LROD: APPROFONDIMENTO

Silvio Venturino
s.venturino@studenti.unisa.it
Università degli studi di Salerno
Salerno, Italia

Catello Staiano
c.staiano14@studenti.unisa.it
Università degli studi di Salerno
Salerno, Italia

KEYWORDS

Overlap detection, aligning, long reads, solid k-mers, the third generation sequencing technology (TGS), algorithms, github, frequency, recall, precision, F1-score, C++, Hash table.

ACM Reference Format:
Silvio Venturino and Catello Staiano. 2018. LROD: APPROFONDIMENTO. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

1.1 Il problema

Le tecnologie di sequenziamento di terza generazione frammentano il genoma in un gran numero di long reads e il processo di ricombinazione di queste letture in una sequenza di DNA completa è chiamato assemblaggio del genoma [7]. L'overlap detection tra due long reads è utile per il processo di assemblaggio del genoma [8], allineando e unendo i frammenti di DNA in un'unica sequenza continua. A differenza delle short reads, le long reads permettono un assemblaggio più accurato, evitando regioni ripetute e regioni più complesse. Tuttavia l'alto numero di errori che derivano dal sequenziamento di terza generazione (TGS) implica che ottenere l'overlap detection è ancora un compito impegnativo.

1.2 Obiettivo di LROD

In questo studio gli autori presentano un algoritmo di overlap detection a lettura lunga (LROD) che può migliorare l'accuratezza dei risultati delle sovrapposizioni tra long reads. L'obiettivo è quello di trovare sovrapposizioni in base alla distribuzione dei k-mer, ovvero delle sottostringhe del genoma di lunghezza k.

1.3 LROD

Diversamente da altri algoritmi che utilizzano i k-mer, LROD conserva innanzitutto solo i k-mer solidi comuni tra le long reads che, rispetto ai k-mers tradizionali, hanno una frequenza più elevata. Dato che le tecniche TGS hanno un alto tasso di errore e le regioni ripetitive complicano il processo di overlap detection, LROD tenta di risolvere il problema sfruttando i solid k-mers. LROD utilizza una strategia che si adopera in varie fasi:

(1) innanzitutto trova l'insieme di k-mers comuni solidi;
(2) in secondo luogo trova una catena che include i k-mers comuni consistenti. La consistenza della catena è determinata da alcune condizioni che vedremo durante lo studio degli algoritmi;
(3) infine, tramite la catena, valuta la regione di overlap candidata e la restituisce.

1.4 Risultati ottenuti

Nella prima parte del paper, nelle sezioni 2 e 3, analizzeremo il lavoro svolto dai ricercatori di LROD, con i risultati ottenuti dagli stessi e le conclusioni. Nella seconda parte del paper approfondiremo l'implementazione dell'algoritmo e i punti che secondo noi sono critici per le prestazioni di LROD. I risultati ottenuti dai ricercatori affermano che LROD vince in quasi tutti i casi in termini di *recall*, *precision* e *F1-score*, rispetto a MHAP [1] e Minimap2 [4]; MHAP risulta il peggiore in ogni ambito. Rispetto a Minimap2, LROD risulta essere più oneroso in termini di utilizzo di memoria e tempo di esecuzione. Come affermato dal team stesso "Sebbene LROD funzioni bene, secondo i risultati sperimentali, ha un'ovvia carenza in termini di tempo di esecuzione. In futuro ci concentreremo sul miglioramento del modulo delle prestazioni di calcolo di LROD, aumentando la velocità di calcolo e riducendo il tempo di esecuzione". Detto ciò, nella parte finale del paper traiamo le nostre conclusioni ed evidenziamo i punti critici dell'implementazione fornita che fanno calare drasticamente le prestazioni di tempo di esecuzione e utilizzo della memoria.

1.5 Obiettivo del paper

Dal canto nostro, nella sezione 4 spiegheremo l'implementazione dell'algoritmo e tutta la fase di preprocessing delle strutture che lo stesso utilizzerà. Infine analizzeremo l'algoritmo e i suoi punti critici che in seguito potranno essere migliorati.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

2 LROD: OVERLAP DETECTION TRA DUE LONG READS

LROD utilizza i k-mers comuni tra due reads per determinare se esiste una regione di overlap. Tuttavia, l'elevato tasso di errore di sequenziamento del TGS di solito porta a k-mers comuni negativi e le regioni ripetitive possono causare una contraddizione di posizione tra i k-mers comuni [6]. Per un dataset di long reads, un k-mer con una piccola frequenza include comunemente errori di sequenziamento, mentre un k-mer con una grande frequenza solitamente proviene da una regione ripetitiva [5]. Pertanto, LROD seleziona solo k-mers le cui frequenze sono nell'intervallo [f_{min}, f_{max}] come k-mers solidi, dove f_{min} e f_{max} sono due soglie calcolate da LROD. L'utilizzo solo di k-mers solidi consente a LROD di evitare alcuni problemi causati da errori di sequenziamento e regioni ripetitive. Dopo aver determinato l'intervallo, i k-mers le cui frequenze non rientrano in esso vengono ignorati nei passaggi successivi. Per un dataset di long reads, LROD utilizza innanzitutto DSK [9], un programma di conteggio dei k-mers, per calcolare la frequenza di ciascun k-mer nel dataset. Se la frequenza di un k-mer è uguale a uno, allora solo una lettura contiene questo k-mer, il che significa che è inutile per trovare eventuali sovrapposizioni tra due long reads. Per LROD, la frequenza minima del k-mer è 2, ovvero f_{min} = 2 per impostazione predefinita. LROD sviluppa un metodo per calcolare f_{max} in base alla frequenza dei k-mers. F(x) si riferisce al numero di k-mer la cui frequenza è x, (x = 1, 2, 3. . . , h, dove h è la frequenza massima del k-mer). Ad esempio, esiste un insieme di k-mers {AAT, ATA, TAG, TAG, AGT, ATA, AAT, AGT, AGT}. Per questo insieme di k-mers, F(1) = 0 in quanto nessun k-mer appare una volta. F(2) = 3, il che significa che tre k-mer compaiono due volte, ovvero "AAT, ATA, TAG". Infine F(3) = 1, il che significa che solo un k-mer, cioè "AGT", viene ripetuto tre volte. Quindi, S(y) viene utilizzato per calcolare la somma cumulativa di F(x), come descritto nell'equazione 1. Quando f è il più piccolo valore tale che $S(f) > \theta * S(h)$, settiamo f_{max} = f, mentre θ = 0.9 di default; S(h) è la frequenza totale dei k-mers i cui valori delle frequenze non sono inferiori a 2.

$$\sum_{x=f_{min}}^{y} F(x) \tag{1}$$

- S(y) indica la somma cumulativa delle frequenze dei k-mers fino ad una specifica frequenza "y".
- Si calcola sommando i valori di F(x) per tutte le frequenze dalla frequenza minima (f_{min}) fino a "y."
- In parole semplici, S(y) fornisce il numero totale di k-mers che appaiono un numero di volte minore o uguale a "y" nel dataset.

Consideriamo il seguente esempio: supponendo di avere l'insieme di kmers {AAT, ATA, TAG, TAG, AGT, ATA, AAT, AGT, AGT}, il calcolo di f_{max} avviene in quattro passi:

(1) Calcolo di F(x) per tutti i valori di x:
   - F(1) = 0
   - F(2) = 3
   - F(3) = 1

(2) Calcolo della somma cumulativa di S(y) per tutti i valori di y:
   - S(1) = F(1) = 0
   - S(2) = F(1) + F(2) = 0 + 3 = 3
   - S(3) = F(1) + F(2) + F(3) = 0 + 3 + 1 = 4

(3) Calcolo di S(h), la frequenza totale di k-mers con frequenza non inferiore a 2:
   - S(h) = F(2) + F(3) = 3 + 1 = 4

(4) Calcolo di f_{max}:
   Quando f è il più piccolo valore tale che S(f) > θ * S(h), settiamo f_{max} = f. (θ è la soglia impostata a 0.9 per impostazione predefinita). Sostituendo i valori nella formula, S(f) > 0.9 * 4, cioè S(f) > 3.6; quindi:
   - Se f = 1, S(1) = 0 < 3.6
   - Se f = 2, S(2) = 3 < 3.6
   - Se f = 3, S(3) = 4 > 3.6

Siccome f = 3 è il più piccolo valore tale che S(f) > θ * S(h), settiamo f_{max} = 3. In conclusione l'intervallo [f_{min}, f_{max}] è [2,3]. Dopo aver determinato l'intervallo [f_{min}, f_{max}], i k-mers le cui frequenze non rientrano in questo intervallo vengono ignorati nei passaggi successivi. L'overlap detection utilizzando solo k-mers solidi può ridurre al minimo l'impatto degli errori di sequenziamento e delle regioni ripetitive e migliorare l'accuratezza dei risultati [6]. I rimanenti k-mers solidi vengono indicizzati utilizzando una tabella hash con i k-mers come chiavi. Per un k-mer specifico, la tabella hash consente a LROD di identificare rapidamente le long reads che lo includono. Per due long reads, LROD utilizza l'algoritmo 1 per rilevare eventuali sovrapposizioni tra di loro; LROD trova innanzitutto l'insieme di k-mers comuni (*CKS*) tra due long reads R_1 e R_2. In secondo luogo, sulla base del *CKS*, LROD tenta di cercare una catena di k-mers comuni consistenti, che corrispondano a una sovrapposizione candidata. In terzo luogo, LROD valuta ulteriormente il candidato e determina la sovrapposizione finale. L'algoritmo 2 mostra lo pseudocodice del concatenamento. L'algoritmo 3 viene utilizzato per verificare se due k-mers sono consistenti chiamando l'algoritmo 4 per valutare se vengono rispettate quattro condizioni. Se queste ultime non vengono rispettate, viene chiamato l'algoritmo 5 che valuta la consistenza di due k_s-mer con k_s < k. L'algoritmo 6 viene utilizzato per la valutazione del candidato per la regione di overlap; in particolare vengono valutate tre condizioni. Maggiori dettagli riguardo queste condizioni, saranno forniti nei paragrafi successivi. Infine LROD restitusce le regioni di overlap.

2.1 Algoritmo 1

L'algoritmo 1 è il principale che invoca tutti gli altri e infine determina la regione di sovrapposizione tra le long reads; rappresenta il "main" dell'algoritmo. Esso seleziona due long reads R_1 e R_2 per rilevare se si sovrappongono. In caso affermativo LROD fornirà la regione in R_1 che si sovrappone ad un'altra regione in R_2. I passi principali di questo algoritmo sono:

(1) trovare l'insieme di k-mers comuni (CKS) tra R1 e R2;
(2) rimuovere i k-mers forward o reverse, mantenendo quelli presenti in numero maggiore e ignorando gli altri;
(3) ordinare i k-mers comuni;
(4) creare una catena di k-mers comuni consistenti;
(5) determinare la regione di overlap tramite la catena, e se presente restituirla, altrimenti restituire NULL.

Algorithm 1: Finding_overlap_region (R_1, R_2, k, k_s, α, β, γ, ε)

```
Input: two long reads
Output: determines the final overlap
Begin
    Finding the common k-mer set CKS between R1
     and R2;
    Removing forward or reverse common k-mers
     from CKS;
    Sorting common k-mers in CKS;
    m ← |CKS|;
    count ← 5;
    if m < count then
        return NULL;
    end if
    i ← 0;
    while i < m do
        if CKS[i] is not visited then
            CKS[i] is visited;
            chain ← Chaining_from_start (CKS, i, k, ks,
             α, β, γ);
            if chain != NULL then
                all common k-mers in the chain are
                 visited;
                region ← Evaluate_candidate
                  overlap_region
                (CKS, chain, α, γ, ε);
                if region != NULL then
                    return region;
                else
                    i++;
                end if
            else
                i++;
            end if
        end if
    end while
    return NULL;
End
```

LROD prende in input, oltre alle due long reads R_1 e R_2, i parametri k, k_s, α, β, γ, ϵ.

- k indica la lunghezza dei k-mers presi in considerazione. Un valore maggiore di k aiuta a risolvere i problemi legati alla ripetizione, ma diminuisce il numero di k-mers comuni tra due long reads. Un valore più piccolo introdurrà più k-mers comuni negativi e complicherà il processo di overlap detection; k=15 per impostazione predefinita.
- k_s rappresenta la lunghezza dei k-mers più piccoli considerati durante l'analisi. Si riferisce alla dimensione delle sottosequenze più brevi prese in considerazione. L'utilizzo di "k_s" consente di esaminare dettagli specifici o variazioni localizzate all'interno delle sequenze, consentendo un'analisi più dettagliata dei k-mers comuni (k_s = 9 di default).
- α rappresenta la soglia della distanza tra due k-mers comuni. Questo parametro è utilizzato per valutare se due k-mers sono abbastanza vicini l'uno all'altro nelle sequenze per essere considerati consistenti. Una distanza maggiore di α potrebbe indicare una discrepanza tra i k-mers e potrebbe influenzare la loro considerazione come consistenti. È importante impostare α in modo da bilanciare l'accettazione di k-mers distanti e la riduzione di falsi positivi. α assume il valore 400 di default.
- β è un parametro utilizzato per limitare la ricerca di k-mers comuni nella seconda fase di valutazione della consistenza tra i k-mers. Viene utilizzato per definire una distanza massima tra due k-mers comuni al fine di evitare ricerche troppo estese e limitare il tempo computazionale richiesto per l'analisi. β assume il valore 1500 di default.
- γ è utilizzato per valutare la differenza massima tra le distanze D1 e D2 tra due k-mers comuni. Questo parametro è importante per stabilire quanto due k-mers devono essere simili tra loro in termini di posizione nelle sequenze per essere considerati consistenti. Una differenza maggiore di γ potrebbe indicare una discrepanza tra i k-mers che potrebbe influenzare l'essere consistenti o meno. γ assume il valore 0.3 di default.
- ϵ è la soglia minima di lunghezza della sovrapposizione (500 per impostazione predefinita).

Innanzitutto, LROD estrae tutti i k-mers da R_1 spostandosi a destra di una posizione e seleziona i k-mers che appaiono in R_2 (attraverso la tabella hash dei k-mers). In altre parole, LROD estrae i k-mers da R_1 e controlla se ciascun k-mer è presente anche in R_2. Questi k-mers comuni vengono salvati in CKS. Se un k-mer in una long read appare due o più volte in un'altra long read, LROD lo elimina dal *CKS*. I restanti k-mers comuni nel *CKS* sono ordinati in ordine crescente in base alla loro posizione in R_1. LROD utilizza M per rappresentare il numero di k-mers comuni positivi (in forward), cioè stessa direzione in entrambe le sequenze e N per indicare il numero di k-mers comuni opposti (in reverse), ossia orientamenti opposti nelle due sequenze di riferimento. Se M > N e M > count (count = 5), LROD mantiene i k-mers comuni positivi e ignora i k-mers comuni opposti. Se N > M e N > count, LROD mantiene i k-mers opposti e ignora quelli positivi. Count = 5 è il numero minimo di k-mers forward o reverse che devono essere presenti nel *CKS* affinchè sia possibile costruire una catena di k-mers consistenti.

Infatti se il valore maggiore tra M ed N risulta essere minore di *count*, l'algoritmo restituisce *null*; questo significa che il numero di k-mers presenti in CKS non è sufficiente per costruire una catena. In caso contrario, un ciclo *while* itera su questo insieme e man mano che scorre i k-mers, contrassegna quelli visitati. A questo punto l'algoritmo 1 chiama l'algoritmo 2 passandogli l'indice i-esimo da cui far partire la catena; infatti il k-mer con l'indice i viene sempre aggiunto alla catena e a partire da questo si valuta se il successivo è consistente. Quindi ad ogni chiamata all'algoritmo 2 vengono costruite diverse catene fino a quando non si riesce a costruire una catena che abbia almeno tre k-mers e che ci consenta di determinare la regione di overlap finale. Ad ogni chiamata l'algoritmo 2 restituisce la catena di k-mers consistenti costruita. Se la catena non è vuota, tutti i k-mers comuni nella catena sicuramente sono stati visitati. Successivamente viene chiamato l'algoritmo 6 che valuta il candidato per la regione di overlap attarverso tre condizioni. Se vengono rispettate, restituisce la regione, altrimenti restituisce *null*. Alla successiva invocazione dell'algoritmo 2, la costruzione della catena partirà dal kmer successivo. Infatti l'indice i viene incrementato di una posizione e sarà il nuovo punto di partenza della catena. Fino a quando non si ottiene la regione di overlap, l'indice i sarà incrementato, diventando di volta in volta il punto di inizio per la costruzione della nuova catena. Una volta ottenuta la regione, viene restituita e l'algoritmo termina. Se si esce dal ciclo *while* senza aver ottenuto la catena, l'algoritmo 1 restituisce *null*.

2.2 Algoritmo 2

Il processo di chaining è una fase cruciale dell'algoritmo LROD, poiché mira a trovare una catena dall'insieme di common k-mers che consista di alcuni k-mers consistenti e che corrisponda ad una sovrapposizione candidata tra due long reads R_1 e R_2. Questo algoritmo prende in input *CKS*, *start*, k, k_s, α, β e γ.

- *CKS* è l'insieme di k-mers comuni.
- *start* rappresenta l'indice del k-mer nell'insieme *CKS* da cui partire ad ogni chiamata per determinare la catena di consistenza. Nell'algoritmo 1, infatti, questo indice viene incrementato di una posizione finchè non si riesce a creare una catena che ci permetta di determinare la regione di overlap.

α, β e γ rappresentano i parametri di consistenza che i k-mers della catena devono rispettare. La catena di k-mers consistenti, ad ogni chiamata, viene inizializzata a NULL, quindi non è presente nessun k-mer nella catena. Ci sono due indici per lavorare sulla catena:

- *start* è il primo k-mer da cui partire e viene sempre aggiunto alla catena;
- *end* è il k-mer successivo a *start* e bisogna valutarne l'aggiunta o meno alla catena;

Innanzitutto, il k-mer comune iniziale CKS[start] viene aggiunto alla catena. Quindi, LROD cerca il primo k-mer comune successivo, che sia consistente con il k-mer comune precedente nella catena. Per valutare ciò, inizia un ciclo *while* all'interno del quale viene invocato l'algoritmo 3 che restituisce *true* in caso affermativo e *false* in caso negativo. In quest'ultimo caso, si passa al k-mer successivo (*continue*); quindi l'*end* viene incrementato di una posizione e lo *start* rimane invariato poichè lo scopo è proprio quello di creare una catena di k-mers che siano consistenti con l'ultimo kmer appena aggiunto in questa catena. In caso contrario, se i kmer analizzati

Algorithm 2: Chaining_from_start (CKS, start, k, k_s, α,β, γ)

Input: the starting common k-mer
Output: find a chain which consists of some consistent common k-mers
Begin
    $m \leftarrow$ |CKS|;
    end $\leftarrow$ start + 1;
    chain $\leftarrow$ NULL;
    Adding CKS[start] to the chain;
    while start $<$ m and end $<=$ m **do**
        result $\leftarrow$ Determine_consistent (CKS[start], CKS[end], k, k_s, α,β, γ);
        if result != true **then**
            end++;
            continue;
        end if
        Adding CKS[end] to the chain;
        start $\leftarrow$ end;
        end $\leftarrow$ end + 1;
    end while
    if |chain| $>$ 2 **then**
        return chain;
    else
        return NULL;
    end if
End

soddisfano i parametri di consistenza, CKS[end] viene aggiunto alla catena in quanto consistente con *CKS*[*start*]. Successivamente vengono aggiornati gli indici:

- *start* $\leftarrow$ *end*;
- *end* $\leftarrow$ *end*+1;

LROD ripete questo processo finché non vengono visitati tutti i k-mers comuni; infine, LROD ottiene una catena. La questione più importante nel trovare una catena è come decidere se due k-mers comuni sono consistenti. Per due k-mers comuni, è possibile calcolare le loro **distanze** nelle due long reads. Quando le due distanze sono grandi o differiscono troppo, i due k-mers comuni potrebbero essere inconsistenti.

2.3 Algoritmo 3

L'algoritmo 3 *Determine_consistent* invoca *Determine_consistent_1* per vedere se due k-mers sono consistenti:

Algorithm 3: Determine_consistent (CKS[start], CKS[end], k, k_s, α,β,γ)

Input: common k-mers
Output: whether two common k-mers are consistent
Begin:
  if determine_consistent_1(CKS[start], CKS[end], α, γ) != true **then**
    return determine_consistent_2(CKS[i], CKS[j], k, k_s, β);
  endif
  return true;
End

Nell'algoritmo 4 LROD presenta alcune condizioni da valutare. Se non possono essere determinate in tale circostanza, LROD invoca l'algoritmo 5 per analizzare ulteriormente la loro consistenza basandosi su più piccoli k_s-mers (k_s < k). Questo algoritmo prende in input *CKS*[*start*], *CKS*[*end*], k, k_s, α, β e γ.

- *CKS*[*start*] inizialmente è l'indice del primo k-mer nell'insieme di common k-mers.
- *CKS*[end] inizialmente è l'indice del secondo k-mer nell'insieme di common k-mers.

Se questo algoritmo ritorna *true*, significa che i due k-mers in questione sono consistenti.

2.4 Algoritmo 4

In questo algoritmo vengono valutate alcune condizioni per vedere se due k-mers sono consistenti. Esso prende in input *CKS*[*start*], *CKS*[*end*], k, k_s, α e γ. L'i-esimo k-mer comune in *CKS* è rappresentato da quattro tuple (P_{1i}, O_{1i}, P_{2i}, O_{2i}):

- P_{1i} e P_{2i} sono le posizioni iniziali del k-mer comune rispettivamente in R_1 e R_2;
- O_{1i} e O_{2i} sono gli orientamenti del k-mer comune rispettivamente in R_1 e R_2.

Per **orientamento** si intende il verso di lettura di una sequenza. Se l'i-esimo k-mer comune ha lo stesso orientamento nelle due sequenze di riferimento ($O_{1i} = O_{2i}$), il k-mer comune è positivo (forward); questo significa che il kmer appare nella stessa direzione in entrambe le sequenze. In caso contrario si tratta di un k-mer comune opposto (reverse), cioè orientamenti opposti nelle due sequenze di riferimento. Per due k-mers comuni (P_{1i}, O_{1i}, P_{2i}, O_{2i}) e (P_{1j}, O_{1j}, P_{2j}, O_{2j}), due distanze $D1 = |P_{1j} - P_{1i}|$ e $D2 = |P_{2j} - P_{2i}|$ possono essere calcolate. P_{1j} e P_{2j} sono rispettivamente le posizioni di arrivo del k-mer comune in R_1 e R_2. LROD valuta quattro condizioni C1, C2, C3 e C4 per valutare la loro consistenza:

- C1: $P_{1i} < P_{1j}$ e $P_{2i} < P_{2j}$;
- C2: $P_{1i} < P_{1j}$ e $P_{2i} > P_{2j}$;
- C3: $D1 < \alpha$ e $D2 < \alpha$;
- C4: (Max(D1, D2) − Min(D1, D2)) / Max(D1, D2) < γ

Algorithm 4: Determine_consistent_1 (CKS[start], CKS[end], k, k_s, α,γ)

Input: common k-mers
Output: whether two common k-mers are consistent
Begin
  Getting the positions of the start and end common k-mers:
  P_{1i}, P_{2i}, P_{1j}, P_{2j} ($P_{1i} < P_{1j}$);
  $D_1 \leftarrow |P_{1j} - P_{1i}|$;
  $D_2 \leftarrow |P_{2j} - P_{2i}|$;
  if forward common k-mers **then**
    if ($P_{1i} < P_{1j}$ and $P_{2i} < P_{2j}$) and ($D_1 < \alpha$ && $D_2 < \alpha$) and((Max(D_1, D_2) − Min(D_1,D_2)) / Max(D_1,D_2) < γ) **then**
      return true;
    else
      return false;
    end if
  end if
  if reverse common k-mers **then**
    if ($P_{1i} < P_{1j}$ and $P_{2i} > P_{2j}$) and ($D_1 < \alpha$ and $D_2 < \alpha$) and ((Max(D_1, D_2) − Min(D_1,D_2)) / Max(D_1,D_2) < γ) **then**
      return true;
    else
      return false;
    end if
  end if
End

La condizione C1 indica che i due k-mers sono in *forward*; prendiamo in considerazione un k-mer comune alle due long reads R_1 e R_2. La condizione di *forward* si verifica quando la posizione di partenza P_{1i} del k-mer comune nella sequenza R_1 viene prima della posizione di arrivo P_{1j} del k-mer, e contemporaneamente la posizione di partenza P_{2i} del k-mer comune nella sequenza R_2 viene prima della posizione di arrivo P_{2j} del k-mer. Invece la condizione C2 mostra che i due k-mers sono in *reverse*; prendendo sempre in considerazione un k-mer comune alle due long reads R_1 e R_2, la condizione di *reverse* si verifica quando la posizione di partenza del k-mer comune P_{1i} nella sequenza R_1 viene prima della posizione di arrivo P_{1j} del k-mer ma, contrariamente alla condizione C1, la posizione di partenza P_{2i} del k-mer comune nella sequenza R_2 viene dopo la posizione di arrivo P_{2j} del k-mer.

A causa dell'elevato tasso di errore di sequenziamento del TGS, dovremmo consentire una *distanza* tra due k-mers comuni consistenti consecutivi. Tuttavia, maggiore è la distanza, maggiore è il numero di errori di sequenziamento esistenti [6]. Quindi C3 specifica la soglia della distanza massima α tra due k-mers comuni consistenti; tuttavia, α è difficile da determinare. Un α piccolo non considererebbe alcuni k-mers comuni consistenti mentre un α grande accetterebbe più k-mers comuni inconsistenti. In questa fase, LROD adotta un valore piccolo di α (400 per impostazione predefinita) per selezionare k-mers comuni consistenti con alta affidabilità. C4 specifica la differenza massima tra le distanze D1 e

D2 (γ = 0.3); se il rapporto in questione è inferiore a γ, la condizione è soddisfatta. Questo criterio mira a stabilire un limite relativo alla differenza tra le distanze per identificare k-mers comuni consistenti in modo più robusto. Ovviamente le 4 condizioni non possono essere rispettate tutte contemporaneamente in quanto le condizioni C1 e C2 sono mutuamente esclusive. Per i forward common k-mers devono essere rispettate le condizioni C1, C3 e C4. Per i reverse common k-mers devono essere rispettate le condizioni C2, C3 e C4. Quando i due k-mers comuni soddisfano le condizioni, LROD li considera consistenti. Se l'algoritmo 4 restituisce *false*, LROD utilizzerà l'algoritmo 5 per valutare ulteriormente la consistenza tra i due k-mers comuni, basandosi su k_s-mer più piccoli (k_s < k).

2.5 Algoritmo 5

L'algoritmo 5 viene invocato se non si riesce ad ottenere una catena di k-mers consistenti con i k-mers di taglia 15. Infatti LROD tenta di costruire una catena con k-mers di lunghezza inferiore.

Algorithm 5: Determine_consistent _2 (CKS[start], CKS[end], k, k_s, β)

Input: common k-mers
Output: whether two common k-mers are consistent
Begin:
Getting the positions of the start and end common k-mers:
$P_{1i}, P_{2i}, P_{1j}, P_{2j}$ ($P_{1i} < P_{1j}$);
$D_1 \leftarrow |P_{1j} - P_{1i}|$ and $D_2 \leftarrow |P_{2j} - P_{2i}|$;
if $D_1 > \beta$ or $D_2 > \beta$ **then**
  return false;
end if
For forward common k-mers, find the common k_s-mer set between $[P_{1i} + k - k_s, P_{1j} + k_s]$ in R_1 and $[P_{2i} + k - k_s, P_{2j} + k_s]$ in R_2. For reverse common k-mers, find the common k_s-mer set between $[P_{1i} + k - k_s, P_{1j} + k_s]$ in R_1 and $[P_{2j} - k_s, P_{2i} - k + k_s]$ in R_2.
if there is a chain which starting from the starting point to the ending point **then**
  return true;
else
  return false;
end if
End

Parametri fondamentali:

- k_s, lunghezza dei k-mers inferiore (k_s=9);
- β valore grande (1.500 per impostazione predefinita) che rappresenta la distanza massima tra due common k-mers in R1 e R2;

In primis, LROD trova piccoli k_s-mer (k_s < k) da due regioni in R_1 e R_2 tra due k-mers comuni. Quindi inizialmente l'algoritmo prende in input le posizioni di inizio e fine dei common k-mers. Poi calcola le distanze D1 e D2 e verfica che entrambe non siano maggiori di β, altrimenti i due k-mers non sarebbero consistenti, perchè troppo distanti . Poi calcola l'insieme di k_s-mer comuni a partire dai forward common k-mers nell'intervallo [P_{1i}+k-k_s, P_{1j}+k_s] in R_1 e nell'intervallo [P_{2i}+k-k_s, P_{2j}+k_s] in R_2. Successivamente calcola l'insieme di k_s-mer comuni a partire dai reverse common k-mers nello stesso intervallo considerato in precedenza nel caso di R_1 e nell'intervallo [P_{2j}-k_s, P_{2i} - k + k_s] in R_2. Se LROD riesce a trovare un percorso dal k_s-mer comune iniziale al k_s-mer comune finale, l'algoritmo 5 restituisce *true*. Dopo aver ottenuto la catena, il numero di k-mers comuni nella catena dovrebbe essere maggiore di 2. Infine, se i k-mers comuni nella catena sono positivi, LROD conclude che R_1 e R_2 probabilmente provengono dallo stesso filamento, altrimenti potrebbero provenire da filamenti inversi [6]. Inoltre, LROD ottiene due bozze di sovrapposizione [P_1, P_n+k] e [Q_1, Q_n+k] su R_1 e R_2, rispettivamente. L'intervallo [P_1, P_n+k] indica la sovrapposizione sulla long read R_1 e va da P_1, la posizione di partenza del primo k-mer nella catena, fino a P_n+k, la posizione del k-mer finale nella catena, spostato di k basi in avanti. In altre parole, questa sovrapposizione indica l'intervallo di R_1 che corrisponde alla sequenza sovrapposta con R_2. L'intervallo [Q_1, Q_n+k] indica la sovrapposizione sulla long read R_2 e va da Q_1, la posizione del primo k-mer nella catena in R_2, fino a Q_n+k, la posizione del k-mer finale nella catena in R_2, spostato di k basi in avanti. In sintesi indica l'intervallo di R_2 che corrisponde alla sequenza sovrapposta con R_1.

2.6 Algoritmo 6

L'algoritmo 6 è quello che valuta la regione di overlap, sulla base della catena di k-mers consistenti trovata con i precedenti algoritmi.

A causa di errori di sequenziamento, la sovrapposizione del candidato di cui sopra, potrebbe discostarsi leggermente dalla sovrapposizione reale. Supponiamo che la vera sovrapposizione su R_1 sia [SP_1, EP_1] e che la vera sovrapposizione su R_2 sia [SP_2, EP_2]; le lunghezze di R_1 e R_2 sono rispettivamente Len_1 e Len_2. LROD utilizza il seguente metodo per valutare la sovrapposizione candidata e ottenere la reale sovrapposizione per R_1 e R_2; i casi di allineamento valutati da LROD sono mostrati in Figura 1.

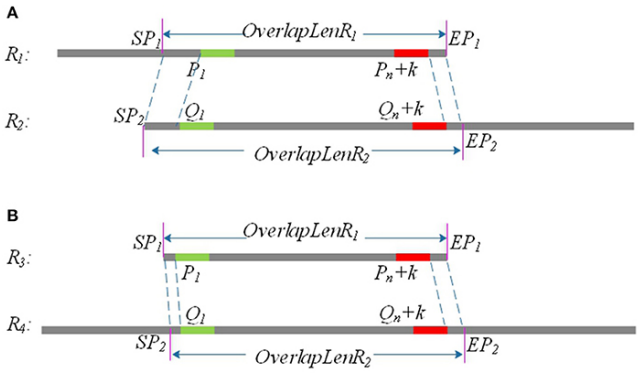

Figure 1: overlap parziale (A) e overlap totale (B)

Gli indici sono:

- P_1 e Q_1, le *start* positions dei positive common k-mers su R_1 e R_2 rispettivamente;
- P_n+k e Q_n+k, le *end* positions dei positive common k-mers su R_1 e R_2;
- SP_1 e EP_1, la *start* position e la *end* position della regione di overlap su R_1;
- SP_2 e EP_2, la *start* position e la *end* position della regione di overlap su R_2;

Le condizioni per i due casi di ovelap (parziale e totale):

(1) Se $P_1 > Q_1$ e $Len_1 - P_n <= Len_2 - Q_n$, $SP_1 = P_1 - Q_1$, $EP_1 = Len_1$; $SP_2 = 1$, $EP_2 = Q_n + Len_1 - P_n+1$, come mostrato in figura 1A. *Partial overlap*: l'estremità destra di una long read si allinea con l'estremità sinistra dell'altra long read.
(2) Se $P_1 < Q_1$ e $Len_1 - P_n <= Len_2 - Q_n$, $SP_1 = 1$, $EP_1 = Len_1$; $SP_2 = Q_1 - P_1$, $EP_2 = Q_n + Len_1 - P_n+1$, come mostrato nella Figura 1B. *Total overlap*: una long read è completamente allineata ad una parte dell'altra long read.
(3) Se $P_1 > Q_1$ e $Len_1 - P_n > Len_2 - Q_n$, $SP_1 = P_1 - Q_1$, $EP_1 = P_n + Len_2 - Q_n$; $SP_2 = 1$, $EP_2 = Len_2$.
(4) Se $P_1 < Q_1$ e $Len_1 - P_n > Len_2 - Q_n$, $SP_1 = 1$, $EP_1 = P_n + Len_2 - Q_n$; $SP_2 = Q_1 - P_1$, $EP_2 = Len_2$.

Algorithm 6: Evaluate_candidate_overlap_region(CKS, chain, α, γ, ε)

Input: CKS, chain, α, γ, ε
Output: True or False
Begin
  Getting draft overlap based on chain: $[SP_1, EP_1]$ and $[SP_2, EP_2]$;
  $OverlapLenR_1 \leftarrow EP_1 - SP_1$;
  $OverlapLenR_2 \leftarrow EP_2 - SP_2$;
  $MaxOverlapLen \leftarrow \max(OverlapLenR_1, OverlapLenR_2)$;
  $MinOverlapLen \leftarrow \min(OverlapLenR_1, OverlapLenR_2)$;
  if $(EP_1 - SP_1) - (P_n + k - P_1) < \alpha$ and $(EP_2 - SP_2) - (Q_n + k - Q_1) < \alpha$ and $MinOverlapLen > \varepsilon$ and $(MaxOverlapLen - MinOverlapLen) / MaxOverlapLen < \gamma$ **then**
    return true;
  end if
  return false;
End

Verificati i casi di allineamento, si può ottenere la reale sovrapposizione su R_1 e R_2. Come mostrato nella Figura 1, la lunghezza della sovrapposizione su R_1 è $OverlapLenR_1 = EP_1 - SP_1$ e la lunghezza della sovrapposizione su R_2 è $OverlapLenR_2 = EP_2 - SP_2$. Usiamo *MaxOverlapLen* e *MinOverlapLen* per rappresentare rispettivamente la lunghezza massima e la lunghezza minima di sovrapposizione: $MaxOverlapLen = max(OverlapLenR_1, OverlapLenR_2)$ e $MinOverlapLen = min(OverlapLenR_1, OverlapLenR_2)$.

L'algoritmo 6 prende in input *CKS*, *chain*, α, γ e ϵ; *chain* fa riferimento alla catena di k-mers consistenti creata con gli algoritmi precedenti. Quando R_1 e R_2 soddisfano le seguenti tre condizioni, LROD considera che R_1 e R_2 abbiano una sovrapposizione. Le sovrapposizioni sono $[SP_1, EP_1]$ e $[SP_2, EP_2]$ rispettivamente su R_1 e R_2; altrimenti non esiste alcuna sovrapposizione tra loro. ϵ è la soglia della lunghezza di sovrapposizione (500 per impostazione predefinita). In particolare vengono valutate le seguenti tre condizioni:

(1) $(EP_1 - SP_1)-(P_n+k - P_1) < \alpha$ e $(EP_2 - SP_2)-(Q_n+k - Q_1) < \alpha$; controlla se la lunghezza della sovrapposizione calcolata su R_1 e R_2 è vicina alla lunghezza effettiva dell'overlap. Se questa differenza è minore di α (400), allora la sovrapposizione su R_1 e R_2 è considerata attendibile.
(2) MinOverlapLen > ϵ; se la sovrapposizione è più lunga di ϵ (500), viene considerata significativa. ϵ è il numero minimo di basi che la sovprapposizione dovrà contenere.
(3) (MaxOverlapLen - MinOverlapLen) / MaxOverlapLen < γ; questa condizione controlla quanto le lunghezze massime e minime della sovrapposizione differiscono. Se questa differenza è inferiore a un certo valore γ, allora la sovrapposizione è considerata coerente. Questo aiuta a valutare la consistenza delle lunghezze della sovrapposizione su entrambe le reads.

Se tutte le condizioni sono soddisfatte, LROD considera che le due reads abbiano una sovrapposizione significativa, altrimenti non viene identificata nessuna sovrapposizione tra di loro.

3 RISULTATI OTTENUTI DA LROD

Per verificare l'efficacia del metodo proposto in questo documento, sono stati utilizzati tre dataset simulati e tre reali per confrontare LROD, MHAP e Minimap2; le prestazioni sono state verificate con k-mers di lunghezza k=13 e k=15.

3.1 Discussione

I tre dataset provengono da genomi di *Escherichia Coli (E. coli)*, *Caenorhabditis elegans (C. elegans)*, e *humans*, che furono sequenziati da $SMRT_s$. I dataset reali collegati ad *E. coli* e *C. elegans* sono disponibili su **http://schatzlab.cshl.edu/data/ectools/**. Il real human dataset è NA20300 (SRR9683669). I tre dataset reali sono qui indicati come *E. coli_Real, C. elegans_Real e Human_Real*. In questo paper, è stato usato SURVIVOR [3] per ottenere tre dataset simulati: 10X coverage E. coli (*E. coli-10*), 20X coverage E. coli (*E. coli-20*), e 10X coverage human chromosome 20 (*chr20-10*). Per questi dataset di long reads, sono state mantenute le long reads la cui lunghezza era superiore a 2.000 coppie di basi (bp) nei seguenti esperimenti.

La figura 2 mostra i dettagli dei dataset di long reads, inclusa la lunghezza genomica, la lunghezza media delle letture, il numero di letture e la copertura. Per i tre dataset simulati, è possibile ottenere direttamente le sovrapposizioni reali tra le long reads. Per i tre dataset reali, viene usato BLASR [2] per allineare queste long reads rispetto ai genomi di riferimento. Sono state mantenute solo quelle letture la cui qualità di allineamento era > 85% e le long reads erano completamente allineate sul genoma di riferimento. Successivamente è stato possibile acquisire vere e proprie sovrapposizioni tra queste long reads in base alle loro posizioni di allineamento.

Datasets	Genomic length (Mbp)	Average length of reads(bp)	Number of read	Coverage
E. coli-10	~4.6	6,555	6,955	~10
E. coli-20	~4.6	6,619	13,911	~20
chr20-10	~6.4	6,621	96,574	~10
E. coli_Real	~4.6	4,185	6,972	~7
C. elegans_Real	~99.9	4,091	188,559	~77
Human_Real	~3,157	25,890	461,247	~3.78

Figure 2: dettagli dei dataset

Le sovrapposizioni ottenute sono state utilizzate per valutare le prestazioni degli strumenti di overlap detection. Tutti gli strumenti sono stati eseguiti con 10 threads attivi su un computer con 128 GB di memoria. Durante gli esperimenti, l'intero tempo di LROD può essere ridotto adottando un numero maggiore di threads.

3.2 Risultati

Per il *Human_Real* dataset, quando k = 13, Minimap2 e LROD non si sono conclusi con 10 threads dopo 10 giorni e il requisito di memoria di MHAP era maggiore della capacità di memoria del computer (128 GB). Pertanto, non vengono forniti i risultati per il real human dataset con k = 13. Come mostrato nella Figura 4 e nella Figura 5, LROD ha ottenuto risultati soddisfacenti per questi dataset. Soprattutto nel dataset simulato, nella maggior parte dei casi, *precision*, *recall* e *F1-score* di LROD sono stati superiori a quelli di Minimap2 e MHAP. Sebbene LROD fosse leggermente inferiore a Minimap2 in termini di *F1-score* per E. coli_Real, ha ottenuto performance simili a quelle di Minimap2. Quando k = 15, come mostrato in Figura 4 e 5, LROD è stato superiore agli altri due tools sulla base dell'*F1-score*.

3.3 Dataset simulati

Per *E. coli-10, E. coli-20 e chr20-10*, LROD ha avuto costantemente i migliori risultati in termini di *precision, recall* e *F1-score*. Per questi tre dataset, la *precision* di LROD è stata costantemente superiore al 90%. Sebbene il *recall* di LROD non abbia raggiunto il 90%, tutti i valori erano >85% e superiori a quelli degli altri due tools. Gli *F1-score* medi per LROD sono stati più alti del 5% e del 20% rispetto a quelli di Minimap2 e MHAP, rispettivamente. La *precision* di LROD ha superato il 92% e il *recall* di LROD ha superato l'83%. Sia la *precision* che il *recall* di LROD sono stati superiori a quelli di Minimap2. Quindi, l'*F1-score* di LROD si è classificato al primo posto per questi dataset.

3.4 Dataset reali

Come mostrato nelle Figure 4, 5 e 6 per E. coli_Real, l'*F1-score* di LROD e Minimap2 è stato >95%. Sebbene LROD non abbia ottenuto il più alto *recall*, si è avvicinato molto a quello di Minimap2. Per C. elegans_Real, le *precision* di Minimap2 e MHAP sono state inferiori rispetto a quelle di LROD. Sia Minimap2 che LROD hanno mostrato un buon *recall*, e LROD ha ottenuto il miglior *F1-score*. Per Human_Real, l'*F1-score* di Minimap2 e LROD è stato simile. Da notare che i requisiti di memoria di MHAP superavano la capacità di memoria del computer utilizzato dai ricercatori. Quindi non sono stati forniti i risultati.

3.5 Tempo di esecuzione e requisiti di memoria

Sono stati confrontati i requisiti computazionali tra i tre tools per i sei dataset. I risultati sono mostrati nella Figura 7. Il consumo di memoria di MHAP è stato molto elevato, mentre Minimap2 ha avuto il consumo di memoria più basso. Sebbene il consumo di memoria di LROD sia stato maggiore di quello di Minimap2, risulta inferiore rispetto a quello di MHAP. In termini di tempo di esecuzione (CPU time), LROD è risultato migliore di MHAP. Il tempo di esecuzione sorprendente e il consumo di memoria di Minimap2 hanno attirato l'attenzione di molti ricercatori. Da notare che tutti i tools possono usare più threads per ridurre l'intero tempo.

3.6 Metriche usate per la valutazione degli algoritmi

- *Precision* è l'accuratezza della predizione delle classi positive. E' definita come il rapporto tra i veri positivi e la somma di veri positivi e falsi positivi: $\frac{TP}{TP+FP}$. Tuttavia la *precision* da sola non basta ed è associata alla metrica *recall*.
- *Recall* indica il rapporto di istanze positive correttamente individuate sul totale dei casi. E' definita come il rapporto tra i veri positivi e la somma di veri positivi e falsi negativi: $\frac{TP}{TP+FN}$.
- *F1-score* è una misura dell'accuratezza di un test; è la media armonica di *precision* e *recall*: $\frac{2}{\frac{1}{r}+\frac{1}{p}} = 2 \cdot \frac{p \cdot r}{p+r}$. Può assumere valori compresi fra 0 e 1. Viene attribuito un peso maggiore ai valori piccoli; questò fa sì che un classificatore ottenga un alto *F1-score* solo quando *precision* e *recall* sono entrambi alti. **https://it.wikipedia.org/wiki/F1_score**.

		Realtà: Positivo	Realtà: Negativo
Predizione	Positivo	TP	FP
Predizione	Negativo	FN	TN

Figure 3: matrice di confusione

k	Dataset	Precision			Recall			F1-score		
		MHAP	Minimap2	LROD	MHAP	Minimap2	LROD	MHAP	Minimap2	LROD
k = 13	*E. coli-10*	0.871	0.866	0.935	0.599	0.837	0.887	0.710	0.851	0.910
	E. coli-20	0.859	0.855	0.924	0.597	0.827	0.875	0.704	0.841	0.899
	chr20-10	0.685	0.752	0.933	0.612	0.829	0.893	0.646	0.788	0.912
	E. coli_Real	0.967	0.987	0.976	0.875	0.969	0.948	0.919	0.978	0.962
	C. elegan_Real	0.362	0.685	0.752	0.746	0.917	0.909	0.487	0.785	0.824
k = 15	*E. coli-10*	0.878	0.834	0.941	0.490	0.759	0.849	0.629	0.795	0.893
	E. coli-20	0.866	0.825	0.930	0.487	0.751	0.831	0.624	0.786	0.878
	chr20-10	0.734	0.851	0.942	0.504	0.746	0.855	0.598	0.795	0.896
	E. coli_Real	0.963	0.957	0.964	0.798	0.953	0.924	0.873	0.955	0.943
	C. elegan_Real	0.729	0.789	0.897	0.706	0.947	0.958	0.717	0.861	0.926
	Human_Real	–	0.779	0.736	–	0.667	0.706	–	0.719	0.720

Figure 4: overlap detection sui dataset

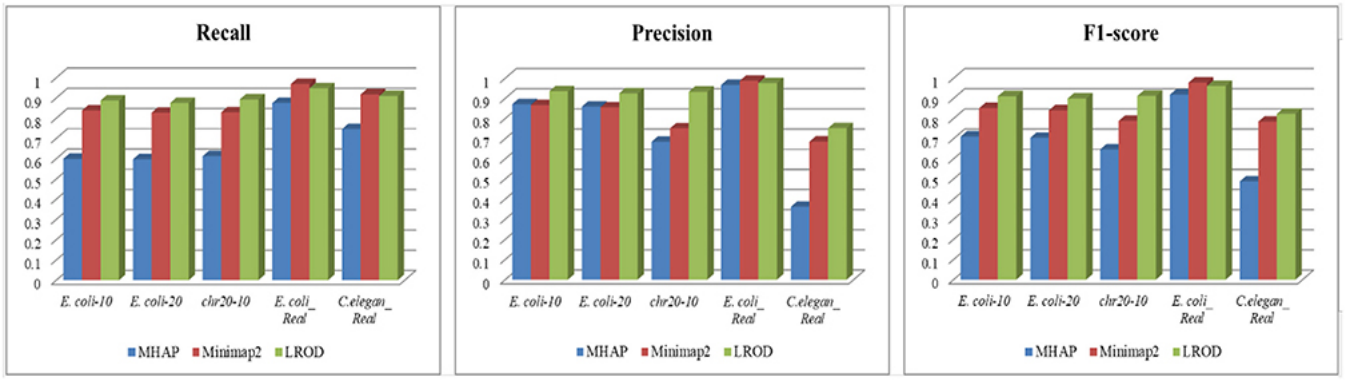

Figure 5: overlap detection con k = 13

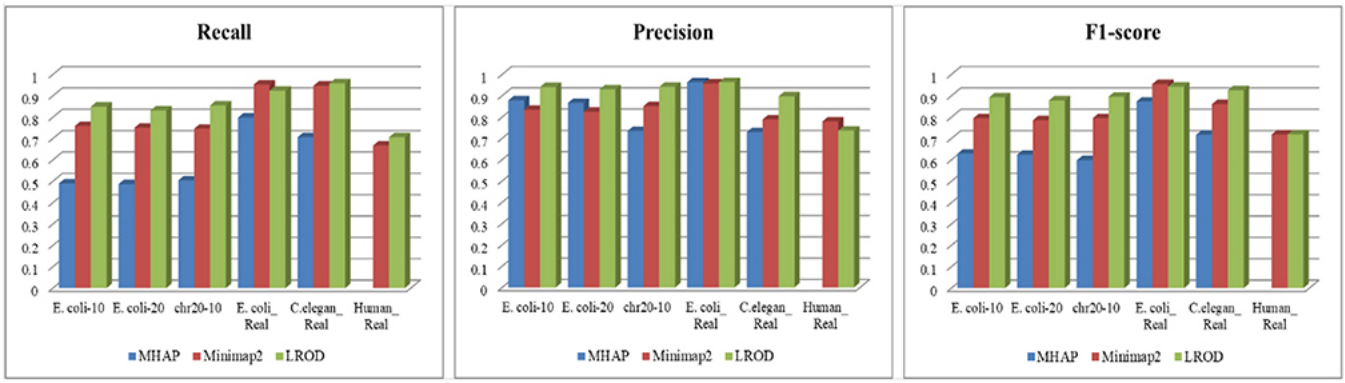

Figure 6: overlap detection con k = 15

4 LROD: IMPLEMENTAZIONE

LROD è distribuito su GitHub al link **https://github.com/luojunwei/LROD**. Il linguaggio utilizzato è prevalentemente il C, anche se l'estensione dei files e il compilatore usati sono C++. L'eseguibile "LROD" viene dato in output eseguendo il comando make.

k	Dataset	MHAP		Minimap2		LROD	
		Running time	Memory (Mb)	Running time	Memory (Mb)	Running time	Memory (Mb)
k = 13	*E. coli-10*	4 m 29 s	39,703	0 m 45 s	1,251	2 m 4 s	1,491
	E. coli-20	9 m 15 s	39,971	2 m 30 s	1,612	5 m 5 s	2,336
	chr20-10	87 m 44 s	42,748	125 m 26 s	8,441	59 m 1 s	11,487
	E. coli_Real	4 m 14 s	39,501	0 m 26 s	1,129	1 m 36 s	1,238
	C. elegans_Real	28,813 m 15 s	42,156	1,753 m 38 s	25,940	14,625 m 23 s	36,708
k = 15	*E. coli-10*	6 m 3 s	41,163	0 m 17 s	2,644	2 m 14 s	3,326
	E. coli-20	12 m 34 s	42,959	0 m 48 s	2,972	5 m 51 s	4,248
	chr20-10	88 m 18 s	43,641	21 m 52 s	7,625	46 m 38 s	11,906
	E. coli_Real	4 m 42 s	41,099	0 m 1 s	2,513	1 m 40 s	3,036
	C. elegans_Real	377 m 17 s	44,414	292 m 45 s	15,522	620 m 16 s	17,418
	Human_Real	–	–	424 m 42 s	35,814	4,402 m 10 s	51,906

Figure 7: tempo di esecuzione e memoria

4.1 Struttura del progetto

La struttura del progetto, ottenuta con il comando tree, è la seguente:

```
LROD
|---- aligning.cpp
|---- aligning.h
|---- bitarray.cpp
|---- bitarray.h
|---- kmer.cpp
|---- kmer.h
|---- LROD
|---- main.cpp
|---- makefile
|---- read.cpp
|---- read.h
|---- README.md
|---- test
      |---- command.txt
      |---- kmer_file.txt
      |---- long_read.fa
      |---- result.csv
      +---- unit_test
            |---- data
            |---- kmer_frequency.txt
            +---- readme.md
      |---- makefile
      |---- README.MD
      +---- unit_test.cpp
```

4.2 Pre-condizioni e Post-condizioni

Prima di eseguire LROD si deve installare DSK, un tool che prende in input un file di long reads e restituisce un file contente i kmer con la frequenza associata. Un esempio dell'output è il seguente:

```
AAAAAAAAAAAAAAA 1945
AAAAAAAAAAAAAAC 76
AAAAAAAAAAAAAAT 38
AAAAAAAAAAAAAAG 39
AAAAAAAAAAAAACA 50
AAAAAAAAAAAAACC 18
AAAAAAAAAAAAACT 10
AAAAAAAAAAAAACG 3
```

Il comando da eseguire è:

dsk2ascii -file kmer_reads.h5 -out kmer-freq-file.txt

LROD sfrutterà questi kmer per determinare l'overlap tra le reads. Link al github: **https://github.com/GATB/dsk**. Per le post-condizioni, LROD restituisce un file .csv chiamato result.csv che rappresenta le regioni di overlap.

Il comando per eseguire LROD è il seguente:

LROD -r long_read.fa -c kmer_file.txt -o result

4.3 Fasi dell'implementazione

Per facilitare la comprensione, abbiamo diviso l'implementazione in varie fasi.

4.3.1 Fase 1: preprocessing kmerHashTable. Nella funzione **GetKmerHashTableHead** a riga 378 in kmer.cpp viene costruita una tabella hash per ricercare i k-mer all'interno delle read. Si ha una coppia chiave-valore "key=indice hash del kmer" e "value=kmerInteger". Inizialmente si trasforma il kmer, attraverso operazioni bit a bit, in kmerInteger con la procedura SetBitKmer(kmerReal:AAAT). Successivamente si calcola l'indice della tabella hash in cui verrà posizionato il k-mer con la procedura Hash(kmerInteger). Infine, in posizione hashIndex, si salverà il kmerInteger, valutando anche le condizioni sulla frequenza (frequenza compresa tra max e min).

La tabella Hash ha molteplici scopi:

- ricerca dei k-mers;
- verifica dei k-mers presenti nelle reads.
- calcolo delle posizioni dei k-mers.

Criteri per l'inserimento:

- frequenza compresa tra max e min
- lettere non tutte uguali (AAAAAA non viene considerato)

Strutture in C:

```
//nodo della kmer hash
typedef struct KmerHashNode{
    unsigned int kmer; //kmerInteger
int startPositionInArray; //posizione del kmer in read
}KmerHashNode;

//lista di kmerHashNode
typedef struct KmerHashTableHead{
    KmerHashNode * kmerHashNode; // lista di kmer
   unsigned long int allocationCount; //# nodi allocati
   long int min;
    long int max;
}KmerHashTableHead;
```

4.3.2 Fase 2: preprocessing kmerRead. La seconda fase è il preprocessing della struttura kmerRead che avviene a riga 600 nella funzione **InitKmerReadNodeHeadSub** in kmer.cpp; tale struttura memorizza i kmer presenti nella read i-esima considerata e che si trovano anche nella tabella hash. Il kmer viene memorizzato insieme all'indice della read e alla sua posizione all'interno della read.

```
typedef struct KmerReadNode{
    unsigned int kmer;
    unsigned int readIndex;
    unsigned int position;
    bool orientation;
}KmerReadNode;

typedef struct KmerReadNodeHead{
    KmerReadNode * kmerReadNode;
    long int realCount;
    long int allocationCount;
    long int kmerLength;
    long int \textit{start}ReadIndex;
    long int endReadIndex;
}KmerReadNodeHead;
```

Vediamo i passi, presenti nella funzione:

- Si analizza un batch di 50.000 reads alla volta;
- per ogni read e per ogni k-mer nella read si verifica la presenza nella tabella hash.
- se presente, si aggiunge un nodo KmerReadNode in posizione realCount (inizializzato a 0 e incrementato ogni volta che viene inserito il kmer), inserendo il k-mer, readIndex, posizione del k-mer nella read;
- Si aggiorna anche la KmerHashTable.startPositionInArray = posizione del kmer.
  L'obiettivo finale è salvare il riferimento di tutti i k-mer presenti nella HashTable e nelle reads considerate, per determinare il commonKmerSet ovvero l'insieme di k-mers comuni.

4.3.3 Fase 3: Preprocessing threads, Common k-mers. L'ultima fase del preprocessing riguarda i threads e il common k-mers set. LROD sfrutta la libreria *p_threads* per gestire il multithreading. Nella funzione **GetCommonKmerHeadAllThread** in aligning.cpp a riga 149 inizialmente si alloca un array di threads:

pthread_t tid[totalThreadNumber];

In seguito si alloca la struttura che ogni thread avrà a disposizione per gestire il lavoro. Tale struttura contiene i riferimenti alle strutture menzionate in precedenza:

```
typedef struct GetCommonKmerHeadP{
    KmerHashTableHead * kmerHashTableHead;
    KmerReadNodeHead * kmerReadNodeHead;
    ReadSetHead * readSetHead;
    long int kmerLength;
    char * readFile;
    char * outputFile;
    long int step;
    long int threadIndex;
    long int totalThreadNumber;
    long int smallIntervalDistance;
    long int largeIntervalDistance;
    long int overlapLengthCutOff;
    long int smallKmerLength;
    float lengthRatio;
    long int startReadIndex;
}GetCommonKmerHeadP;
```

Si calcola dapprima il numero di reads che ciascun thread processerà. In un ciclo for si inizializza la struttura e si avviano i threads, ognuno dei quali eseguirà la funzione GetCommonKmerHeadThread, responsabile della ricerca dei common k-mers sulle reads assegnate.

pthread_create(&tid[i], NULL, GetCommonKmerHeadThread, (void *)&getCommonKmerHeadP[i])

Ogni thread alloca le strutture per gestire i k-mers comuni alle reads:

```
typedef struct CommonKmer{ //lista di common k-mers
    //indice della read in cui si trova il common kmer
        long int readIndex;
    // posizione del kmer nella kmerReadNode[i]
        long int leftPosition;
    //posizione del kmer nella read[i] considerata
        long int rightPosition;
    //se 1 -> kmer è in forward, se 0 è in reverse.
        bool orientation;
}CommonKmer;

typedef struct CommonKmerHead{ //Testa della lista
    CommonKmer * commonKmer;
    long int readIndex;
    long int realCount;
    long int allocationCount;
}CommonKmerHead;
```

Usando la kmerReadNode, la kmerHashTable e leggendo i kmers dalla read corrente, ogni thread inserisce il k-mer nella struttura CommonKmer, invocando la funzione **InsertCommonToTwoReadAligningHead** a riga 335 in aligning.cpp. Questo viene fatto da ogni thread per ogni k-mer presente nel batch di reads; vediamo i passi per l'inserimento dei common k-mers.

Per ogni read nel batch:

- preleva un k-mer alla volta e verifica la presenza nella tabella hash;

- se presente, sfruttando la kmerReadNode come riferimento, inserisce il commonKmer:
  - CommonKmer.leftPosition=posizione del k-mer nella read considerata
  - CommonKmer.rightPosition=posizione del kmer i-esimo in kmerReadNode.
  - readIndex = indice di dove trova il commonKmer rispetto alla kmerReadNode (che contiene tutti i riferimenti dei kmer alle read)
- se il kmer non è presente nella tabella hash, passa al k-mer successivo.
- terminata la i-esima read e riempita la lista di commonKmerHead, che contiene gli indici delle read in cui vengono trovati k-mers comuni, rimuove gli indici duplicati e la riordina.

Nella fase iniziale di questa parte, vengono allocate anche le strutture relative all'allineamento, quali i grafi:

```
//Nodo del grafo
typedef struct AdjGraph{
   //leftPosition di commonKmer, quindi KmerReadNode[i]
    long int dataLeft;
    //position del kmer nella read[i]
    long int dataRight;
    bool visit;
}AdjGraph;

typedef struct AdjGraphHead{ //testa del grago
    AdjGraph * graph;
    long int realCountGraph;
    long int allocationCountGraph;
    AdjGraph * reverseGraph;
    long int reverseRealCountGraph;
    long int reverseAllocationCountGraph;
    ArcIndex * arcIndex;
    long int realCountArc;
    long int allocationCountArc;
    float * distanceToSource;
    bool * visited
    long int leftStart;
    long int rightStart;
    long int leftEnd;
    long int rightEnd;
    char * localLeftRead;
    char * localRightRead;
    float lengthRatio;
    long int overlapLengthCutOff;
}AdjGraphHead;
```

4.3.4 Fase 4: Allineamento. Dopo l'inserimento dei common k-mers, alla read i-esima si verifica l'overlap, invocando la funzione **GetOverlapResult** a riga 373 in aligning.cpp; qui inizia l'algoritmo 1 presentato dai ricercatori di LROD. Nella funzione GetOverlapResult a riga 1221, per ogni elemento del common k-mer set, LROD aggiorna i campi dell'i-esimo grafo nel seguente modo:

- AdjGraph[i].dataLeft = commonKmer.leftPosition, riferimento al k-mer nella kmerReadNode, chiamiamola R_2
- AdjGraph[i].dataRight= commonKmer.rightPosition, riferimento al k-mer della read corrente, chiamiamola R_1
- marca il grafo come visitato, quindi il campo *visit* viene posto uguale a 1.

Se i grafi di adiacenza sono più di 5, LROD valuta la consistenza di ciascun common k-mer salvato in ciascun grafo. Per valutare la consistenza, LROD nella funzione GetOverlapResult invoca la funzione **CreateGraphSinglePath** a riga 1325, in riferimento all'algoritmo 2 **Chaining_from_start**. Questa funzione ha lo scopo di creare un singolo percorso nel grafo di adiacenza. Inizializza gli indici *start* e *end* necessari per determinare la consistenza dei k-mers e li aggiorna ad ogni invocazione della funzione. Finchè non arriva alla fine del grafo verifica se è possibile aggiungere un arco tra i due grafi (LROD chiama la funzione **AddEdgeInGraph** riga 1123 nella funzione *CreatGraphSinglePath*); L'obiettivo è quello di determinare una catena di k-mers consistenti, dove per consistenza si intende verificare le 4 condizioni dell'algoritmo 4 analizzate nella funzione **AddEdgeInGraph**. Visto che ogni grafo contiene i riferimenti ai k-mers comuni, ragioneremo direttamente sui grafi. Ovvero:

sia G = [G1,G2,G3,G4,G5,..., G_realCountGraph], l'array di grafi. LROD valuta le condizioni dell'algoritmo 4 su G1 e G2:

- verifica che la distanza tra dataRight e dataLeft, quindi tra i common k-mers di R_1 e di R_2 non sia maggiore di maxIntervalDistance = 1500, ossia il β nella condizione dell'algoritmo 5 (riga 986 in *aligning.cpp* nella funzione *AddEdgeInGraph*).
- verifica che la diffenza tra maxvalue - minvalue /maxvalue < G->lenghtRatio=0,3, ossia la condizione C4 dell'algoritmo 4 (riga 998 in *aligning.cpp*).
- verifica che la distanza tra dataRight e dataLeft, quindi tra i common k-mers di R_1 e di R_2 non sia maggiore di largestIntervalDistance = 400, ossia α nella condizione 3 dell'algoritmo 4 (riga 1000 in *aligning.cpp*).
- verifica se l'orientamento dei k-mers è in forward o in reverse, quindi le condizioni 1 e 2 dell'algoritmo 4 (riga 1005 in *aligning.cpp*).

Se tutte le condizioni sono rispettate, tra i common k-mers contenuti in G1 e G2, si aggiunge un arco tra i due grafi (*edge==1*) e successivamente vengono aggiornati sia lo *startIndex* che l'*endIndex* (da riga 1125 a 1132 in *aligning.cpp*) per confrontare G2 con G3. In caso contrario se non è stato aggiunto l'arco (*edge==0*), viene aggiornato solo l'*endIndex* (riga 1134 in *aligning.cpp*) e si confronta G1 con G3 e cosi via. Fa riferimento alla creazione della catena di common k-mers con i relativi aggiornamenti degli indici spiegati nel paragrafo 2.2. Ricordiamo che la struttura AdjGraphHead memorizza una lista di archi.

```
// Archi del grafo
typedef struct ArcIndex{
    long int startIndex;
    long int endIndex;
    float weight;
}ArcIndex;
```

Ci devono essere almeno due k-mers consistenti nella catena (riga 1140 in *aligning.cpp*); LROD quindi verifica la presenza di almeno un arco tra due grafi. Verificata la consistenza si determina l'eventuale overlap tramite la funzione **Overlap_Display_Graph** chiamata

da CreatGraphSinglePath, in riferimento all'**algoritmo 6 Evaluate_candidate _overlap _region**. La funzione valuta le 3 condizioni dell'algoritmo 6:

- verifica inizialmente che la regione abbia dimensione maggiore di 300 (riga 694 in *aligning.cpp*);
- verifica che la distanza tra i k-mers sia minore di 400 (riga 724 in *aligning.cpp*), ovvero α nella condizione 1; altrimenti verifica se è compresa tra 400 e 1500, ovvero la condizione 3 (riga 695 in *aligning.cpp*); in questo caso LROD analizza k-mers di dimensione inferiore.
- verifica che (MaxOverLen - MinOverLen)/MaxOverLen > G->lengthRatio=0.3, ovvero la condizione 3 (riga 695 in *aligning.cpp*);

Infine scrive la regione di overlap nel file assegnato al thread.

5 CONCLUSIONI FINALI

5.1 Punti critici

(1) Nel file *read.cpp* la funzione *getReadSetHead* da riga 36 a 82 apre due volte il file *long_read.fa* contenente le 500 long_reads separate dal carattere ">":
- la prima volta per incrementare il contatore delle long reads (da riga 36 a 49 in *read.cpp*);
- la seconda volta scorre per eliminare il carattere di terminazione "/n" o "/r" e per popolare la struttura readSetHead (da riga 58 a 82 in *read.cpp*).

Avrebbero potuto fare entrambe le operazioni in una sola volta.

(2) Nel file *kmer.cpp* la funzione *GetKmerHashTableHead* da riga 380 a 534 apre tre volte il file *kmer_file.txt* contenente i k-mers con le frequenze associate ed effettua dei confronti servendosi della funzione *DetectSameKmer* presente in *kmer.cpp* da riga 143 a 155.
- La prima volta copia solo le frequenze "*kmerF*" dei k-mers e le analizza; fa il break se almeno un carattere della frequenza (un numero in questo caso) è diverso (da riga 380 a 452 in *kmer.cpp*). Essenzialmente sono due i problemi di questo confronto:
  - dovrebbe escludere i k-mer che hanno una frequenza con tutti numeri uguali, anche se non ci sono motivi validi nel fare questa cosa; cioè non dovrebbe prendere in considerazione quelli con frequenza 11, 22,...111,...,2222 e cosi via ma questi non vengono esclusi (spiegazione sotto);
  - viene effettuato all'interno di un ciclo for che itera da zero a kmerLength dove kmerLength = 15 ma *kmerF* contiene al massimo 4 numeri in *kmer_file.txt*; quindi questo confronto restituisce *true* anche se analizza la frequenza 1111 e il k-mer con quest'ultima frequenza non viene eliminato. Un k-mer, per essere escluso, dovrebbe avere una frequenza formata da 15 numeri, ad esempio 111111111111111. Quest'ultimo non verrebbe preso in considerazione nei passaggi successivi.
- La seconda volta (da riga 458 a 488 in *kmer.cpp*). si copia solo il k-mer (senza la frequenza) ed effettua correttamente il confronto sui caratteri del k-mer considerando solo quelli che differiscono in almeno un carattere per ottenere il conteggio totale dei k-mer, tenendo conto anche dell'intervallo delle frequenze. Con questo confronto, la stringa *AAAAAAAAAAAAAA*, per esempio, non viene presa in considerazione.
- La terza volta effettua nuovamente il confronto del punto precedente e poi converte i kmer in bytes e li memorizza in una tabella hash (da riga 498 a 534 in *kmer.cpp*).

(3) Operazioni ripetitive legate alla kmerHashTable, effettuate per tutti i k-mers delle reads:
- costruzione della tabella stessa, riga 378 in kmer.cpp;
- inserimento dei k-mers nella KmerReadNode, riga 600 in kmer.cpp;
- inserimento dei k-mers nella CommonKmer, riga 230 in aligning.cpp.

sono:
- prelievo del kmer dal file (strncpy);
- conversione in kmerInteger (SetBitKmer);
- calcolo dell'hashIndex;
- verifica della presenza del kmer nella HashTable (SearchKmerHashTable), tramite hashIndex;

(4) La funzione è GetCommonKmerHeadThread in *aligning.cpp* a riga 230, non effettua un controllo sulla presenza del k-mer in reverse, all'interno della hash table. La conseguenza è che a riga 360 esegue inutilmente la funzione *InsertCommonToTwoReadAligningHead* nel file *aligning.cpp*.

(5) Memoria e tempo di esecuzione, allocazione di molte strutture e aggiornamento:
- ReadSetHead, ReadSet;
- KmerHashTableHead, KmerHashNode;
- KmerReadNodeHead, KmerReadNode;
- CommonKmerHead, CommoKmer;
- AdjGraphHead, AdjGraph, ArcIndex;
- GetCommonmerHeadP;

Ciò crea un aggravio prestazionale sia per il tempo di esecuzione sia per l'utilizzo della memoria.

(6) Altri punti critici:
- mancanza di commenti specifici;
- nomi delle variabili fuorvianti;
- presenza di molti cicli innestati che scorrono le varie strutture e le aggiornano;
- non utilizzo del C++ e di librerie, ma solo del C puro. Questo potrebbe essere un vantaggio dal punto di vista prestazionale, anche se non sfruttato, ma potrebbe anche essere uno svantaggio in termini di leggibilità del codice;
- utilizzo di un tool esterno come DSK per ottenere i k-mers solidi;
- non utilizzo di container per eseguire il tool. Questo costringe ad installare tutte le dipendenze per l'esecuzione dei tools.

REFERENCES

[1] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. 2015. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature biotechnology* 33, 6 (2015), 623–630.

[2] Mark J Chaisson and Glenn Tesler. 2012. Mapping single molecule sequencing reads using basic local alignment with successive refinement (BLASR): application and theory. *BMC bioinformatics* 13 (2012), 1–18.

[3] Daniel C Jeffares, Clemency Jolly, Mimoza Hoti, Doug Speed, Liam Shaw, Charalampos Rallis, Francois Balloux, Christophe Dessimoz, Jürg Bähler, and Fritz J Sedlazeck. 2017. Transient structural variations have strong effects on quantitative traits and reproductive isolation in fission yeast. *Nature communications* 8, 1 (2017), 14061.

[4] Heng Li. 2018. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics* 34, 18 (2018), 3094–3100.

[5] Binghang Liu, Yujian Shi, Jianying Yuan, Xuesong Hu, Hao Zhang, Nan Li, Zhenyu Li, Yanxiang Chen, Desheng Mu, and Wei Fan. 2013. Estimation of genomic characteristics by analyzing k-mer frequency in de novo genome projects. *arXiv preprint arXiv:1308.2012* (2013).

[6] Junwei Luo, Ranran Chen, Xiaohong Zhang, Yan Wang, Huimin Luo, Chaokun Yan, and Zhanqiang Huo. 2020. LROD: An Overlap Detection Algorithm for Long Reads Based on k-mer Distribution. *Frontiers in Genetics* 11 (2020). https://doi.org/10.3389/fgene.2020.00632

[7] Niranjan Nagarajan and Mihai Pop. 2013. Sequence assembly demystified. *Nature Reviews Genetics* 14, 3 (2013), 157–167.

[8] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. 2001. An Eulerian path approach to DNA fragment assembly. *Proceedings of the national academy of sciences* 98, 17 (2001), 9748–9753.

[9] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. 2013. DSK: k-mer counting with very low memory usage. *Bioinformatics* 29, 5 (2013), 652–653.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009