Relazione sul Progetto

Kfinger - Implementazione in C

(esame di Strumenti Formali per la Bioinformatica)

Antonio Allocca

Università di Salerno, Fisciano, Italia

Abstract. L'algoritmo Kfinger si pone l'obiettivo del calcolo degli Overlap tra lunghe reads genomiche, compito fondamentale, ad esempio, per costruire l'overlap graph. La tecnica proposta utilizza le fattorizzazioni di Lyndon [1], come introdotto in [3]. E' ben noto che la fattorizzazione di Lyndon di una stringa è una sequenza non crescente di fattori, che sono parole di Lyndon. Tale fattorizzazione è unica e può essere calcolata in tempo lineare [2]. Viene utilizzata una rappresentazione delle reads come sequenza di interi, precisamente data dalla lunghezza delle parole di Lyndon che compongono la fattorizzazione della read (Lyndon Fingerprints). E' stato dimostrato in [4] che due stringhe che condividono un overlap comune, condividono anche un insieme di fattori consecutivi nella loro fattorizzazione. Quindi, in [3] viene mostrato come individuare regioni comuni tra reads, utilizzando queste k-uple di interi (chiamate k-fingers o k-mers). Qui proponiamo una versione in C più efficiente di quella proposta originariamente in Python.

1 Algoritmo iniziale

L' algoritmo proposto in [3] si divide in quattro fasi principali:

- Lettura delle fingerprint
- Popolazione del dizionario *dict_occ_kmers*
- Popolazione dei dizionari *min_sharing_dict* e *matches_dict*
- Calcolo e riconciliazione degli overlaps

Nella prima parte dell'algoritmo si impostano i parametri che andranno poi ad influire sulla precisione e sulla velocità dell'algoritmo.

Quindi si procede con la lettura delle fingerprints, già inserite in un file di testo, e con la memorizzazione di queste ultime. Si lavorerà con le k-fingers (k-mers) cioè con sequenze di k fattori di tali interi.

Precisiamo che le reads in input vengono duplicate, cioé per ogni read si ottiene il suo reverse and complement (r&c); si fattorizza il read originale e anche la sua versione r&c. Di conseguenza per ogni read si hanno due fingerprint, una con l'ID del read a cui è stato aggiunto _0 (fingerprint del read originale) e l'altra con l'ID del read a cui è stato aggiunto _1 (fingerprint del read r&c). Precisiamo anche che le reads vengono in realtá fattorizzate per segmenti successivi di 300

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.

basi (tranne eventualmente l'ultimo che potrebbe essere ovviamente piú corto) e la fingerprint é semplicemente la concatenazione delle fingerprints dei successivi segmenti. La segmentazione serve solo per limitare la lunghezza dei fattori, che altrimenti crescerebbe da sinistra verso destra lungo il read.

Viene popolato il dizionario *dict_occ_kmers* in modo tale che ad ogni entry che ha come chiave un k-mer viene associata una lista di coppie (READ,START) che corrispondono rispettivamente all'ID della read in cui appare il k-mer e l'indice in cui appare il k-mer all'interno della fingerprint (read).

Si procede poi con la creazione di altri 2 dizionari:

- *min_sharing_dict*: contiene il numero di volte in cui due read ID sono presenti all'interno delle coppie (READ,START) relative ad un k-mer
- *matches_dict*: Rightmost e Leftmost k-mer in comune, per ogni coppia di read.

Il primo dizionario, nell'ultima implementazione in C, è stato accorpato all'altro.

Le chiavi nei dizionari sono liste di interi per velocizzare la ricerca e l'inserimento delle entry presenti all'interno dei dizionari. Con tale approccio infatti possiamo procedere ad effettuare confronti bit a bit e possiamo creare copie molto più velocemente. I valori dei dizionari invece se sono liste a strutture allora avremo una lista di puntatori a queste ultime.

Vengono poi prodotte in output le regioni comuni e gli overlap, in questo modo. Una volta calcolate le posizioni dei k-mer leftmost L e rightmost R, per ogni coppie di reads, si deve verificare che L ed R diano origine ad una regione comune. Questo si verifica solo se lo start di L della seconda read è $\leq$ dello start di R nella seconda read. Da una regione comune viene prodotto un overlap se la regione copre a sufficienza l'overlap. Al termine viene effettuato il calcolo e la riconciliazione degli overlap. La riconciliazione è causata dal fatto che le reads sono state duplicate e quindi serve per ricondurre gli overlap all'effettivo strand dei reads in input.

2 Parametri

I parametri considerati in entrambe le implementazioni sono i seguenti:

- Dimensione e minima lunghezza totale dei k-mers: `k = 7, min_total_length = 40`.
- Minimo numero di k-mers (unici) che due reads in overlap devono condividere (in assoluto): `min_shared_kmers = 4`.
- Massimo numero di occorrenze di un k-mer nei reads (valore compatibile con la coverage dell'input): `max_kmer_occurrence = -1`. Un valore pari a -1 indica assenza del controllo.
- Parametri di filtraggio delle regioni comuni in output:
  - massima differenza percentuale tra le lunghezze (bp) delle due stringhe della regione comune (mettere 0.0 nel caso di reads senza errore), `max_diff_region_percentage = 0.0`.

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.

- minima lunghezza (bp) delle due stringhe della regione comune, `min_region_length = 100`.

– Parametro di filtraggio delle regioni comuni:
- minima percentuale di copertura dei k-mers contigui rispetto alla lunghezza della regione comune. Una regione lunga L bp deve contenere almeno L/min_total_length di k-mers comuni, `min_region_kmer_coverage = 0.27`

– Parametri di filtraggio degli overlap in output:
- minima copertura percentuale (bp) della regione comune rispetto all'overlap, `min_overlap_coverage = 0.70`
-minima lunghezza dell'overlap da produrre in output (1200 per read senza errore), `min_overlap_length = 600`

3 Implementazione in Python

L'implementazione in Python è servita per delineare quelli che erano i passi dell'algoritmo fornendo un codice intuitivo che ha aiutato in seguito alla sua traduzione nel linguaggio C, nonostante tutte le complicazioni che ne derivano.

3.1 Lettura delle fingerprint

Nel file testuale ogni fingerprint è su un'unica riga e le fingerprints sono rappresentate come sequenze ID 45 7 9 1 1 | 7 65 2 3 54 |, dove ogni | era stato introdotto per separare i segmenti della fattorizzazione e ID è semplicemente n_x, dove n identifica l'n-esima read e x è un flag che indica se la fingerprint é del read originale ($x = 0$) oppure del read dopo reverse and complement ($x = 1$).

Per la realizzazione di liste di fingerprint, per ogni riga del file in input vengono innanzitutto rimosse tutte le occorrenze del carattere |. Successivamente dalla prima stringa presente all'interno della riga vengono estratti i dati della read. In particolare otteniamo un intero che rappresenta il suo **ID** e un valore booleano.

Se il valore booleano assume valore 0, allora la read è originale, altrimenti è la sua versione duplicata a di cui ne viene calcolato il r&c.

```
1  with open('./inputtest.txt', 'r') as input_file:
2      file_rows = input_file.readlines()
3      whole_rows = [re.findall(r'[^\s|]+', row) for row in
       file_rows]
4      read_ids = []
5      fingerprint_list = []
6      for row in whole_rows:
7          read_ids.append(row.pop(0))
8          fingerprint_list.append(list(map(int, row)))
9      end_time = time.perf_counter()
10     print('Upload the fingerprint: ', end_time-start_time)
11
12     len(fingerprint_list)
```

Listing 1.1. Lettura fingerprint in Python

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.

3.2 Popolazione del dizionario *dict_occ_kmers*

Per la creazione delle occorrenze dei k-mer, si deve leggere ciascuna delle fingerprint. Per ogni intero presente nella fingerprint si procede a comporre in k-mer che è formato dall'intero e, quando possibile, dai $k-1$ interi successivi.

Si può inoltre, se impostato, controllare l'unicità del k-mer all'interno delle reads, nel senso che vengono memorizzate per un dato read solo le occorrenze di k-mers che appaiono una sola volta nel read stesso.

Ogni k-mer (sequenza di interi) è associato ad una sottostringa di read (concatenazione dei fattori corrispondenti). Per evitare k-mer che supportano sottostringhe troppo corte (che porterebbero a trovare overlap errati), si controlla se la somma dei valori che compongono il k-mer sia maggiore o uguale della grandezza minima totale, parametro fissato all'inizio. Solo in questo caso viene creata all'interno del dizionario una entry che avrà per chiave una rappresentazionne del $k$mer come stringa e come valore una lista di coppie (READ,START). Precisamente, START è la posizione di inizio dell'occorrenza del k-mer chave all'interno della fingerprint della READ. Per costruzione, le tuple sono ordinate per valore crescente del valore READ e poi per START. Vengono poi eliminati i k-mer che occorrono una sola volta nell'insieme in input o troppe volte.

```
def compute_kmer_occurrences(fingerprint_list):
    start_time_begin = time.perf_counter()
    kmer_occurrences = {}
    for (j,finger) in enumerate(fingerprint_list):
        check_unique = []
        occ_kmer_list = []
        for (i,c) in enumerate(finger):
            kmer = tuple(finger[i:i+k])
            if len(kmer) == k:
                check_unique.append(kmer)
                occ_kmer_list.append((j,i,kmer))
        c = Counter(check_unique)
        for kmer_t in occ_kmer_list:
            #if c[kmer_t[2]] == 1:
            if c[kmer_t[2]] >= 1:
                cfr_kmer = kmer_t[2]
                if sum(cfr_kmer) >= min_total_length:
                    cfr_kmer = '_'.join(list((map(str,
    cfr_kmer))))
                    value = kmer_occurrences.get(cfr_kmer,
    [])
                    value.append((kmer_t[0], kmer_t[1]))
                    kmer_occurrences[cfr_kmer] = value
    for kmer in kmer_occurrences:
        kmer_occurrences[kmer] = tuple(kmer_occurrences[kmer
    ])
    end_time = time.perf_counter()
    print('compute_kmer_occurrences: ', end_time - start_time
    )
```

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.

```
    print(kmer_occurrences)
    return kmer_occurrences
```

Listing 1.2. Popolazione del dizionario *dict_occ_kmers*

```

h_kmer_occurrences = dict()
for kmer in kmer_occurrences:
    size = len(kmer_occurrences[kmer])
    if size > 1 and (max_kmer_occurrence == -1 or size <=
    max_kmer_occurrence):
        h_kmer_occurrences[kmer] = kmer_occurrences[kmer]

kmer_occurrences = h_kmer_occurrences
```

Listing 1.3. Eliminazione di alcuni k-mers

3.3 Popolazione dei dizionari *min_sharing_dict* e *matches_dict*

In questa fase vengono trovati quelli che sono i leftmost e i rightmost k-mer per ogni coppia di reads. Inoltre viene creato un dizionario per tenere conto di quanti k-mers sono condivisi da due reads,

Questa è sicuramente la fase più onerosa in quanto abbiamo dei for innestati.

```
def compute_matches(kmer_occurrences):
    start_time_begin = time.perf_counter()
    min_sharing_dict = {}
    matches_dict = {}
    for (p, kmer) in enumerate(kmer_occurrences):
        occ_list = kmer_occurrences[kmer]
        for (i, first_occ) in enumerate(occ_list):
            read1 = first_occ[0]
            for second_occ in occ_list[i+1:]:
                read2 = second_occ[0]
                min_sharing_dict[(read1, read2)] =
    min_sharing_dict.get((read1, read2), 0) + 1
                value = matches_dict.get((read1, read2), [-1,
     -1, -1,-1])
                if value[0] == -1 or value[0] > first_occ[1]:
                    value[0] = first_occ[1]
                    value[1] = second_occ[1]
                if value[2] == -1 or value[2] < first_occ[1]:
                    value[2] = first_occ[1]
                    value[3] = second_occ[1]
                matches_dict[(read1, read2)] = value
    end_time = time.perf_counter()
    print('compute_matches: ', end_time - start_time)
    print(min_sharing_dict)
    print(matches_dict)
```

This work is licensed under a Creative Commons “Attribution-NonCommercial-NoDerivatives 4.0 International” license.

```
24      return (min_sharing_dict, matches_dict)
```

Listing 1.4. Popolazione dei dizionari *min_sharing_dict* e *matches_dict*

Dai dizionari dei leftmost e rightmost k-mers si inferiscono gli overlap per il set dei reads duplicati. Subito dopo gli overlap vengono riconciliati per avere gli overlap per il set di reads originali.

Successivamente i risultati prodotti vengono controllati (parte attualmente non tradotta in C).

4 Implementazione in C

L'implementazione in C ha avuto difficoltà legate soprattutto alla scelta delle strutture di dati da utilizzare, come accederci (inserimento e ricerca di un elemento) e la memoria allocata. Infatti, abbiamo a che fare con grandi quantità di dati e non dobbiamo tenere conto solamente della velocità di esecuzione, ma anche alla memoria utilizzata. Il codice è disponibile nella seguennte repository `https://github.com/Tonin-ai/pattern-matching-2`.

4.1 Utilizzo di Hashtable

Per emulare il comportamento del dizionario di Python, si è scelto di implementare delle Hashtable. L'idea è quella di avere un tempo di accesso costante a un dizionario in cui le key delle entry hanno un valore hash che collide. I dizionari vengono allocati dinamicamente per non consumare inutilmente memoria.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <malloc.h>
5  #include <limits.h>
6  #include "matches_dict.h"
7
8  typedef struct hash_mdict{
9
10         matches_dict** dicts_list; //lista di puntatori a
       dizionari
11         int size; //numero di dizionari
12
13 } hash_mdict;
14
15 void initHashMDict(hash_mdict* mdict,int size);
16 int add_in_hash_mdict(matches_dict** dict,int* reads,int
       first, int second);
17 void viewHashMDict(hash_mdict* mdict);
18
19 /*
20 in: mdict,dize
21 out: void
```

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.

```
viene inizializzato il dizionario con un numero di entry passate come parametro
*/

void initHashMDict(hash_mdict* mdict,int size){
        mdict->size=size;
        mdict->dicts_list=(matches_dict**)malloc(mdict->size * sizeof(matches_dict*));
        for(int i=0;i<mdict->size;i++) mdict->dicts_list[i]= NULL;
}

/*
in: dict,reads,first,second
oud: 1 se viene inserito correttemente la coppia (FIRST_READ, SECOND_READ) all'inerno della entry con chiave reads nel dizionario dict
*/

int add_in_hash_mdict(matches_dict** dict,int* reads,int first, int second){
        if(!*dict){//viene controllato che \'e allocata memoria per il dizionario dict
                *dict = (matches_dict*)malloc(sizeof(matches_dict));
                initMDict(*dict);
        }
        return add_in_mdict(*dict,reads,first,second);
}

/*
in: mdict
out: void
viengono visualizzati tutti i dizionari presenti nella lista di dizionari
*/

void viewHashMDict(hash_mdict* mdict){
        for(int i=0;i<mdict->size;i++){
                if(mdict->dicts_list[i]){
                        t[i]->size);
                         viewMDict(mdict->dicts_list[i]);
                }
        }
}
```

Listing 1.5. Hashtable per match table

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.

Funzione hash La funzione hash utilizzata è una funzione poco resistente a collisioni: più si ha a disposizione memoria per allocare entry all'Hashtable e più si può utilizzare una funzione esistente a collisione (come ad esempio CRC).

Tali funzioni devono avere un buon compromesso tra velocità di calcolo e randomicità dell'algoritmo.

```
int getEntryKey(int* kmer,int hdictsize,int k){

        unsigned int result = 0x55555555;

        for(int i=0;i<k;i++){
                result ^= kmer[i];
                result = result<<5;
        }
        result%=hdictsize;
        return result;
}
```

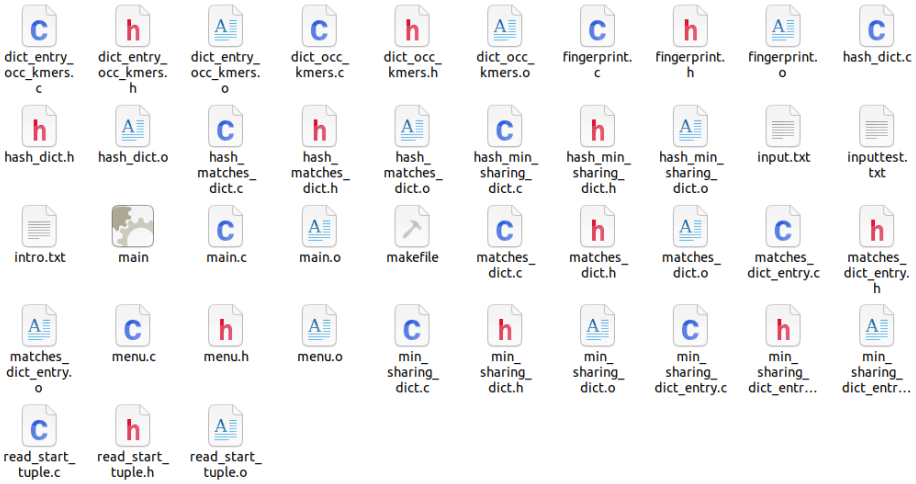
Listing 1.6. Funzione per calcolo dell'hash

Con questa funzione, ogni intero presente all'interno della chiave (stiamo parlando di kmer) contribuisce alla randomicità della funzione.

4.2 Suddivisione codice in librerie

Ogni hashtable, dizionario ed entry sono stati meticolosamente scritti in modo tale che il codice sia riproducibile, facile da leggere e l'accesso a tali funzioni viene fatto solamente tramite funzione.

Questo implica che ogni componente è modificabile. Ad esempio se si trova una soluzione più efficiente dell'hashtable, si possono lasciare intatti i dizionari, senza dover modificare l'intero codice, azione che portrebbe ad errori anche gravi.



This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.

4.3 Allocazione dinamica della memoria

Fattore principale dell'implementazione è proprio l' allocazione dinamica della memoria. Questo è dovuto all'enorme quantità di dati che il programma dovrà gestire e all'impredicibilità di essi. Possiamo infatti allocare troppe entry a un dizionario rendendolo così molto efficiente ma rischiando di saturare la memoria oppure allocarne troppe poche rendendo l'algoritmo più lento, ma con un consumo della memoria più efficiente.

Per questo si èscelto di gestire tutto tramite puntatori e l'uso della libreria malloc.h che permette l' allocazione, la de-allocazione e la re-allocazione della memoria.

In numero di entry dell' hashtable invece è costante e deve essere un valore abbastanza grande. E' stato stimato in

Uso massivo dei puntatori L'utilizzo dei puntatori è stato decisivo per lo sviluppo di tutto l'algoritmo. Si sono creati puntatori per creare le strutture dati e si è usata l'aritmetica dei puntatori per andare ad operare direttamente sugli indirizzi di memoria per ridurre, quando era possibile, ancora di più i tempi per l'esecuzione dei programmi.

Inoltre si è preferito utilizzare operazione direttamente sui bit per rendere il processo efficiente. Inoltre, le k-finger (k-mer) non sono rappresentati come stringhe, ma come array in modo da poter confrontare usando la memcmp di C.

Liste concatenate e doppiamente concatenate Uno degli utilizzi dei puntatori è quello di creare strutture dati dinamiche. In questa implementazione si è fatto uso sia di liste concatenate che doppiamente concatenate.

Ogni entry di un dizionario è collegata tramite un puntatore alla entry successiva. Tale struttura dati, se salviamo il puntatore all'ultimo elemento della lista, permette di avere un inserimento costante a discapito del tempo di ricerca. Se vogliamo visitare tutta la lista, infatti, dovremo per forza attuare una visita sequenziale che, per array molto grandi (decine di miliardi di entry), è un'operazione molto onerosa.

Le lista doppiamente concatenate, invece, sono state utilizzate per implementare le liste di coppie (READ,START) relative ad ogni k-mer. Questo è stato fatto perchè, per costruzione, le liste sono già ordinate, ma se si volesse parallelizzare il processo si può optare per un insertion-sort.

```
1 #include <stdlib.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <malloc.h>
5 typedef struct read_start_tuple{
6
7         int read;  //posizione della read all'interno della
    lista di fingerprint
8         int start; //posizione occorrenza kmer nella
```

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.

```
9
10          //Le tuple sono ordinate per valore crescente del
        valore read e poi per start
11          //una lista doppiamente concatenata ci velocizza il
        processo di inserimento e cancellazione
12          struct read_start_tuple* prev;
13          struct read_start_tuple* next;
14
15  } rs_tuple;
16
17  rs_tuple* create_rs_tuple(int r, int s, rs_tuple* p, rs_tuple
        * n){
18          rs_tuple* p1 = (rs_tuple*)malloc(sizeof(rs_tuple));
19
20          p1->read  = r;
21          p1->start = s;
22
23          p1->next = n;
24          p1->prev = p;
25
26          return p1;
27  }
```

Listing 1.7. Prima versione coppia (READ,START)

4.4 Dizionari e strutture implementate

- `read_star_tuple`: rappresenta la coppia (READ,START), quindi l'intero associato alla read e l'indice da cui parte un k-finger.
- `dict_entry_occ_kmer`: ad ogni k-mer associa la lista delle coppie (READ,START), sopra definita. Viene costruito automaticamente in ordine sulle read e, per ogni read, in ordine crescente su START.
  Tale struttura rappresenterà le entry del dizionario dict_occ_kmers. E' stato fornito anche un costruttore per ottene un puntatore alla entry con i parametri passati.
- `dict_occ_kmers`: è il dizionario che contiene per ogni k-mer la lista sopra definita. Sono definite le funzioni per cercare un k-mer (iterativa), inserire un k-mer se non trovato (aggiunge in coda).
  Inoltre è stata sviluppata una visita dell'intero dizionario per averne una rappresentazione grafica,
  Quando si vuole inserire all'interno del dizionario una coppia read, start viene prima cercata la entry all'interno del dizionario che che ha per chiave il k-mer a cui fanno riferimento. Se tale entry viene trovata, viene aggiunta la coppia alla lista, altrimenti viene allocata memoria per creare una nuova entry che avrà per chiave il k-mer e come valore una lista vuota alla quale andiamo poi ad aggiungere la coppia.

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.

- `hash_dict`: è il dizionario formato dai dict_occ_kmers. In altre parole, è la lista di puntatori alle liste che rappresentano dict_occ_kmers Questa struttura oltre a gestire l'inserimento e l'inizializzazione dell'hash table, si occupa pure della visualizzazione di tutti i dizionari contenuti al suo interno,
  Tale implementazione ha fornito di abbattere di molto di tempi di esecuzione in quanto abbiamo tempo di acccesso costante a dizionari di dimensioni molto ridotte
- `match_dict_entry`: viene memorizzata una coppia $(key, value)$, dove key rappresenta una coppia di read (r_1, r_2) e il $value$ rappresenta una quadrupla $(l_1, l_2, r_1, r_2$ che sono rispettivamente le posizioni leftmost (le prime due) e rightmost (le ultime due) di r_1 e di r_2 rispettivamente.
  I controlli per l'aggiornamento dei valori all'interno della lista vengono fatti in una funzione dedicata al momento dell'inserimento. In altre parole, se la lista associata deve essere aggiunta, modificata oppure no, viene verificato con un' unica funzione.
- `matches_dict`: considera tutte le entry e gestisce gli inserimenti, le ricerche all'interno del dizionario ed offre la possibilità di visualizzare il dizionario
- `min_sharing_dict_entry`: per ogni coppia (r_1, r_2) valuta quante volte compare nell'altro dizionario E' stato eliminato o, per meglio dire, incorporato all'interno di matches_dict per motivi di efficienza in termini di spazio e velocità.
  Se si necessita comunque tale implementazione e' comunque disponibile all'interno del git.
- `dict_overlap`: ad ogni coppia di read (r_1, r_2) associa nove valori interi, come scritto nel notebook.

5 Il metodo

Per effettuare la lettura delle fingerprint si acquisisce una stringa alla volta. Se è una stringa del tipo id_bool, allora sarà instanziata una fingerprint con valore id pari all'id letto e il parametro isreverse assumerà il valore bool. In seguito verrà letta la lista di interi che compongono la fingerprint.

Dopo questa fase, si passa al riempimento del dizionario delle occorrenze dei k-mer temporaneo. Per ogni k-mer che viene letto, viene prima controllato che soddisfi alcuni requisiti, come la grandezza totale del k-mer maggiore o uguale di una grandezza fissata. In seguito viene considerata la read e l'indice in cui tale k-mer appare, e nel dizionario la chiave k-mer, avrà come valore la coppia read e start. Quando deve essere inserita una coppia, prima si cerca all'interno del dizionario l'entry con chiave ottenuta applicando la funzione hash sul k-mer, ottenendo così un dizionario (la lista associata a tale k-mer). Sul dizionario così ottenuto, viene ricercata la presenza o meno della entry che ha per chiave il k-mer. Se non è presente alloco una nuova entry che ha per chiave il k-mer. Dopo aver effettuato tale controllo, alla lista di coppie associata al k-mer viene aggiunta la nuova coppia appena letta.

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.

Una volta riempito il dizionario temporaneo, si fa il filtering delle entry che hanno al massimo un numero di coppie associate ad esso pari al parametro `max_kmer_occurence` stabilito all'inizio.

Successivamente viene computato il dizionario dei match utilizzando il dizionario delle occorrenze. Per ogni entry del dizionario delle occorrenze, vengono analizzate le coppie associate ad ogni k-mer e viene calcolata la lista di interi che rappresenta la posizione leftmost e quella rightmost del k-mer. Le coppie vengono analizzate 2 alla volta. Così facendo le due read rispettive diventano la chiave della entry del dizionario dei match e il valore associato a tale chiave sarà la lista di interi che stabilisce il leftmost ed il rightmost. Contemporanemente viene salvato anche il numero di volte in cui una coppia di read viene aggiornata (non cè più quindi il dizionario min_sharing_dict).

I controlli per l'aggiornamento o meno dei valori leftmost e rightmost viene fatta direttamente all'interno dell'update per alleggerire il codice nel main.

Infine viene calcolato il dizionario degli overlap associando ad ogni coppia di read una lista parametri (descritti precedentemente) se e solo se rispettano determinate condizioni descritte all'interno dell'algoritmo. Quindi, gli overlap riconciliati saranno prodotti come record dei campi seguenti:

- id del primo read (senza il terminatore di strand)
- lunghezza del primo read
- posizione 0-based di inizio dell'overlap sul primo read
- posizione 1-based di fine dell'overlap sul primo read
- id del secondo read (senza il terminatore di strand)
- lunghezza del secondo read
- posizione 0-based di inizio dell'overlap sul secondo read
- posizione 1-based di fine dell'overlap sul secondo read
- strand del secondo read rispetto al primo (0: se uguale; 1: se opposto)

5.1 Algoritmo di calcolo degli overlap

Il dizionario dei leftmost e rightmost k-mers contiene tutte le coppie di fingerprints (chiavi del dizionario) che sono candidati a dare un overlap.

Per ogni coppia (r_1, r_2) nel dizionario si ottiene il leftmost k-mer L e il rightmost k-mer R.

Vengono poi fatte le due seguenti verifiche:

(1) r_1 e r_2 devono condividere almeno `min_shared_kmers` (parametro in input) k-mers che sono unici nelle due reads,

(2) L deve venire prima di R in r_2 (per costruzione L viene prima di R in r_1).

Se tali verifiche hanno esito positivo, allora si determina la coppia di sottostringhe s_1 e s_2 (cioè la regione comune) indotte dai k-mers L e R sui corrispondenti reads. Cioè, s_1 è la sottostringa della read che corrisponde a r_1 che inizia nella stessa posizione di inizio della sottostringa corrispondente a L su r_1 e finisce nella stessa posizione di fine della sottostringa corrispondente a

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.

R su r_1. Analogamente, s_2 è la sottostringa di r_2 che inizia nella stessa posizione di inizio della sottostringa corrispondente a L su r_2 e finisce nella stessa posizione di fine della sottostringa corrispondente a R su r_2. Viene calcolato poi un valore "atteso" minimo di k-mers condivisi tra le due reads sulla base dei parametri in input `min_region_kmer_coverage`, `min_total_length` e della lunghezza della regione comune (minimo tra le due lunghezze di s_1 e s_2). Se il numero di k-mers condivisi da r_1 e r_2 supera questo minimo atteso, e inoltre si ha che (1) la differenza di lunghezza tra s_1 e s_2 non supera una certa percentuale `max_diff_region_percentage` del massimo tra le due lunghezze di s_1 e s_2 e (2) il massimo delle lunghezze di s_1 e s_2 é oltre la soglia `min_region_length`, allora si procede a calcolare l'overlap tra r_1 e r_2. Questo é praticamente ottenuto estendendo la regione comune in modo da ottenere un overlap prefisso-suffisso oppure un overlap in cui una delle due reads corrisponde a una sottostringa (anche non propria) dell'altro. A questo punto, se il minimo tra le due lunghezze di s_1 e s_2 è almeno una percentuale `min_overlap_coverage` della lunghezza di tale overlap e la lunghezza dell'overlap é almeno il parametro `min_overlap_length`, allora l'overlap viene tenuto (altrimenti viene scartato).

Si considerino ora, dati due reads in input n_1 e n_2 tutti gli overlap trovati, che saranno al massimo 4 dal momento che ogni read viene duplicato. Per ogni coppia n_1, n_2 viene tenuto l'overlap (tra quelli trovati) di lunghezza massima. La riconciliazione rimappa eventualmente l'inizio e la fine dell'overlap selezionato rispetto alle reads originali. Ad esempio, se si ha che la versione originale di n_1 ha un suffisso che coincide con un prefisso della versione r&c di n_2, allora devo rimappare l'overlap in un prefisso della versione originale di n_2.

6 Esecuzione e Performance

Il tool sviluppato consente di essere eseguito in due modalità:

- se si esegue con ./main, sarà un'esecuzione che calcolerà esclusivamente gli overlap
- se si esegue con ./main -l, viene creato un file di log dei dizionari e delle fingerprint.

Per controllare l'equivalenza dei due codice, si è prodotto uno script Python che, legge l'output prodotto da entrambe le implementazioni, creare il dizionario e vede se per ogni entry cè la rispettiva entry nel output in Python. Non sono stati identificati errori.

Il file utilizzato è di ... MB per un totale di ... reads.

[RR: Il dataset utilizzato per la sperimentazione é stato ottenuto utilizzando il tool `DeepSimulator` e contiene 10 000 error-free long reads (che diventano 20 000 dopo la duplicazione), simulati dalla regione del cromosoma umano 21, tra le posizioni 32 000 000 e 34 000 000 (regione di lunghezza 2 000 000bp). Sono stati trovati in tutto 419292 coppie in overlap di cui 18 coppie presentano una differenza tra le due regioni in overlap (essendo reads senza errori queste coppie non sono attese, ma il metodo consente di trovare overlap con errori e quindi

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.

le trova, bisognerebbe andare a vedere - calcolando la distanza di edit - qual è la differenza tra le due regioni. In ogni caso, la lunghezza di questi overlap non supera le 1450bp e quindi é una percentuale bassa rispetto alla lunghezza dei reads che é sull'ordine delle 13000-15000 bp).]

Caratteristiche sistema

Nome dispositivo LAPTOP-26FE8SRB
Processore Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
RAM installata 16,0 GB (15,8 GB utilizzabile)
Tipo sistema Sistema operativo a 64 bit, processore basato su x64

Python		**C**	
Operazione	**Tempo(s)**	**Operazione**	**Tempo(s)**
Secondi lettura linee	10.5283788	Secondi lettura linee	2
Secondi creazione dizionario	100.20265909999999	Secondi creazione dizionario	25
Secondi creazione msdict e mdict	185.9841576	Secondi creazione msdict e mdict	67
Secondi calcolo overlaps	87.86921159999997	Secondi calcolo overlaps	5

References

1. Lyndon, R. C.: On Burnside's problem. Transactions of the American Mathematical Society **77**(2), 202–215 (1954)
2. Duval, J.-P., Factorizing Words over an Ordered Alphabet, J. Algorithms, vol. 4 (4), pp. 363-381, 1983.
3. Bonizzoni, P., Petescia, A., Pirola, Y., Rizzi, R., Zaccagnino, R., Zizza, R.: KFinger: Capturing Overlaps Between Long Reads by Using Lyndon Fingerprints. Bioinformatics and Biomedical Engineering - 9th International Work-Conference, IWBBIO 2022, Spain, June 27-30, 2022, Lecture Notes in Computer Science, vol. 13347, pp. 436-449.
4. Bonizzoni, P., De Felice, C., Zaccagnino, R., Zizza, R.: Lyndon words versus inverse Lyndon words: Queries on suffixes and bordered words. In: LATA 2020, LNCS, vol. 12038, pp. 385–396. Springer (2020). `https://doi.org/10.1007/978-3-030-40608-0\_27`

This work is licensed under a Creative Commons "Attribution-NonCommercial-NoDerivatives 4.0 International" license.