista3[p].duplicated_records[i-1],
196         lista4[p].duplicated_records[i-1]));
197     }
198     parametersChangesVector.push_back(d);
199   }
200 }

string RunAlgorithm::getStringMedia(int i1, int i2, int i3, int i4) {
  return std::to_string((int)((i1 + i2 + i3 + i4)/4));
}

void RunAlgorithm::printRecordsCsv(vector<outputValues> lista, int k) {
  output_csv_record_selected << "k = " + std::to_string(k) + ";";
  for (const outputValues &out : lista) {

```

```

200     output_csv_record_selected << std::to_string(out.inputRecordSelected) + ";" ;
201 }
202 output_csv_record_selected << " ;";
203 }
204
205 void RunAlgorithm::printWhileIterationsCsv(vector<parametersChangesObject> v) {
206     for (const parametersChangesObject &obj : v) {
207         output_csv_parameter_changes << "n = " + obj.n + ", k = " + obj.k + "; ; ; ; ; ";
208     }
209     output_csv_parameter_changes << endl;
210     for (int i = 0; i < (int) v.size(); i++) {
211         output_csv_parameter_changes << "Iterations:" + ";";
212         output_csv_parameter_changes << "m decrease:" + ";";
213         output_csv_parameter_changes << "n decrease:" + ";";
214         output_csv_parameter_changes << "c values:" + ";";
215         output_csv_parameter_changes << "Duplicated records:" + ";";
216         output_csv_parameter_changes << " ;";
217     }
218     output_csv_parameter_changes << endl;
219     for (int i = 1; i <= input.input_dataset_size; i++) {
220         for (const parametersChangesObject &obj : v) {
221             if (i <= obj.whileIterations+1) {
222                 output_csv_parameter_changes << std::to_string(i) + ";" ;
223                 output_csv_parameter_changes << obj.m_dec[i-1] + ";" ;
224                 output_csv_parameter_changes << obj.n_dec[i-1] + ";" ;
225                 if (i <= obj.whileIterations) {
226                     output_csv_parameter_changes << obj.c_val[i-1] + ";" ;
227                     output_csv_parameter_changes << obj.duplicated[i-1] + ";" ;
228                 } else {
229                     output_csv_parameter_changes << ";;;" ;
230                 }
231             } else {
232                 output_csv_parameter_changes << ";;;;;" ;
233             }
234         }
235         output_csv_parameter_changes << endl;
236     }
237 }
238
239 void RunAlgorithm::closeOutputCsvFile() {
240     output_csv_wall.close();
241     output_csv_CPU.close();
242     output_csv_record_selected.close();
243     output_csv_parameter_changes.close();
244 }
245

```

6.1.2 Codice sorgente delle classi singolari GUI

6.1.2.1 Codice sorgente di mainwindow.h

```
1  ifndef MAINWINDOW_H
2  define MAINWINDOW_H
3
4  #include "common.h"
5  #include "run_algorithm.h"
6
7  using namespace std;
8
9  namespace Ui {
10 class MainWindow;
11 }
12
13
14 class MainWindow : public QMainWindow
15 {
16     Q_OBJECT
17
18 public:
19     explicit MainWindow(QWidget *parent = nullptr);
20     ~MainWindow();
21
22 private slots:
23     void on_run_button_clicked();
24     void on_input_file_field_editingFinished();
25     void on_output_file_field_editingFinished();
26     void on_parameter_n_field_editingFinished();
27     void on_parameter_k_field_editingFinished();
28     void clearOutput();
29     bool splitStringByComma(string stringToSplit, vector<int> &v);
30     int getFileSize(string fileName);
31     void setCsvFilesName();
32     void connectSignals(RunAlgorithm *run);
33     void updateTestsValue(bool isFull, bool isIterate, int value);
34     void getMessageBox(string message, string title);
35     void getMessageBox(string message);
36
37 private:
38     Ui::MainWindow *ui;
39     inputValues input;
40     vector<int> n_values;
41     vector<int> k_values;
42 };
43
44 #endif // MAINWINDOW_H
45
```

6.1.2.2 Codice sorgente di mainwindow.cpp

```
1 #include "mainwindow.h"
2 #include "ui_mainwindow.h"
3
4 MainWindow::MainWindow(QWidget *parent) :
5     QMainWindow(parent),
6     ui(new Ui::MainWindow)
7 {
8     ui->setupUi(this);
9     ui->input_file_check->setVisible(false);
10    ui->output_file_check->setVisible(false);
11    ui->parameter_n_check->setVisible(false);
12    ui->parameter_k_check->setVisible(false);
13    ui->run_button->setStyleSheet("background-color: #919191; color : white");
14 }
15
16 MainWindow::~MainWindow()
17 {
18     delete ui;
19 }
20
21 void MainWindow::on_input_file_field_editingFinished()
22 {
23     input.input_file_path = "";
24     string inputValue = ui->input_file_field->text().toStdString();
25     if (inputValue.empty()) {
26         ui->input_file_check->setVisible(false);
27     } else if (inputValue.compare(input.input_file_path) != 0) {
28         const char *input_path = inputValue.c_str();
29         int flag = access(input_path, F_OK);
30         if (flag == -1) {
31             ui->input_file_check->setVisible(false);
32             ui->input_file_field->clear();
33             getMessageBox("The inserted path is not a valid path; retry!");
34             return;
35         }
36         struct stat sb;
37         if(stat(input_path, &sb) == 0 && !S_ISREG(sb.st_mode)) {
38             ui->input_file_check->setVisible(false);
39             ui->input_file_field->clear();
40             getMessageBox("The inserted path doesn't match with a regular file;
41             retry!");
42             return;
43         }
44         input.input_file_path = inputValue;
45         ui->input_file_check->setVisible(true);
46         input.input_dataset_size = getFileSize(inputValue);
47         if (input.output_dataset_size != -1 && input.input_dataset_size <
48             input.output_dataset_size) {
49             ui->parameter_n_field->clear();
50             input.output_dataset_size = -1;
51         }
52     }
53 }
54
55 void MainWindow::on_output_file_field_editingFinished()
56 {
57     input.output_file_dir = "";
58     string inputValue = ui->output_file_field->text().toStdString();
59     if (inputValue.empty()) {
60         ui->output_file_check->setVisible(false);
61     } else if (!inputValue.empty() && inputValue.compare(input.output_file_dir) !=
62     0) {
63         const char *output_path = inputValue.c_str();
64         int flag = access(output_path, F_OK);
65         struct stat sb;
66         if (flag == -1) {
67             ui->output_file_check->setVisible(false);
68             ui->output_file_field->clear();
69             getMessageBox("The inserted path is not a valid path; retry!");
70             return;
71         }
72         if(flag == 0 && stat(output_path, &sb) == 0 && !S_ISDIR(sb.st_mode)) {
73             ui->output_file_check->setVisible(false);
```

```

71         ui->output_file_field->clear();
72         getMessageBox("The inserted path doesn't match with a directory; retry!");
73         return;
74     }
75     input.output_file_dir = inputValue;
76     ui->output_file_check->setVisible(true);
77 }
78
79 void MainWindow::on_parameter_n_field_editingFinished()
80 {
81     n_values.clear();
82     input.output_dataset_size = -1;
83     string inputValue = ui->parameter_n_field->text().toStdString();
84     if (inputValue.empty()) {
85         ui->parameter_n_check->setVisible(false);
86     } else if (!inputValue.empty() &&
87         inputValue.compare(to_string(input.output_dataset_size)) != 0) {
88         if (!splitStringByComma(inputValue, n_values)) {
89             ui->parameter_n_check->setVisible(false);
90             ui->parameter_n_field->clear();
91             return;
92         }
93         ui->parameter_n_check->setVisible(true);
94     }
95 }
96
97 void MainWindow::on_parameter_k_field_editingFinished()
98 {
99     k_values.clear();
100    input.parameter_k = -1;
101    string inputValue = ui->parameter_k_field->text().toStdString();
102    if (inputValue.empty()) {
103        ui->parameter_k_check->setVisible(false);
104    } else if (!inputValue.empty() &&
105        inputValue.compare(to_string(input.parameter_k)) != 0) {
106        if (!splitStringByComma(inputValue, k_values)) {
107            ui->parameter_k_check->setVisible(false);
108            ui->parameter_k_field->clear();
109            return;
110        }
111        ui->parameter_k_check->setVisible(true);
112    }
113 }
114 int MainWindow::getFileSize(string filename) {
115     string line;
116     ifstream input_file;
117     input_file.open (filename);
118     int length = 0;
119     while (getline(input_file, line)) {
120         length++;
121     }
122     input_file.close();
123     return length/4;
124 }
125
126 void MainWindow::on_run_button_clicked()
127 {
128     clearOutput();
129     if (input.input_file_path.empty()) {
130         getMessageBox("Insert the desired input-file path!");
131         return;
132     }
133     if (input.output_file_dir.empty()) {
134         getMessageBox("Insert the desired output-file directory!");
135         return;
136     }
137     if (n_values.empty()) {
138         getMessageBox("Insert the desired sample size (parameter n)!");
139         return;
140     }
141     if (k_values.empty()) {

```

```

142     getMessageBox("Insert the desired parameter k!");
143     return;
144 }
145 QApplication :: setOverrideCursor (Qt::BusyCursor);
146 setCsvFileName();
147 RunAlgorithm *run = new RunAlgorithm(input, n_values, k_values);
148 connectSignals(run);
149 run->run();
150 QApplication :: restoreOverrideCursor();
151 getMessageBox("Process completed successfully", "Success!");
152 }
153
154 void MainWindow::clearOutput() {
155     ui->csv_name_partial->clear();
156     ui->csv_name_wall->clear();
157     ui->csv_name_p_changes->clear();
158     ui->csv_name_r_selected->clear();
159
160     ui->test_f_it->clear();
161     ui->test_f_m->clear();
162     ui->test_p_it->clear();
163     ui->test_p_m->clear();
164     ui->test_tot->clear();
165 }
166
167 bool MainWindow::splitStringByComma(string stringToSplit, vector<int> &v) {
168     if (!regex_match(stringToSplit,
169         regex("[1-9]+[0-9]*[,][\\s]*([1-9]+[0-9]*[,]?[\\s]*)?")) {
170         getMessageBox("The inserted values are not in correct format; retry!");
171         return false;
172     }
173     string delimiter = ",";
174     int start = 0;
175     int end = stringToSplit.find(delimiter);
176     while (end != -1) {
177         int t = stoi(stringToSplit.substr(start, end - start));
178         if (t > input.input_dataset_size || t < 1) {
179             getMessageBox("The parameter must be between 1 and input-dataset size;
180             retry");
181             return false;
182         }
183         start = end + delimiter.size();
184         end = stringToSplit.find(delimiter, start);
185         v.push_back(t);
186     }
187     if (!stringToSplit.substr(start, end - start).empty()) {
188         int t = stoi(stringToSplit.substr(start, end - start));
189         if (t > input.input_dataset_size || t < 1) {
190             getMessageBox("The parameter must be between 1 and input-dataset size;
191             retry");
192             return false;
193         }
194         v.push_back(t);
195     }
196     return true;
197 }
198
199 void MainWindow::updateTestsValue(bool isFull, bool isIterate, int value) {
200     if (isFull && isIterate) {
201         ui->test_f_it->setText(QString::fromStdString(to_string(value)));
202     } else if (isFull && !isIterate) {
203         ui->test_f_m->setText(QString::fromStdString(to_string(value)));
204     } else if (!isFull && isIterate) {
205         ui->test_p_it->setText(QString::fromStdString(to_string(value)));
206     } else if (!isFull && !isIterate) {
207         ui->test_p_m->setText(QString::fromStdString(to_string(value)));
208     }
209     int tot_test = (ui->test_tot->text().isEmpty()) ? 0 :
210     stoi((ui->test_tot->text()).toStdString());
211     ui->test_tot->setText(QString::fromStdString(to_string(tot_test + 1)));
212     qApp->processEvents();
213 }

```

```

211 void MainWindow::setCsvFilesName() {
212     ui->csv_name_wall->setText(QString::fromStdString("executionTimesWall.csv"));
213     ui->csv_name_partial->setText(QString::fromStdString("executionTimesCPU.csv"));
214
215     ui->csv_name_r_selected->setText(QString::fromStdString("inputRecordSelected.csv"))
216     );
217     ui->csv_name_p_changes->setText(QString::fromStdString("parametersChanges.csv"));
218     qApp->processEvents();
219 }
220
221 void MainWindow::connectSignals(RunAlgorithm *run) {
222     ui->progressBar->setRange(0, n_values.size()*k_values.size()*4);
223     connect(run, SIGNAL(testValueChanged(bool, bool, int)), this,
224             SLOT(updateTestsValue(bool, bool, int)));
225     connect(run, SIGNAL(progressBarValueChanged(int)), ui->progressBar,
226             SLOT(setValue(int)));
227 }
228
229 void MainWindow::getMessageBox(string message, string title) {
230     QMessageBox msgBox;
231     msgBox.setWindowTitle(title.c_str());
232     string text = "<p align='center'>" + title + "!</p>";
233     msgBox.setText(text.c_str());
234     message = "<p align='center'>" + message + "</p>";
235     msgBox.setInformativeText(message.c_str());
236     msgBox.setFont(QFont ("MS Shell Dlg 2", 11));
237     msgBox.setStandardButtons(QMessageBox::Ok);
238     msgBox.setDefaultButton(QMessageBox::Ok);
239     msgBox.setStyleSheet("QLabel{min-width:400 px;}");
240     msgBox.button(QMessageBox::Ok)->setStyleSheet("background-color: #919191; color
241 : white");
242     msgBox.exec();
243 }
244
245 void MainWindow::getMessageBox(string message) {
246     getMessageBox(message, "Warning!");
247 }
248
249 }
```

6.1.2.3 Codice sorgente di main.cpp

```

1 #include "mainwindow.h"
2
3 #include <QApplication>
4 #include <QScreen>
5 #include <QDesktopWidget>
6
7 int main(int argc, char *argv[])
8 {
9     QApplication a(argc, argv);
10    MainWindow m;
11    m.setFixedSize(m.width(), m.height());
12
13    QRect rec = QGuiApplication::screenAt(m.pos())->geometry();
14    QSize size = m.size();
15    QPoint topLeft = QPoint((rec.width() / 2) - (size.width() / 2),
16                           (rec.height() / 2) - (size.height() / 2));
17    m.setGeometry(QRect(topLeft, size));
18
19    m.showNormal();
20    a.exec();
21    return 0;
22 }
```

6.1.3 Codice sorgente delle classi singolari CLI

6.1.3.1 Codice sorgente di main.cpp

```
1 #include "common.h"
2 #include "run_algorithm.h"
3
4 using namespace std;
5
6 void printAlgorithmDescription();
7 void setInputParameters();
8 void inputFilePath();
9 void outputFilePath();
10 void parameterN();
11 void parameterK();
12 string splitStringByComma(string stringToSplit, vector<int> &v);
13 int getFileSize(string fileName);
14
15 inputValues input;
16 vector<int> n_values;
17 vector<int> k_values;
18
19 int main () {
20     printAlgorithmDescription();
21     setInputParameters();
22     RunAlgorithm *run = new RunAlgorithm(input, n_values, k_values);
23     run->run();
24     printf("\nProcess completed successfully\n");
25     system("pause");
26     printf("\nApplication stopped.\n");
27     return 0;
28 }
29
30 void printAlgorithmDescription() {
31     printf("Input:\n");
32     printf(" - dataset D with |D| = m reads of 4 lines (only 3^ is
33 significant)\n");
34     printf(" - sample size n\n");
35     printf(" - parameter k ∈ [1, m]\n");
36
37     printf("Output:\n");
38     printf(" - file with n reads of D chosen at random uniformly with
39 replacement\n");
39     printf(" - log file with execution times\n");
40
41     printf("Version:\n");
42     printf(" - Full : in the output dataset are written all four lines\n");
43     printf(" - Partial : in the output dataset is written only significant line\n");
44 }
45
46 void setInputParameters() {
47     inputFilePath();
48     input.input_dataset_size = getFileSize(input.input_file_path);
49     outputFilePath();
50     parameterN();
51     parameterK();
52 }
53
54 void inputFilePath() {
55     bool b = true;
56     while (b) {
57         printf("\nEnter the desired input-file path:\n");
58         cin >> input.input_file_path;
59         const char *input_path = input.input_file_path.c_str();
60         int flag = access(input_path, 00);
61         if (flag == -1) {
62             printf("The inserted path is not a valid path; retry!\n");
63             cin.clear();
64             continue;
65         }
66         struct stat sb;
67         if(stat(input_path, &sb) == 0 && !(((sb.st_mode) & 0xF000) == 0x8000)) {
68             printf("The inserted path doesn't match with a regular file; retry!\n");
69             cin.clear();
70             continue;
71         }
72         b = false;
73 }
```

```

72         cin.clear();
73     }
74 }
75
76 void outputPath() {
77     bool b = true;
78     while (b) {
79         printf("\nInsert the desired output-file path\n");
80         string out;
81         cin >> input.output_file_dir;
82         const char *output_path = input.output_file_dir.c_str();
83         int flag = access(output_path, 00);
84         struct stat sb;
85         if (flag == -1) {
86             printf("The inserted path is not a valid path; retry!");
87             cin.clear();
88             continue;
89         }
90         if(flag == 0 && stat(output_path, &sb) == 0 && !(((sb.st_mode) & 0xF000) ==
91             0x4000)) {
92             printf("The inserted path doesn't match with a directory; retry!");
93             cin.clear();
94             continue;
95         }
96         b = false;
97     }
98 }
99
100 void parameterN() {
101     bool b = true;
102     while (b) {
103         printf("\nInsert the desired parameters n (separated by semicolon):\n");
104         string tmp;
105         cin >> tmp;
106         string error = splitStringByComma(tmp, n_values);
107         if (!error.empty()) {
108             printf(error.c_str());
109             cin.clear();
110             continue;
111         }
112         b = false;
113     }
114 }
115
116
117 void parameterK() {
118     bool b = true;
119     while (b) {
120         printf("\nInsert the desired parameters k (separated by semicolon):\n");
121         string tmp;
122         cin >> tmp;
123         string error = splitStringByComma(tmp, k_values);
124         if (!error.empty()) {
125             printf(error.c_str());
126             cin.clear();
127             continue;
128         }
129         b = false;
130     }
131 }
132
133
134 int getFileSize(string filename) {
135     string line;
136     ifstream input_file;
137     input_file.open (filename);
138     int length = 0;
139     while (getline(input_file, line)) {
140         length++;
141     }
142     input_file.close();
143     return length/4;

```

```

144     }
145
146     string splitStringByComma(string stringToSplit, vector<int> &v) {
147         if (!regex_match(stringToSplit,
148             regex("([1-9]+[0-9]*[;](\\s*)*)*([1-9]+[0-9]*[;]?\\s*)?"));
149         )
150             return "The inserted values are not in correct format; retry!";
151
152         string delimiter = ";";
153         int start = 0;
154         int end = stringToSplit.find(delimiter);
155         while (end != -1) {
156             int t = stoi(stringToSplit.substr(start, end - start));
157             if (t > input.input_dataset_size || t < 1) {
158                 return "The parameter must be between 1 and input-dataset size; retry";
159             }
160             start = end + delimiter.size();
161             end = stringToSplit.find(delimiter, start);
162             v.push_back(t);
163         }
164         if (!stringToSplit.substr(start, end - start).empty()) {
165             int t = stoi(stringToSplit.substr(start, end - start));
166             if (t > input.input_dataset_size || t < 1) {
167                 return "The parameter must be between 1 and input-dataset size; retry";
168             }
169             v.push_back(t);
170         }
171     }

```