SecureIt: home security has never been so easy
Anno Accademico 2011/2012

Luca Bonato, Marco Ziccardi
Corso di studi in Informatica
Facoltà di Scienze MM.FF.NN.
Università degli Studi di Padova
Email: {lohathe,marco.ziccard}@gmail.com

***Abstract*—Nowadays different applications offer a mechanism to monitor your home environment in order to detect any intrusion. None of that solutions completely exploits phone functionalities to find intruders: accelerometer, camera or microphone are often mutually exclusive. Moreover none of these applications tracks phone's position after an intrusion has been identified. The proposed application aims to collect different solution for motion/presence detection and to track phone's position using any kind of connectivity.**

I. Introduction

The aim of the application development was not only the creation of a home security system but the discovery of phone capabilities too. In fact operations performed by the application are complex and really resource needful:

- Accelerometer to detect orientation and shake
- Camera to detect motion processing frames
- Microphone to detect environmental sound level
- Wifi or 3G connectivity to upload images and audio
- GPS to get current location
- Bluetooth to upload location in absence of connectivity
- In app encryption in order to securely authenticate the user

To the mobile application is associated a back-end where the application itself uploads images and audio got at the instant of intrusion detection.

II. Intrusion detection

Intrusion detection is done in three complementary ways:

- **Accelerometer**
- **Camera**
- **Microphone**

A. Accelerometer

The accelerometer is used to detect motion of the phone itself. Different sensitivity can be set in order to detect different kinds of motions.
The accelerometer returns a *triple* of three acceleration values on the three axes (*X*, *Y* and *Z*).

Those values allow to detect every orientation taken by the phone except for phone rotations on the *Z* axis when the phone is vertical (this is obvious according to the fact that in such a position the phone is not subject to acceleration variations on that axis).

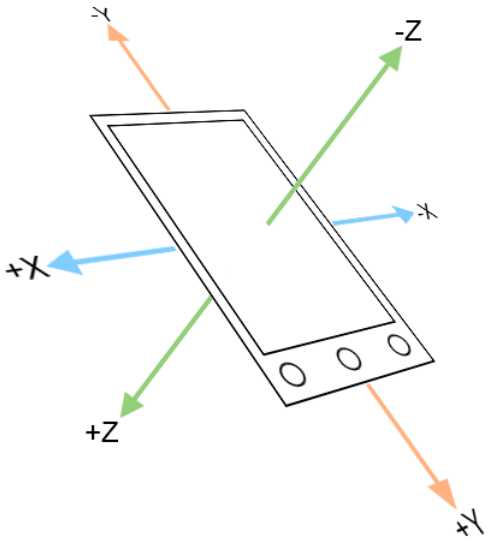

Fig. 1. Values and signs of detected acceleration

Those acceleration values are used to move a 3D animation made in OpenGL ES (running on the GPU) that simply uses them to show phone's orientation, see figure 2.

Acceleration triple is also used to detect any variation in the phone acceleration. Accelaration variation gives an idea of intensity of phone's *shake* and it is computed as in equation 1.

$$variation = \frac{(\Delta accelX) + (\Delta accelY) + (\Delta accelZ)}{\Delta t} \quad (1)$$

Three levels of sensitivity are defined:

- `LOW: SHAKE_THRESHOLD = 5000`
- `MEDUM: SHAKE_THRESHOLD = 4000`
- `HIGH: SHAKE_THRESHOLD = 3000`

If $variation > SHAKE\_THRESHOLD$ the application starts to send alerts according to the specified options.

Fig. 2. OpenGL ES 3D animation to show phone orientation

B. *Camera*

The camera is used to detect motion in the surrounding environment. The frames captured by the camera are processed to detect if any variation ha occurred since last captured frame. A callback is activated at every captured frame, processing each captured frame would be too expensive for the phone in terms of CPU usage and battery consumption so the processing is reduced at one frame per second.
As said motion detection on the frame matrix is very expensive so the frames are scaled as soon as capture to a size of `640x480`.

Rispetto ad un algoritmo globale un algoritmo locale nel caso in cui Pv, Pd e Po siano tutti negativi può iniziare un nuovo allineamento.
Il punteggio di un allineamento viene calcolato in maniera diversa nel caso in cui si stia lavorando su sequenze di aminoacidi o di nucleotidi:

- **Aminoacidi**: il punteggio di allineamento è dato dalla somma di tutti i punteggi di appaiamento estratti da una *scoring matrix* (sia essa PAM o BLOSUM) meno le penalità per i *gap*. Le penalità per i *gap* sono date da due componenti: la penalità di apertura di un *gap* e la penalità conseguente all'allungamento di un residio (aminoacido) del *gap*
- **Nucleotidi**: il punteggio è calcolato come sopra tranne per il fatto che la matrice di sostituzione è semplificata: +1 per un *match*, -1 per un *mismatch*

Le penalità di *gap* sono variabili e dipendono dal contesto di utilizzo.

C. *FASTA*

Gli algoritmi di programmazione dinamica hanno risolto il problema dell'allineamento di due sequenze ma sono inutilizzabili (troppo lenti) per allineare una sequenza *query* su tutte le sequenze *subject* di una banca dati.

FASTA (Lipman e Pearson, 1985) è la prima soluzione euristica al problema dell'allineamento utilizzabile su una banca dati.
FASTA si basa su una strategia di indicizzazione di parole di `ktup` residui (nucleotidi o aminoacidi) dette *k-mers*. Tipicamente `ktup` = 6 per gli allineamenti di sequenze di nucleotidi e `ktup` = 2 per sequenze di aminoacidi.
FASTA scorre la *query* e memorizza in un indice per ciascuna parola le posizioni in cui essa compare (costo lineare nella lunghezza della *query*). Successivamente scorre ciascuna sequenza *subject k-mer* per *k-mer*, quando uno di questi compare nell'indice l'algoritmo memorizza la diagonale. Vengono poi calcolati i punteggi degli allineamenti di *k-mer* tramite una matrice di sostituzione (PAM o BLOSUM a tolleranza variabile dipendentemente dal contesto), questo calcolo permette di escludere quelle sequenze *subject* che non superano una certa soglia di similarità.
FASTA procede ad estendere ciascuno di questi allineamenti applicando opportune penalità per i *gap* e anche in questo caso scarta le estensioni che non rientrano in una certa soglia di punteggio.
In ultima istanza viene eseguito Smith e Waterman modificato per considerare solo porzioni ristrette adiacenti alle estensioni appena calcolate ($\pm$ 20 aminoacidi).

FASTA per mezzo dell'indicizzazione delle parole identifica i *match* possibili prima di effettuare una ricerca di allineamento più onerosa tramite Smith e Waterman.

D. *BLAST*

BLAST (Altshul et al., 1990) rappresenta ad oggi l'algoritmo più usato per l'allineamento generale di proteine e DNA.
Risulta essere particolarmente efficiente negli allineamenti proteici poichè, invece di creare un indice basato sul *matching* esatto delle parole (come FASTA), crea un indice basato sulla similarità calcolata per mezzo di una matrice di sostituzione (PAM o BLOSUM).
La lunghezza delle parole è definita da un parametro `W`, tipicamente nel caso di allineamento proteici `W`=3.
Dopo aver creato l'indice di similarità per la stringa *query* l'algoritmo ricerca i *W-mers* in tutte le sequenze *subject* della base dati. Ogni corrispondenza può potenzialmente essere parte di un allineamento più lungo, si procede quindi all'estensione di tali corrispondenze in entrambe le direzioni ottendendo segmenti di allineamento locale non ulteriormente estendibili detti HSP (*High-Scoring Segment Pair*). Un ulteriore parametro `X` determina quanto punteggio negativo può accumularsi nel tentativo di estendere una corrispondenza.

Tutti gli HSP che superano una certa soglia `S` sono ritenuti significativi.

Esistono diverse versioni di blast che realizzano compiti specifici:

- **blastp**: ricerca gli allineamenti di una sequenza di aminoacidi su una banca dati di sequenze di aminoacidi
- **blastn**: ricerca gli allineamenti di una sequenza di nucleotidi su una banca dati di sequenze di nucleotidi
- **blastx**: ricerca allineamenti in una banca dati di sequenze proteiche a partire da una sequenza *query* di nucleotidi dopo aver tradotto la *query* in una sequenza di aminoacidi secondo tutte le 6 possibili fasi di lettura (*6-frame translations*)
- **tblastn**: ricerca gli allineamenti di una sequenza proteica su una banca dati di sequenze di nucleotidi calcolando per ciascuna sequenza *subject* della banca dati tutte le possibili sequenze di aminoacidi (*6-frame translations*)
- **tblastx**: ricerca gli allineamenti di tutte le possibili traduzioni in sequenza di aminoacidi di una sequenza di nucleotidi con tutte le traduzioni in aminoacidi delle sequenze di una banca dati di nucleotidi. Rappresenta la versione più lenta dell'algoritmo BLAST, confronta sequenze aminoacidiche al posto di nucleotidi per offrire maggior tolleranza (effettua 36 confronti per ogni coppia di sequenze utilizzando matrici di sostituzione) e identificare relazioni di omologia tra sequenze nucleotidiche distanti

E. Burrows-Wheeler Transformation (BWT)

Permutations	Sorted permutations	Last column
googol$	$googol	**l**
oogol$g	gol$goo	**o**
ogol$go	googol$	**$**
gol$goo	l$googo	**o**
ol$goog	ogol$go	**o**
l$googo	ol$goog	**g**
$googol	oogol$g	**g**

TABLE I
BWT ESEGUITA SULLA STRINGA GOGOL ($ INDICA LA FINE DELLA STRINGA)

Con l'avvento dei sequenziatori di nuova generazione il numero di *reads* prodotte è aumentato notevolmente e consistentemente anche la quantità di genoma da allineare. Occorrevano metodi sempre più ottimizzati per la realizzazione dell'allineamento.
BWT (Burrows et al., 1994) è un algoritmo utilizzato nella compressione dei dati, sfruttato nel campo della bioinformatica per la creazione di indici di piccole dimensioni e di rapida consultazione.

BWT non cambia i valori dei caratteri che compaiono in una stringa ma semplicemente attua delle permutazioni tali che se la stringa in input conteneva caratteri ripetuti nella stringa trasformata questi appariranno di fila, rendendo così più facile la compressione.
La trasformazione viene banalmente fatta ordinando lessicograficamente tutte le permutazioni della stringa in input e prendendo l'ultima colonna.

III. ALLINEAMENTO DI READS CORTE

A. MAQ

MAQ (Li et al., 2008), *Mapping and Assembly with Qualities*, è un programma di allineamento che opera su *reads* di piccole dimensioni prodotte da sequenziatori di nuova generazione, in particolare è progettato per lavorare con Illumina ma è in grado di lavorare anche con *reads* prodotte da SOLiD (e quindi in *color-space*).

MAQ differiva dalla maggior parte dei prodotti esistenti. In primo luogo calcola una misura di qualità per ciascun allineamento, misurante la probabilità dell'allineamento di essere sbagliato (questo approccio aiuta notevolmente l'identificazione delle variazioni). In secondo luogo mappa ogni *read* che abbia un *match*, distribuendo casualmente le *reads* ripetute in una delle alternative a parità di bontà, questo accorgimento solleva dall'ambiguità di definire il concetto di unica occorrenza di una *read*. Inoltre MAQ allinea *mate-pair reads* in una *sliding window*, andando ad esaminare le posizioni accoppiate con poco costo computazionale e permettendo di allineare correttamente anche *reads* ripetute se le loro *mate reads* si trovano in regioni uniche.

Alcuni punti chiave dell'algoritmo MAQ sono:

- **Indicizzazione delle reads e scorrimento del genoma di riferimento**: le *reads* sono indicizzate per mezzo di *seeds* di 28 basi, si scorre il genoma di riferimento alla ricerca di *seeds hit* con al più due *mismatch*
- **Assegnazione di punteggi agli hit e mappatura casuale**: Ad un *hit* viene assegnato un punteggio di $2^{24} * q + h$, dove `q` è la somma delle qualità delle basi che non si allineano correttamente e `h` è un intero in 24 bit derivante dall'*hash* delle coordinate dell'*hit* e dell'identificatore della *read*. Come risultato MAQ mappa casualmente una *read* se ci sono *hit* ugualmente buone a causa di ripetizioni nel genoma
- **Qualità di allineamento**: La qualità di allineamento è la probabilità che l'allineamento di una *read* sia sbagliato. Dato un genoma di riferimento `x` lungo `L` e una *read* `z` lunga `l` la probabilità che `z` sia mappata in `u` è:
$$p_s(u|x,z) = \frac{p(z|x,u)}{\sum_{v=1}^{L-l+1} p(z|x,v)}$$
Dove `p(z|x,u)` equivale al prodotto delle probabilità di errori delle basi che non si allineano. La qualità di un allineamento è:

$$Q_s(u|x,z) = -10log_{10}\left[1 - p_s(u|x,z)\right]$$

- **Indels corti**: per le coppie di *reads* tali che solo una viene allineata è utilizzato Smith e Waterman con *gaps* per la *read* associata in una regione ristretta determinata a seconda della dimensione dell'inserto, come in figura **??**
- **Identificazione consensus**: forniti dei dati `D` di allineamento su una posizione si calcola $P(D| < b1, b1 >)$, $P(D| < b2, b2 >)$ e $P(D| < b1, b2 >)$, assumento che il campione su cui si sta lavorando sia diploide. La conoscenza della probabilità a priori di un eterozigote $< b1, b2 >$ permette di calcolare la probabilità di ciascun genotipo. Il consensus $\hat{g}$ è il genotipo che massimizza la probabilità a posteriori e la sua qualità è $-10log_{10}\left[1 - P(\hat{g}|D)\right]$

MAQ utilizza per l'allineamento 6 ore di CPU e 800MB di memoria per 1 milione di coppie di *reads* ed è facilmente parallelizzabile su *clusters*. Genera file binari compressi sia per gli allineamenti (1 byte per ciascun nucleotide delle *reads*) sia per il consensus (4 byte per ciascun nucleotide sul genoma di riferimento). Fornisce inoltre un visualizzatore dei risultati dell'allineamento detto *maqview* che permette di identificare graficamente la posizione di una qualsiasi *read* sul genoma.

Le prestazioni di MAQ sono riassunte in figura **??** e sono state estratte dai dati di allineamento di 100 millioni di coppie di *reads* da 35 basi (estratte da un campione diploide con 0.1% di sostituzioni e 0.01% di *indels* di una base) sul genoma umano.

B. Bowtie

Bowtie (Langmead et al., 2009) usa un indice costruito tramite *Burrows-Wheeler Transformation* e promette di avere una piccola occupazione di memoria (circa 1.3GB per l'intero genoma umano).

Bowtie effettua alcuni compromessi per fornire la sua velocità e il suo scarso utilizzo di memoria, esso infatti non garantisce la più alta qualità di *mapping* delle *reads* se non esiste alcun *match* esatto. Inoltre può fallire nell'allineare alcune *reads* quando configurato per eseguire alla massima velocità. Possiede diverse opzioni per bilanciare questo *trade-off*.

C. BWA

BWA (Li e Durbin, 2009), *Burrows-Wheeler Aligner*, può essere visto come una evoluzione di MAQ. Mentre MAQ utilizza un indice basato su *hash* per cercare il genoma di riferimento BWA utilizza un indice costruito con BWT che permette ricerche più veloci del suo predecessore.

1) Prefix trie e string matching: Un *prefix trie* per una stringa `X` è un albero in cui ogni arco è etichettato con un simbolo e la concatenazione delle etichette nel cammino da una foglia alla radice rappresenta un prefisso di `X`. Dato un *prefix trie* cercare se una *query* `W` è una sottostringa esatta di `X` equivale a cercare il nodo che rappresenta `W` e ciò può essere fatto in tempo `O(|W|)`. Per consentire *mismatch* possiamo compiere un attraversamento esaustivo dell'albero e confrontare `W` con ogni possibile cammino.

2) BWT e suffix array: Supponiamo di aver calcolato la *Burrows-Wheeler Transformation* e di averla memorizzata in `B`. Un *suffix array* `S` di una stringa `X` lunga `n` è una permutazione degli indici `0 ... n-1` tale che `S[i]` è la posizione iniziale dell'i-esimo suffisso più piccolo.

La maggior parte degli algoritmi per la costruzione di un *suffix array* richiedono `O(nlogn)` in memoria, risultando in $12\tilde{G}B$ per l'intero genoma umano, recentemente Hon et al. (2007) hanno presentato una nuova tecnica che permette di costruire l'albero dei suffissi per l'intero genoma umano in spazio (nei picchi) <1GB. Tale tecnica è adottata da BWA.

3) Intervalli nel suffix array: Se una stringa `W` è una sottostringa di `X`, la posizione di ogni occorrenza di `W` in `X` si troverà in un intervallo nel *suffix array* dal momento che tutti i suffissi contenenti `W` sono ordinati insieme. Definiamo quindi:

$$\bar{R}(\mathrm{W}) = \max\{\mathrm{k} : \mathrm{W} \text{ è prefisso di } \mathrm{X}_{S(k)}\}$$
$$\underline{R}(\mathrm{W}) = \min\{\mathrm{k} : \mathrm{W} \text{ è prefisso di } \mathrm{X}_{S(k)}\}$$

4) Matching esatto: Sia `C(a)` il numero di simboli in `X[0,n-2]` che sono lessicograficamente minori di un `a` appartenente all'alfabeto e `O(a,i)` il numero di occorrenze di `a` in `B[0,i]`. Ferragina e Manzini (2000) provarono che, se `W` è una sottostringa di `X`, allora:

$$\bar{R}(\mathrm{aW}) = \mathrm{C(a)+O(a,}\underline{R}\mathrm{(W))}$$
$$\underline{R}(\mathrm{aW}) = \mathrm{C(a)+O(a,}\bar{R}\mathrm{(W)-1)+1}$$

Questo risultato permette di controllare se `W` è una sottostringa di `X` e contare le occorenze di `W` in tempo `O(|W|)` calcolando iterativamente $\bar{R}$ e $\underline{R}$ a partire dalla fine di `W`. Questa procedura è chiamata *backward search*.

5) Matching non esatto: L'algoritmo data una stringa `W` cerca gli intervalli nel *suffix array* di `X` delle sottostringhe che hanno un *match* con `W` con non più di `z` differenze. Essenzialmente la ricerca di un *match* non esatto usa *backward search* per campionare sottostringhe distinte dal genoma. Questo processo è limitato dall'*array* `D`, tale che `D[i]` è il limite inferiore al numero di differenze in `W[0,i]`. Meglio è stimato `D` più si restringe lo spazio di ricerca nel *suffix trie* e quindi più è efficiente l'algoritmo. Un *bound* semplice per `D` è `D[i]=0` per ogni `i` ma questo risulta in un algoritmo molto inefficiente.

Program	Single-end Time(s)	Conf(%)	Err(%)	Paired-end Time(s)	Conf(%)	Err(%)
Bowtie-32	1271	79.0	0.76	1391	85.7	0.57
BWA-32	823	80.6	0.30	1224	89.6	0.32
MAQ-32	19797	81.0	0.14	21589	87.2	0.07
SOAP2-32	256	78.6	1.16	1909	86.8	0.78
Bowtie-70	1726	86.3	0.20	1580	90.7	0.43
BWA-70	1599	90.7	0.12	1619	96.2	0.11
MAQ-70	17928	91.0	0.13	19046	94.6	0.05
SOAP2-70	317	90.3	0.39	708	94.5	0.34
Bowtie-125	1966	88.0	0.07	1701	91.0	0.37
BWA-125	3021	93.0	0.05	3059	97.6	0.04
MAQ-125	17506	92.7	0.08	19388	96.3	0.02
SOAP2-125	555	91.5	0.17	1187	90.8	0.14

TABLE II
PRESTAZIONI DI BWA PER L'ALLINEAMENTO DI UN MILIONE DI COPPIE DI *reads* DI 32, 70 E 125 BASI SIMULATE DA UN GENOMA UMANO CON FREQUENZA DI SNP (*Single Nucleotide Polimorphism*) PARI A 0.09%, UNA FREQUENZA DI *indels* PARI A 0.01% UNA FREQUENZA UNIFORME DI ERRORI DI SEQUENZIAMENTO DI 2%. SONO MOSTRATI I TEMPI DI ESECUZIONE (CALCOLATI SU UN *core* DEL PROCESSORE 2.5 GHZ XEON E5420I), LA STIMA PERCENTUALE DI *reads* ALLINEATE CORRETTAMENTE E LA PERCENTUALE DI ERRORE IN TALE STIMA

6) Valutazione delle prestazioni: Nel valutare le prestazioni di BWA sono state utilizzate quelle di altri tre programmi: MAQ, SOAPv2 e Bowtie. MAQ indicizza le *reads* tramite una *hash table* mentre SOAPv2 e Bowtie sono altri due algoritmi che utilizzano BWT. Sono stati condotti esperimenti sia con dati simulati sia con dati reali.

- **Dati simulati:** Sono state simulate *reads* dal genoma umano con lo strumento *wgsim* incluso nel pacchetto *SAMtools* e successivamente sono stati eseguiti i quattro programmi per allineare tali *reads* sul genoma umano. La tabella II mostra come BWA e MAQ raggiungano la stessa accuratezza nell'allineamento mentre BWA risulta essere più preciso sia di SOAPv2 che di Bowtie. Per quanto concerne la velocità SOAPv2 è decisamente il più veloce per l'allineamento *single-end* ma viene recuperato nell'allineamento *paired-end* nel quale BWA risulta essere migliore in due casi su tre (*reads* di 32 e 125 basi).
  Per quello che riguarda la memoria SOAPv2 usa 5.4GB. Sia Bowtie che BWA utilizzano 2.3GB per l'allineamento *single-end* mentre circa 3GB per il *paired-end*, più grande della memoria necessaria a MAQ che è di circa 1GB. Tuttavia l'utilizzo di memoria negli algoritmi basati su BWT non dipende dal numero di *reads* da allineare mentre l'occupazione di MAQ è lineare in esso. Inoltre gli allineatori centrati su BWT supportano il *multi-threading*.
- **Dati reali:** L'allineamento è stato fatto utilizzando 12.2 milioni di coppie di *reads* di 51 basi prodotte dal sequenziatore Illumina. Le prestazioni misurate sono mostrate in tabella III.

Program	Time(h)	Conf(%)	Paired(%)
Bowtie	5.2	84.4	96.3
BWA	4.0	88.9	98.8
MAQ	94.9	86.1	98.7
SOAP2	3.4	88.3	97.5

TABLE III
SONO MOSTRATI I TEMPI DI ESECUZIONE (CALCOLATI SU UN *core* DEL PROCESSORE 2.5 GHZ XEON E5420I), LA STIMA PERCENTUALE DI *reads* ALLINEATE CORRETTAMENTE E LA STIMA PERCENTUALE CON LE *mate-pairs* MAPPATE NEL CORRETTO ORIENTAMENTO DA BWA

D. mrFAST e mrsFAST

I programmi mr- e mrsFAST (Alkan et al., 2009; Hach et al., 2010) sono degni di nota dal momento che restituiscono tutti i possibili allineamenti di una *read* su un genoma, non soltanto l'unico allineamento migliore. L'abilità di riportare tutte le possibili locazioni del genoma è molto utile per l'identificazione delle variazioni strutturali, infatti questi due algoritmi sono stati sviluppati proprio a questo scopo.

mr- e mrsFAST utilizzano un metodo *seed-and-extend* per l'allineamento, creano indici di una *hash table* per il genoma di riferimento, ciascuna *read* viene suddivisa in *first middle* e *last k-mers* (di default `k`=12), ciascuno di questi *k-mer* viene cercato nel *hash* per fissare i *seeds* di allineamento.

E. BFAST

BFAST (Homer et al., 2009) è un algoritmo per allineare *reads* corte su un genoma di riferimento di grandi dimensioni. Realizza l'allineamento in due fasi:

- Crea indici multipli sul genoma di riferimento
- Identifica le CAL (*Candidate Alignment Locations*) per ciascuna *read*, successivamente queste CAL sono allineate localmente usando *gap* per identificare il miglior *match*. Questa tecnica si applica alle sequenze in *base-space* (Illumina e 454) e in *color-space* (SOLiD). L'allineamento usando *gap* permette di identificare SNP (*Single Nucleotide Polimorphism*), inserzioni e delezioni, oltre che errori nella sequenza in *color-space*

BFAST utilizza un indice multiplo prestando bene attenzione a sfruttarlo il più possibile quando esso viene caricato in RAM. Ciascun indice è un *suffix array* ottimizzato del genoma di riferimento che indicizza un *seed*, una stringa

di 0 e 1 che inizia e finisce con 1, che definisce le basi nella *read* che si stanno considerando in un *lookup*.
Chiamiamo k (*key size*) il numero di 1 in un *seed* mentre w (*key width*) il numero di 0 e 1. Il numero di CAL identificate dall'algoritmo dipende da k e dalla complessità e quantità di ripetizioni presenti nel genoma di riferimento. Con k maggiori in media si ha un *lookup* più probabilmente unico nell'indice ma si riduce anche il numero di *lookup* totali, rendendo l'algoritmo meno sensibile.
Solitamente per il genoma umano viene preferito k = 18 per *reads* corte ($\leq$ 40 bp) e k = 22 per *reads* più lunghe ($\geq$ 40 bp) per bilanciare sensibilità e unicità del *lookup*. Scegliere k di minori dimensioni è utile per le *reads* corte dal momento che così aumenta il numero di *offsets* e la sensibilità migliora. BFAST implementa un *hash* verso l'indice che riduce il tempo di *lookup* e consiste delle prime j basi (j è detta *hash width*) degli indici di riferimento, con $j \leq k$. *Hash width* è sempre più corta di *key size*.

La RAM necessaria per caricare un intero indice è approssimativamente 17GB, che porta in pratica alla necessità di 24GB di RAM per prestazioni ottime, quantità elevata ma raggiungibile dai moderni computer. Inoltre BFAST è in grado di scomporre l'indice in 4^n componenti: in questo modo, nonostante la fase di *lookup* richieda più tempo dal momento che essa andrà replicata su ogni indice parziale, BFAST per l'intero genoma riesce ad eseguire su computer con solo 4GB di RAM.

BFAST ammette allineamenti *paired-end* o *mate-pair*, realizzando allineamenti indipendenti per ciascun *end* e annotando il collegamento tra *pair-ends*.

Per mostrare le performance di BFAST esso è stato confrontato con altri software disponibili utilizzando *reads* diverse per dimensione e tipo. A questo scopo sono stati utilizzati sia dati simulati che dati reali.

Nelle figure **??**, **??**, **??** viene effettuata una valutazione degli algoritmi di allineamento su un insieme di 10,000 *reads* simulate in *base-space*.

In tabella IV i tempi di allineamento sono stati calcolati su due tipologie di *reads*, prodotte in un caso da Illumina in *base-space* e nel secondo caso da SOLiD in *color-space*, è mostrata inoltre la percentuale di *reads* allineate correttamente.

F. Novoalign

Novoalign è un prodotto proprietario di Novocraft (Novocraft, 2000) che utilizza una strategia di *hashing* simile a MAQ (Li et al., 2008).

Novoalign è diventato piuttosto popolare nelle recenti pubblicazioni a causa della sua accuratezza, esso inoltre ammette fino a 8 *mismatch* per *read* nell'allineamento di una sola *end*.

	Illumina 3.5M 55bp Time (s)	Illumina 3.5M 55bp % mapped	ABI SOLiD 1M 50bp Time (s)	ABI SOLiD 1M 50bp % mapped
BFAST	47,474	69.6	42,856	72.5
BOWTIE	857	55.7	not supported	not supported
BWA	4,883	59.3	845	47.8
MAQ	126,541	73.6	6,680	68.1
SHRiMP	324,380	83.3	32,644	70.4
SOAP	131,248	62.4	131,248	62.4

TABLE IV
RISULTATI NEI TEMPI DI ALLINEAMENTO E NEL NUMERO DI *reads* ALLINEATE CORRETTAMENTE DI VARI ALGORITMI SU DUE *datasets* REALI DIFFERENTI IN RELAZIONE A TEMPI E PERCENTUALI DI BFAST

G. SHRiMP

SHRiMP (Rumble et al., 2009) è un programma di allineamento specializzato nel mappare *reads* in *color-space* generate dal sequenziatore SOLiD. Dal momento che la conversione di sequenze da *base-space* a *color-space* è univoca è possibile utilizzarlo anche per quest'ultime.
SHRiMP utilizza tutti i recenti avanzamenti nell'allineamento di sequenze:

- *Q-gram filters* (Rasmussen et al., 2005): che consentono di cominciare l'allineamento con *seeds* che abbiano più di un *match*
- *Spaced seeds* (Califano and Rigoutsos, 1993): che ammettono sezioni di *mismatch* predeterminate nelle sequenze *seed*s
- Hardware specializzato: implementazioni ed istruzioni particolari che permettono di velocizzare l'algoritmo standard Smith e Waterman

SHRiMP è stato ulteriormente migliorato nella sua versione successiva SHRiMP2 (David et al. 2010). Quest'ultima versione si distingue dalla precedente andando essa ad indicizzare non le *reads* ma il genoma di riferimento. Tale modifica ha notevolmente migliorato le prestazioni permettendo l'aggiunta di una modalità per l'allineamento *paired-end* e ha aperto la strada al *multi-threading*.

L'indice del genoma di riferimento è caricato in RAM, per l'intero genoma umano arriva ad occupare 48GB. SHRiMP2 offre tuttavia degli strumenti per scomporre tale indice in componenti che rientrino in una certa quantità di RAM. L'*overhead* prodotto da questa suddivisione può essere trascurato.

Diversi *threads* sono usati per allineare le *reads*. Utilizzando dei *seeds* sono letti e cercati nell'indice tutti i possibili *k-mer* della *reads*. Tali posizioni dell'indice corrispondono a delle diagonali nella matrice di allineamento. Date queste diagonali

è possibile costruire un insieme di CML (*Candidate Mapping Locations*).

SHRiMP2 utilizza un meccanismo di *caching* per velocizzare l'allineamento di *reads* provenienti da sequenze ripetute: dopo ciascun all'ineamento viene calcolato un *hash* della regione e salvato il punteggio, in caso di successive richieste per quella regione il punteggio non sarà ricalcolato.

L'unico programma che sia effettivamente confrontabile con SHRiMP2 è BFAST in quanto gli intenti di sviluppo sono praticamente gli stessi: preferire la correttezza alla velocità. Come BFAST, SHRiMP2 garantisce alta precisione anche per *reads* molto polimorfe. Nei test SHRiMP2 raggiunge una precisione simile o addirittura maggiore a quella di BFAST per tutti i tipi di polimorfismo, eseguendo inoltre tra le 2 e le 5 volte più velocemente.

H. ABMapper

Gli allineatori di *reads* corte generate dai moderni sequenziatori forniscono un'idea dell'allineamento ma non sono in grado di gestire adeguatamente *reads* provenienti da sequenziamento di RNA che si trovino a cavallo di posizioni tra le quali è avvenuto *splicing* o che appartengano a sequenze ripetute e che possono quindi essere mappate su più posizioni nel genoma.

ABMapper (Chan et al., 2010) vuole essere uno strumento in grado di analizzare *reads* a cavallo di regioni che abbiano subito *splicing* o appartenenti a regioni ripetute.

La sequenza genomica di riferimento è inizialmente inidicizzata in *k-mers* secondo un `k` definito dall'utente. Dopo l'indicizzazione per ciascuna *read* che si vuole allineare sono estratti due *seeds* (chiamati `A` e `B`, dai quali deriva il nome del programma) di lunghezza `k` da ciascuna estremità della *read* e sono estesi tramite allineamento uno verso l'altro, come in figura **??**.

Due scenari possono manifestarsi durante l'estensione di un *seed*:

- *Exonic alignment*: la *read* non si trova a cavallo di due regioni che hanno subito *splicing*, viene esteso soltanto un *seed*
- *Spliced alignment*: entrambi i *seeds* sono estesi finchè non viene trovato un punto di terminazione. Un buon allineamento in questo caso prevede che i due *seeds* siano estesi indipendentemente fino a che l'intera *read* non è coperta. Se l'allungamento tollera errori però un *seed* può essere esteso oltre il suo reale punto di terminazione. In questi casi l'algoritmo tollererà e correggerà le sovrapposizioni

ABMapper supporta diversi formati in input: semplici file di testo, FASTA e FASTQ. Inoltre ABMapper è in grado di estrarre *reads* mappate su più di una posizione nel genoma o non mappate da file SAM/BAM prodotti da BWA per tentare un ulteriore allineamento.

La lunghezza `k` di un *seed* determina la velocità di ricerca, la sensibilità e la precisione, più essa è corta più l'allineamento diventa accurato ma aumentano i tempi di esecuzione. ABMapper permette all'utente di definire tale dimensione coerentemente con la dimensione del campione (numero di *reads* da allineare) e la potenza di calcolo a disposizione. La dimensione di default è 10.

ABMapper tollera dinamicamente l'errore ammettendo che alcune *reads* siano ulteriormente estese oltre il loro limite eventualmente accettando un ulteriore errore che supera quello di *cutoff*. Questa tecnica è molto utile nel caso di *paired-end reads* che sono affette dal problema noto secondo il quale la seconda *read* ha una qualità inferiore (errori maggiormente frequenti) rispetto alla prima.
Un'altra funzionalità importante è quella di ammettere sovrapposizioni tra frammenti di una *read* che abbia subito *splicing*, la regione sovrapposta può essere ulteriormente analizzata.

ABMapper è in grado di identificare probabili siti di *splicing* utilizzando motivi canonici come `GT-AG`, `GC-AG` e `AT-AC` per riconoscere possibili giunzioni, specialmente nel caso in cui nell'estensione siano identificati due frammenti sovrapposti.

Elementi ripetuti nella sequenza possono essere filtrati secondo un numero di ripetizioni soglia specificato dall'utente.

ABMapper cerca di fornire più informazioni utili possibili oltre al semplice allineamento (in formato SAM e BED), come dettagli sull'allineamento, siti di *splicing* e *reads* ripetute.

I. rNA

rNA (Policriti et al., 2011) è una allineatore numerico di *reads* corte.
Gli strumenti di allineamento di *reads* di piccole dimensioni sono sviluppati tenendo conto di due fattori:

- Migliorare la velocità: al fine di mantenere il passo della mole sempre più ingente di dati prodotti
- Migliorare la correttezza: ovvero massimizzare il numero di *reads* posizionate correttamente sul genoma di riferimento, essendo sicuri di averle allineate su tutte le possibili locazioni

Solitamente gli algoritmi di allineamento sacrificano leggermente la correttezza per ottenere maggiore velocità, ammettendo un numero limitato di *mismatches* per ogni *read*. Per massimizzare questo *trade-off* strumenti come Bowtie (Langmead et al., 2009) e BWA (Li and Durbin, 2009) utilizzano l'euristica *seed-and-extend*: al fine di allineare una *read* `r` una condizione necessaria è che `l` basi con $l < |r|$ della `read` abbiano un *match* quasi esatto sul genoma di riferimento. BFAST invece opera favorendo la correttezza

piuttosto che la velocità permettendo allineamenti con un numero grande di *mismatch* e *indels*, purtroppo per fare questo è di un ordine di grandezza più lento dei precedenti algoritmi.

rNA (*randomized Numerical Aligner*) supporta FASTA e FASTQ come formati in input e SAM/BAM come formati in output, è in grado di allineare sia *single reads* che *paired-end reads* e può eseguire sia in architetture parallele che distribuite.

rNA è uno strumento accurato per l'allineamento di *reads* con grande polimorfismo, presenza di errori e *indels* di piccole dimensioni, è creato per lavorare in *base-space* e quindi maggiormente indicato per Illumina *reads* ma può lavorare anche su SOLiD *reads* dopo una opportuna conversione da *color* a *base-space*.
Come la maggior parte dei software di allineamento anche rNA è diviso in due fasi:

- Pre-elaborazione del genoma di riferimento: costruisce un *hash* del genoma di riferimento dato in input
- Allineamento: utilizza l'*hash* creata nella pre-elaborazione per allineare le *reads* contro il genoma, questa fase può essere parallelizzata e distribuita sui diversi nodi di un *cluster*

1) Hashing e strategia di allineamento: rNA è basato su un metodo semplice ma efficiente originalmente proposto da Rabin e Karp (Rabin e Karp, 1987). Un *pattern* di lunghezza `l` in un alfabeto $\sum$ può essere rappresentato come un numero in base $|\sum|$. Nelle applicazioni pratiche questo numero non rientra nella dimensione di una parola della memoria, tale condizione è molto sfavorevole dal momento che le operazioni numeriche sono ritenute eseguire in tempo costante solo per quei numeri che stanno in una parola. A tale scopo Rabin e Karp definiscono il concetto di *fingerprint* ovvero la rappresentazione numerica di un *pattern* modulo un dato numero primo `q` tale che il valore di *fingerprint* rientri in una parola di memoria. Rabin e Karp hanno dimostrato che calcolando tutti i *fingerprint* delle sottostringhe lunghe `l` di un testo `T` la ricerca di *match* avviene in tempo `O(|T|)` (si noti che confrontanto i *fingerprint* non si ottengono i *match* esatti poichè la codifica numerica modulo `q` non è univoca). Tale approccio viene leggermente esteso da Olicriti et al. per gestire efficientemente anche *mismatches*. La *hash table* dei *fingerprints* viene salvata con spazio proporzionale a `|T|` (dimensione del genoma di riferimento) e `q` (dimensione della *hash table*), richiedendo `4q + 5|T|` bytes.

2) Indels: Quando rNA non riesce ad allineare una *read* con la tecnica basata su Rabin e Karp con l'estensione per i *mismatch* prova un allineamento con una versione modificata di Smith e Waterman che supporta piccoli *indels*.

3) Δ-search: Identificare se una *read* compare una sola volta nel genoma ha un importante valore biologico. Le tecniche adottate precedentemente prevedevano di identificare il *match* migliore per una *read* (minimo numero di *mismatch* `k`) e se nessun altro *match* lo uguagliava la *read* era ritenuta unica sul genoma. rNA introduce una tecnica detta Δ-*search* che permette di tenere in considerazione anche le *reads* con numero di *mismatch* tra `k+1` e `k+`Δ.

	BFAST	BOWTIE	BWA	rNA
time (hh:mm:ss)	510:33:32	01:21:02	8:05:30	20:10:35
% align	74.12% (94.03%)	69.19%	76.25%	79.4%
RAM	17.4GB	3.5GB	3.7GB	19.8GB

TABLE V
TEMPI E PERCENTUALI DI ALLINEAMENTI CORRETTI OTTENUTI DA RNA DI 166,622,914 READS PRODOTTE DAL SEQUENZIATORE ILLUMINA

Le prestazioni di rNA sono state confrontate con i maggiori programmi di allineamento: BFAST (Homer et al., 2009), Bowtie (Langmead et al., 2009), and BWA (Li e Durbin, 2009). Il lavoro di confronto è stato portato avanti sia su dati simulati che su dati reali su una macchina con un processore 8-core 2.5Ghz Intel(R) Xeon(R) e 32GB RAM, sempre usando 8 *threads*.

Tutti i programmi di allineamento sono stati eseguiti con i parametri di default fatta eccezione per rNA per il quale è stata disabilitata l'opzione di *auto trimming*. Dal momento che BFAST è stato creato per allineare *reads* corte a grande distanza per esso sono state filtrate le *reads* allineate con più di 7 *mismatches* (senza limiti per il numero di *indels*), tra parentesi viene espresso il suo output originale. BFAST è uno strumento per allineare tutte le reads ma nell'utilizzo pratico richiede troppo tempo. Al contraro Bowtie è lo strumento più veloce ma le sue prestazioni sono ottenute al prezzo di un peggiore allineamento. rNA e BWA raggiungono risultati simili: rNA allinea circa 3% in più delle *reads* ma d'altro canto BWA è più veloce.

J. SOAP

SOAP è un algoritmo di allineamento specificatamente creato per identificare SNPs (*Single Nucleotide Polymorphisms*). Si è evoluto in tre versioni, SOAPv2 (Li et al., 2009) come BWA e il suo predecessore MAQ migliora SOAPv1 (Li et al., 2008) utilizzando un indice basato su *Burrows-Wheeler Transformation* (BTW). Questo ha migliorato notevolmente la velocità di allineamento e l'utilizzo di memoria.
SOAPv2 determina *match* costruendo una *hash table* che accelera ulteriormente le ricerce nell'indice del genoma di riferimento.

1) SOAPv3: Utilizzando il genoma umano come riferimento, allineare 70 milioni di coppie di read (equivalente al prodotto di una corsa del sequenziatore Illuminal HiSeq 2000) con al massimo 4 *mismatches* richiede più di 3.5 ore utilizzando l'allineatore più veloce esistente.

Per allineare 1G di coppie di *reads* (che conducono approssimativamente ad una copertura 30x del genoma umano) sono necessari più di due giorni per completare il processo di allineamento. Detto questo appare molto difficile ridurre i tempi utilizzando una singola CPU.

SOAPv3 (Li et al., 2012) utilizza i multi-processori in una GPU (*Graphic Processing Unit*) per migliorare sensibilmente le prestazioni. SOAPv3 sviluppa una versione per GPU delle strutture dati per l'indicizzazione compressa del genoma di riferimento utilizzata da SOAPv2 e basate su BWT.
SOAPv3 introduce a tale scopo due novità.

La ricerca di *pattern* utilizzando BWT richiede un numero piuttosto alto di accessi in RAM. Una implementazione diretta della tecnica per *single* CPU su GPU porterebbe ad avere un numero notevole di *threads* in accesso concorrente alla memoria, degradando le prestazioni complessive dell'algoritmo. Questo problema è stato risolto modificando la struttura dati per ridure quanto più possibile gli accessi a memoria pur mantenendo l'efficienza dell'indicizzazione.

La seconda difficoltà incontrata risiedeva nel fatto che le GPU lavorano in modalità SIMT (*Single-Instruction Multiple-Thread*). Processori nella stessa unità (detta *Streaming Multi-processor* - SM) devono eseguire la stessa istruzione. Troppi *branch* divergenti nel cammino dell'esecuzione parallela possono forzare alcuni processori ad andare in *idle*.

Purtroppo il numero di *branch* divergenti che la ricerca di un *pattern* può introdurre non può essere determinato se non a tempo di esecuzione. SOAPv3 stima un parametro in grado di dire a *runtime* se un *pattern* introdurrà troppi *branch* divergenti (in questo caso si parla di *hard patterns*). SOAPv3 termina l'esecuzione di *hard patterns*, li raggruppa e va ad allinearli in un secondo momento allo scopo di ridurre il tempo di *idle* dei processori.

La versione corrente di SOAPv3 riesce ad allineare *reads* con un numero di *mismatches* non superiore a 4. Le prestazioni di SOAPv3 sono state confrontate con Bowtie e BWA su un computer con 3.07GHz quad-core CPU e 24GB di RAM. SOAPv3 è supportato dalla scheda grafica NVIDIA GTX 580 con 3GB di memoria, necessita infatti di una GPU CUDA con almeno 2.5GB di memoria per indicizzare il genoma umano.

Come appare chiaro dalla tabella VI SOAPv3 è notevolmente più veloce di BWA e Bowtie.
Sono stati inoltre condotti ulteriori esperimenti che confrontano le prestazioni di SOAPv3 con la precedente versione SOAPv2, ottenendo i dati in tabella VII.

Come SOAPv2 il formato dei dati in output di SOAPv3 include testo e formato SAM/BAM.

Attualmente gli sviluppatori di SOAPv3 stanno lavorando ad alcuni miglioramenti per mezzo di programmazione dinamica *GPU-based* in modo da riportare gli allineamenti con *gap* e *indels*. I risultati finora ottenuti mostrano che le percentuali di *reads* allineate dovrebbero migliorare di un 5-6%.

	SOAPv3		SOAPv2	
	(s)	(%)	(s)	(%)
Index loading	133		40	
HiSeq dataset: 70.7M				
Read loading	259		259	
4 mismatch alignment	1447	81.46%	12206	77.25%
3 mismatch alignment	626	79.43%	12198	76.65%
2 mismatch alignment	303	76.48%	4370	76.48%
G. A. II dataset: 25.3M				
Read loading	107		107	
4 mismatch alignment	294	86.58%	3495	84.13%
3 mismatch alignment	212	85.34%	3453	83.63%
2 mismatch alignment	356	83.46%	1465	83.46%

TABLE VII
TEMPI DI ESECUZIONE E PERCENTUALI DI READS ALLINEATE DI SOAPV3 E SOAPV2 SU DUE *paired-end datasets* CON 70.7M E 25.3M COPPIE DI *reads*

IV. ALLINEAMENTO DI READS LUNGHE

Negli ultimi anni sono state sviluppate molte soluzioni in grado di allineare *reads* di piccole dimensioni su un genoma di riferimento, tali soluzioni non sono in grado di gestire *reads* di maggiori dimensioni dal momento che sono specificatamente costruite per lavorare su *reads* corte e con una bassa frequenza di errori. Per *reads* lunghe soluzioni basate sull'*hashing* come BLAT e SSAHA2 rappresentano l'unica scelta ma sono notevolmente più lente delle soluzioni per *reads* di piccole dimensioni in termini di basi allineate per unità di tempo.

A. BWA-SW

Nell'allineamento di *reads* corte si cercano quanto più possibile situazioni di *match* esatto fatta eccezione per gli estremi di ciascuna *read* che sono strutturalmente più soggetti ad errori. Questo ragionamento non è valido per le *reads* più lunghe per le quali è più utile ricercare allineamenti locali dal momento che esse sono più soggette a variazioni strutturali ed errori di assemblaggio. Inoltre allineatori di *reads* corte perdono molto in termini di prestazioni quando devono effettuare allineamenti con *gap* (solitamente ammettono un numero limitato di *gap* consecutivi), gli allineatori di *reads* lunghe devono invece essere molto permissivi con i *gap* dal momento che gli *indels* occorrono più frequentemente in *reads* lunghe e possono essere la principale causa di errore in sequenziatori come 454 e Pacific Bioscience.

Quando si considerano algoritmi per velocizzare l'allineamento di *reads* di grandi dimensioni la tecnica dell'*hashing* non è l'unica da tenere in considerazione.

70.7M read pairs

	4 Mismatch		3 Mismatch		2 Mismatch		1 Mismatch		0 Mismatch	
	(s)	(%)	(s)	(%)	(s)	(%)	(s)	(%)	(s)	(%)
SOAPv3	1839	81.46	1019	79.43	695	76.48	521	70.94	452	53.11
BWA	13756	81.03	10590	79.19	8920	76.38	6064	70.90	5272	53.11
Bowtie	Not supported		29178	79.42	2082	76.47	1570	70.93	1216	53.10

25.3M read pairs

	4 Mismatch		3 Mismatch		2 Mismatch		1 Mismatch		0 Mismatch	
	(s)	(%)	(s)	(%)	(s)	(%)	(s)	(%)	(s)	(%)
SOAPv3	735	86.58	453	85.34	356	83.46	291	79.17	452	60.30
BWA	4700	86.17	3803	85.12	3298	83.38	2352	79.17	5272	60.30
Bowtie	Not supported		9486	83.45	617	76.47	1570	79.17	1216	60.13

TABLE VI
TEMPI DI ESECUZIONE E PERCENTUALI DI *reads* ALLINEATE DI SOAPV3, BWA E BOWTIE SU DUE *paired-end datasets* CON 70.7M E 25.3M COPPIE DI *reads*. I TEMPI RIPORTATI INCLUDONO IL TEMPO DI CARICAMENTE DELL'INDICE IN MEMORIA, IL TEMPO DI LETTURA DELLE *reads* ED IL TEMPO DI ALLINEAMENTO VERO E PROPRIO. AL FINE DI MIGLIORARNE LE PRESTAZIONI BWA VIENE ESEGUITO DISABILITANDO L'ALLINEAMENTO CON *gap*

Sono state presentate infatti tecniche che utilizzano un algoritmo dimanico come Smith e Waterman con alberi dei suffissi del genoma di riferimento (Meek et al., 2003), queste tecniche allineano effettivamente la sequenza *query* con ogni sotto-sequenza del riferimento estratta tramite un attraversamento *top-down*. Se le sotto-sequenze equivalenti fossero rappresentate come un unico cammino sul *suffix tree* si risparmirebbe spazio e tempo evitando allineamenti ripetuti su sequenze identiche. In questa direzione Lam et al. (2008) rappresentarono l'albero dei suffissi tramite l'indice di Ferragina e Manzini che è basato su BWT, l'algoritmo che utilizzata tale struttura dati per allineare *reads* lunghe prende il nome di BWT-SW (*Burrows-Wheeler Aligner's Smith-Waterman Alignment*).
BWT-SW è in grado di ottenere risultati identici all'allineamento con Smith e Waterman ma lo fa migliaia di volte più velocemente rispetto ad allineamenti contro il genoma umano. BWT-SW in questa versione semplificata resta più lento di BLAST per lunghe sequenze *query* ma trova tutti gli allineamenti senza utilizzare euristiche.

BWA-SW introduce euristiche per velocizzare l'allineamento. In un certo senso BWA-SW segue il paradigma *seed-and-extends*, ma, diversamente da BLAT e SSAHA2, BWA-SW identifica i *seeds* per mezzo di programmazione dinamica su due indici di Ferraggina Manzini e ammette *mismatch* e *gap* all'interno dei *seeds* stessi. BWA-SW estende un *seed* soltanto quando esso ha poche orrenze nel genoma di riferimento, la velocità viene ottenuta riducendo estensioni non necessarie per sequenze ripetute.

L'algoritmo BWA-SW è implementato come un componente del programma BWA (Li and Durbin, 2009). L'implementazione prende in input un indice BWA e una *query* in formato FASTA o FASTQ e restituisce come risultato l'allineamento in formato SAM. Il file della sequenza *query* contiene tipicamente diverse sequenze (*reads*). Ciascuna *read* viene processata indipendentemente, eventualmente usando più *threads*. L'utilizzo della memoria è dominato dalla dimensione dell'indice di Ferragina e Manzini, 3.7GB per il genoma umano. La memoria necessaria per ogni *read* è approssimativamente proporzionale alla lungezza della *read*, su misure tipiche di *reads* la memoria totale occupata è $<$ 4GB.

Nell'implementazione BWA-SW cerca di aggiustare i propri parametri sulla base della lunghezza delle *reads* e della frequenza di errori di sequenziamento per fare in modo che le impostazioni di default si comportino meglio su input con caratteristiche differenti.

Le prestazioni di BWA-SW sono state misurate confrontandole con quelle di BLAT e SSAHA2 sia su dati simulati che su dati reali. Il confronto su dati reali risulta essere più complesso e meno attendibile dal momento che mancano certezze sui risultati dell'allineamento per *reads* lunghe. Per quello che riguarda i dati simulati in tabella VIII sono stati calcolati su *reads* di diversa lunghezza con una frequenza di errore del 2%.

Dalla tabella VIII è possibile vedere che BWA-SW è chiaramente il programma più veloce, molte volte più di BLAT e SSAHA2 su La precisione di BWA-SW è paragonabile a quella di SSAHA2 quando la query è di grandi dimensioni, quando invece è più piccola SSAHA2 è più accurato anche se deve spendere molto più tempo per l'allineamento. SSAHA2 non viene confrontato con le *reads* di 10,000 basi dal momento che non è stato inizialmente pensato per tale compito e si comporta conseguentemente male. BLAT sembra essere più performante di SSAHA2 ma soltanto perchè viene eseguito con l'opzione `-fastMap` e

Programma-Metriche	100bp	200bp	500bp	1,000bp	10,000bp
BLAT-time	685	819	1078	1315	2628
BLAT-Q20%	68.7	92.0	97.1	97.7	98.4
BLAT-errAlg%	0.99	0.55	0.17	0.01	0.00
BWA-SW-time	165	222	249	234	158
BWA-SW-Q20%	85.1	93.8	96.1	96.9	98.4
BWA-SW-errAlg%	0.01	0.00	0.00	0.00	0.00
SSAHA2-time	4872	1932	3311	1554	-
SSAHA2-Q20%	85.5	93.4	96.6	97.7	-
SSAHA2-errAlg%	0.00	0.01	0.00	0.00	-

TABLE VIII
CIRCA 10,000,000 DI BASI RAGGRUPPATE IN *reads* DI DIVERSA LUNGHEZZA SONO SIMULATE A PARTIRE DAL GENOMA UMANO. IL 20% DEGLI ERRORI SONO *indels* CON LUNGHEZZA CHE EVOLVE SECONDO UNA DISTRIBUZIONE GEOMETRICA. QUESTE *reads* SIMULATE SONO ALLINEATE SUL GENOMA UMANO CON BLAT, BWA-SW E SSAHA2. LE COORDINATE DI ALLINEAMENTO SONO POI CONFRONTATE CON LE COORDINATE SIMULATE PER TROVARE GLI ERRORI DI ALLINEAMENTO. I DATI RAPPRESENTATI SONO IL TEMPO SU UN *core* DI UN PROCESSORE INTEL E5420 2.5 GHZ CPU, LA PERCENTUALE DI *reads* ALLINEATE CON QUALITÀ MAGGIORE O UGUALE A 20 (Q20) E LA PERCENTUALE DI ALLINEAMENTI ERRATI IN TALE Q20

paga questa velocità con meno precisione, con le opzioni di default BLAT è più accurato ma 10 volte più lento di SSAHA2 e comunque non accurato quanto BWA-SW.

Per quello che riguarda la memoria sia BWA-SW che BLAT utilizzano circa 4GB. SSAHA2 utilizza 2.4GB per *reads* più lunghe di 500 basi e 5.3GB per reads più corte (utilizzando l'opzione -454 che aumenta il numero di sequenze *seed* salvate nell'*hash* e quindi la memoria utilizzata). Inoltre BWA-SW supporta il *multi-threading* quindi può richiedere meno memoria per ciascun *core* se è eseguito su un *multi-core*. SSAHA2 e BLAT non supportano per ora il *multi-threading*.

REFERENCES

[1] Altschul,S,. et al. (1990) Basic local alignment search tool. Journal of Molecular Biology, 215, 403–410.

[2] Burrows,M. and Wheeler,D. (1994) A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation.

[3] Chan,T. et al. (2010) ABMapper: a suffix array-based tool for multi-location searching and splice-junction mapping. Bioinformatics, 27, 421-422.

[4] David,M. et al. (2011) SHRiMP2: Sensitive yet Practical Short Read Mapping. Bioinformatics, 27, 1011–1012.

[5] Ferragina,P. and Manzini,G. (2000) Opportunistic data structures with applications. In Proceedings of the 41st Symposium on Foundations of Computer Science (FOCS 2000), IEEE Computer Society, 390–398.

[6] Homer,N. et al. (2009) Bfast: an alignment tool for large scale genome resequencing. PLoS ONE, 4, e7767.

[7] Karp,R. and Rabin,M. (1987) Efficient randomized pattern-matching algorithms. IBM J. Res. Develop., 31, 249–260.

[8] Li,H. et al. (2008) Mapping short DNA sequencing reads and calling variants using mapping quality scores. Genome Res., 18, 1851–1858.

[9] Li,H. and Durbin,R. (2009) Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics, 25, 1754–1760.

[10] Li,H. and Durbin,R. (2010) Fast and accurate long-read alignment with Burrows-Wheeler transform. Bioinformatics, 26, 589–595.

[11] Li,R. et al. (2008) SOAP: short oligonucleotide alignment program. Bioinformatics, 24, 713–714.

[12] Li,R. et al. (2009) SOAP2: an improved ultrafast tool for short read alignment. Bioinformatics, 25, 1966–1967.

[13] Li,R. et al. (2012) SOAP3: Ultra-fast GPU-based parallel alignment tool for short reads.

[14] Lipman,D.J. and Pearson,W.R. (1988) Improved tools for biological sequence comparison. Biochemistry, 85, 2444-2448.

[15] Policriti,A. et al. (2011) rNA: a Fast and Accurate Short Reads Numerical Aligner. Bioinformatics Advance Access.

[16] Ruffalo,M. et al. (2011) Comparative analysis of algorithms for next-generation sequencing read alignment. Bioinformatics, 27, 2790-2796.

[17] Smith,T.F. and Waterman,M.S. (1981) Identification of common molecular subsequences. J. Mol. Biol., 147, 195–197.