# Nubeam-dedup: a fast and RAM-efficient tool to de-duplicate sequencing reads without mapping

Veltri Lorenzo, Saguto Federica, Sestito Martina

## 1 Introduzione

L'algoritmo Nubeam-dedup, presentato nell'articolo *Nubeam-dedup: a fast and RAM-efficient tool to de-duplicate sequencing reads without mapping* di Hang Dai e Yongtao Guan (Dipartimento di Biostatistica e Bioinformatica, dalla Duke University School of Medicine, Durham, NC 27705, USA), è stato progettato per rimuovere i duplicati di DNA senza utilizzare un genoma di riferimento, poiché in alcuni casi i genomi di riferimento possono non essere disponibili, incompleti o di bassa qualità.
Le duplicazioni delle letture di DNA provengono dal processo di reazione a catena della polimerasi (PCR), un tecnica utilizzata per amplificare o arricchire molecole di DNA, ottenendo in questo modo delle copie degli stessi frammenti. La PCR può creare artefatti che potrebbero suggerire che una molecola sia più abbondante di un'altra.
L'algoritmo Nubeam-dedup rimuove i duplicati derivanti dalla PCR calcolando, per ogni lettura di DNA, i "numeri Nubeam" nel seguente modo:

1. rappresenta i nucleotidi come matrici;
2. trasforma le letture in prodotti di matrici;
3. assegna un numero univoco (Nubeam) a ciascuna lettura.

Il Nubeam-dedup fornisce una funzione hash per le sequenze di DNA, consentendo una deduplicazione facile ed efficiente in termini di RAM.

# 2 L'algoritmo

L'algoritmo di Hang Dai e Yongtao Guan assume che le letture abbiano la stessa lunghezza $L$ e siano composte da 4 nucleotidi:

- A: Adenina;
- T: Timina;
- C: Citosina;
- G: Guanina;

Si procede costruendo quattro sequenze binarie a partire da una lettura, utilizzando ciascuno dei quattro nucleotidi come riferimento. Indichiamo con 1 il nucleotide di riferimento, mentre gli altri nucleotidi li indicheremo con 0. Successivamente, concateniamo le sequenze binarie ottenute in una singola sequenza binaria $B$ (di lunghezza $4L$) e trasformiamo questa sequenza in una matrice prodotto.
Definiamo la matrice

$$M_1 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \tag{1}$$

per rappresentare 1 e la sua trasposta

$$M_0 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \tag{2}$$

per rappresentare lo 0.
Per la sequenza binaria $B{=}b_1, .., b_{4L}$ , otteniamo una matrice prodotto

$$M_B = \prod_{j=1}^{4L} M_{bj}. \tag{3}$$

Sia

$$W = \begin{pmatrix} 1 & \sqrt{3} \\ \sqrt{2} & \sqrt{5} \end{pmatrix} \tag{4}$$

una matrice peso, definiamo il numero Nubeam come:

$$Nubeam = tr(WM_B) \tag{5}$$

tale numero indica l'unicità della lettura.
Quindi, a sequenze diverse corrispondono numeri Nubeam diversi e viceversa. Due sequenze che hanno stesso numero Nubeam devono essere necessariamente uguali.

# 3 Implementazione su Spark

Per implementare l'algoritmo su PySpark, bisogna creare una connessione con Spark per poi aprire il file FASTQ scelto tramite il comando *.textFile()*.
Il file verrà trattato come un documento di testo composto da "blocchi" di 4 elementi ciascuna, uno per riga:

1. codice identificativo contrassegnato da una "@" iniziale;
2. sequenza di DNA formata da nucleotidi;
3. codice identificativo identico al precedente contrassegnato da un "+" iniziale (opzionale);
4. valutazione di qualità della sequenza formata da una stringa di simboli di lunghezza pari a quella della sequenza di nucleotidi. Ogni simbolo ha un valore ASCII che assegna il punteggio di qualità della singola base.

I dati necessari per la deduplicazione sono le prime due righe di tali blocchi, ovvero l'Identificatore (per valutare se si tratta di un frammento proveniente dal capo del filamento, /1, o dalla coda, /2) e la Sequenza. In base a ciò, si può decidere quale dei due metodi di deduplicazione sia più appropriato utilizzare: **Metodo Single-End o Metodo Paired-End**.
Cominciamo con il dividere le varie sezioni utilizzando il comando *.zipWithIndex()*, che assegna un indice crescente a ciascuna riga del documento scelto, seguito da un *.map()* che rende uguali gli indici delle righe appartenenti ad uno stesso blocco (per esempio le

prime quattro righe = indice 0, seconde quattro righe = indice 1, e così via). In questo modo, è possibile utilizzare il comando *.reduceByKey()* per raggruppare le righe con il medesimo indice.
Prima di eseguire la deduplicazione, isoliamo le sequenze creando due RDD tramite il comando *.filter()* e le chiamiamo dDati1_lecture1 e dDati1_lecture2, composte rispettivamente da frammenti iniziali (/1 o 1:N:0:CAGATC) e finali (/2 o 2:N:0:CAGATC) delle sequenze.
In base al tipo di lettura eseguita sul DNA, il Nubeam_dedup rimuove i duplicati seguendo una logica diversa per SE e PE:

- **Il Metodo Single-End** consiste nel confrontare i numeri Nubeam di tutti i frammenti iniziali delle sequenze per cercare i duplicati. Partendo dall' RDD dDati1_lecture1 creiamo due nuove RDD con il comando *.map()*.
  L'RDD nominata dNubeamSE sarà composta dai numeri Nubeam delle sequenze, ottenuti grazie alla funzione *numero_Nubeam_da_sequenza_basi_azotate()*, come chiave, e dalle sequenze stesse come valore. Mentre dNubeam_compSE avrà la stessa logica della precedente ma si andranno a considerare le sequenze complementari di quelle selezionate piuttosto che quelle originali. Questa scelta è necessaria per non selezionare le sequenze di DNA prese dalla fine del filamento.
  Attraverso il comando *.union()* aggreghiamo le due RDD in una sola (dSE) ed applichiamo il comando *.distinct()* per togliere tutti i duplicati e lasciare la RDD con solo elementi unici. Di conseguenza, dUnordered_set sarà una RDD formata da sequenze uniche e potrà servire per determinare se eventuali sequenze aggiunte in un secondo momento sono o meno duplicati; tale operazione sarà svolta attraverso il confronto dei numeri Nubeam.

- **Il Metodo Paired-End (PE)** consiste nel confrontare le coppie di numeri Nubeam associati a due letture di DNA provenienti da uno stesso frammento. Tale metodo, infatti, considera sia la lettura che si trova all'inizio, sia quella che si trova alla fine del frammento di DNA considerato.
  Utilizziamo *.reducebeKey()* per accoppiare i frammenti iniziali e finali con la stessa chiave, ottenendo l' RDD dDati1_unif, che contiene la chiave, e le coppie di se-

quenze associate alla stessa chiave.

Nell'RDD dNubeamPE calcoliamo il numero Nubeam per ogni coppia di sequenze. Analogamente nell'RDD dNubeam_compPE calcoliamo il numero Nubeam per i complementari.

Quello che otteniamo alla fine sono 2 RDD che contengono le sequenze di ogni filamento con associato il numero Nubeam. Con *.union()* creiamo l'RDD dPE, che contiene tutti i numeri Nubeam delle sequenze e dei rispettivi complementari.

Infine, rimuoviamo i duplicati utilizzando *.distinct()*, che elimina le coppie di frammenti con lo stesso numero Nubeam. In particolare, si vanno a rimuovere le coppie di numeri Nubeam che presentano lo stesso accoppiamento. Quindi, se in una coppia solo uno dei due numeri coincide, o sono entrambi diversi, la coppia viene mantenuta e quindi non è una sequenza duplicata.

# 4 Implementazione sul database NoSQL Neo4j

Abbiamo scelto di implementare il database su Neo4j in quanto questa piattaforma ci consente di rappresentare in modo efficiente i duplicati di sequenze di DNA attraverso un modello grafico.

Grazie alla struttura a grafo di Neo4j, è possibile visualizzare le connessioni tra le sequenze duplicate. Questo ci permette di individuare relazioni complesse che potrebbero essere difficili da identificare con altri tipi di database. Inoltre, questo approccio ci permette di eseguire query efficienti per l'analisi dei legami tra i dati, migliorando la comprensione delle sequenze duplicate.

L'implementazione del database su Neo4j viene fatta caricando i dati iniziali che contengono anche i duplicati. A tal fine si considera l' RDD *dPE*, che contiene:

- i **numeri Nubeam** per tutte le sequenze (/1 o 1:N:0:CAGAT e /2 o 2:N:0:CAGAT) e per i loro complementari;
- le **sequenze** e i loro **complementari**.

Sull'RDD viene eseguito uno *.zipwithIndex()* per aggiungere ad ogni riga un indice che servirà per collegare la lettura iniziale e quella finale di ogni filamento. Successivamente

si esegue un *.map()* per trasformare i dati nel formato **Sequenza1, Nubeam1, Sequenza2, Nubeam2, riga**.
I dati vengono scritti in un file .csv, tramite PySpark, e salvati nella cartella *import* di Neo4j.
Successivamente si crea la connessione tra PySpark e Neo4j e si eseguono le query grazie a cui vengono creati:

- un nodo per ogni sequenza /1 (o 1:N:0:CAGAT) e /2 (o 2:N:0:CAGAT) e per i loro complementari (**284500 nodi**).
  Ad ogni nodo si assegnano le **proprietà**:
  - *seq*: rappresenta la sequenza di nucleotidi
  - *n1*: rappresenta il numero Nubeam della sequenza
  - *r1*: rappresenta l'indice di riga, ovvero la riga del file *.csv* in cui è salvata la sequenza.
- un nodo per ogni numero nubeam delle sequenze /1 (o 1:N:0:CAGAT), eliminando i duplicati grazie alla funzione `DISTINCT` di Neo4j (**132520 nodi**);
- un nodo per ogni numero nubeam delle sequenze /2 (o 2:N:0:CAGAT), eliminando i duplicati grazie alla funzione `DISTINCT` di Neo4j (**132866 nodi**);
- un arco, di tipo *Ass1*, che colleghi ogni sequenza /1 (o 1:N:0:CAGAT) al corrispondente numero Nubeam calcolato a partire da questa (**142250 archi**);
- un arco, di tipo *Ass2*, che colleghi ogni sequenza /2 (o 2:N:0:CAGAT) al corrispondente numero Nubeam calcolato a partire da questa (**142250 archi**);
- un arco, di tipo *Coppia*, che colleghi ogni sequenza /1 (o 1:N:0:CAGAT) alla corrispondente sequenza /2 (o 2:N:0:CAGAT). Sequenze con lo stesso indice di riga si uniscono tramite l'arco *Coppia* (**142250 archi**).

Inoltre sono state scritte due query per mostrare graficamente:

- i **duplicati della single-end**

```
MATCH p=(n1)-[r:ASS1]->(s1)
WITH n1, collect(p) AS percorsi
WHERE size(percorsi)>1
RETURN percorsi
```

Eseguendo questa query sul dataset "giocattolo" ciò che si ottiene è un grafo di questo tipo:

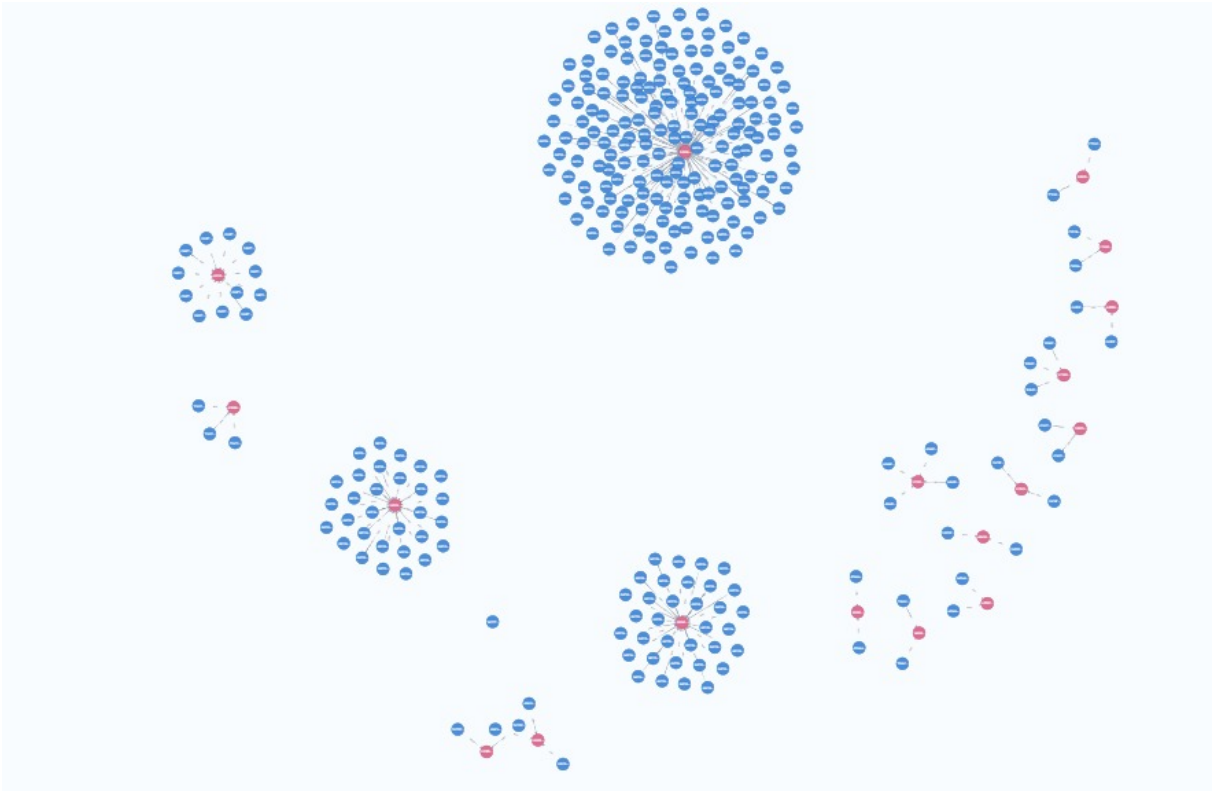

Figure 1: Grafo duplicati per il metodo single-end sul dataset "giocattolo"

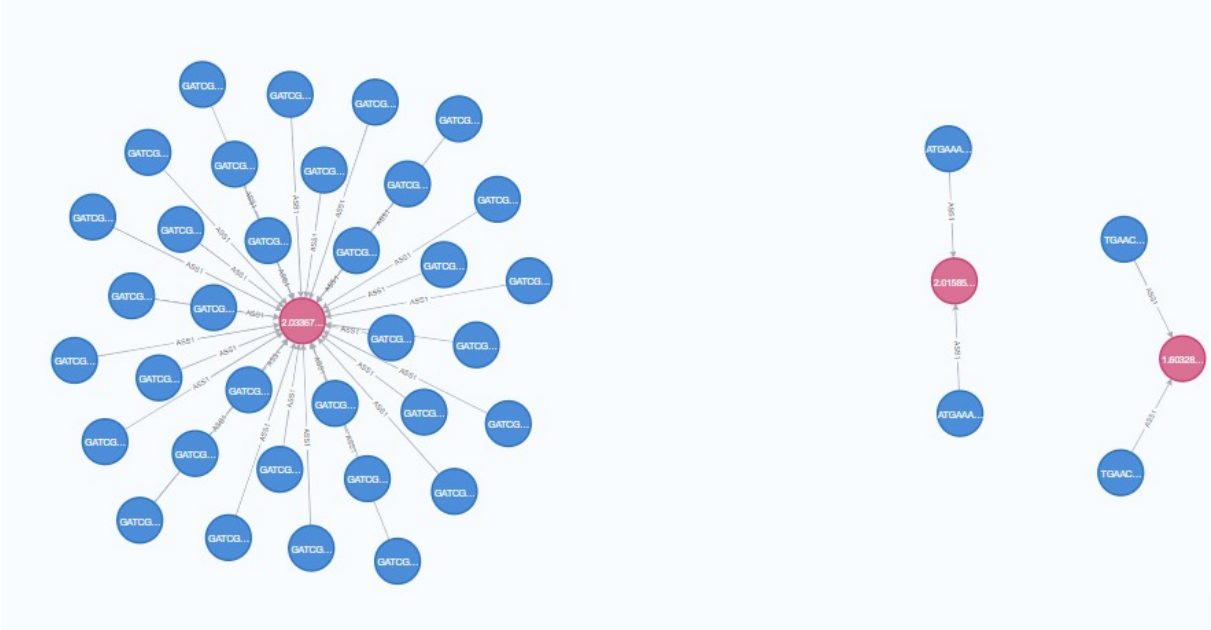

Figure 2: Ingrandimento di un nodo del grafo del dataset "giocattolo" per il metodo single-end

Quello che si nota da questo grafo è che per una sequenza può esserci anche più di un duplicato (più di due archi tra un nodo Nubeam (rosso) e un nodo sequenza (blu)).

Eseguendo la query sul dataset contenente il DNA dei girasoli si ottiene un grafo di questo tipo:

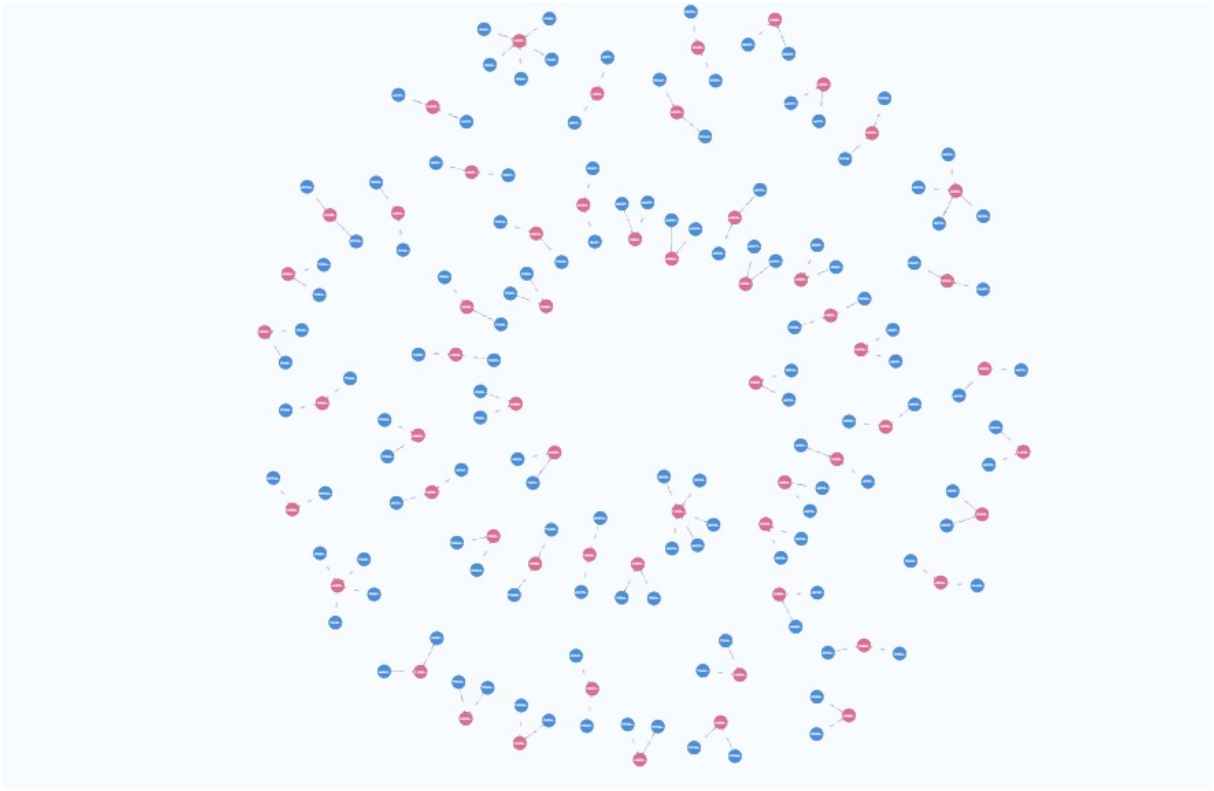

Figure 3: Grafo duplicati per il metodo single-end sul dataset girasoli

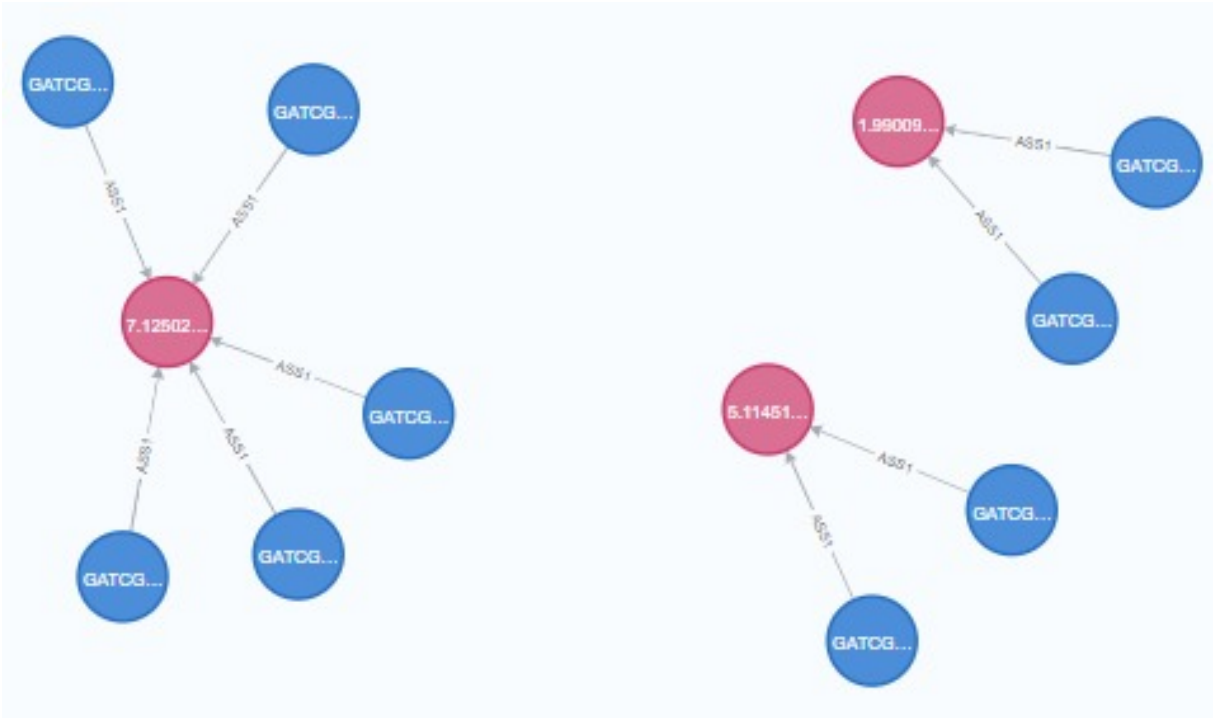


Figure 4: Ingrandimento di un nodo del grafo del dataset girasoli per il metodo single-end

Questi grafi mostrano che la maggior parte delle sequenze presenta un solo duplicato.

Questo può essere spiegato considerando la natura dei dataset. Il dataset "giocattolo" è stato progettato appositamente per lo sviluppo dell'algoritmo, quindi mostra anche il caso in cui una sequenza abbia duplicati multipli. Al contrario il dataset del DNA dei girasoli rappresenta una situazione più reale in cui non è noto come le sequenze siano state processate prima di entrare nel processo di deduplicazione.

- i **duplicati della paired-end**

```
MATCH p = (n1)-[a1:ASS1]-(s1)-[*]->(n2)
WITH n1, n2, collect(p) AS percorsi
WHERE size(percorsi) >= 2
RETURN n1,n2,percorsi
```

Come per il metodo single-end, anche in questo caso, l'output dipende dal dataset per cui si esegue la query. Nel caso del dataset "giocattolo" si ottiene un grafo di questo tipo:

Figure 5: Grafo duplicati per il metodo paired-end sul dataset "giocattolo"

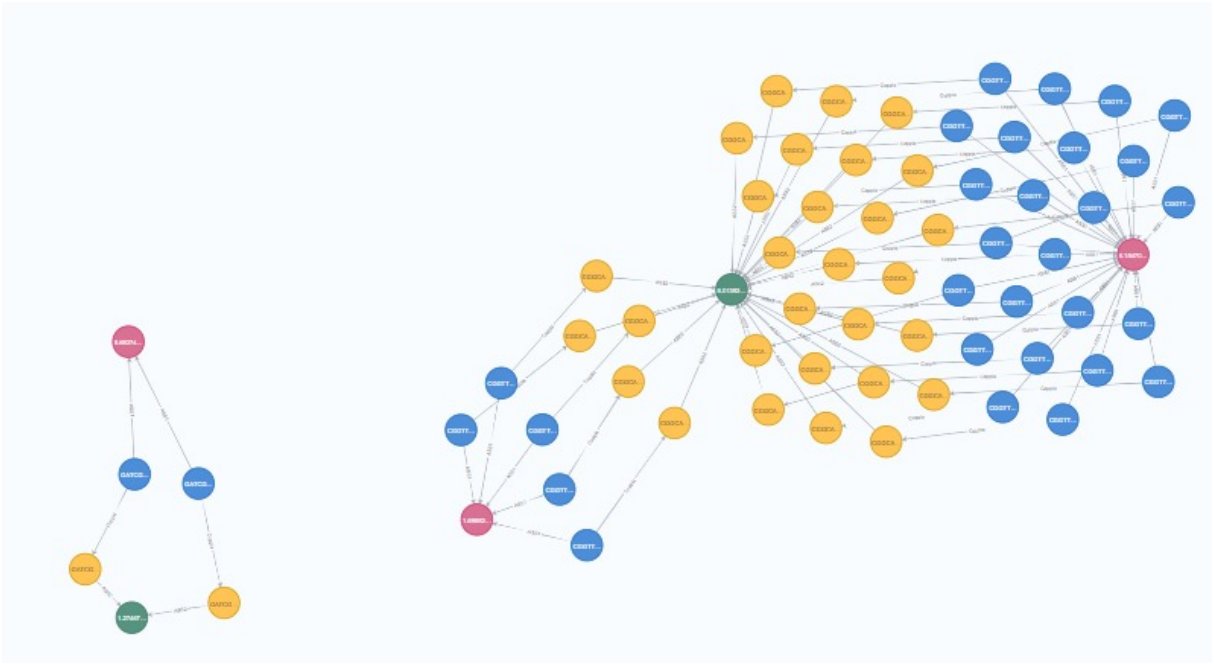

Figure 6: Ingrandimento di un nodo del grafo del dataset "giocattolo" per il metodo paired-end

Nel caso del dataset del DNA dei girasoli si ottiene un grafo di questo tipo:

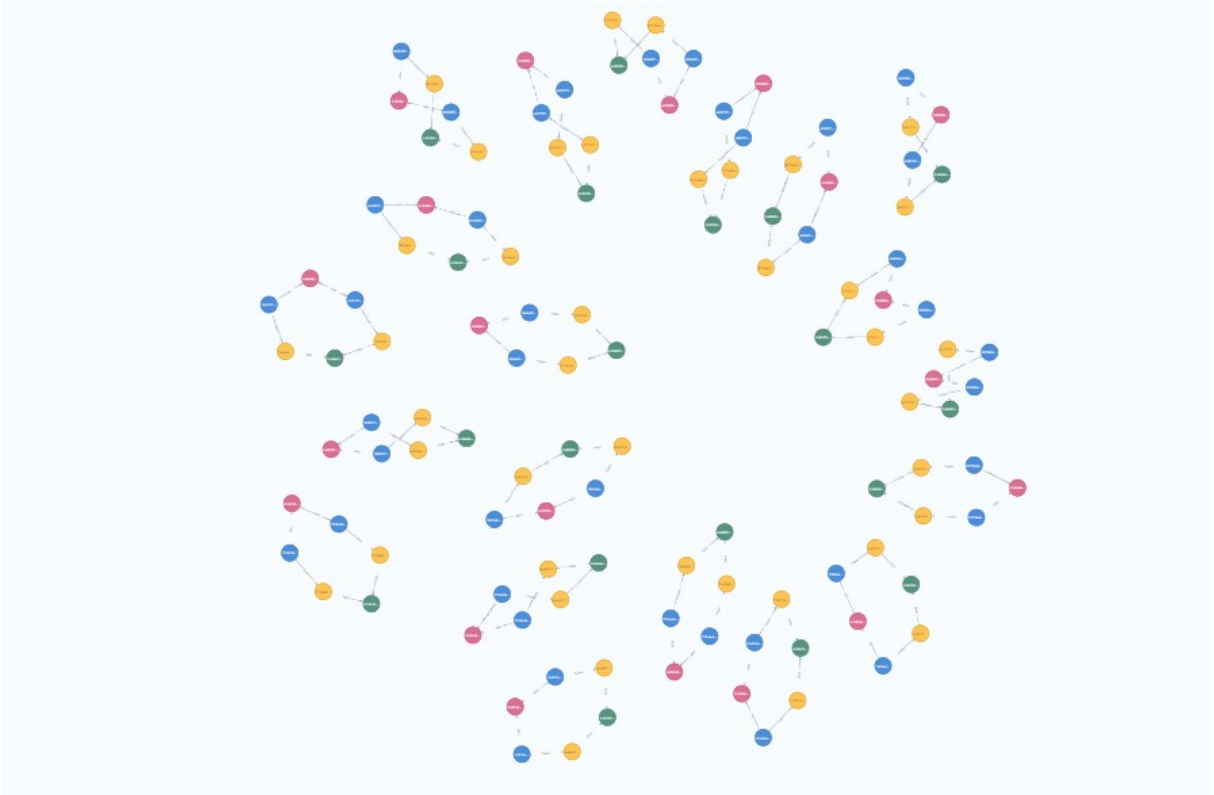

Figure 7: Grafo duplicati per il metodo paired-end sul dataset girasoli

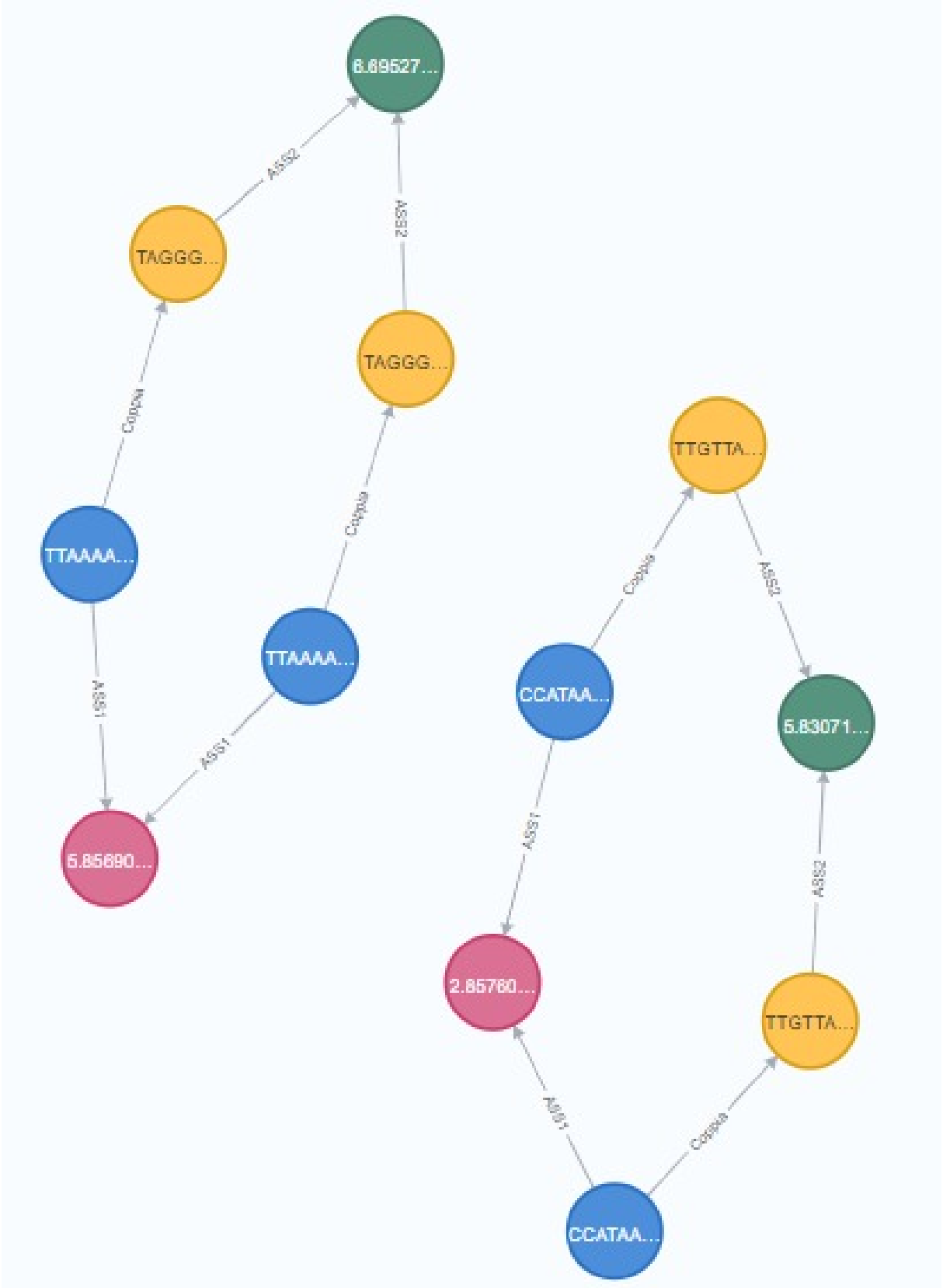


Figure 8: Ingrandimento di un nodo del grafo del dataset girasoli per il metodo paired-end

La differenza tra i due è dovuta allo stesso motivo illustrato per il metodo single-end.

# 5 Risultati

In questa sezione vengono analizzati i risultati dell'esecuzione dell'algoritmo **Nubeam-dedup** successivamente all'implementazione su *Apache Spark*.
In primo luogo si analizzano, distinguendo tra i vari dataset su cui è stato eseguito l'algoritmo, i risultati statistici dell'output ottenuto:

- Considerando il **dataset "giocattolo"** costruito da *Hang Dai* e *Yongtao Guan* appositamente per implementare l'algoritmo si ha che le letture di DNA presenti nel file sono **142250**, di cui metà (**71125**) sono letture della parte iniziale del filamento e l'altra metà (**71125**) sono letture della parte finale del filamento.
  Con il **metodo della single-end**, quindi analizzando solo le letture della parte iniziale del filamento di DNA, vengono eliminati **9730 duplicati**. Questi corrispondo al 13.68% **del totale delle letture della parte iniziale** del filamento e al 6.84% **del totale delle letture** presenti nel dataset.
  Con il metodo della paired-end vengono eliminati **2252 duplicati** che corrispondo all' 1.45% **del totale delle letture** presenti nel dataset.

- L'algoritmo è stato applicato anche a un altro dataset che contiene **279535 letture** di DNA proveniente dal DNA dei girasoli, di queste metà (**137757**) sono letture della parte iniziale del filamento e l'altra metà (**137757**) sono letture della parte finale del filamento. In questo caso con il metodo della single-end sono stati eliminati **4089 duplicati** che corrispondo all' **2.96% del totale delle letture della parte iniziale del filamento** e all'**1.46% del totale delle letture** presenti nel dataset.
  Con il metodo della paired-end sono stati eliminati **4041 duplicati**, **l'1.46% del totale delle letture** presenti nel dataset.

L'analisi dei risultati computazionali ci mostra che:

- nel dataset "giocattolo" l'algoritmo impiega i tempi mostrati in tabella per completare le varie azioni

| Passo | Descrizione | Durata | Tasks | Dati input | Shuffle read | Shuffle write |
|---|---|---|---|---|---|---|
| 5 | **distinct** | 31 s | 8/8 | | 21.3 MiB | 20.0 MiB |
| 3 | **reduceByKey** | 0.6 s | 4/4 | | 41.9 MiB | 10.6 MiB |
| 2 | **distinct** | 21 s | 4/4 | | 41.9 MiB | 9.6 MiB |
| 1 | **reduceByKey** | 0,8 s | 2/2 | 37.8 MiB | | 20.9 MiB |
| 0 | **zipWithIndex** | 0,9 s | 2/2 | 37.8 MiB | | |

Table 1: Risultati computazionali algoritmo Nubeam-dedup dataset "giocattolo"

Ciò che si evince dalla tabella è che per la **preparazione dei dati alla duplicazione l'algoritmo impiega circa 2 secondi** di cui 0.9 secondi sono impiegati per lo *zipWithIndex()* e 0.8 secondi per il *reduceByKey()*. L'algortimo prende in input 37.8 MiB di dati che processa tramite lo *zipWithIndex()* e il *reduceByKey()* per renderli adatti alla deduplicazione. Queste operazioni richiedono entrambe l'uso di 2 processori, i dati vengono divisi in due parti e ridistribuiti a due nodi del sistema; ognuno dei quali è responsabile per quella parte. Inoltre il *reduceByKey()* causa lo scambio di 20.9 MiB di dati in scrittura fra i nodi del sistema. Questa fase è comune a entrambi i metodi di deduplicazione per questo verrà considerata per entrambi nel calcolo del tempo totale di esecuzione e dei dati scambiati.

**L'eliminazione dei duplicati per il metodo single-end avviene in 22.7 secondi**, di questi: 1.7 secondi sono impiegati per la preparazione dei dati alla deduplicazione e 21 secondi per il *distinct()*, cioè per la rimozione vera e propria dei duplicati. Il *distinct()* elimina i duplicati e per farlo utilizza 4 processori, questo significa che i dati vengono divisi in 4 parti che vengono ridistribuite tra 4 nodi. Per svolgere il suo compito il *distinct()* scambia tra i nodi 41.9 MiB di dati in lettura e 9.6 Mib di dati in scrittura, per un totale di 51.5 MiB di dati scambiati tra i nodi. **L'eliminazione dei duplicati con il metodo single-end comporta lo scambio tra i nodi di 72.4 MiB di dati per il totale delle operazioni**.

**L'eliminazione dei duplicati per il metodo paired-end richiede 33.3 sec-**

**ondi** di cui 1.6 secondi sono impiegati per la preparazione dei dati alla deduplicazione, 0.6 secondi per il *reduceByKey()* e 31 secondi per l'eliminazione vera e propria dei duplicati (*distinct()*).

Come si può notare dalla tabella è assente il passo 4, questo perchè Spark utilizza dei risultati che sono già stati ottenuti per ridurre lo scambio di dati e il tempo dell'intera operazione

Per eseguire il *reduceByKey()*, nel metodo paired-end, vengono utilizzati 4 processori con uno scambio di dati pari a 41.9 MiB in lettura e 10.6 MiB in scrittura. Per la deduplicazione vengono utilizzati 8 processori con uno scambio di dati pari a 21.3 MiB in lettura e 20.0 MiB in scrittura. **Per eseguire la deduplicazione con il metodo paired-end vengono scambiati tra i nodi un totale di 114.7 MiB di dati**. In termini di tempo e di dati scambiati fra i nodi del sistema, **l'eliminazione dei duplicati tramite il metodo paired-end è il più costoso** (33.3 second e 114.7 MiB).

- nel secondo dataset contenente letture di DNA dei girasoli l'algoritmo impiega i tempi mostrati in tabella per completare le varie azioni

| Passo | Descrizione | Durata | Tasks | Dati input | Shuffle read | Shuffle write |
|---|---|---|---|---|---|---|
| 5 | **distinct** | 1.7 min | 12/12 | | 61.9 MiB | 59.4 MiB |
| 3 | **reduceByKey** | 1 s | 6/6 | | 102.2 MiB | 30.9 MiB |
| 2 | **distinct** | 56 s | 6/6 | | 102.2 MiB | 29.6 MiB |
| 1 | **reduceByKey** | 1 s | 3/3 | 95.2 MiB | | 51.1 MiB |
| 0 | **zipWithIndex** | 1 s | 3/3 | 95.2 MiB | | |

Table 2: Risultati computazionali algoritmo Nubeam-dedup dataset girasoli

Ciò che si evince dalla tabella è che per la **preparazione dei dati alla duplicazione l'algoritmo impiega 2 secondi** di cui 1 secondo è impiegati per lo *zipWithIndex()* e 1 secondo per il *reduceByKey()*. L'algortimo prende in input 95.2 MiB di dati che processa tramite lo *zipWithIndex()* e il *reduceByKey()* per renderli adatti alla deduplicazione. Queste operazioni richiedono entrambe l'uso di 3 processori, i dati vengono divisi in tre parti e ridistribuiti a tre nodi del sistema; ognuno

dei quali è responsabile per quella parte. Inoltre il *reduceByKey()* causa lo scambio di 51.1 MiB di dati in scrittura fra i nodi del sistema. Questa fase è comune a entrambi i metodi di deduplicazione per questo verrà considerata per entrambi nel calcolo del tempo totale di esecuzione e dei dati scambiati.

**L'eliminazione dei duplicati per il metodo single-end avviene in 58 secondi**, di questi: 2 secondi sono impiegati per la preparazione dei dati alla deduplicazione e 56 secondi per il *distinct()*, cioè per la rimozione vera e propria dei duplicati. Il *distinct()* elimina i duplicati e per farlo utilizza 6 processori, questo significa che i dati vengono divisi in 6 parti che vengono ridistribuite tra 6 nodi. Per svolgere il suo compito il *distinct()* scambia tra i nodi 102.2 MiB di dati in lettura e 29.6 Mib di dati in scrittura, per un totale di 131.8 MiB di dati scambiati tra i nodi. **L'eliminazione dei duplicati con il metodo single-end comporta lo scambio tra i nodi di 182.9 MiB di dati per il totale delle operazioni**.

**L'eliminazione dei duplicati per il metodo paired-end richiede 2 minuti** di cui 2 secondi sono impiegati per la preparazione dei dati alla deduplicazione, 1 secondo per il *reduceByKey()* e 1 minuto e 17 secondi per l'eliminazione vera e propria dei duplicati (*distinct()*).
Come si può notare dalla tabella è assente il passo 4, questo perchè Spark utilizza dei risultati che sono già stati ottenuti per ridurre lo scambio di dati e il tempo dell'intera operazione
Per eseguire il *reduceByKey()*, nel metodo paired-end, vengono utilizzati 6 processori con uno scambio di dati pari a 102.2 MiB in lettura e 30.9 MiB in scrittura. Per la deduplicazione vengono utilizzati 12 processori con uno scambio di dati pari a 61.9 MiB in lettura e 59.4 MiB in scrittura. **Per eseguire la deduplicazione con il metodo paired-end vengono scambiati tra i nodi un totale di 305.5 MiB di dati** per il totale delle operazioni. In termini di tempo e di dati scambiati fra i nodi del sistema, **l'eliminazione dei duplicati tramite il metodo paired-end è il più costoso** (2 minuti e 305.5 MiB).

# References

[1] Hang Dai and Yongtao Guan, *Dai2020, Nubeam-dedup: a fast and RAM-efficient tool to de-duplicate sequencing reads without mapping*, Bioinformatics,Oxford University Press,https://academic.oup.com/bioinformatics/article/36/10/3254/5753947

[2] Rieseberg Lab, *FastQ Examples*, GitHub Repository, `https://github.com/rieseberglab/fastq-examples`

[3] Apache Software Foundation, *PySpark API Reference*, Apache Spark Documentation, `https://spark.apache.org/docs/latest/api/python/reference/pyspark.html`