LROD

Progetto di Bioinformatica

Staiano Catello

Venturino Silvio

Programma

1. Obiettivi paper, LROD
2. Algoritmo
3. Implementazione

Obiettivi paper

LROD (long read overlap detection) è un algoritmo che tenta di migliorare il rilevamento delle sovrapposizioni.
Sfrutta i kmer, sottostringhe del genoma di lunghezza k.
Obiettivo:

- trovare le sovrapposizioni tra due long reads, in base alla distribuzione dei kmers solidi

Risvolto:

- Trovare sovrapposizioni è utile per il processo di **assemblaggio** del genoma. Gli overlap vengono utilizzati per allineare e unire i frammenti di DNA in un'unica sequenza continua.

Obiettivi paper

Problemi durante il processo di detection dell'overlap:
1. alto tasso di errori di sequenziamento TGS
2. Regioni ripetitive complicano il processo di rilevamento delle sovrapposizioni

LROD sfrutta i kmers solidi per risolvere tali problemi.
Un **solid kmer** è un kmer presente molto frequentemente che potrebbe non includere errori di sequenziamento, il che aiuta a semplificare il processo di rilevamento della sovrapposizione.

Le **regioni ripetitive** vanno evitate poichè i kmer presenti in queste regioni potrebbero essere interpretati erroneamente come solid kmer.

Solid kmer

Per un set di dati a lettura lunga:

- un kmer con una piccola frequenza include comunemente errori di sequenziamento
- un kmer con una grande frequenza solitamente proviene da una regione ripetitiva.

Pertanto, LROD seleziona solo k-mer le cui frequenze sono nell'intervallo [fmin, fmax] come kmer solidi, dove fmin e fmax sono due soglie calcolate da LROD. L'utilizzo solo di kmer solidi consente a LROD di evitare alcuni problemi causati da errori di sequenziamento e regioni ripetitive.
Dopo aver determinato l'intervallo, i k-mer le cui frequenze
non rientrano in esso vengono ignorati nei passaggi successivi.

Scelta del valore k

Prima di calcolare i valori di fmin e fmax, LROD dovrebbe determinare il valore di k:

- Un valore maggiore di k aiuta a risolvere i problemi legati alla ripetizione, ma diminuisce il numero di k-mer comuni tra due letture lunghe.
- Un valore più piccolo di k introdurrà più k-mer comuni negativi e complicherà il processo di rilevamento della sovrapposizione.

LROD imposta k su 15 per impostazione predefinita.

Algoritmo, Pre-condizioni e Post-condizioni

1) Prima di chiamare LROD si deve installare DSK, un tool che prende in input un file di long reads e restituisce un file contente i kmer (solidi) con la frequenza associata.

```
1 AAAAAAAAAAAAAAA 1945
2 AAAAAAAAAAAAAAC 76
3 AAAAAAAAAAAAAAT 38
4 AAAAAAAAAAAAAAG 39
```

2) LROD restituisce un file csv che rappresenta la regione di overlap

Link al github: https://github.com/GATB/dsk

Algoritmo, Passi fondamentali

1. Trovare il set di k-mer comuni
2. Creazione di una catena di kmer consistenti
3. Determinarne la consistenza
4. Valutazione del candidato per la regione di overlap

Algoritmo, Passi fondamentali

1) Trovare il CKS

- Identificazione del common k-mer set tra R1 e R2. Ausilio di una tabella Hash
- Eliminazione dei k-mer duplicati e riordinamento (forward o reverse)
- Per applicare LROD, CKS deve contenere almeno 5 elementi

Algoritmo, Passi fondamentali

2) Creazione di una catena di k-mer

Identificazione di una **catena di k-mer consecutivi, comuni e consistenti** all'interno di CKS, che corrisponda ad una possibile sovrapposizione candidata tra le lunghe sequenze R1 e R2.

La catena di kmer consistenti è NULL all’inizio.
Due indici per lavorare sulla catena:

- start, primo kmer alla catena
- end, kmer successivo a start di cui bisogna valutare l’aggiunta o meno alla catena

Quindi determinare la consistenza tra start ed end

Algoritmo, Passi fondamentali

3) Determinare la consistenza

Quindi dal CKS che stiamo costruendo, si preleva la coppia di kmer start ed end e si valutano le condizioni nelle reads:

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

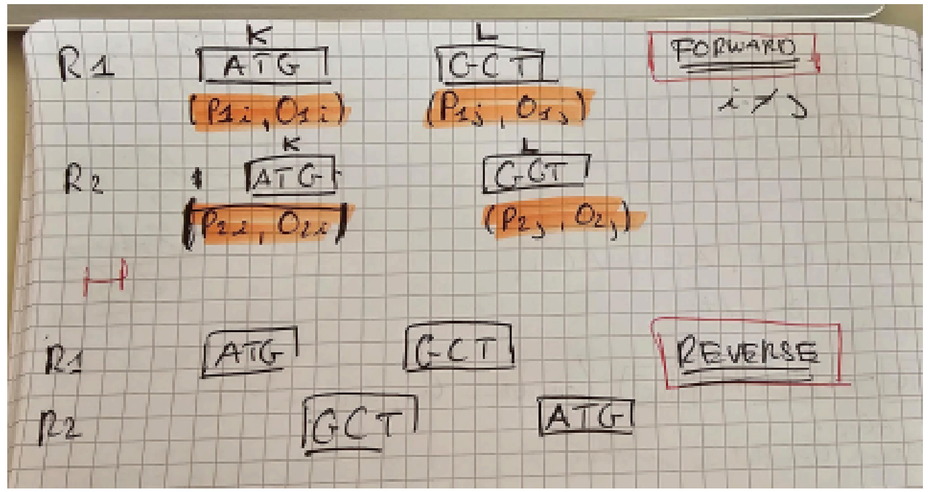
Input: common k-mers
Output: whether two common k-mers are consistent
Begin
Getting the positions of the start and end common k-mers:
$P_{1i}, P_{2i}, P_{1j}, P_{2j}$ ($P_{1i} < P_{1j}$);
$D_1 \leftarrow |P_{1j} - P_{1i}|$;
$D_2 \leftarrow |P_{2j} - P_{2i}|$;
if forward common k-mers **then**
  if ($P_{1i} < P_{1j}$ 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

Quattro condizioni:

1. **C1: P1i < P1j and P2i < P2j (kmer in forward)**
2. **C2: P1i < P1j and P2i > P2j; (reverse)**
3. **D1 < α and D2 < α; (α= 400) soglia**
4. **C4: (Max(D1, D2) – Min(D1,D2)) / Max(D1 ,D2) < γ (y=0.3) limite alla differenza tra le distanze**

Algoritmo, Passi fondamentali

3) Determinare la consistenza



Quattro condizioni:

1. **C1: P1i < P1j and P2i < P2j (kmer in forward)**
2. **C2: P1i < P1j and P2i > P2j; (reverse)**
3. **D1 < α and D2 < α; (α= 400) soglia**
4. **C4: (Max(D1, D2) – Min(D1,D2)) / Max(D1 ,D2) < γ (γ=0.3) limite alla differenza tra le distanze**

Algoritmo, Passi fondamentali

4) Valutazione del candidato per la regione di overlap

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

Algoritmo, Passi fondamentali

4) Valutazione del candidato per la regione di overlap

Condizioni

1. **(EP1 − SP1) − (Pn +k−P1) < α:** controlla se la lunghezza della sovrapposizione calcolata su R1 è vicina alla lunghezza effettiva dell'overlap. Se questa differenza è minore di α (400), allora la sovrapposizione su R1 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:** controlla quanto le lunghezze massime e minime della sovrapposizione differiscono. Se questa differenza è inferiore a un certo valore γ (0.3), allora la sovrapposizione è considerata coerente (per entrambe le reads)

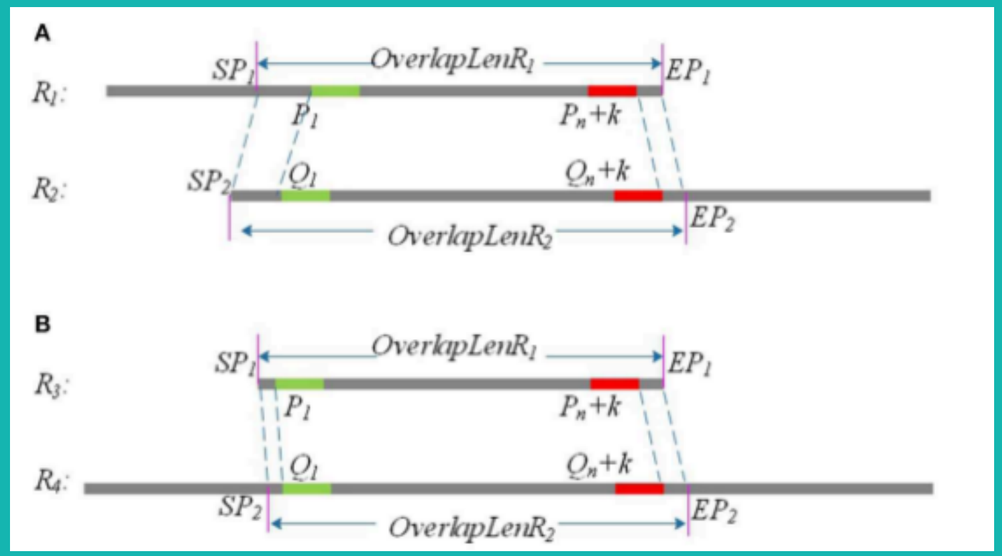
Se le condizioni sono soddisfatte, abbiamo un overlap significativo e si restituisce la regione.

Algoritmo, Passi fondamentali

4) Valutazione del candidato per la regione di overlap

Overlap parziale

Overlap totale



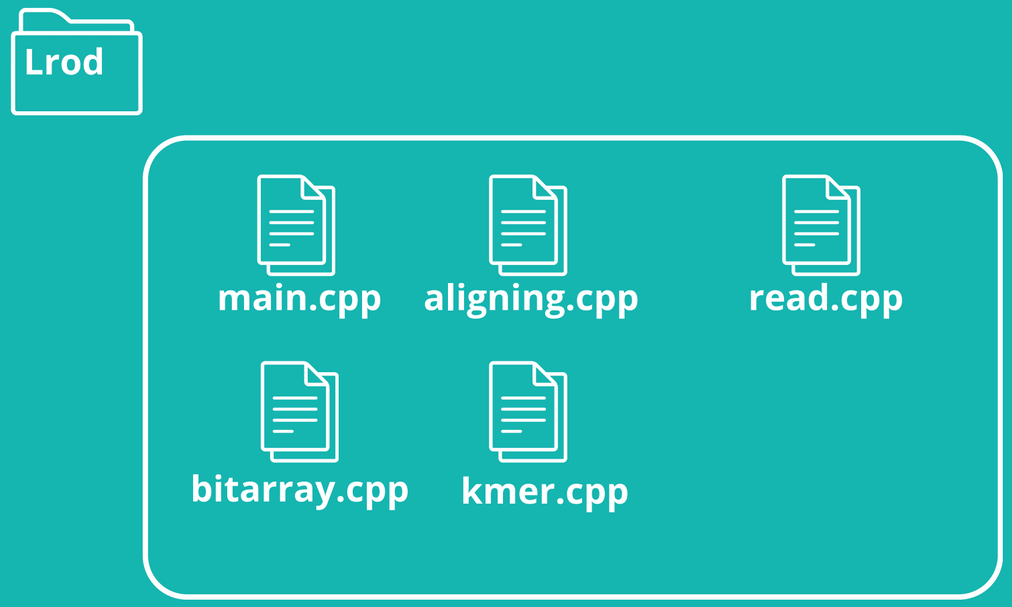
- P1 e Q1 sono le start positions dei positive common k-mers su R1 e R2 rispettivamente. **P1 e Q1 sono il primo k-mer della catena di k-mer calcolata.**
- Pn+k e Qn+k sono le end positions dei positive common k-mers su R1 e R2. **Pn e Qn sono gli ultimi k-mer della catena di k-mer calcolata.**

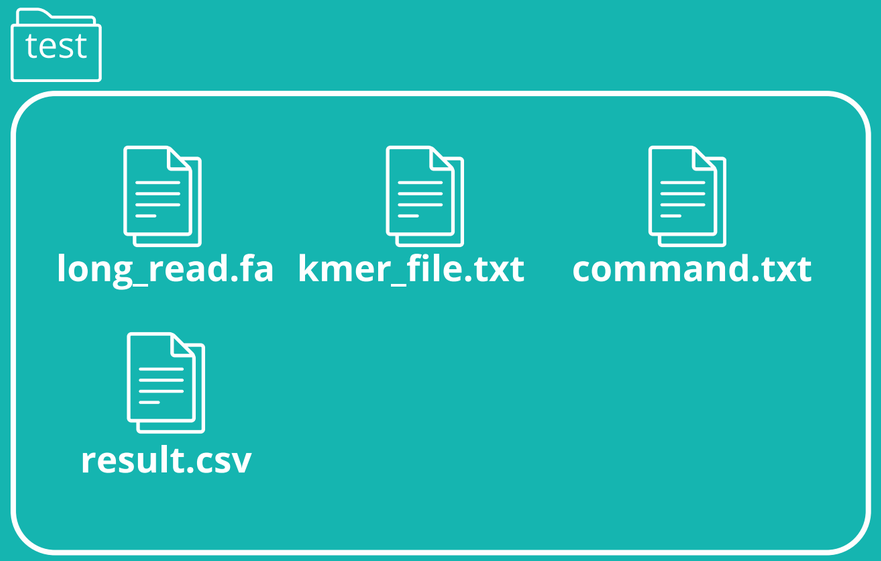
- SP1 e EP1 sono la start position e la end position della regione di overlap su R1;
- SP2 e EP2 sono la start position e la end position della regione di overlap su R2

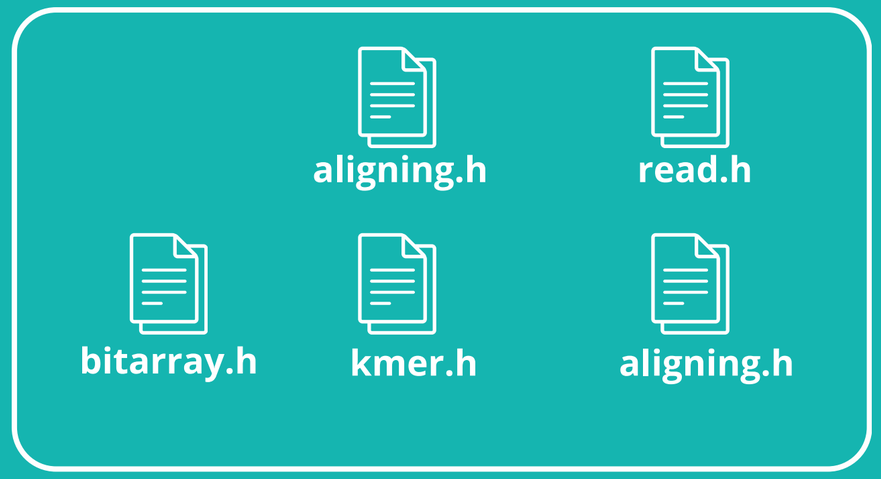
Implementazione: fasi

1. Pre-processing reads, kmer e kmerHashTable, kmerReadNode
2. Pre-processing threads e strutture, CommonKmer, grafi
3. Allineamento

Implementazione: Struttura del progetto







Makefile nella cartella Lrod per la compilazione delle librerie e ottenimento dell'eseguibile **LROD**

Implementazione: fasi

1)Pre-processing **kmerHashTable**

Costruzione di una tabella hash per ricercare i kmer.

Coppia chiave valore: H(hashIndex) -> kmerInteger

dove hashIndex = SetBitkmer(kmerReale:AAAT).

La chiave per accedere al k-mer è il k-mer stesso.

Criteri per inserimento:

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

Altri campi:

- posizione all’interno della read
- numero di nodi allocati
- max e min

```c
//nodo della kmer hash
typedef struct KmerHashNode{
    unsigned int kmer;
    int startPositionInArray;
    //unsigned int fre;
}KmerHashNode;

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

Implementazione: fasi

1)Pre-processing **kmerRead** (struttura di appoggio)

Si analizzano 50k reads alla volta e :

```
typedef struct KmerReadNode{
    unsigned int kmer;
    unsigned int readIndex; // Indica
    unsigned int position;  //Indicat
    bool orientation;  //Indicates th
}KmerReadNode;

typedef struct KmerReadNodeHead{
    KmerReadNode * kmerReadNode;
    long int realCount;
    long int allocationCount;
    long int kmerLength;
    long int startReadIndex;
    long int endReadIndex;
}KmerReadNodeHead;
```

- per ogni read e per ogni kmer nella read si verifica la presenza nella tabella hash.
- Se presente, si aggiunge un nodo KmerReadNode in posizione realCount, inserendo il kmer, **readIndex**, **posizione del kmer nella read**...
- Si aggiorna anche la KmerHashTable.startPositionInArray= posizione del kmer
- **Obiettivo**: salvare tutti i kmer (di tutte le reads) presenti nella hashtable, per determinare il commonKmerSet.

Implementazione: fasi

2)Pre-processing **threads e strutture**

- Creazione array di threads
- Creazione della struttura dati utilizzata da ogni threads
- Divisione del carico di lavoro, quindi delle reads.
- Ogni thread esegue **GetCommonKmerHeadThread**
- Infine ogni threads dà in output il proprio risultato nel proprio file, e poi si fonderanno in un unico file result.csv

```
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;
```

Implementazione: fasi

2)Pre-processing **grafi**

funzione **GetCommonKmerHeadTHread**

- alloca un array di grafi AdjGraphHead
- alloca un array di grafi reverse
- alloca un array di archi
- lista di CommonKmer

```
typedef struct AdjGraphHead{
    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;
    long int * edgeToSource;
    bool * visited;
    long int largestIntervalDistance;
    long int nodeCount;
    long int leftStart;
    long int rightStart;
    long int leftEnd;
    long int rightEnd;
    char * localLeftRead;
    char * localRightRead;
    long int smallKmerLength;
    long int kmerLength;
    float lengthRatio;
    long int overlapLengthCutOff;
}AdjGraphHead;
```

```
typedef struct ArcIndex{
    long int startIndex;
    long int endIndex;
    float weight;
}ArcIndex;

typedef struct AdjGraph{
    long int dataLeft; //assumono
    long int dataRight;//position
    bool visit;
}AdjGraph;
```

Implementazione: fasi

```
typedef struct CommonKmer{
    long int readIndex;
    long int leftPosition;
    long int rightPosition;
    bool orientation;
}CommonKmer;

typedef struct CommonKmerHead{
    CommonKmer * commonKmer;
    long int readIndex;
    long int realCount;
    long int allocationCount;
}CommonKmerHead;
```

2)Pre-processing **CommonKmer**

funzione **GetCommonKmerHeadTHread**

chiama **InsertCommonToTwoReadAligningHead**

- Per ogni read nell'intevallo (50k reads alla volta)
- preleva un kmer alla volta, verifica la presenza nella tabella hash
- Se presente, sfruttando la **kmerReadNode** come riferimento, riempie la commonKmer:
  - CK.leftPosition=posizione del k-mer nella read considerata
  - CK.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, passo al k-mer successivo

- terminata la i-esima read e riempita la lista di **commonkmerHead** (contiene gli indici delle read), rimuove i kmer duplicati e la riordina

Implementazione: fasi

3) Allineamento

La funzione **GetCommonKmerHeadThread** chiama sull'i-esima read, alla fine, **GetOverlapResult** e in input:

- G -> lista di grafi
- CK -> la lista di kmerComuni trovati fino a quel momento
- localG -> grafo locale alla read

Itera su CK trovai all'-iesima read:
Se valuta CK readIndex in un'altra read, diversa da quella in cui è stata trovato il common kmer:

- verifica la presenza di almeno 5 elementi nel grafo
- verifica l'orientamento
- chiama **createGraphSinglePath**

```
typedef struct CommonKmer{
    long int readIndex;
    long int leftPosition;
    long int rightPosition;
    bool orientation;
}CommonKmer;

typedef struct CommonKmerHead{
    CommonKmer * commonKmer;
    long int readIndex;
    long int realCount;
    long int allocationCount;
}CommonKmerHead;
```

Implementazione: fasi

3) Allineamento

Aggiornamento dei nodi nel grafo i-esimo

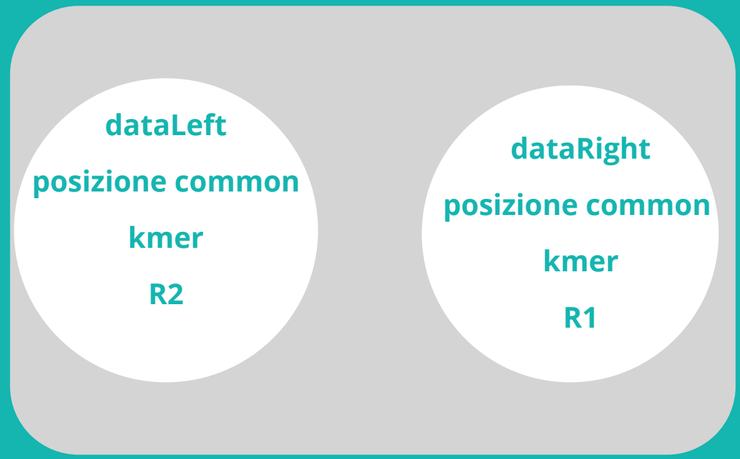
- AdjGraph[i].dataLeft = CK.leftPosition (R2) ->KmerRead
- AdjGraph[i].dataRight= CK.rightPosition (R1) ->Read
- terminata la i-esima read e riempita la lista di commonkmer (contiene gli indici delle read),
- rimuove i kmer duplicati e la riordina

Di volta in volta si calcolano i kmer comuni tra le read prese in considerazione.

```
typedef struct CommonKmer{
    long int readIndex;
    long int leftPosition;
    long int rightPosition;
    bool orientation;
}CommonKmer;
```

```
typedef struct ArcIndex{
    long int startIndex;
    long int endIndex;
    float weight;
}ArcIndex;

typedef struct AdjGraph{
    long int dataLeft; //assumono
    long int dataRight;//position
    bool visit;
}AdjGraph;
```



Singolo grafo

Implementazione: fasi

3) Allineamento
Calcolati i commonKmer, determinare overlap sulla read i-esima

```
typedef struct ArcIndex{
    long int startIndex;
    long int endIndex;
    float weight;
}ArcIndex;

typedef struct AdjGraph{
    long int dataLeft; //assumono
    long int dataRight;//position
    bool visit;
}AdjGraph;
```

```
typedef struct AdjGraphHead{
    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;
    long int * edgeToSource;
```

- **GetOverlapResult** (chiamata per ogni read)
- Itera sui common kmer trovati alla i-esima read
- Valuta la consistenza di ciascun kmer attraverso la funzione **createGraphSinglePath (determine_consistent_1)**

Implementazione: fasi

3) Allineamento

Determinare una catena di kmer consistenti

Sia ck= [ck1,ck2,ck3,ck4,ck5…]

- Sulla lista di common kmer si **verificano le condizioni di consistenza di determine_consistent_1 (alg 4)**
- Se sono rispettate, tra ck1 e ck2, si aggiunge un arco tra ck1 e ck2. Altrimenti
- Si confronta ck1 con ck3 e se sono rispettate si traccia l’arco tra ck1 e ck3.

Si continua da ck3

Ci devono essere almeno 2 kmer consistenti nella catena , quindi almeno un arco.

```
typedef struct ArcIndex{
    long int startIndex;
    long int endIndex;
    float weight;
}ArcIndex;

typedef struct AdjGraph{
    long int dataLeft; //assumono
    long int dataRight;//position
    bool visit;
}AdjGraph;
```

```
typedef struct AdjGraphHead{
    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;
    long int * edgeToSource;
```

Implementazione: fasi

3) Allineamento

Calcolata la catena, si procede alla fase di determinazione dell'overlap (sull'i-esima read)

```
typedef struct ArcIndex{
    long int startIndex;
    long int endIndex;
    float weight;
}ArcIndex;

typedef struct AdjGraph{
    long int dataLeft; //assumono
    long int dataRight;//position
    bool visit;
}AdjGraph;
```

Funzione overlap_display_graph(riferimento algoritmo 6)

- Verifica delle condizioni 1,2,3
- Verifica i casi di allineamento
- Scrive i risultati nel file (gestito dal thread)
- Infine si fondono i risultati di ogni thread.

```
typedef struct AdjGraphHead{
    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;
    long int * edgeToSource;
```

Risultato overlap

File.csv

```
1  113,1,1,1,2528,2528,321,2932,7610
2  175,1,1,1,3860,6789,1,3896,7610
3  220,1,1,1,1750,4621,1,1746,7610
4  433,3,1,1,3706,3706,1005,4545,4653
5  225,5,0,1688,3205,3205,1,1565,6292
6  122,6,1,1371,4075,4445,1,2654,2654
7  136,6,0,502,3115,6632,1,2654,2654
8  469,6,0,1206,3240,3240,1,1822,2654
9  98,8,0,1,1435,2738,1460,2867,2867
10 192,8,0,290,2669,2669,1,2377,2867
```

- **se (leftIndex > rightIndex) i parametri sono stampati nel seguente ordine:**

leftIndex, rightIndex, orien, leftStartpos, leftEndpos, leftLen, rightStartpos, rightEndpos, rightLen

- **altrimenti: rightIndex,leftIndex,orien, rightStartpos, rightEndpos, rightLen, leftStartpos, leftEndpos, leftLen**

Risultato overlap (File.csv)

1. **leftIndex = commonKmerHead->readIndex; (indice read corrente R1)**
2. **rightIndex = commonKmerHead->commonKmer[0].readIndex;**
3. **orien: forward (0) o reverse (1)**
4. **leftStartpos = graph[iniStartIndex].dataLeft (read corrente R1)**
5. **leftEndpos = graph[lastEndIndex].dataLeft**
6. **leftLen = readSetHead->readSet[leftIndex - 1].readLength;**
7. **rightStartpos = graph[iniStartIndex].dataRight (read R2)**
8. **rightEndpos = graph[lastEndIndex].dataRight**
9. **rightLen = readSetHead->readSet[rightIndex - 1].readLength;**

Risultati del paper

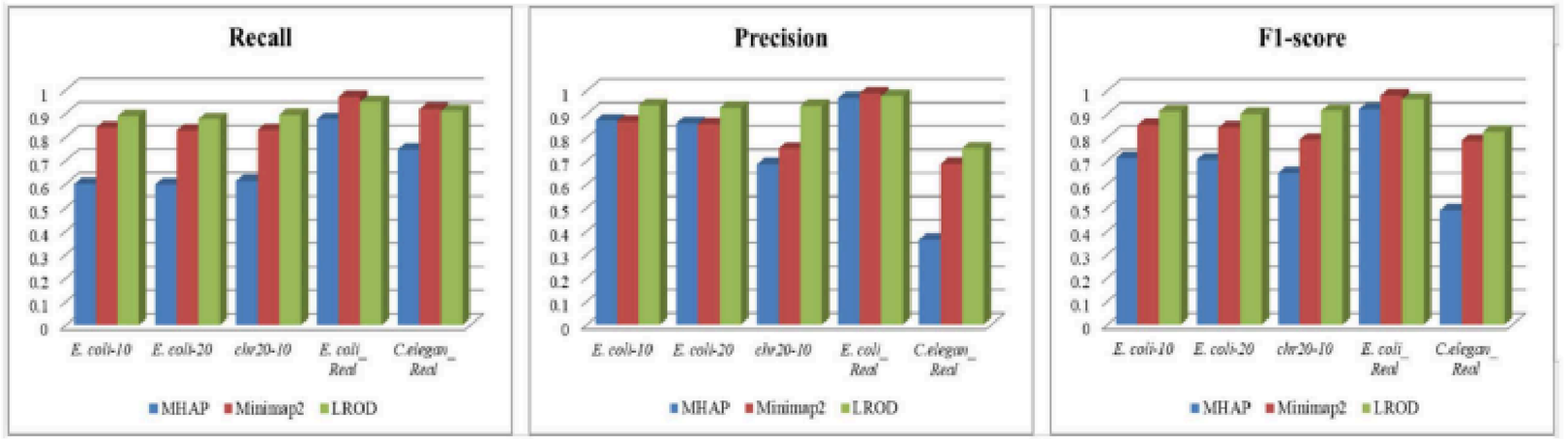


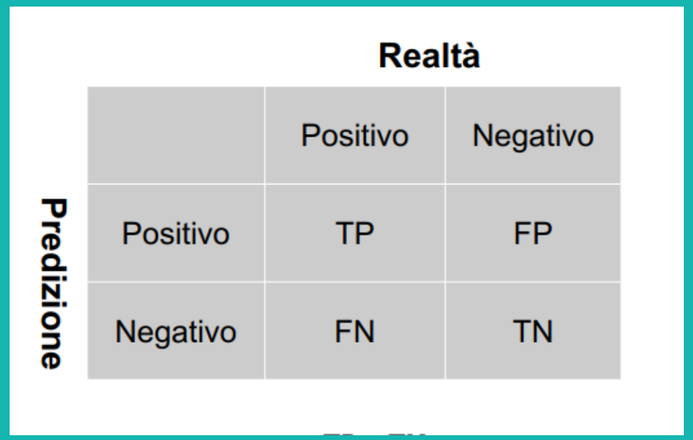
FIGURE 3 | Overlap detection with $k = 13$.

TABLE 3 | Running time and memory.

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

Metriche usate per la valutazione dell’algoritmo

- **Recall** indica il rapporto diistanze positive correttamente individuate (sul totale dei casi)
- **Precision** è l’accuratezza della predizione delle classi positive
- **F1 score** media armonica di precision e recall. Attribuisce un peso maggiore ai valori piccoli. Questò fa sì che un classificatore ottenga un alto punteggio F1 solo quando precisione e recupero sono entrambi alti. **(wikipedia e slides cyber security)**



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

$$\text{Precisione} = \frac{veropositivo}{veropositivo + falsopositivo}$$

$$\text{Recupero} = \frac{veropositivo}{veropositivo + falsonegativo}$$

$$F_1 = \frac{2}{\frac{1}{\mathrm{r}} + \frac{1}{\mathrm{p}}} = 2 \cdot \frac{\mathrm{p} \cdot \mathrm{r}}{\mathrm{p} + \mathrm{r}}.$$

Metriche usate per la valutazione dell’algoritmo

- **Tempo di esecuzione (Totale, considerando il pre-processing)**
- **Ram utilizzata**
- **Lunghezza k-mers**
- **Dataset usato (reale o generato)**

Risultati del paper

- **Parametri:**
  - **HW = 128 gb, kmer-lenght=15-13, 10 threads**

Lrod vince in quasi tutti i casi in termini di **Recall, Precision, F1-score,** rispetto a MHAP e Minimap2. MHAP risulta il peggiore in ogni ambito.

Rispetto a Minimap2, Lrod risulta essere piu oneroso in termini di utilizzo di memoria e tempo di esecuzione.

Punti critici getReadSetHead in read.cpp

Il file long_read.fa contente le 500 long reads separate dal carattere “>” viene aperto due volte:

1. La prima volta per incrementare il contatore delle long reads
2. La seconda volta scorre per eliminare il carattere di terminazione “/n” o “/r” e per popolare la struttura readSetHead

Punti critici: getKmerHashTableHead in kmer.cpp

Il file kmer_file.txt contente i 131.071 kmer con le frequenze associate viene aperto per tre volte:

1. La prima volta copia solo le frequenze e le inserisce nell'array kmer; analizza le frequenze dei kmer e fa il break se almeno un numero della frequenza è diverso. Il problema è che effettua un confronto all'interno di un ciclo for che itera da zero a kmerLength dove kmerLength = 15 ma kmer contiene al massimo 4 numeri. Quindi, anche leggendo 1111 restituisce sempre TRUE.
2. La seconda volta si copia solo il kmer ed effettua correttamente il confronto sui caratteri del kmer considerando solo quelli che differiscono almeno in un carattere per ottenere il conteggio totale dei kmer, tenendo conto anche dell’intervallo delle frequenze.
3. La terza volta effettua di nuovo il confronto del punto precedente e poi converte i kmer in bytes e li memorizza in una tabella hash.

Punti critici:

Operazioni ripetitive legate alla kmerHashTable, effettuate durante:

- **Costruzione della tabella stessa**
- **Inserimento dei k-mers nella KmerReadNode**
- **Inserimento dei k-mers nella CommonKmer**

Sono:

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

Punti critici:

Mancanza di un controllo sulla presenza del kmer in reverse, all’interno della hash table.

Conseguenza: esecuzione inutile della funzione **InsertCommonToTwoReadAligningHead**

Funzione: GetCommonKmerHeadThread **(riga 473)**

Punti critici: memoria e tempo di esecuzione

Allocazione di molte strutture:

1. **ReadSetHead, ReadSet**
2. **KmerHashTableHead, KmerHashNode**
3. **KmerReadNodeHead, KmerReadNode**
4. **CommonKmerHeadArray, CommonKmerHead, CommoKmer**
5. **AdjGraphHead, AdjGraph, ArcIndex**
6. **GetCommonmerHeadP**

Altri punti critici

1. Mancanza di commenti
2. Nomi delle variabili fuorvianti
3. Utilizzo di molte strutture dati, memorizzazione delle reads, dei kmer, dei common k-mer ...
4. Presenza di molti cicli innestati che scorrono le varie strutture e le aggiornano
5. Non utilizzo del C++ e di librerie, ma solo del C puro.
    a. Questo potrebbe essere un vantaggio dal punto di vista prestazionale (anche se non sfruttato);
    b. Potrebbe essere uno svantaggio in termini di leggibilità del codice.
6. Utilizzo di un tool esterno come **DSK** per ottenere i k-mers solidi.

Commenti finali

“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 LROD, aumentando la velocità di calcolo e riducendo il tempo di esecuzione”