Tool per l'assemblaggio di grafici di DeBruijn

Il tool è stato creato utilizzando linguaggio Python ed è composto da due file, il primo, **graph.py** contenente la classe **DeBruijnGraph** e relativi metodi e un file utils.py contenente metodi utili alla lettura di reads e la loro divisione in kmers per il successivo assemblaggio.

Il file **assembler.py** contiene la lettura del file di input in formato FASTA (**input**.**fa**) e il finale assemblamento della sequenza per l'output finale che viene salvato nel file di testo 'output.txt' mentre il grafo viene scritto sul file '**graph.gv**' in formato DOT

Il file main.py è utilizzato per fornire un esempio del funzionamento del tool evocando il metodo '**assemble_reads**' di assembler, che prende in input il file FASTA e il valore K con cui creare i k-mer per l'assemblaggio

Utils.py

- **read_input(path)**
  metodo che prende in input il file FASTA e ne riporta i contenuti in una lista di oggetti di tipo Sequence

- **splice(read, k)**
  metodo che prende in input una lista di reads e il valore numerico K e ritorna la lista di stringhe formata dai relativi k-mer

Graph.py

- **class Vertice**
  Oggetto principale del grafo usato per rappresentare i nodi, contenente **label** (k-1-mer) e due contatori (**in_degree, out_degree**) per tenere traccia del grado del nodo.
  - **isSemiBalanced(self)**
    ritorna True se il nodo è semi bilanciato
  - **isBalanced(self)**
    ritorna True se il nodo è bilanciato
- **__init__(kmers)**
  Costruttore della classe, prende in input una lista di k-mers e costruisce il grafo su un multi-dizionario dal formato {Nodo_sorgente : [lista_nodi_destinazione]}
  Ogni k-mer viene diviso in due k-1-mers e, in caso non siano gia presenti come nodi, vengono creati due oggetti Vertice con per label il relativo k-1-mer.

Viene poi controllato se il k-1-mer di sinistra (vertice_L) esista gia come chiave nel dizionario per aggiungerci l'arco relativo al k-1-mer di destra (vertice_R), in caso contrario vertice_L viene aggiunto come chiave con relativo primo arco verso vertice_R.

Una copia del dizionario è creata poi prima che gli archi duplicati siano rimossi per l'assemblamento della superstringa finale

- **isDupe(verticeL, vericeR)**
  Metodo che controlla se un arco da verticeL a verticeR esista gia, utilizzato per il conteggio degli archi di entrata e di uscita di un nodo
- **balanceCount()**
  Metodo che conta il numero di nodi bilanciati, semibilanciati e sbilanciati presenti nel grafo, usato per riconoscere se il grafo costruito presenti un cammino Euleriano o meno e per salvare la posizione di testa e di coda del grafo
- **eulerianPath()**
  Metodo che esegue la lettura durante il cammino euleriano del grafo, dopo essersi assicurato che esso sia Euleriano (tips e bubbles non sono gestite quindi risultano in errori)
  Viene creato poi un ciclo euleriano (cosi da avere la lettura anche della coda del grafo per l'assemblamento finale, perché le labels sono lette dai nodi)
  La visita del grafo viene effettuata in maniera ricorsiva, per ogni chiave del dizionario (nodo sorgente) vengono visitati i nodi destinazione presenti nella lista a esso associata che vengono poi usati come nodo sorgente di un ulteriore chiamata del metodo 'visita', ogni label dei nodi visitati viene poi inserita in coda alla lista '**result'**, essendo questa una map di oggetti **Vertice** sará poi ritornata convertita in lista contenente le labels dei relativi nodi per l'assemblaggio finale in '**assembler.py'**
- **isEulerian()**
  Metodo che ritorna True se il grafo è Euleriano utilizzando i metodi **isBalanced()** e **isSemiBalanced()** per controllarne il bilanciamento
- **hasEulerianPath()**
  metodo che controlla l'esistenza o meno di un cammino euleriano (al massimo 2 nodi semi-bilanciati)
- **hasEulerianCycle()**
  metodo che controlla l'esistenza o meno di un ciclo euleriano (tutti i nodi bilanciati)
- **printGraph()**
  Metodo utilizzato per la scrittura del file '**graph.gv'**, utilizza il multi-dizionario copiato in precedenza per stampare in formato DOT il grafo con i relativi pesi degli archi come label semplicemente scorrendo ogni Nodo come chiave e la relativa lista di nodi destinazione associata ad esso tenendo traccia di possibili archi duplicati

Esempio:

INPUT: File FASTA con due read, K = 5:

>Read001

ATCGATAGATGG

>Read002

AGATGGAAG

OUTPUT:

- **Grafo su file graph.gv:**

  ```
  digraph "DeBruijn Graph" {
    bgcolor="white";
    ATCG -> TCGA [label="1"] ;
    TCGA -> CGAT [label="1"] ;
    CGAT -> GATA [label="1"] ;
    GATA -> ATAG [label="1"] ;
    ATAG -> TAGA [label="1"] ;
    TAGA -> AGAT [label="1"] ;
    AGAT -> GATG [label="2"] ;
    GATG -> ATGG [label="2"] ;
    ATGG -> TGGA [label="1"] ;
    TGGA -> GGAA [label="1"] ;
    GGAA -> GAAG [label="1"] ;
  }
  ```

  Grafo scritto in formato DOT (graphviz) in cui sono presenti tutti gli archi con relativo peso (in caso di archi multipli), questo file puó poi essere trasformato in un file .png con dot

- **Superstringa assemblata su file output.txt:**

  ATCGATAGATGGAAG

NB: I risultati di questo test sono inclusi nell'archivio compresso