Integrazione e Ottimizzazione dell'Algoritmo LROD per il Rilevamento delle Sovrapposizioni nelle Letture Lunghe: Un Approccio Senza Dipendenze Esterne

Manuel Sica matr. 0522501870

Università degli Studi di Salerno, Via Giovanni Paolo II, 132, Fisciano, SA, 84084, Italia

ARTICLE INFO

Keywords:
rilevamento sovrapposizioni
k-mer
sequenziamento di terza generazione
DSK
LROD

ABSTRACT

Le tecnologie di sequenziamento di terza generazione (TGS) come SMRT (Single Molecule Real-Time) e ONT (Oxford Nanopore Technologies) producono letture molto lunghe, cruciali per l'assemblaggio del genoma. Tuttavia, queste letture sono affette da un alto tasso di errore di sequenziamento, complicando il rilevamento accurato delle sovrapposizioni. L'algoritmo LROD (Long Read Overlap Detection) è stato progettato per migliorare l'accuratezza del rilevamento delle sovrapposizioni tra queste letture lunghe attraverso l'analisi della distribuzione dei k-mer. Questo studio descrive le modifiche apportate all'algoritmo LROD per incrementarne l'efficienza e per eliminare la dipendenza dal software DSK, integrando il calcolo delle frequenze dei k-mer direttamente all'interno dell'algoritmo, rendendolo un approccio più autonomo e efficiente per il rilevamento delle sovrapposizioni nelle letture lunghe.

1. Introduzione

Il rilevamento delle regioni sovrapposte tra lunghe letture di DNA è un passaggio cruciale nell'assemblaggio del genoma. Le tecnologie di sequenziamento di terza generazione (TGS) come SMRT (Single Molecule Real-Time) e ONT (Oxford Nanopore Technologies) offrono letture molto lunghe che sono fondamentali per questo compito. Tuttavia, queste tecnologie sono caratterizzate da un alto tasso di errore, rendendo il rilevamento accurato delle sovrapposizioni una sfida significativa.

L'algoritmo LROD (Long Read Overlap Detection) [1] è stato sviluppato per migliorare l'accuratezza del rilevamento delle sovrapposizioni sfruttando la distribuzione dei k-mer. Originariamente, LROD richiedeva l'uso del software DSK per calcolare le frequenze dei k-mer, un passaggio necessario prima di eseguire il rilevamento delle sovrapposizioni.

In questo articolo, ci concentreremo sulle modifiche apportate all'algoritmo LROD per migliorare la sua efficienza e per eliminare la dipendenza dal software DSK. L'integrazione del calcolo delle frequenze dei k-mer direttamente all'interno di LROD rappresenta un passo significativo verso un approccio più autonomo e ottimizzato per il rilevamento delle sovrapposizioni nelle lunghe letture. Inoltre, verranno presentati i risultati dei benchmark che dimostrano le performance migliorate del nuovo algoritmo rispetto alla versione originale.

2. Stato dell'Arte

Il rilevamento delle sovrapposizioni è un problema ben noto nell'assemblaggio del genoma. Strumenti come MHAP [2] e Minimap2 [3] sono ampiamente utilizzati per questo scopo. MHAP utilizza un approccio basato su MinHash per rilevare sovrapposizioni, mentre Minimap2 sfrutta i minimizer per rappresentare le sequenze in modo compatto.

BLASR [4] è stato progettato per allineare lunghe letture a un genoma di riferimento e può essere utilizzato per rilevare sovrapposizioni tra letture lunghe. DALIGNER [5] è un altro strumento specificamente progettato per rilevare sovrapposizioni, concentrandosi sull'ottimizzazione dell'efficienza di esecuzione.

LROD si distingue da questi strumenti grazie al suo utilizzo della distribuzione dei k-mer per migliorare l'accuratezza del rilevamento delle sovrapposizioni, specialmente in presenza di alti tassi di errore di sequenziamento.

2.1. Definizione di k-mer

Un **k-mer** [6] è una sottosequenza di lunghezza k derivata da una sequenza di DNA, RNA o proteine. Nel contesto del sequenziamento genomico, i k-mer sono utilizzati per rappresentare frammenti di sequenze più lunghe.

I k-mer hanno diversi utilizzi nei sequenziamenti:

- **Identificazione delle Somiglianze tra Sequenze:** I k-mer sono utilizzati per confrontare le sequenze genomiche e identificare somiglianze e differenze.
- **Assemblaggio del Genoma:** Nell'assemblaggio del genoma, i k-mer aiutano a ricostruire la sequenza completa del DNA da letture corte, identificando le regioni sovrapposte.
- **Errori di Sequenziamento:** I k-mer possono essere utilizzati per identificare e correggere errori di sequenziamento.

La scelta del valore di k è cruciale. Un valore troppo piccolo può non catturare sufficientemente le caratteristiche uniche della sequenza, mentre un valore troppo grande può ridurre il numero di k-mer comuni rilevati tra le sequenze, rendendo più difficile il rilevamento delle sovrapposizioni.

2.1.1. k-mer Solidi

I **k-mer solidi** [7] sono quei k-mer che hanno una frequenza compresa tra due soglie predefinite, $f_{\min}$ e $f_{\max}$. Questi k-mer sono considerati affidabili per l'analisi perché:

- **Frequenze basse** (inferiori a $f_{\min}$) possono indicare errori di sequenziamento.
- **Frequenze alte** (superiori a $f_{\max}$) possono indicare regioni ripetitive del genoma.

Selezionando solo i k-mer con frequenze all'interno di questo intervallo, si riduce l'impatto degli errori di sequenziamento e delle regioni ripetitive.

3. Algoritmo LROD

L'algoritmo si articola in tre fasi principali:

1. **Selezione dei k-mer Solidi:** Si scelgono i k-mer con frequenze comprese tra due soglie, $f_{\min}$ e $f_{\max}$, per ridurre l'impatto degli errori di sequenziamento e delle regioni ripetitive.
2. **Costruzione della Catena di k-mer:** Si cerca una catena di k-mer consistenti che rappresentano una potenziale sovrapposizione tra due letture.
3. **Valutazione delle Sovrapposizioni:** Si utilizzano due fasi per determinare la consistenza dei k-mer e si revisionano le sovrapposizioni candidate per ottenere quelle reali.

3.1. Selezione dei k-mer Solidi

Obiettivo: Ridurre l'impatto degli errori di sequenziamento e delle regioni ripetitive selezionando solo i k-mer con frequenze adeguate.

Passaggi:

1. **Calcolo della Frequenza dei k-mer:** Utilizzando strumenti come DSK [8], si calcola la frequenza di ogni k-mer all'interno del dataset di letture lunghe.
   - **DSK:** È un software veloce e leggero per calcolare la frequenza dei k-mer in grandi dataset genomici.
   - **Frequenza dei k-mer:** Rappresenta quante volte un k-mer specifico appare nel dataset di letture.
2. **Determinazione degli Intervalli di Frequenza:** I k-mer con frequenze al di fuori di un intervallo predefinito $[f_{\min}, f_{\max}]$ vengono eliminati.
   - $[f_{\mathbf{min}}, f_{\mathbf{max}}]$**:** Questi parametri definiscono l'intervallo di frequenza accettabile per considerare un k-mer come "solido".
3. **Costruzione della Tabella Hash dei k-mer:** I k-mer solidi vengono indicizzati in una tabella hash per una ricerca rapida durante il confronto tra letture.
   - **Tabella Hash:** Una struttura dati che permette una ricerca rapida dei k-mer comuni tra le letture.

3.2. Rilevamento delle Sovrapposizioni

Obiettivo: Identificare i k-mer comuni tra due letture e costruire catene di k-mer consistenti che rappresentano potenziali sovrapposizioni.

Passaggi:

1. **Rilevamento dei k-mer Comuni:** Identificare il set comune di k-mer (CKS) tra due letture.
2. **Rimozione dei k-mer Comuni Ripetitivi:** Eliminare i k-mer comuni che si ripetono troppe volte per evitare ambiguità.
3. **Creazione di Catene di k-mer Consistenti:** Costruire una catena di k-mer comuni consistenti che rappresentano una potenziale sovrapposizione tra le letture.

3.3. Valutazione delle Sovrapposizioni Candidate

Obiettivo: Valutare la consistenza dei k-mer comuni e determinare le sovrapposizioni reali tra le letture.

Passaggi:

1. **Strategia in Due Fasi:**
   - **Fase 1: Distanza tra i k-mer:** Si verifica se la distanza tra i k-mer è consistente in entrambe le letture.
   - **Fase 2: Posizioni e Orientamenti:** Si controllano le posizioni e gli orientamenti dei k-mer per confermare la sovrapposizione.
2. **Revisione delle Sovrapposizioni Candidate:** Le sovrapposizioni identificate vengono riviste e corrette per eventuali deviazioni dovute agli errori di sequenziamento.

3.4. Determinazione della Sovrapposizione Finale

Obiettivo: Ottenere una sovrapposizione precisa e corretta tra le letture lunghe.

Passaggi:

1. **Modifica delle Sovrapposizioni Candidate:** Le sovrapposizioni candidate vengono ulteriormente modificate per correggere eventuali errori e ottenere la sovrapposizione finale.
2. **Determinazione della Sovrapposizione Finale:** Viene determinata la sovrapposizione finale tra le due letture lunghe, che rappresenta la regione comune tra le letture.
3. **Validazione della Sovrapposizione:** La sovrapposizione finale viene validata per assicurarsi che sia biologicamente rilevante e che corrisponda alle attese sulla struttura genomica.

4. Modifiche al codice LROD

Il codice presentato dagli autori [1] evidenzia delle lacune in termini di ottimizzazione, sia dal punto di vista della memoria che delle prestazioni. In questa sezione verranno esplorati i codici modificati [9] per affrontare queste problematiche.

4.1. Modifiche nel file `read.cpp`

Il file `read.cpp` conteneva una funzione `getReadSetHead` che presentava un'inefficienza significativa nell'apertura e lettura del file contenente le lunghe letture (`long_read.fa`) in due passaggi distinti. Questa doppia operazione comportava un inutile overhead di I/O, rallentando l'intero processo.

4.1.1. Descrizione del Problema

La funzione originale `getReadSetHead` eseguiva le seguenti operazioni[10]:

1. Apertura del file `long_read.fa` e conteggio del numero di letture presenti nel file (righe da 36 a 49).
2. Chiusura del file.
3. Riapertura del file `long_read.fa` per eliminare i caratteri di terminazione (`\n` o `\r`) e popolare la struttura `readSetHead` con le letture (righe da 58 a 82).

Questa doppia lettura del file non era necessaria e introduceva un overhead aggiuntivo.

4.1.2. Risoluzione del Problema

La funzione `getReadSetHead` è stata riscritta per eseguire entrambe le operazioni (conteggio delle letture ed eliminazione dei caratteri di terminazione) in un unico ciclo di lettura del file.

Le modifiche principali apportate sono le seguenti:

1. La funzione ora apre il file una sola volta.
2. Durante la lettura del file, la funzione conta il numero di letture e contemporaneamente popola la struttura `readSetHead`.
3. È stata introdotta una logica per gestire la riallocazione dinamica della memoria per il `readSet`. Inizialmente, la memoria viene allocata per un numero di letture predefinito (ad esempio 100), e viene incrementata dinamicamente in blocchi di 100 letture se necessario, riducendo così il numero di riallocazioni.
4. Sono stati aggiunti controlli aggiuntivi per la corretta gestione della memoria e per assicurarsi che tutte le letture siano gestite in modo appropriato, evitando potenziali fughe di memoria.

4.1.3. Dettagli dell'Implementazione

La nuova funzione `getReadSetHead` è implementata come segue:

```
ReadSetHead * GetReadSetHead(char *filename, char *StrLine, long int maxSize) {
    printf("\nfunzione GetReadSetHead\n");
    ReadSetHead * readSetHead = (ReadSetHead *)malloc(sizeof(ReadSetHead));
    if (!readSetHead) {
        printf("Memory allocation error!\n");
        return NULL;
    }

    readSetHead->readSet = NULL;
    readSetHead->readCount = 0;

    FILE *fp;
    if ((fp = fopen(filename, "r")) == NULL) {
        printf("error opening file!\n");
        free(readSetHead);
        return NULL;
    }

    long int readIndex = -1;
    long int allocatedReads = 0;

    // Conta le reads e popola la struttura readSetHead in un unico ciclo
    while ((fgets(StrLine, maxSize, fp)) != NULL) { // legge maxSize caratteri in ogni ciclo
        if (StrLine[0] == '>') { // se il primo carattere corrisponde a > -> e viene rilevata una read -> incrementa il contatore
            readSetHead->readCount++;
            readIndex++;
            if (readIndex >= allocatedReads) {
                // Rialloca memoria per readSet
                allocatedReads += 100; // incrementa di 100 alla volta per ridurre le riallocazioni
                readSetHead->readSet = (ReadSet *)realloc(readSetHead->readSet, sizeof(ReadSet) * allocatedReads);
                if (!readSetHead->readSet) {
                    printf("Memory allocation error!\n");
                    fclose(fp);
                    free(readSetHead);
                    return NULL;
                }
                // Inizializza le nuove reads allocate
                for (long int i = readIndex; i < allocatedReads; i++) {
                    readSetHead->readSet[i].readLength = 0;
                    readSetHead->readSet[i].read = NULL; // Inizializza i puntatori a NULL
                }
            }
            continue;
        }

        // Elimina il carattere di terminazione \n o \r
        long int readLength = strlen(StrLine);
        while (readLength > 0 e (StrLine[readLength - 1] == '\n' o StrLine[readLength - 1] == '\r')) {
            readLength--;
        }
        StrLine[readLength] = '\0'; // Aggiunge il terminatore nullo

        // Popolare la struttura readSetHead
        if (readIndex >= 0 e readIndex < readSetHead->readCount) {
            readSetHead->readSet[readIndex].readLength = readLength;
            readSetHead->readSet[readIndex].read = (char *)malloc(sizeof(char) * (readLength + 1));
            if (!readSetHead->readSet[readIndex].read) {
                printf("Memory allocation error!\n");
                fclose(fp);
                for (long int i = 0; i <= readIndex; i++) {
```

```
                free(readSetHead->readSet[i].read);
            }
            free(readSetHead->readSet);
            free(readSetHead);
            return NULL;
        }
        strncpy(readSetHead->readSet[readIndex].read, StrLine, readLength);
        readSetHead->readSet[readIndex].read[readLength] = '\0';
    }
}

fclose(fp);

printf("Number of long reads: %ld;\n", readSetHead->readCount);

return readSetHead;
}
```

Questa nuova implementazione riduce significativamente l'overhead di I/O, migliorando l'efficienza complessiva della funzione e semplificando il codice. La gestione della memoria è stata migliorata per evitare fughe e garantire che tutte le letture siano correttamente allocate e inizializzate.

4.2. Modifiche nel file `kmer.cpp`

Nel file `kmer.cpp` la funzione `GetKmerHashTableHead` apriva tre volte il file `kmer_file.txt` contenente i k-mer con le frequenze associate ed eseguiva dei confronti servendosi della funzione `DetectSameKmer`. Questo processo inefficiente è stato migliorato per evitare le aperture multiple del file e ottimizzare il flusso di lavoro.

4.2.1. Descrizione del Problema

La funzione originale `GetKmerHashTableHead` presentava le seguenti problematiche [10]:

1. **Prima Apertura del File:** Copiava solo le frequenze (`kmerF`) dei k-mer e le analizzava. Faceva un break se almeno un carattere della frequenza era diverso, ma non escludeva correttamente i k-mer con frequenze formate da cifre uguali (ad esempio, 111, 222) perché la funzione `DetectSameKmer` confrontava erroneamente la lunghezza dei k-mer (15 caratteri) con la lunghezza delle frequenze (massimo 4 cifre).
2. **Seconda Apertura del File:** Copiava solo i k-mer (senza la frequenza) ed eseguiva correttamente il confronto sui caratteri del k-mer, escludendo quelli che differivano in almeno un carattere, ottenendo il conteggio totale dei k-mer.
3. **Terza Apertura del File:** Eseguiva nuovamente il confronto del punto precedente, poi convertiva i k-mer in bytes e li memorizzava in una tabella hash.

4.2.2. Risoluzione del Problema

Le modifiche apportate hanno ottimizzato il codice in modo che il file `kmer_file.txt` venga aperto una sola volta, eseguendo tutti i passaggi necessari in un'unica lettura. Questo ha comportato i seguenti miglioramenti:

1. **Lettura Unica del File:** Il file viene letto una sola volta, riducendo il tempo di I/O e migliorando l'efficienza.
2. **Eliminazione dei k-mer con Frequenze Non Valide:** Durante la lettura del file, i k-mer con frequenze formate da cifre uguali vengono correttamente esclusi.
3. **Popolazione della Tabella Hash:** I k-mer validi vengono convertiti in bytes e memorizzati nella tabella hash durante la stessa lettura del file.

4.3. Dettagli dell'Implementazione

Il nuovo flusso di lavoro per la funzione `GetKmerHashTableHead` è stato implementato come segue:

```
KmerHashTableHead* GetKmerHashTableHead(const char* address, ReadSetHead* readSetHead, long int kmerLength, long int step, long int min, float maxRatio) {
    std::cout << "\nFunzione GetKmerHashTableHead\n";

    std::ifstream file(address);
    if (!file.is_open()) {
        std::cerr << address << ", does not exist!" << std::endl;
        exit(EXIT_FAILURE);
    }

    auto kmerHashTableHead = new KmerHashTableHead();

    long int arrayCount = 1000000;
    std::vector<int> freArray(arrayCount, 0);

    long int kmerCount = 0;
    long int allKmerFrequency = 0;
    std::string line;

    // Prima lettura del file per calcolare frequenze e range
    while (std::getline(file, line)) {
        std::string kmer = line.substr(0, kmerLength);
        if (!DetectSameKmer(kmer.c_str(), kmerLength)) {
            continue;
        }
        int frequency = std::stoi(line.substr(kmerLength + 1));
        if (frequency >= arrayCount - 10) {
            continue;
        }
        freArray[frequency]++;
        kmerCount++;
        allKmerFrequency += frequency;
    }

    std::cout << "The number of kmer types: " << kmerCount << ";" << std::endl;
    std::cout << "Sum of frequencies for different kmer: " << allKmerFrequency << ";" << std::endl;

    float acc = 0;
    long int max = 0;
    for (long int i = 0; i < arrayCount; i++) {
        if (freArray[i] != 0) {
            acc += static_cast<float>(freArray[i] * i) / allKmerFrequency;
            if (acc > 0.9 e max == 0) {
```

```
            max = i;
            break;
        }
    }
}

long int min = std::max(2L, static_cast<long int>(frequencyCutOff));
if (min > max) {
    std::cerr << "min is larger than max!" << std::endl;
    exit(EXIT_FAILURE);
}
std::cout << "The range of kmer frequency is: [" << min << "," << max << "];" << std::endl;

// Reinizializza variabili per la seconda lettura
file.clear();
file.seekg(0, std::ios::beg);

kmerCount = 0;
allKmerFrequency = 0;

kmerHashTableHead->allocationCount = kmerCount * 1.2;
kmerHashTableHead->kmerHashNode = new KmerHashNode[kmerHashTableHead->allocationCount]();

unsigned long int kmerInteger = 0;
while (std::getline(file, line)) {
    std::string kmer = line.substr(0, kmerLength);
    if (!DetectSameKmer(kmer.c_str(), kmerLength)) {
        continue;
    }
    int frequency = std::stoi(line.substr(kmerLength + 1));
    if (frequency >= arrayCount - 10 o frequency < min o frequency > max) {
        continue;
    }

    SetBitKmer(&kmerInteger, kmerLength, kmer.c_str());
    long int hashIndex = Hash(kmerInteger, kmerHashTableHead->allocationCount);
    while (true) {
        if (kmerHashTableHead->kmerHashNode[hashIndex].kmer == 0) {
            kmerHashTableHead->kmerHashNode[hashIndex].kmer = kmerInteger + 1;
            break;
        } else {
            hashIndex = (hashIndex + 1) % kmerHashTableHead->allocationCount;
        }
    }
}

auto kmerReadNodeHead = new KmerReadNodeHead();
kmerReadNodeHead->realCount = 0;
kmerReadNodeHead->allocationCount = allKmerFrequency * 1.1;
kmerReadNodeHead->kmerReadNode = new KmerReadNode[kmerReadNodeHead->allocationCount]();

long int readLength = 0;
char* kmer1 = new char[kmerLength + 1];

for (long int i = 0; i < readSetHead->readCount; i++) {
    readLength = readSetHead->readSet[i].readLength;
    for (int j = 0; j readLength - kmerLength + 1 - step; j += step) {
        strncpy(kmer1, readSetHead->readSet[i].read + j, kmerLength);
        kmer1[kmerLength] = '\0';
        if (!DetectSameKmer(kmer1, kmerLength)) {
            continue;
        }
        SetBitKmer(&kmerInteger, kmerLength, kmer1);
        long int hashIndex = SearchKmerHashTable(kmerHashTableHead, kmerInteger);
        if (hashIndex != -1) {
            auto& node = kmerReadNodeHead->kmerReadNode[kmerReadNodeHead->realCount++];
            node.kmer = kmerInteger;
            node.readIndex = i + 1;
            node.position = j;
            node.orientation = true;
        } else {
            ReverseComplementKmer(kmer1, kmerLength);
            SetBitKmer(&kmerInteger, kmerLength, kmer1);
            hashIndex = SearchKmerHashTable(kmerHashTableHead, kmerInteger);
            if (hashIndex != -1) {
                auto& node = kmerReadNodeHead->kmerReadNode[kmerReadNodeHead->realCount++];
                node.kmer = kmerInteger;
                node.readIndex = i + 1;
                node.position = j;
                node.orientation = false;
            }
        }
    }
}

sort(kmerReadNodeHead->kmerReadNode, 0, kmerReadNodeHead->realCount - 1);

kmerInteger = kmerReadNodeHead->kmerReadNode[0].kmer + 1;
for (long int i = 0; i < kmerReadNodeHead->realCount; i++) {
    if (kmerReadNodeHead->kmerReadNode[i].kmer != kmerInteger) {
        kmerInteger = kmerReadNodeHead->kmerReadNode[i].kmer;
        long int hashIndex = SearchKmerHashTable(kmerHashTableHead, kmerInteger);
        kmerHashTableHead->kmerHashNode[hashIndex].startPositionInArray = i;
    }
}

delete[] kmer1;
kmerReadNodeHead->kmerLength = kmerLength;
```

```
    return kmerReadNodeHead;
}
```

Questa implementazione migliora l'efficienza riducendo il numero di letture del file, ottimizzando il calcolo delle frequenze e la costruzione della tabella hash dei k-mer.

5. Implementazione di LROD senza l'Uso di DSK

Questa sezione descrive in dettaglio l'implementazione di LROD per il rilevamento delle sovrapposizioni in lunghe letture senza l'uso del software DSK. Il calcolo delle frequenze dei k-mer è stato integrato direttamente all'interno di LROD, mantenendo la funzionalità del rilevamento delle sovrapposizioni in long reads.

5.1. Modifiche al Codice

5.1.1. Rimozione della Dipendenza da DSK

Inizialmente, LROD utilizzava DSK per generare un file contenente le frequenze dei k-mer dalle lunghe letture. Questo processo avveniva in due passaggi:

1. Esecuzione di DSK per generare il file delle frequenze dei k-mer.
2. Conversione del file binario di output di DSK in un file di testo leggibile da LROD.

Per eliminare la dipendenza da DSK, abbiamo integrato il calcolo della frequenza dei k-mer direttamente nel codice di LROD.

5.1.2. Implementazione del Calcolo delle Frequenze dei k-mer

Abbiamo aggiunto funzioni per leggere le lunghe letture dal file di input, generare k-mer dalle sequenze e calcolare le loro frequenze. Questo è stato realizzato con i seguenti passaggi:

Lettura del File di Input Una funzione legge il file FASTA contenente le lunghe letture.

```
std::string readLongReadFile(const std::string& filePath) {
std::ifstream file(filePath);
std::string sequence;
std::string line;
while (std::getline(file, line)) {
if (line[0] != '>') {
sequence += line;
}
}
return sequence;
}
```

Generazione dei k-mer Una funzione genera i k-mer dalla sequenza di lettura specificata. Ad esempio, per una sequenza di DNA `ACGTGCA` e una lunghezza k-mer di 3, i k-mer generati saranno `ACG`, `CGT`, `GTG`, `TGC` e `GCA`.

```
std::vector<std::string> generateKmers(const std::string& ↩
    ↪ sequence, int k) {
std::vector<std::string> kmers;
for (size_t i = 0; i <= sequence.size() - k; ++i) {
kmers.push_back(sequence.substr(i, k));
}
return kmers;
}
```

Calcolo delle Frequenze dei k-mer Una funzione calcola le frequenze dei k-mer generati e le memorizza in una tabella hash. Ad esempio, se i k-mer generati sono `ACG`, `CGT`, `ACG`, la frequenza di `ACG` sarà 2 e quella di `CGT` sarà 1.

```
std::unordered_map<std::string, int> ↩
    ↪ computeKmerFrequencies(const ↩
    ↪ std::vector<std::string>& kmers) {
std::unordered_map<std::string, int> kmerFrequencies;
for (const auto& kmer : kmers) {
kmerFrequencies[kmer]++;
}
return kmerFrequencies;
}
```

5.1.3. Integrazione nel Workflow di LROD

Abbiamo integrato queste funzioni nel flusso di lavoro di LROD per garantire che le frequenze dei k-mer siano calcolate direttamente durante l'esecuzione del software.Tutte le funzioni introdottoe sono state integrate nel file principale `main.cpp`:

```
int main(int argc, char* argv[]) {
// Parsing degli argomenti di comando
// ...

// Lettura del file di long read
std::string sequence = readLongReadFile(readFile);

// Generazione dei k-mer
std::vector<std::string> kmers = generateKmers(sequence, ↩
    ↪ kmerLength);

// Calcolo delle frequenze dei k-mer
auto kmerFrequencies = computeKmerFrequencies(kmers);

// Stampa delle frequenze dei k-mer per verifica
for (const auto& entry : kmerFrequencies) {
    std::cout << "kmer: " << entry.first << " frequency: " ↩
        ↪ << entry.second << std::endl;
}

// Procedere con il rilevamento delle sovrapposizioni ↩
    ↪ utilizzando kmerFrequencies
// ...
}
```

5.1.4. Utilizzo di NtHash

NtHash[11] è un algoritmo di hashing che consente di calcolare efficientemente gli hash per k-mer. Utilizzando

una combinazione di tecniche di hashing avanzate e ottimizzazioni specifiche per le sequenze di DNA, NtHash offre prestazioni superiori rispetto ad altri algoritmi di hashing tradizionali. Nel nostro contesto, NtHash è stato utilizzato per calcolare rapidamente e accuratamente gli hash per i k-mer estratti dalle lunghe letture, permettendo un calcolo efficiente delle frequenze dei k-mer.

```
std::unordered_map<std::string, int> computeKmerFrequencies(const std::string& sequence, int kmerLength, int step) {
std::unordered_map<std::string, int> kmerFrequencies;
NtHash nth(sequence, 1, kmerLength);
while (nth.roll()) {
std::string kmer = sequence.substr(nth.get_pos(), kmerLength);
kmerFrequencies[kmer]++;
}
return kmerFrequencies;
}
```

6. Benchmark e Risultati

I test sono stati effettuati utilizzando un MacBook Pro con processore Apple M3 Pro, 18 GB di memoria e un totale di 11 core (5 di prestazioni e 6 di efficienza). Per i test sono stati utilizzati tre dataset: uno artificiale creato dagli sviluppatori di LROD denominato `long_read.fa`, un dataset di *E. coli*, e un dataset denominato w303. Il file delle frequenze dei k-mer è stato generato utilizzando il comando del codice modificato `-generate-kmer-file-only`.

I risultati dei benchmark per entrambi i codici sono riportati di seguito.

Codice Originale

- **Dataset artificiale:** Tempo di esecuzione: 4s, Memoria: 43424Mb
- **Dataset *E. coli*:** Tempo di esecuzione: 18s, Memoria: 93184Mb
- **Dataset w303:** Tempo di esecuzione: 124s, Memoria: 250912Mb

Codice Modificato

- **Dataset artificiale:** Tempo di esecuzione: 1s, Memoria: 43424Mb
- **Dataset *E. coli*:** Tempo di esecuzione: 12s, Memoria: 93184Mb
- **Dataset w303:** Tempo di esecuzione: 93s, Memoria: 250000Mb

6.1. Analisi dei Risultati

Il codice modificato ha mostrato un miglioramento significativo nel tempo di esecuzione per tutti e tre i dataset testati. In particolare:

- Per il dataset artificiale, il tempo di esecuzione è stato ridotto da 4 secondi a 1 secondo, mantenendo lo stesso consumo di memoria di 43424 Mb.
- Per il dataset E. coli, il tempo di esecuzione è stato ridotto da 18 secondi a 12 secondi, mantenendo lo stesso consumo di memoria di 93184 Mb.
- Per il dataset w303, il tempo di esecuzione è stato ridotto da 124 secondi a 93 secondi, con una leggera riduzione del consumo di memoria da 250912 Mb a 250000 Mb.

Questo dimostra che il codice modificato è significativamente più efficiente in termini di tempo di esecuzione rispetto al codice originale. La riduzione del tempo di esecuzione è stata ottenuta senza aumentare il consumo di memoria per i dataset artificiale ed E. coli, e con una leggera riduzione per il dataset w303.

Questi risultati indicano che le modifiche apportate al codice hanno migliorato l'efficienza temporale senza compromettere l'efficienza della memoria, rendendo il codice modificato una scelta migliore per scenari in cui il tempo di esecuzione è un fattore critico.

7. Futuri Sviluppi

Il miglioramento dell'algoritmo LROD apre diverse prospettive per futuri sviluppi. Alcune delle direzioni più promettenti includono:

7.1. Ottimizzazione del Codice

Ulteriori ottimizzazioni del codice possono portare a un incremento delle prestazioni, sia in termini di tempo di esecuzione che di consumo di memoria. Come ad esempio:

- **Parallelizzazione e Multi-threading:** Implementare il supporto per l'esecuzione parallela e il multi-threading per sfruttare al meglio le capacità dei processori multi-core moderni.
- **Algoritmi di Hashing Avanzati:** Sperimentare con nuovi algoritmi di hashing nel codice kmer.cpp.

7.2. Espansione delle Applicazioni

Espandere l'applicazione dell'algoritmo LROD a nuovi ambiti e migliorare la sua adattabilità:

- **Sequenziamento delle Proteine:** Esplorare l'applicabilità di LROD al sequenziamento delle proteine, utilizzando k-mer di aminoacidi invece di nucleotidi.
- **Sequenziamento dell'RNA:** Modificare l'algoritmo per lavorare con dati di sequenziamento dell'RNA, che presentano specifiche sfide come la presenza di introni e l'editing dell'RNA.

8. Conclusione

Le modifiche apportate all'algoritmo LROD, attraverso l'integrazione del calcolo delle frequenze dei k-mer direttamente all'interno del software, hanno eliminato la dipendenza da strumenti esterni come DSK. Questo ha comportato un

significativo miglioramento nell'efficienza del rilevamento delle sovrapposizioni nelle lunghe letture di DNA, riducendo i tempi di esecuzione senza aumentare il consumo di memoria. I test condotti su vari dataset hanno dimostrato che il nuovo approccio è più veloce e altrettanto accurato rispetto alla versione originale di LROD.

Il miglioramento dell'efficienza temporale del software è cruciale per applicazioni su larga scala e per l'analisi di dataset complessi, dove i tempi di esecuzione ridotti possono fare una grande differenza. La riduzione del tempo di esecuzione osservata nei benchmark indica che il codice modificato può gestire dataset di grandi dimensioni in modo più efficace, rendendolo uno strumento potente per il rilevamento delle sovrapposizioni in progetti di assemblaggio del genoma.

References

[1] Junwei Luo, Ranran Chen, Xiaohong Zhang, Yan Wang, Huimin Luo, Chaokun Yan, and Zhanqiang Huo. Lrod: An overlap detection algorithm for long reads based on k-mer distribution. *Frontiers in Genetics*, 11:632, 2020.

[2] Kevin Berlin, Sergey Koren, Chen-Shan Chin, Jarrett P. Drake, Jane M. Landolin, and Adam M. Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature Biotechnology*, 33:623–630, 2015.

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

[4] Mark J. Chaisson and Glenn Tesler. Mapping single molecule sequencing reads using basic local alignment with successive refinement (blasr): application and theory. *BMC Bioinformatics*, 13:238, 2012.

[5] Eugene W. Myers. Efficient local alignment discovery amongst noisy long reads. In *International Workshop on Algorithms in Bioinformatics*, pages 52–67. Springer, 2014.

[6] Katharine M. Jenike, Lucía Campos-Domínguez, Marilou Boddé, José Cerca, Christina N. Hodson, Michael C. Schatz, and Kamil S. Jaron. Guide to k-mer approaches for genomics across the tree of life. *arXiv preprint*, 2024.

[7] Bo Liu, Yu Shi, Jun Yuan, Xiang Hu, Huan Zhang, Ning Li, Zhen Li, Yan Chen, Ding Mu, and Wen Fan. Estimation of genomic characteristics by analyzing k-mer frequency in de novo genome projects. *arXiv preprint*, 2013.

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

[9] Manuel Sica. https://github.com/strumenti-formali-per-la-bioinformatica/lrod-optimization-long-reads.git, 2024.

[10] Catello Staiano Silvio Venturino. Lrod: Approfondimento. 2024.

[11] Parham Kazemi, Johnathan Wong, Vladimir Nikolić, Hamid Mohamadi, René L. Warren, and Inanç Birol. nthash2: recursive spaced seed hashing for nucleotide sequences. *Bioinformatics*, 38(20):4812–4813, 2022.