Assignment 1
Assemblaggio di Reads

1819-1-F1801Q108, Bioinformatica, 2018-2019

Università degli studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione

Matamoros Aragon Ricardo Anibal 807450
Palazzi Bruno 806908
Villa Giacomo 807462

10 Maggio 2019

Indice

1 Sommario

Nel primo assignment è stato richiesto di disegnare e implementare una pipeline di assemblaggio di reads provenienti da un genoma; in particolare si richiedeva che, dato in input un file in formato FASTQ, venisse fornito in output un grafo di Overlap (o de Bruijn), utile per l'assemblaggio delle reads.
Noi abbiamo deciso di implementare entrambi i grafi, per poi passare ad un attività di confronto degli stessi. Abbiamo quindi scritto un programma che prende in input la qualità richiesta per le basi e la lunghezza minima delle reads pulite, date queste informazioni sarà poi eventualmente possibile effettuare il trimming; si passerà successivamente alla creazione di entrambi i grafi. Nelle successive sezioni si tratterà di tutti i punti fondamentali dell'assignment.
L'intero progetto è stato scritto utilizzando il linguaggio Python 3.7.3.

2 Main

Lo script che prende il nome di Main.py è il launcher dell'intera applicazione; in questo script viene richiamata una funzione, denominata `main()` appunto, la quale si occupa di chiamare in sequenza le operazioni di Trimmer, Overlap Graph e De Bruijn Graph.

Per ogni operazione richiamata, tolto il trimming, viene calcolato l'utilizzo di memoria RAM e il tempo di esecuzione.

3 Trimming

3.1 Introduzione

Nella fase di Trimming siamo partiti dal file FASTQ fornito (dotato quindi sia delle read sia della stringa phred values) per poi giungere alla creazione di un file (`CleanReadsNGS.txt`) contenente il trimming delle reads.

L'obiettivo principale di questo step è quindi quello di eseguire un filtraggio dell'insieme iniziale di reads, con il quale si genera un insieme con la stessa cardinalità, ma che potrebbe presentare una lunghezza inferiore dati i singoli elementi (in funzione della qualità e lunghezza minima definita). Come soglia di qualità si utilizza il valore q (Phred Score) associato ad ogni singola base per ogni read.

La qualità q di ogni base è stata codificata in un carattere **c** attraverso la seguente formula: $c = ASCII(min(q, 93) + 33)$

Gli script contenti le funzioni qui definite sono:

- **trimming.py**: `trimmer()`
- **readsOperation.py**: `trim_operation(qualityString, qualityValue, lengthValue)`, `convert_ASCII_to_decimal(ASCII_letter)`

3.2 Idea di base

Il concetto alla base di questa funzionalità può essere sintentizzato nell'espressione *"another view about it "*; partendo dalla stringa di qualità volevamo una struttura che rappresentasse la read in funzione della qualità, quindi che indicasse le sottostringhe con qualità maggiore uguale a quella impostata. In questo senso, come successivamente verrà spiegato, è stato deciso di creare una lista di triple contenenti i due indici della sottostringa (inizio e fine) e la lunghezza di questa; i primi servivano per sapere in che punto effettuare il *"cut"*, l'ultimo per avere un parametro utile sul quale ordinare al fine di restituire la sottostringa di lunghezza massima.

3.3 Funzione `trimmer()`

Questa funzione, una volta richiamata dal main, permette di scegliere il file FASTQ e definire la qualità e la lunghezza minima richiesta per l'operazione di trimming.

Il file è scelto mediante l'utilizzo della libreria `Tkinter` che permette di implementare una sorta di GUI come visibile in figura 1, tuttavia in questa fase è commentato; è stato notato che, a causa di un conflitto causato dalla libreria memory check, solo sui sistemi macOS vi è l'impossibilità di scegliere un file, con Ubuntu 18.04 invece non è sorto tale problema.

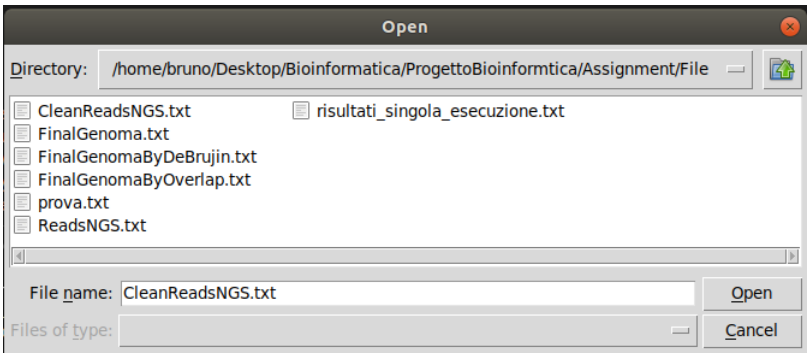


Figura 1: Interfaccia grafica di Tkinter per la scelta del file in Ubuntu 18.04

In ogni caso la scelta del file è possibile "manualmente" inserendo nella cartella `Assignment/File` un file denominato `ReadNGS.txt`.

In seguito alla scelta del file e dell'input vengono effettuati dei controlli sui valori di qualità e lunghezza, inoltre se fosse inserita una qualità maggiore di 93 verrebbe automaticamente settata a tale valore. Il tutto è inserito in un ciclo nel quale si uscirà una volta che l'input utente risulterà corretto.

Una volta che l'input risulterà corretto si scandirà il file selezionato, identificando di volta in volta la riga con la read e la riga della corrispettiva phred. Verrà quindi richiamato il metodo `trimOperation(qualityString, qualityValue, lengthValue)` passando, come si evince dalla signature, stringa phred, qualità richiesta e lunghezza minima richiesta.

Nel caso in cui questo metodo dovesse restituire `None` significa che non è stato rispettato il vincolo sulla lunghezza minima data una read; in questo caso si interrompe il ciclo e viene mostrato a schermo un messaggio di errore con il numero della read che non rispetta il vincolo. Se invece il vincolo sulla lunghezza minima non fosse stato violato, le reads trimmate verranno salvate su un file (`CleanReadsNGS.txt`) in funzione di ciò che la funzione ha restituito.

3.4 Funzione `trimOperation(qualityString, qualityValue, lengthValue)`

Questa funzione è richiamata, da parte `trimmer()`, ad ogni phred letta; come parametri prende dunque la stringa phred, la qualità richiesta e la lunghezza minima.
L'attività di questa funzione è principalmente analizzare la phred e valutare, carattere per carattere, che il livello di qualità richiesto sia per lo meno raggiunto. In questo modo si vanno a definire degli intervalli di indici, i quali identificano le sottostringhe che rispettano il vincolo di qualità, data ovviamente la read in questione. Una volta definiti tutti questi intervalli si selezionerà quello con lunghezza maggiore, si valuterà se la lunghezza di questo è maggiore di quella imposta; in caso positivo si procederà al return della tripla:

```
(StartIndex, EndIndex, TotalLength)
```

in caso contrario si restituirà `None`.

```
[[0, 19, 19], [20, 27, 7], [30, 34, 4], [35, 55, 20], [56, 63, 7], [66, 70, 4], [71, 77, 6]]
```

Figura 2: Struttura creata durante un'operazione di trimming con qualità 55 e lunghezza 10 su di una generica read

```
[35, 55, 20]
```

Figura 3: Tripla di lunghezza massima ritornata data immagine precedente

3.5 Funzione `convertASCIIToDecimal(ASCII_letter)`

Per completezza descriviamo questa funzione che viene utilizzata per convertire un carattere codificato ASCII nel corrispettivo valore decimale, tenendo ovviamente conto dell'aggiunta del valore 33.
Utilizzata unicamente nella funzione precedentemente mostrata al fine di alleggerire il codice.

4 Overlap Graph

4.1 Introduzione

In questa fase siamo partiti dal file generato durante le operazioni di trimming, con l'obiettivo di ottenere un grafo di Overlap per la ricostruzione del genoma.
I passaggi fondamentali in questa fase sono i seguenti:

- Identificazione della sovrapposizione da tutte le reads a tutte le reads (logica All vs. All).
- Identificazione dei valori massimi di sovrapposizione tra le reads.
- Ricostruzione del genoma.

A tal propositivo le classi, con i corrispondenti metodi, che permetto di effettuare le operazioni descritte sopra sono le seguenti:

- **OverlapG.py**: `overlapGraph()`
- **OverlapGraphOperation.py** : `graphMatrix(reads)`, `buildGraph(graphMat)`, `printFinalGenoma(graph, reads)`

4.2 Idea di base

Per l'implementazione del grafo di Overlap, innanzitutto, ci siamo interrogati su quale struttura dati utilizzare al fine di mantenere tutti i valori di overlap tra le varie reads, abbiamo quindi pensato di creare una matrice quadrata di dimensione pari al numero di reads. Ogni cella di tale matrice contiene un valore di overlap (quanto la read `i` dovrà "overlappare" la read `j`), questo è ottenuto mediante un allineamento basato sulla programmazione dinamica. Per ogni allineamento avevamo bisogno di due matrici: una contenente i punteggi e una per risalire e osservare se l'overlap risultante fosse il più possibile privo di errori.

4.3 Funzione `overlapGraph()`

Questa funzione è richiamata dal Main una volta che le operazioni di trimming sono terminate; ha come scopo quello di leggere le reads dal file creato dalla precedentemente fase, al fine di creare una lista contenente appunto le reads "pulite". La lista così creata verrà poi utilizzata nelle funzioni che seguono, le quali saranno richiamate nello stesso ordine con cui sono riportate in questa documentazione.

4.4 Funzione `graphMatrix(reads)`

Questa funzione rappresenta il "core" dell'intera sezione, il suo compito è creare e popolare con dei valori una matrice (la core matrix `graphMat`) con un numero di righe (e di conseguenza colonne) pari al numero di reads. Ogni cella dunque conterrà lo score di allineamento (numero Match sommato al numero di Mismatch) date le due reads; ovviamente la diagonale non verrà considerata (read contro se stessa).

I punteggi assegnati ai vari casi sono quindi i seguenti:

- Match: 10
- Mismatch: -1
- Indel: -4

Viene dunque creata la core matrix, in questo caso date le reads una 18x18, e si inizia a iterare. Per ogni cella verranno create due matrici:

- `dynamicProgMat`: matrice utilizzata per la programmazione dinamica, di dimensioni pari a lunghezza read sulle riga per lunghezza read sulle colonna, ovviamente riga e colonna rispetto alla `graphMat`.
- `directMat`: matrice utilizzata per le operazioni di ricostruzione dell'allineamento, le dimensioni sono le stesse della matrice precedente. I valori di questa matrice possono essere 0 (`STOP`), 1 (`UP`), 2 (`LEFT`) e 3 (`DIAGONAL`).

Una volta create le matrici sopra, abbiamo implementato l'algoritmo di programmazione dinamica mostrato a lezione; una volta definiti i casi base:

$$D(0,0) = 0$$
$$D(i,0) = i * d$$
$$D(0,j) = 0$$

Siamo passati a riempire le celle della `dynamicProgMat` utilizzando la seguente logica:

$$D(i,j) = \max \begin{cases} D(i-1,j-1) + \delta(a_i, b_j) \\ D(i-1,j) + d \\ D(i,j-1) + d \end{cases}$$

In funzione di cosa inserivamo di cella in cella, la matrice `directMat` veniva anch'essa popolata, in accordo con quello scelto e con quanto sopra definito. Dunque un generico confronto tra read porta alla generazione delle seguenti matrici, rispettivamente `dynamicProgMat` e `directMat`:

```
[[  0.   0.   0.   0.   0.]
 [ -4.  -1.  -1.  -1.  -1.]
 [ -8.   6.   2.  -2.  -2.]
 [-12.   2.   5.   1.  -3.]
 [-16.  -2.  12.  15.  11.]]
```

```
[[0. 0. 0. 0. 0.]
 [1. 3. 3. 3. 3.]
 [1. 3. 2. 3. 3.]
 [1. 3. 3. 3. 3.]
 [1. 1. 3. 3. 2.]]
```

Figura 4: Matrici nel caso di quality 64

A questo punto siamo andati a selezionare, data l'ultima colonna, il valore massimo così da poter poi risalire utilizzando la matrice di direzione; in funzione della direzione presa verranno aumentati dei contatori che tengono nota del numero di Match, Mismatch e Indel.

Sono state inserite in fase di testing due liste le quali, se stampate insieme, formano una matrice di allineamento. È stato comunque deciso di mantenerle, anche se con stampa commentata, in quanto utili per ciò che concerne la comprensione delle operazioni; avremo dunque, date le matrici precedentemente mostrate, la seguente matrice di allineamento:

```
['A', 'G', 'G', 'C', '-']
['-', 'G', 'C', 'C', 'T']
```

Dunque, una volta performato quanto sopra riportato è necessario capire se il punteggio può essere inserito nella `graphMat`; questa decisione è presa osservando se il numero di Mismatch sommato al numero di Indel risulta essere inferiori al 3%; è stato deciso di definire tale valore in quanto pensiamo possa essere sufficientemente alto da eliminare gli overlap totalmente scorretti e mantenere dunque solo quelli corretti. In questo senso, se la risposta a quanto prima riportato è positiva vengono salvati il numero di Match e di Mismatch data la cella; nel caso in cui fosse negativa viene salvato il valore -1.

La matrice finale ottenuta è la seguente:

```
[[ 0. 67. 50. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1.  0. 63. 51. 38. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1.  0. 68. 55. 50. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1.  0. 67. 62. 41. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1.  0. 75. 54. 36. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1.  0. 59. 41. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1.  0. 62. 48. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1.  0. 66. 47. 33. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1.  0. 61. 47. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1.  0. 66. 51. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1. -1.  0. 65. 46. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.  0. 61. 42. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.  0. 61. 42. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.  0. 61. 44. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.  0. 63. 43.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.  0. 60.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.  0.]]
```

Figura 5: `graphMat` delle read dato il trimming con quality 50

Si vuole sottolineare come nella fase di salvataggio dei valori nella `graphMat`, si è deciso di salvare a indici inversi; se si sta valutando l'overlap della read i-esima (sulle righe) contro la read j-esima (sulle colonne), il valore risultato viene salvato nella cella `[j][i]`. Questo perché è la read j-esima che va in overlap (quindi "sovrasta") con la read i-esima; è stato deciso di seguire questa logica in quanto va a semplificare sensibilmente gli step successivi.

In ultimo, in questa sezione, sono state mostrate diverse matrici facenti riferimento a diversi possibili test legati alla qualità; questo è stato necessario in quanto, definendo una qualità tendente al 50, le matrici di allineamento, di direzione e di programmazione dinamica risultavano graficamente poco chiare a causa della dimensione delle stesse.

4.5 Funzione `buildGraph(graphMat)`

Dopo aver creato la matrice con i vari punteggi di overlap tra tutte le read è necessario capire se è possibile creare un grafo. In caso positivo bisognerà poi definire tale grafo, mediante un determinato formato; il compito di questo metodo è appunto questo.

Si parte leggendo la matrice, riga per riga, al fine di identificare il valore massimo di overlap; partendo dalla prima riga dunque si cercherà la colonna con valore massimo, in questo modo viene identificata la read di partenza, quella di arrivo e l'overlap della prima sulla seconda.
Il grafo di overlap è dunque rappresentato mediante una lista contenente le seguenti triple:

```
(FromRead, OverlapValue, ToRead)
```

In questa funzione viene, come precedentemente accennato, controllato se è possibile creare il grafo; sostanzialmente viene controllato che esista un singolo nodo pozzo, quindi un singolo nodo con valore di overlap pari a 0.
Nel caso in cui ve ne fossero più di uno verrebbe restituito `None`, comportando la stampa di un messaggio di errore dato il metodo successivo.

```
[[0, 67.0, 1], [1, 63.0, 2], [2, 68.0, 3], [3, 67.0, 4], [4, 75.0, 5], [5, 59.0, 6], [6, 62.0, 7], [7, 66.0, 8], [8, 61.0, 9], [9, 66.0, 10], [10, 65.0, 11],
 [11, 61.0, 12], [12, 61.0, 13], [13, 61.0, 14], [14, 63.0, 15], [15, 60.0, 16], [16, 55.0, 17], [17, 0.0, -1]]
```

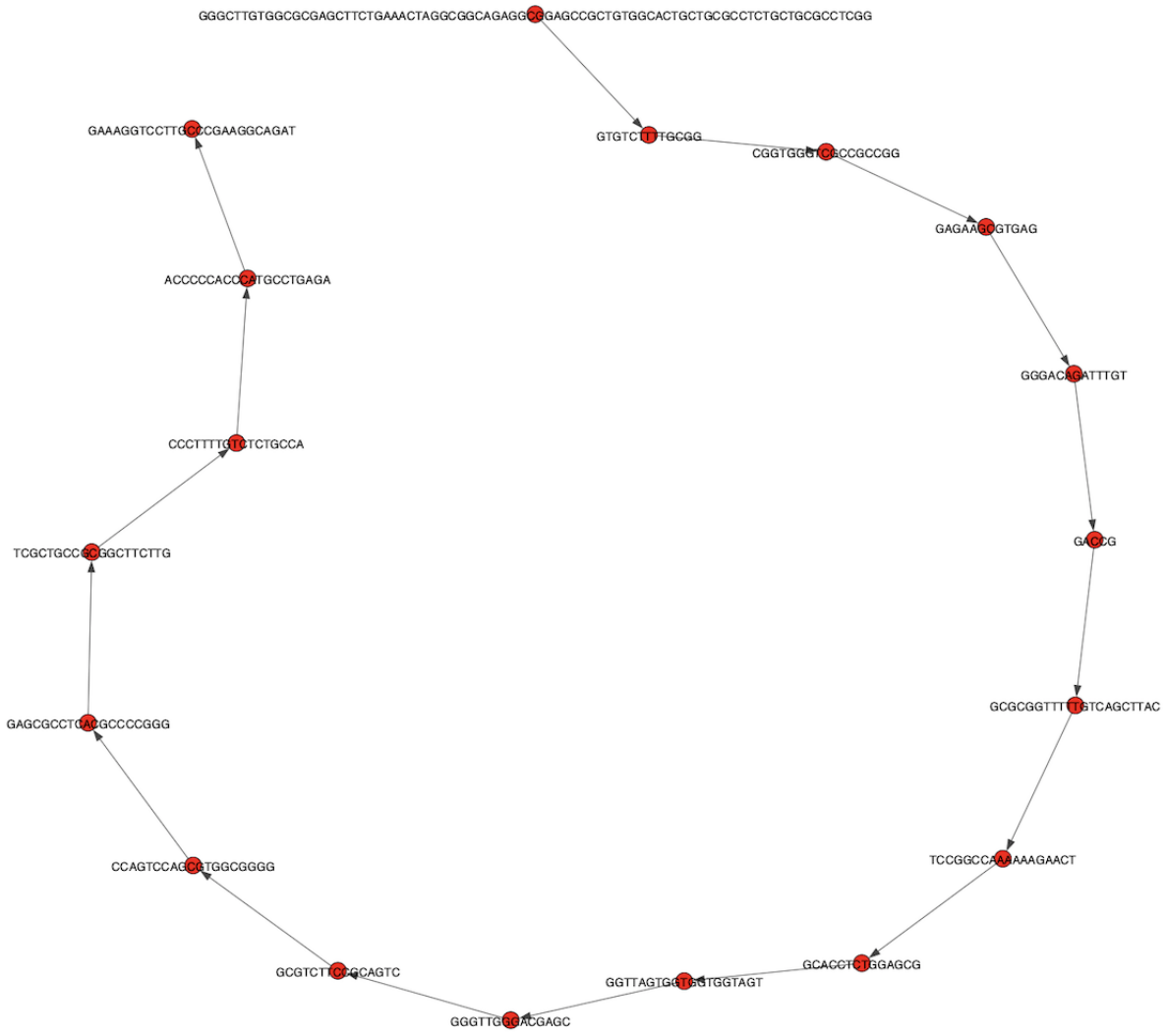
Figura 6: Rappresentazione del grafo di overlap data un'operazione di trimming con qualità 50

4.6 Funzione `printFinalGenoma(graph, reads)`

Quest'ultima funzione si occupa di ricavare, qualora il grafo sia diverso da `None` (vedi sezione 4.5), la sequenza finale. Viene dunque identificato il nodo di partenza, andando a cercare l'indice `FromRead` che non compare mai come indice `ToRead`; una volta identificato tale valore si procederà a salvare la read di partenza su di una stringa rappresentante il genoma finale, per poi passare alla read successiva andando a "tagliare" in funzione dell'overlap. Andiamo quindi ad utilizzare un approccio **right extension**.

Una volta terminato di scandagliare la lista rappresentante il grafo, si valuterà se il numero di "passi" eseguiti è pari al numero di read; se così non fosse verrebbe stampato un messaggio di errore, infatti tale situazione indica che non è stato performato un cammino hamiltoniano.
In caso contrario la stringa ottenuta viene salvata su di un file (`FinalGenomaByOverlap.txt`) così da poter essere poi confrontata con quella fornita.

Dato il numero limitato di reads è stato possibile plottare il grafo di overlap, i nodi rappresentano la parte di read aggiunta alla stringa rappresentate il genoma finale (approccio right extension):


GGGCTTGTGGCGCGAGCTTCTGAAACTAGGCGGCAGAGGCGGAGCCGCTGTGGCACTGCTGCGCCTCTGCTGCGCCTCGG
GTGTCTTTTGCGG
CGGTGGGTCGCCCGCCGG
GAGAAGCGTGAG
GGGACAGATTTGT
GACCG
GCGCGGTTTCTGTCAGCTTAC
TCCGGCCAAAAAAGAACT
GCACCTCTGGAGCG
GGTTAGTGGTGGTGGTAGT
GGGTTGGGACGAGC
GCGTCTTCCGCAGTC
CCAGTCCAGCCTGGCGGGG
GAGCGCCTCACGCCCCGGG
TCGCTGCCCCGGCTTCTTG
CCCTTTTGTCTCTGCCA
ACCCCCACCCATGCCTGAGA
GAAAGGTCCTTGCCCGAAGGCAGAT

5 De Bruijn Graph

5.1 Introduzione

Anche in questa fase siamo partiti dal file generato durante le operazioni di trimming, con l'obiettivo di ottenere un grafo di De Bruijn per la ricostruzione del genoma. Questa fase si divide principalmente in tre ulteriori sotto fasi:

- Lettura delle read pulite nella fase di trimming.
- Costruzione del grafo di De Bruijn partendo dalle read pulite dal trimming.
- Visita del grafo costruito e ricostruzione del genoma.

Successivamente è stata dedicata attenzione all'eventuale presenza di bubbles e tips, provando ad individuarle e a prendere il giusto percorso nel grafo (per evitare di commettere errori).
Le classi e i corrispondenti metodi, che permettono di effettuare le operazioni descritte sopra sono le seguenti:

- **DebrujinG.py** : `debrujin()`
- **ClassDebrujin.py**: `class Edge`, `class Vertex`, `buildGraph(reads, k)`, `readReads(fileReads)`, `printFinalGenoma(debrujin_g)`

5.2 Idea di base

L'idea di utilizzare un approccio ad oggetti è sorta in seguito allo studio del problema e a ciò che ci è stato fatto vedere durante le lezioni; sicuramente un utilizzo degli oggetti "Node" ed "Edge" semplifica di molto la comprensione del grafo effettivo. Abbiamo deciso di utilizzarli anche per poter gestire (o tentare di gestire) gli errori che possono essere presenti in questa tipologia di grafo.

5.3 Funzione `debrujin()`

In questa funzione viene definita la lunghezza dei k-mer, nel nostro caso abbiamo deciso di impostare una valore di k pari a 31; la scelta di tale valore è stata guidata dalle spiegazioni dei professori e dal fatto che, maggiore è questo valore, maggiori sono le possibilità di evitare "ambiguità" nel grafo date regioni simili del genoma. Abbiamo comunque notato che il genoma è correttamente ricostruito con k maggiore o uguale a 13.

In seguito verranno richiamate tutte le funzione che portano alla creazione del grafo di De Bruijn.

5.4 `classe Edge`

La struttura della classe per l'arco del grafo è molto semplice, infatti un arco viene rappresentato solo con la label del vertice a cui porta (nello specifico il testo del k-mero del vertice in cui entra l'arco).

```
#class edge of the DeBrujin graph
class Edge:
    def __init__(self, lab):
        self.label = lab
```

5.5 classe `Vertex`

La struttura della classe per il vertice del grafo non è molto più complessa di quella dell'arco. Rappresentiamo un vertice con una label (stringa k-mero), con indegree e outdegree (rispettivamente il numero di archi entranti e uscenti) e con un occorrenza pensato per memorizzare quante volte abbiamo incontrato questo vertice nella costruzione del grafo; questo perché presupponiamo che i vertici con più occorrenze abbiano più probabilità di essere corretti, mentre quelli con poche occorrenze (solitamente una sola) siano probabilmente errati, portando alla creazione di bolle e/o tips.

```
#class vertex of the DeBrujin graph
class Vertex:
    def __init__(self, lab):
        self.indegree = 0
        self.outdegree = 0
        self.label = lab
        self.occurence = 1
```

5.6 Funzione `readReads(fileReads)`

Questa prima funzione riceve in input il nome di un file e si occupa di recuperare le read trimmate da questo, ritorna la lista delle read lette; pronta per essere passata alla successiva funzione.

5.7 Funzione `buildGraph(reads, k)`

Questa funzione è la "*core function*" dell'intera sezione; crea il grafo ricevendo in input la lista delle read trimmate e un parametro k che indica la lunghezza delle sottostringhe che andremo ad estrarre da ogni read.

Questa funzione lavora su tutti i reads e ad ogni iterazione estrae una coppia di k-meri vicini, partendo dalla coppia 0 - 31 e 1 - 32; data ogni coppia vi sono 4 casi:

- **Entrambi assenti**: si istanziano i due vertici, vengono aumentati, rispettivamente, `indegree` e `outdegree`, il secondo è inserito nella lista di adiacenza del primo.
- **Primo presente, secondo assente**: si aumenta di 1 gli attributi `occurence` e `outdegree` del primo, si istanzia il secondo e gli si aumenta il valore di `indegree` di una unità. Il secondo è inserito nella lista di adiacenza del primo.
- **Primo assente, secondo presente**: si istanzia il primo e li si aumenta l'attributo `outdegree` di un'unità, si inserisce il secondo nella lista di adiacenza del primo. Al secondo si incrementa `occurence` e `indegree` di un'unità.

- **Entrambi presenti**: si aumenta di un'unità l'attributo `occurence` di entrambi.

In sostanza creiamo due dizionari: `vertex` ed `edge`; il primo rappresentante i vertici univoci trovati, il secondo la lista di adiacenza per ognuno di questi. Il primo così strutturato:

```
{k-merLabel:  <Vertex object>}
```

Il secondo:

```
{k-merLabel:  [<Edge object>]}
```

Una volta terminata la creazione dei k-meri per ogni read, si ritorna la coppia di dizionari rappresentanti, di fatto, il nostro grafo di De Bruijn.

5.8 Funzione `printFinalGenoma(debrujin_g)`

Questa funzione riceve una lista contenente i due dizionari sopracitati, il suo compito è restituire il genoma finale. Si effettua in prima battuta un controllo sull'esistenza dei k-mer, questo perché, data la funzione di trimming e il valore di k, non è detto che si sia riusciti effettivamente a creare i k-mer (es: qualità troppo alta, read corte e k troppo alto).

Tolta l'identificazione degli errori, la quale verrà trattata nella prossima sottosezione, individuiamo il vertice con `indegree` minore (nessun arco proveniente da nessun altro vertice porta a lui) e `outdegree` maggiore di 0 (altrimenti la visita finirebbe immadiatamente); da questo preleviamo tutta la stringa `label` (è il primo nodo) e partiamo con la visita. Data la lista di adiacenza di ogni nodo, andiamo a prelevare l'arco che porta al nodo con maggior occorrenza; questo è giustificato dal fatto che assumiamo che un grande valore di `occurence` corrisponda a una minore probabilità di errore. Una volta identificato il nodo sul quale passare, aggiungiamo alla stringa rappresentate il genoma l'ultimo carattere della label del nuovo nodo e reiteriamo quando specificato con il nuovo nodo.

Terminiamo la visita una volta che ci troviamo su di un nodo con lista di adiacenza vuota, il quale rappresenta appunto l'ultimo nodo. A questo punto restituiamo il risultato della visita, lo stesso verrà salvato su di un file.

5.9 Gestione Bubbles e Tip

Grazie all'approccio ad oggetti, agli attributi di questi e a un buon grado di pazienza, siamo riusciti a implementare un primo elementare sistema di "*error detection*" al fine di identificare il numero e la posizione di bubbles e tips.

Nelle righe che vanno dalla 133 alla 170 del file `ClassDebrujin.py` effettuiamo tutte le operazioni in merito. Per prima cosa inizializziamo tre contatori:

- `unbalanced`: numero di vertici sbilanciati con valori di `outdegree` e `indegree` diversi da 0; contatore inizialmente posto a 0. Sostanzialmente nodi che rappresento inizio/fine bubbles o biforcazioni a causa di tips.

- `outDegreeZero`: numero di vertici con `outdegree` a 0; inzialmente posto a -1 in quanto accettiamo di avere un nodo con outdegree uguale a 0, che sarà il nodo finale. Sostanzialmente tips in uscita.

- `inDegreeZero`: numero di vertici con `indegree` a 0; inzialmente posto a -1 in quanto accettiamo di avere un nodo con `indegree` uguale a 0, che sarà il nodo iniziale. Sostanzialmente tips in ingresso.

Iteriamo quindi tutti i nodi incrementando, data la situazione, i valore dei contatori stampando anche le informazioni dei nodi che comportano un incremento di questi. Stampiamo quindi i valori dei contatori. Andiamo a sottrarre dal valore di `unbalanced` la somma di `outDegreeZero` e `inDegreeZero` in quanto ogni Tips crea una biforcazione la quale non è da considerarsi come inizio/fine di una bolla.
Infine, se il valore di `unbalanced` è maggiore di 0, questo rappresenterà il numero di nodi di inizio e fine delle bolle; la metà di questo valore sarà quindi il numero di bolle. Se la somma tra `inDegreeZero` e `outDegreeZero` è maggiore di 0, questo valore rappresentarà il numero di Tips.

Introducendo, ad esempio, il seguente insieme di read (in rosso grassetto gli errori):

```
GGGCTTGTGGCGCGAGCTTCTGAAACTAGGCGGCAGAGGCGGAGCCGCTGTGGCACTGCTGCGCCTCTGCTGCGCCTCGG
GAGCTTCTGAAACTAGGCGGCAGAGGCGGAGCCGCTGTGGCACTGCTGCGCCTCTGCTGCGCCTCGGGTGTCTTTTGCGG
AGGCAGAGGCGGAGCCGCTGTGGCACTGCTGCGCCTCTGCTGCGCCTCGGGTGTCTTTTGCGGCGGTGGGTCGCCGCCGG
AGCCGCTGTGGCACTGCTGCGCCTCTGCTGCGCCTCGGGTGTCTTTTGCGGCGGTGGGTCGCCGCCGGGAGGAGCGTGAG
CTGCTGCGCCTCTGCTGCGCCTCGGGTGTCTTTTGCGGCGGTGGGTCGCCGCCGGGAGAAGCGTGAGGGGACAGATTTGT
GCGCCTCTGCTGCGCCTCGGGTGTCTTTTGCGGCGGTGGGTCGCCGCCGGGAGAAGCGTGAGGGGACAGATTTGTGACCG
TGTCTTTTGCGGCGGTGGGTCGCCGCCGGGAGAAGCGTGAAGGGACAGATTTGTGACCGGCGCGGTTTTTGTCAGCTTAC
GTCGCCGCCGGGAGAAGCGTGAGGGGACAGATTTGTGACCGGCGCGGTTTTTGTCAGCTTACTCCGGCCAAAAAAGAACT
AAGCGTGAGGGGACAGATTTGTGACCGGCGCGGTTTTTGTCAGCTTACTCCGGCCAAAAAAGAACTGCACCTCTGGAGCG
TGTGACCGGCGCGGTTTTTGTCAGCTTACTCCGGCCAAAAAAGAACTGCACCTCTGGAGCGGGTTAGTGGTGGTGGTAGT
TTGTTGTCAGCTTACTCCGGCCAAAAAAGAACTGCACCTCTGGAGCGGGTTAGTGGTGGTGGTAGTGGGTTGGGACGAGC
TCCGGCCAAAAAAGAACTGCACCTCTGGAGCGGGTTAGTGGTGGTGGTAGTGGGTTGGGACGAGCGCGTCTTCCGCAGTC
CACCTCTGGAGCGGGTTAGTGGTGGTGGTAGTGGGTTGGGACGAGCGCGTCTTCCGCAGTCCCAGTCCAGCGTGGCGGGG
TGGTGGTGGTAGTGGGTTGGGACGAGCGCGTCTTCCGCAGTCCCAGTCCAGCGTGGCGGGGGAGCGCCTCACGCCCCGGG
GGACGAGCGCGTCTTCCGCAGTCCCAGTCCAGCGTGGCGGGGGAGCGCGTCACGCCCCGGGTCGCTGCCGCGGCTTCTTG
GCAGTCCCAGTCCAGCGTGGCGGGGGAGCGCCTCACGCCCCGGGTCGCTGCCGCGGCTTCTTGCCCTTTTGTCTCTGCCA
CGGGGGAGCGCCTCACGCCCCGGGTCGCTGCCGCGGCTTCTTGCCCTTTTGTCTCTGCCAACCCCCACCCATGCCTGAGA
CGCTGCCGCGGCTTCTTGCCCTTTTGTCTCTGCCAACCCCCACCCATGCCTGAGAGAAAGGTCCTTGCCCGAAGGCAGAT
```

Figura 7: Read trimmate con quality 50, lunghezza 80, errori aggiunti a "*mano*"

Otteniamo così il seguente output:

```
unbalanced before subtraction:  7
 unbalanced after subtraction:  4
      indegree equals to zero:  2
        outdegree equals zero:  1
There are  2 bubbles in the Bruijin graph
There are  3 tips in the Bruijin graph
```

Per la stampa corretta del genoma ci affidiamo all'ipotesi (e speranza) che ciò che è corretto si ripeta più volte mentre i vari errori (che possono essere anche diversi tra loro) compariranno poche volte (solitamente una volta).

6 Memory Check

6.1 Introduzione

Per misurare la quantità di memoria impiegata da ogni singola fase della pipeline di assemblaggio è stata utilizzata la funzione **measure_memory_usage**, il cui funzionamento è basato sulla funzione **memory_usage** appartenente alla libreria **memory_profiler** fornita da python.
La funzione di controllo è stata definita all'interno dello script **memory_control.py**, e viene richiamata all'interno dello script **main.py** una volta per ogni step da computare.

6.2 Funzione `measure_memory_usage()`

Gli argomenti che questa funzione prende in ingresso sono elencati di seguito nell'ordine da rispettare.

- **target_call :** corrisponde alla chiamata della funzione da analizzare.
- **target_args :** se la funzione da analizzare richiede uno o più argomenti in input, allora devono essere passati come una tupla di argomenti che rispetti l'ordine della funzione sotto analisi.
- **memory_denominator :** questo parametro permette di settare la risoluzione dell'unita di misura da utilizzare per l'output della funzione, il valore di default è in MB (1024**2).
- **memory_usage_refresh :** questo parametro permette di settare la frequenza con la quale si desidera sequenziare la memoria, il valore di default è 0.005 secondi.

L'output generato da questa funzione corrisponde alla quantità di memoria utilizzata dalla funzione passata come argomento, la cui unita di misura viene definita in input.
Per il nostro progetto abbiamo deciso di utilizzare **KB** come risoluzione, questo per rendere possibile un più chiaro confronto tra i grafi.
La funzione è rinvenibile in :
`https://gist.github.com/ds7711/bd2f817090eca8f66b9b23f4805f7cd6`.

7 Confronto e Conclusioni

7.1 Introduzione

Sin dall'inizio dell'assignment la nostra idea era confrontare i due modelli per valutarne la bontà nell'assemblaggio, analizzandone i punti di forza e le debolezze facendo riferimento solo al genoma preso in considerazione. Dato che entrambi i modelli ottengono lo stesso genoma finale, possiamo affermare (banalmente) che servono allo scopo per cui sono stati pensati. Abbiamo scelto di analizzare due misure per provare a capire le prestazioni dei modelli:

- La durata dell'esecuzione completa di ognuna delle due fasi di assemblaggio, misurata con il formato datetime.
- La memoria RAM occupata per completare l'esecuzione dei grafi di assemblaggio, misurata in KB.

Le misure sono state raccolte sulle 2 fasi di assemblaggio genomico che compongono la nostra pipeline al fine di poter confrontare e capire quanto questi due modelli si prestino bene a risolvere il nostro problema.

7.2 Tempo di esecuzione

I tempi di esecuzione dell'overlap graph sono sensibilmente maggiori confrontati al grafo di De Bruijn, questo perché viene creata una matrice 18x18 (logica All vs All) per i punteggi di overlap, questi vengono ottenuti utilizzando due ulteriori matrici 80x80. Nel caso di quality impostata a 50 arriviamo a gestire in totale 578 matrici di dimensione 80x80 l'una. Il tempo si attesta intorno ai 04.536332 secondi.

Il grafo di De Bruijn ottiene performance molto inferiori rispetto all'overlap graph: in media una esecuzione di questo processo impiega un 00.007184 secondi per terminare l'assemblaggio. Questo è dovuto al fatto che il grafo di De Bruijn risultante è molto simile ad una lista: infatti presenta un nodo iniziale con `indegree` = 0 e `outdegree` = 1, uno finale con `indegree` = 1 e `outdegree` = 0 e i restanti nodi hanno `indegree` = 1 e `outdegree` = 1. Quindi una volta identificato il nodo di partenza, si effettua un cammino euleriano (nel caso in cui non vi siano bubbles o tips ovviamente), saltando da un nodo all'altro prelevando, di fatto, il primo nodo dalla lista di adiacenza di ognuno. La "catena" che rappresenta il grafo in questione è composta da 330 nodi, i quali rappresentano i 330 k-meri univoci estratti.

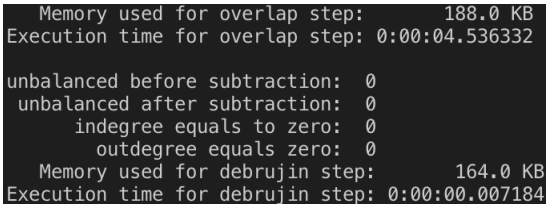
7.3 Memoria in esecuzione

La memoria RAM occupata dall'overlap graph si aggira intorno ai 188.0 KB, questo sempre perché dobbiamo gestire un buon numero di matrici, come sopra riportato.

Il grafo di De Bruijn invece occupa meno memoria RAM rispetto all'overlap graph: in media una esecuzione di questo processo occupa circa 164.0 KB di memoria RAM prima di terminare correttamente. Questo è dovuto al fatto che le strutture dati per archi

e nodi sono relativamente leggere (4 attributi di cui 3 interi per i nodi e 1 per gli archi) ma anche alla poca numerosità degli archi; infatti per ogni nodo abbiamo quasi sempre 1 solo arco uscente e un solo arco entrante, il che implica che le liste di adiacenza di ogni nodo contengono solo un arco (rappresentato dal nodo raggiungibile).

Dato in input il file di reads con errori introdotti appositamente e presentato nella sezione 5.8, o più in generale tenendo conto della possibile presenza di errori, il grafo di De Bruijn impiega più memoria, nel caso specifico 65 KB in più rispetto alla memoria in assenza di errori; questo è principalmente causato dal fatto che vengono creati più k-mer e inoltre le liste di adiacenza dei nodi aumentano di numerosità. La sequenza finale tuttavia risulta essere corretta a differenza del grafo di Overlap, il quale però mantiene costante l'utilizzo di RAM.



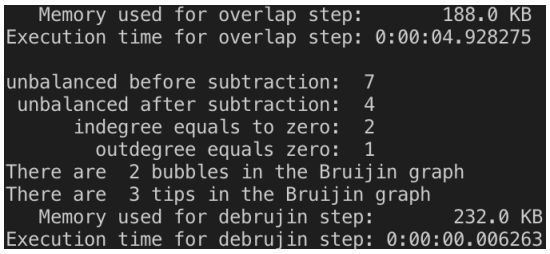


Figura 8: Misure di tempo e memoria in assenza e presenza di errori.

NB: le misure sopra riportate sono state prese da un MacBook Pro con:

- OS: macOS Mojave 10.14.4
- Processore: i7 Inter Core 2,5 GHz
- Memoria: 16 GB LPDDR3

Le misure raccolte potrebbero variare in funzione dell'architettura e delle caratteristiche hardware.

7.4 Conclusioni

Il progetto si è rivelato essere utile al fine di comprendere le strutture e gli algoritmi presentati per l'allineamento, permettendoci inoltre di toccare con mano le principali problematiche di questa fase.

Lo sviluppo di entrambe le metodologie è stato affiancato da uno studio dei grafi e delle situazioni che si potevano venire a creare, possiamo osservare che il grafo di De Bruijn risulta essere molto più "snello" e "solido" rispetto a quello di Overlap; riteniamo sia più solido in quanto, dati gli errori rilevati e gestiti a cui si fa riferimento nella sezione 5.8, questo riesce a restituire la sequenza finale corretta a differenza dell'overlap graph. Il secondo infatti, con soglia errore settata al 3%, accetta errori producendo una sequenza non corretta; nel caso in cui la soglia di errore accettato fosse pari a 0 invece, non riesce a produrre una sequenza finale.

7.5 Suddivisione del lavoro

Il lavoro è stato così suddiviso:

- Matamoros Aragon Ricardo Anibal: `trimming` e `memory check`
- Palazzi Bruno: `De Bruijn Graph`
- Villa Giacomo: `Overlap graph`

In generale, tutto il codice è stato supervisionato e compreso da tutti i membri del gruppo.

Il codice di tutti i file `.py` consegnati risulta essere commentato in ogni sua parte