

Algoritmi per la trasformata di Burrows-Wheeler Posizionale con compressione run-length

Davide Cozzi, 829827, d.cozzi@campus.unimib.it

Introduzione

Negli ultimi anni si è assistiti ad un cambio di paradigma nel campo della *bioinformatica*, ovvero il passaggio dallo studio della *sequenza lineare di un singolo genoma* a quello in cui si studia un grande insieme di genomi, provenienti da diversi individui, al fine di poter considerare anche le *variazioni* o *varianti geniche*. Questo nuovo concetto è stato nominato per la prima volta da Tettelin, nel 2005, con il termine **pangenoma**.

Grazie ai risultati ottenuti in *pangenomica* si sono permessi miglioramenti negli studi sia nel campo della *biologia* che, ad esempio, in quello della *medicina personalizzata*, soprattutto grazie ai **genome-wide association studies (GWAS)**. In tali studi si necessita di studiare grandi collezioni di genomi al fine di poter identificare associazioni tra *varianti geniche* e la propensione ad una certa malattia o ad un suo particolare sintomo.

Dando qualche dato quantitativo possiamo pensare al fatto che il genoma umano, ad esempio prendendo come reference il *GRCh38.p14*, è composto da circa 3.1 miliardi di basi, comportanti circa 59,000 geni. Inoltre, i risultati del *1000 Genome Project* hanno portato ad identificare più di 88 milioni di varianti tra i genomi sequenziati. Tra esse si hanno, ad esempio, 84.7 milioni di **Single Nucleotide Polymorphisms (SNPs)**, ovvero differenze di singole basi azotate, 3.6 milioni di piccole **inserzioni/delezioni (indel)** di basi e circa 60,000 **varianti strutturali**, che coinvolgono più di 50 nucleotidi. Questi numeri fanno capire come, dal punto di vista della *computer science*, siano richiesti algoritmi e strutture dati efficienti in grado di gestire quest'incredibile mole di dati, considerando che è destinata ad aumentare nel prossimo futuro, grazie al miglioramento delle tecnologie di sequenziamento (**Next Generation Sequencing (NGS)** e **Third-Generation Sequencing**).

In tale ottica, sempre da un punto di vista computazionale, il *pangenoma* può essere rappresentato in molteplici modi, tramite strutture dati che permettano di memorizzare tutte le variazioni tra i genomi dei vari individui. Solitamente si hanno due rappresentazioni principali: una tramite un *grafo del pangenoma* e una tramite un *pannello di aplotipi*. In entrambi i casi sono necessarie metodologie efficienti sia per la memorizzazione della struttura dati che per le interrogazioni alla stessa. In questa tesi si è studiata la seconda casistica, ovvero la rappresentazione tramite *pannello di aplotipi*, dove con **aplotipo** si intende l'insieme di alleli, ovvero di varianti, che un organismo eredita da ogni genitore. L'informazione combinata di tutti gli aplotipi in un individuo è detta invece **genotipo**.

In questo contesto trova spazio, ovviamente, uno dei problemi fondamentali della *bioinformatica*, ovvero quello del **pattern matching**. Inizialmente tale concetto era relativo allo studio di un breve pattern all'interno di un testo di grandi dimensioni. Ora, con l'introduzione del *pangenoma*, tale problema si è adattato alle nuove strutture dati. Si ha, ad esempio, il caso studiato in questa tesi: il pattern matching tra un *aplotipo esterno* e un *pannello di aplotipi*.

Lo scopo di questa tesi è stato quello di ottimizzare il problema del pattern matching in una delle strutture dati più utilizzate parlando di pannelli di aplotipi: la **Positional Burrows-Wheeler transform (PBWT)**. Il progetto di tesi, svolto in collaborazione con il laboratorio BIAS, il professor Gagie (Dalhousie University), la professoressa Boucher e il dottor Rossi (University of Florida), ha quindi permesso lo sviluppo di una variante **run-length encoded** della **PBWT** che permettesse di risolvere il problema della ricerca di match massimali tra un pannello di aplotipi e un aplotipo query.

Preliminari

Al fine di comprendere al meglio i metodi usati in questa tesi bisogna introdurre alcuni concetti preliminari.

Il primo è quello della ben nota **Burrows-Wheeler transform (BWT)**, una trasformata reversibile

che ha permesso di ottenere algoritmi efficienti per il pattern matching. Questo è stato permesso grazie all'indicizzazione tramite **FM-index**, un *self-index* che permette di lavorare sulla *BWT* senza averla effettivamente in memoria (avendo in memoria solo l'indice stesso). La *BWT* è fortemente legata al concetto di **Suffix Array (SA)**. Infatti, dato un testo T lungo n si ha che SA è la lista lessicograficamente ordinata delle posizioni di partenza di tutti i suoi suffissi, avendo che $SA[i] = j$ sse $T[j, |T| - 1]$ è l' i -esimo suffisso lessicograficamente minore di T , avendo che $T[SA[i], n - 1] \prec T[SA[j], n - 1]$. A questo punto si può dire che la *BWT* del testo T è tale per cui $BWT[i] = T[SA[i] - 1]$, se $SA[i] \neq 1$, o \$, altrimenti. Un altro concetto spesso usato sono le lunghezze del cosiddetto **Longest Common Prefix (LCP)** tra ogni suffisso consecutivo in SA .

Una caratteristica di questa trasformata è la tendenza a produrre una sequenza che presenta caratteri uguali in posizioni consecutive. Questo fenomeno è dovuto alle ripetizioni di sotto-stringhe nel testo, avendo che tali eventi, per motivazioni biologiche, sono frequenti nell'ambito genomico, dove si ricordi il testo è costruito su un alfabeto $\Sigma = \{a, c, g, t\}$. Al fine di sfruttare tali caratteri uguali consecutivi si è quindi ideata la **Run-length encoded Burrows-Wheeler transform (RLBWT)** dove una sequenza massimale di caratteri uguali, detta *run*, viene memorizzata in modo efficiente come coppia (carattere, lunghezza della run). Ad esempio la stringa `aaaaaa` sarebbe memorizzata come `(a,6)`. Alcuni paper recenti hanno quindi proposto un nuovo tipo di indicizzazione, tramite il cosiddetto **r-index**, per questa struttura dati compressa, al fine di ottenere un indice che non fosse lineare sulla lunghezza del testo ma sul numero di run della *BWT* ottenuto da esso. Tale indice include la *RLBWT* e un *Suffix Array sample*, ovvero i valori di SA all'inizio e alla fine di ogni run. Tali articoli hanno portato alla produzione di due tool, *MONI* e *PHONI*, alle cui tecniche è fortemente ispirata questa tesi. Senza entrare nei dettagli, queste due soluzioni avevano entrambe l'obiettivo di calcolare i **Maximal Exact Matches (MEMs)** tra un testo T , lungo n è un pattern P , lungo m . Si ha infatti che una sotto-stringa, di lunghezza l e iniziante all'indice i , del pattern, ovvero $P[i, i + l - 1]$ è un *MEM* di P in T sse $P[i, i + l - 1]$ è anche una sotto-stringa di T ed essa non può essere stesa in nessuna direzione, avendo che né $P[i - 1, i + l - 1]$ né $P[i, i + l]$ sono sotto-stringhe di T . l'algoritmo per il calcolo dei *MEM* prevede prima la costruzione dell'array delle **Matching Statistics (MS)**, ovvero un array lungo m , di coppie (posizione *pos*, lunghezza *len*), tale che $T[MS[i].pos, MS[i].pos + MS[i].len - 1] = P[i, i + MS[i].len - 1]$ e che $P[i, i + MS[i].len]$ non occorre in T . Quindi si ha un match tra P e T lungo $MS[i].len$ a partire da $MS[i].pos$ in T e da i in P che non è ulteriormente estendibile. Gli autori, per il calcolo delle *Matching Statistics* hanno usato, in *MONI*, anche il concetto di *threshold*, definito come il minimo valore dell'array *LCP*. Un ulteriore miglioramento si ha avuto in *PHONI* con l'uso delle **LCE query** per il calcolo dell'array *MS*. Si ha che, dato un testo T , tale che $|T| = n$, il risultato della **LCE query** tra due posizioni i e j , tali che $0 \leq i, j < n$, corrisponde al più lungo prefisso comune tra le sotto-stringhe che hanno come indice di partenza i e j , avendo quindi il più lungo prefisso comune tra $T[i, n - 1]$ e $T[j, n - 1]$.

I concetti appena espressi verranno riformulati in questa tesi, in quanto essa tratta la costruzione di una versione **run-length encoded** della **Positional Burrows-Wheeler transform (PBWT)**, ovvero la variante posizionale della *BWT*, che viene costruita a partire da un pannello di aplotipi (nello specifico limitandosi al caso bi-allelico avendo che il pannello è composto da simboli nell'alfabeto $\Sigma = \{0, 1\}$). Tale struttura è frutto degli studi fatti da Durbin, nel 2014. Tale struttura dati assume in input un pannello X , composto da M individui/righe e N siti/colonne, e produce, tramite l'*ordinamento dei prefissi inverso*, ad ogni colonna k , due insiemi di array. Tali insiemi sono detti **insieme dei prefix array** e **insieme dei suffix array**. Il primo, denotato a , contiene, per ogni colonna k e ogni posizione i , l'indice dell'aplotipo m nel pannello originale. Si ha quindi che $a_k[i] = m$ sse X_m (denotando la riga m -esima di X) è l' i -esimo aplotipo secondo l'ordinamento inverso fatto in colonna k . Il secondo insieme, denotato con d , indica l'indice della colonna iniziale del suffisso comune più lungo, che termina nella colonna k , tra una riga e la sua precedente all'ordine inverso-prefisso ottenuto per la k -esima colonna. Il pannello ottenuto con le permutazioni dettate dal *prefix array* viene chiamato *matrice PBWT*.

Si può quindi apprezzare il collegamento naturale che si ha tra la *PBWT* e la *BWT*, avendo che il *prefix array* della prima corrisponde al *suffix array* della seconda mentre il *divergence array* altro non è che una diversa rappresentazione dell'array *LCP*.

Cruciale è il fatto che tale struttura permetta di calcolare i match massimali tra un aplotipi esterno e il pannello in tempo $\mathcal{O}(MN)$, mentre una soluzione naive impiegherebbe un tempo proporzionale a $\mathcal{O}(M^2N)$. Il tradeoff di tale algoritmo, conosciuto anche come **algoritmo 5 di Durbin**, è la richiesta in termini di spazio. L'autore stesso infatti stima la richiesta di $13NM$ byte in memoria per poter eseguire l'algoritmo.

Questo ultimo aspetto è stato il punto di partenza di questi progetto di tesi, il quale ha come obbiettivo il creare una variante *run-length encoded* della *PBWT*, detta *RLPBWT*, che permetta, in modo efficiente dal punto di vista della memoria, di poter calcolare *match massimali* tra una sequenza query e il pannello.

Al fine di raggiungere tale obbiettivo si sono usate altre nozioni. In primis, si sono sfruttate le cosiddette **strutture dati succinte**, ovvero strutture per le quali, assumendo \mathcal{X} sia il numero di bit ottimale per memorizzare dei dati, si richiede uno spazio $\mathcal{X} + o(\mathcal{X})$. Nel dettaglio si sono usati i cosiddetti **sparse bitvector**, strutture che richiedono in memoria $\approx m(2 + \log \frac{n}{m})$ bit, con n lunghezza del bitvector e m numero di simboli $\sigma = 1$ in esso. L'efficienza delle operazioni che si possono fare con tali strutture dati, incredibilmente efficienti dal punto di vista dello spazio occupato, sono uno dei punti cruciali del funzionamento della *RLPBWT*.

Infine, al fine di memorizzare il pannello in modo efficiente, si è usata una struttura dati, detta **Straight-Line Programs (SLPs)**. Tale struttura altro non è che una **grammatica context-free**, che genera una e una sola parola, la quale permette sia di effettuare *random access* al testo, non in tempo costante, che di calcolare le *LCE query*.

Metodi

Il processo per ottenere la *RLPBWT* è stato incrementale, iniziando con la creazione di una variante naive, basata sulle intuizioni avute a fine 2021 da Gaggie, che propose una prima variante della struttura senza però specificare come effettuare query alla stessa. Tale proposta, inoltre, presentava alcune ridondanze tra i dati memorizzati.

Il primo approccio aveva come obbiettivo l'ottenere un riadattamento "diretto" dell'*algoritmo 5 di Durbin*, pur tenendo in memoria informazioni legate alle sole run della *matrice PBWT*. Per quanto si siano trovate soluzioni interessanti per gestire il *mapping* tra una colonna e la sua successiva (ma anche tra una colonna e la sua precedente), al fine di poter "seguire" una riga del pannello originale nella *matrice PBWT*, è risultato necessario memorizzare l'intero *divergence array* (in forma di *LCP array* quindi memorizzando la lunghezza del prefisso comune in ordine inverso e non la colonna d'inizio dello stesso) in memoria al fine di poter computare i match con un aplotipo esterno. Inoltre, l'assenza delle informazioni relative al *prefix array* ha impedito di poter annotare quali righe del pannello presentassero un match massimale con l'aplotipo query ma avendo solo informazioni relative alla loro cardinalità. Le informazioni in memoria si è stimato non fossero ottimali, oltre che per il *divergence array*, in quanto non veniva utilizzato alcun approccio tramite *strutture dati succinte* per ottimizzare la memoria necessaria. Si è quindi deciso di cambiare approccio al problema, ispirandosi ai risultati già ottenuti per la *RLBWT*. In primis le strutture necessarie al mapping e all'indicizzazione delle run sono state sostituite da *bitvector sparsi*, creando una prima variante della *RLPBWT* basata su *bitvector*. Ovviamente tale sostituzione non risolveva i limiti dati dall'aver in memoria il *divergence array* e l'algoritmo di ricerca dei match era ancora basato su quello di Durbin.

Al fine di avvicinarsi alle idee proposte in *MONI* e *PHONI* è quindi servito uno studio teorico preliminare al fine di ridefinire concetti come quello di *matching statistics* (compreso quello di *MEMs* calcolabili da esse), di *threshold* e di *LCE query*. Da un punto di vista formale, parlando di *matching statistics*, si ha che essere sono definite, assumendo un pannello X , con M aplotipi, N siti e riga arbitraria i indicabile con la dicitura x_i , come un array, lungo N , di coppie (riga *row*, lunghezza *len*) tale per cui $x_{MS[i].row}[i - MS[i].len + 1, i] = z[i - MS[i].len + 1, i]$ mentre $z[i - MS[i].len, i]$ non è un suffisso terminante in colonna i di un qualsiasi sottoinsieme di righe di X . Data questa definizione, inoltre, si ha che $z[i - l + 1 : i]$ è un *MEM* di lunghezza l in con la riga $MS[i].row$ del pannello X sse $MS[i].len = l \wedge (i = N \vee MS[i].len \geq MS[i + 1].len)$.

Al fine di poter calcolare tale array si è quindi pensato all'utilizzo del concetto di *threshold*, anch'esso riadattato alla *matrice PBWT* come minimo valore dell'array *LCP* all'interno di una run (compre-

dendo anche la testa, qualora esistente, della run successiva, essendo il suo valore *LCP* calcolato sfruttando anche la coda della run corrente). Anche per tale informazioni si è usato un *bitvector sparso*.

Analogamente a quanto visto per l'*r-index*, inoltre, si è provveduto a tenere in memoria i *sample di prefix array* ad inizio e fine di ogni run.

Grazie all'uso delle *threshold* si è potuto sviluppare un algoritmo efficiente dal punto di vista della memoria richiesta per il calcolo delle *matching statistics* in due “sweep”, calcolandone prima le posizioni e poi, tramite *random access* al pannello (tenuto in memoria anch'esso come bitvector), calcolando anche le lunghezze. Tale soluzione, per quanto interessante in merito alle tecniche algoritmiche usate, richiedeva in memoria l'intero pannello, anche se in modo non naive essendo in forma di vettori di bitvector. Esplorando le tecniche presenti negli ultimi sviluppi avuti con la *RLBWT* si è giunti all'uso, come già anticipato, dell'*SLP*. Grazie a ciò, riprendendo quanto fatto in *PHONI* per la *RLBWT*, si è anche deciso di risparmiare ulteriore spazio eliminando l'uso delle *threshold* (che richiedevano un *bitvector sparso*, lungo quanto una colonna della *matrice PBWT* e con un numero di 1 pari a quello delle run). Per ottenere il calcolo dell'array della *matching statistics* si sono quindi usate le *LCE query*, ridefinite, per il caso posizionale. Dato un pannello X , $M \times N$, e due righe x_i e x_j tali che $0 \leq i < m$ e $0 \leq j < M$, con $i \neq j$, si definisce **LCE query** il suffisso comune più lungo tra le due stringhe. Calcolando la lunghezza di tale suffisso comune è possibile, per di più, calcolare le lunghezze delle *matching statistics* contemporaneamente al calcolo delle righe della stessa. Quindi, si è ottenuto il calcolo completo di tale array in un singolo “sweep”.

Al fine di risolvere il problema del calcolo dei match massimali, è servito costruire una struttura dati a supporto che permettesse l'estensione, nel pannello, a tutte le righe che presentassero il *MEM* calcolato con le *matching statistics*. Infatti, con il calcolo di tale array, si aveva nozione di una sola delle righe, $MS[i].row$, presentanti un match lungo $MS[i].len$ fino alla colonna i . Anche tale struttura, seguendo l'ormai evidente correlazione tra *BWT* e *PBWT* (nonché delle versioni *run-length*), si è basata su concetti precedentemente teorizzati, ovvero le **funzioni** φ e φ^{-1} . Parlando di *suffix array* tali funzioni, dato un valore dell'array, ne restituivano i valori adiacenti. In modo analogo si è creata una struttura dati che, data una colonna k e un indice di riga, permettesse di ottenere i valori adiacenti alla riga data nel *prefix array* ottenuto in colonna k . Per le proprietà date dalla costruzione della *PBWT* tutte le righe che presentano un match terminante in colonna k hanno indici adiacenti nel *prefix array*. Quindi, ipotizzando un match massimale terminante in i , partendo da $MS[i].row$, si possono computare tutte le altre righe del pannello che presentano il medesimo match con l'aplotipo query.

Ricapitolando, per computare le *matching statistics* si hanno in memoria, per ogni colonna: un *bitvector sparso* per identificare le run, due per permettere il mapping tramite il numero di 0 e uno fino ad un certo indice, un booleano per capire se la prima run è composta da simboli $\sigma = 0$ (avendo che poi le run hanno simboli ovviamente alternati), il valore complessivo di zeri nella colonna della *matrice PBWT* e, eventualmente, un *bitvector sparso* per le *threshold* (non in memoria se si volessero usare le *LCE query*). In aggiunta l'intera struttura ha in memoria il pannello sotto forma di *SLP* e la struttura computare le funzioni φ e φ^{-1} . A sua volta tale struttura conta due insiemi, di cardinalità pari al numero di righe, di *bitvector sparsi* con una quantità di simboli $\sigma = 1$ pari alle volte che la riga è testa/coda di una run. A questi insiemi si aggiungono altri due insiemi, sempre di cardinalità M , di vettori lunghi quanto il numero di volte che la rispettiva riga è testa/coda di una run. Si segnala che per la costruzione è necessario tenere in memoria anche il vettore a_{n-1} , che può essere eliminato una volta terminata la costruzione della struttura a supporto.

Per concludere questa sezione dedicata ai metodi si ricorda che la variante della **RLPBWT** basata su **SLP** e **LCE query** è stata considerata come il risultato finale ottenuto in questo progetto di tesi.

Risultati

L'obiettivo della tesi era quello di concentrarsi sui limiti dell'*algoritmo 5 di Durbin* dal punto di vista della memoria richiesta. L'utilizzo di strutture dati succinte, la scelta di memorizzare il pannello tramite *SLP* e la necessità di ridurre al minimo le informazioni in memoria hanno comportato inevitabilmente un aumento dei tempi di calcolo. Tale aumento è davvero molto importante, pas-

sando, prendendo ad esempio un pannello 70000×46538 con 30000 query, da $\sim 411s$, ottenuti con l'implementazione originale di Durbin, a ~ 1824 della *RLPBWT* con *SLP* e *LCE query*.

Analizzando però meglio i risultati relativi in termini di memoria richiesta si sono ottenuti dati confortanti. In primis, risulta interessante approfondire la memoria utilizzata dall'*SLP* per il pannello. Per un pannello 100000×358653 , che normalmente occupa $\sim 35gb$ (nel formato *.macs* usato per l'input), si produce un *SLP* che pesa appena $\sim 15mb$.

In secondo luogo si ricordi che, secondo le stime di Durbin, gli array necessari al funzionamento dell'algoritmo 5 richiedono $\sim 13NM$ bytes. Si prenda quindi, ad esempio, il pannello 70000×46538 . Secondo Durbin sarebbero necessari $\sim 40gb$ di memoria. Sperimentalmente si sono registrati picchi di memoria prossimi a quel valore mentre, per la *RLPBWT* con *SLP* e *LCE query*, il picco è stato di appena $\sim 3gb$. In pratica la soluzione *run-length* ha richiesto il 7% di memoria della soluzione di Durbin.

Bisogna però annotare un fatto. Esplorando il codice di Durbin si è scoperta l'esistenza di un algoritmo non approfondito nel paper. In questo caso il calcolo dei match con un pannello di query prevede la fusione di quest'ultimo con il pannello principale, per poi procedere con il calcolo dei match interni al pannello stesso. Tale soluzione può quindi giovare del fatto che non richieda tutte le strutture necessarie al mapping tra le colonne e al sistema di indicizzazione sul *prefix array* atto a tenere traccia dei match. Inoltre necessita di tenere in memoria sono le informazioni (da leggersi: *prefix array* e *divergence array*), calcolate dinamicamente, della memoria corrente. Tale algoritmo è risultato essere superiore sia in termini di tempi (confrontandosi coi risultati appena descritti si parla di 20s di esecuzione) che di memoria (essendo stimata una memoria proporzionale a $O(N + M)$, confermata dal fatto che il picco di memoria in esecuzione sia stato di appena 10,084kb). Del resto tale algoritmo produce i risultati in ordine sparso e, constatando come in letteratura siano presenti solo estensioni e varianti dell'*algoritmo 5*, probabilmente impedisce un facile riadattamento alla risoluzione di altre problematiche che non siano il calcolo dei match massimali con una o più query. La mancanza di una trattazione teorica formale ha comunque impedito ulteriori approfondimenti in merito.

Conclusioni

In conclusione si rileva come i risultati prefissati siano stati raggiunti, producendo una struttura dati efficiente in memoria per la risoluzione del calcolo dei match tra un pannello di aplotipi e un aplotipo esterno. Si segnala inoltre come siano possibili diversi sviluppi futuri. In primis, ovviamente, un'ulteriore ottimizzazione della struttura, sia in termini di gestione del mapping che, eventualmente, di gestione di più query contemporaneamente.

Inoltre sono possibili diverse generalizzazioni rispetto alle caratteristiche del pannello. Per quanto i pannelli di aplotipi prodotti dal sequencing del genoma umano raramente presentino siti multi-allelici si ha una presenza stimata, al momento, di circa il 2% di siti tri-allelici, avendo che tale percentuale risulti fortemente sottostimata. Una prima generalizzazione quindi sarebbe quella di studiare pannelli multi-allelici, riformulando la *RLPBWT* al fine di poter funzionare anche pannelli costruiti su un alfabeto arbitrario (dovendo rinunciare a diverse ottimizzazioni che permette il caso binario). Un'altra generalizzazione interessante riguarda l'ammissione di dati mancanti all'interno del pannello stesso. La maggior parte delle soluzioni attualmente sviluppate sono basate su una forte assunzione: non si hanno dati mancanti. Una variante della *RLPBWT* che sia quindi in grado di lavorare, eventualmente con *algoritmi parametrici* o *algoritmi approssimati*, su pannelli in cui sono presenti wildcard per rappresentare tali dati permetterebbe di fare studi più completi su dati reali, estendendone gli usi nei campi della *medicina personalizzata*, dei *GWAS* etc. ...

Si segnala, infine, sempre in linea con la più recente letteratura sulla *RLBWT*, come la *RLPBWT* sia potenzialmente in grado di risolvere efficientemente anche altri task, come ad esempio la ricerca di **k-MEM**, ovvero match massimali con un aplotipo esterno che coinvolgano esattamente k righe nel pannello.

Le potenzialità di tale struttura sono quindi molteplici e, grazie al ridotto consumo di memoria, si hanno le giuste premesse perché venga utilizzata per gestire e interrogare grosse moli di dati reali, incrementando la capacità di studio, previsione e inferenza che si possono avere grazie al *pangenoma*, sotto forma di pannello di aplotipi.