



Università degli studi di Salerno

STRUMENTI FORMALI PER LA BIOINFORMATICA

ABYSS E BLOOM FILTERS: UN CASO DI STUDIO

Dipartimento di Informatica

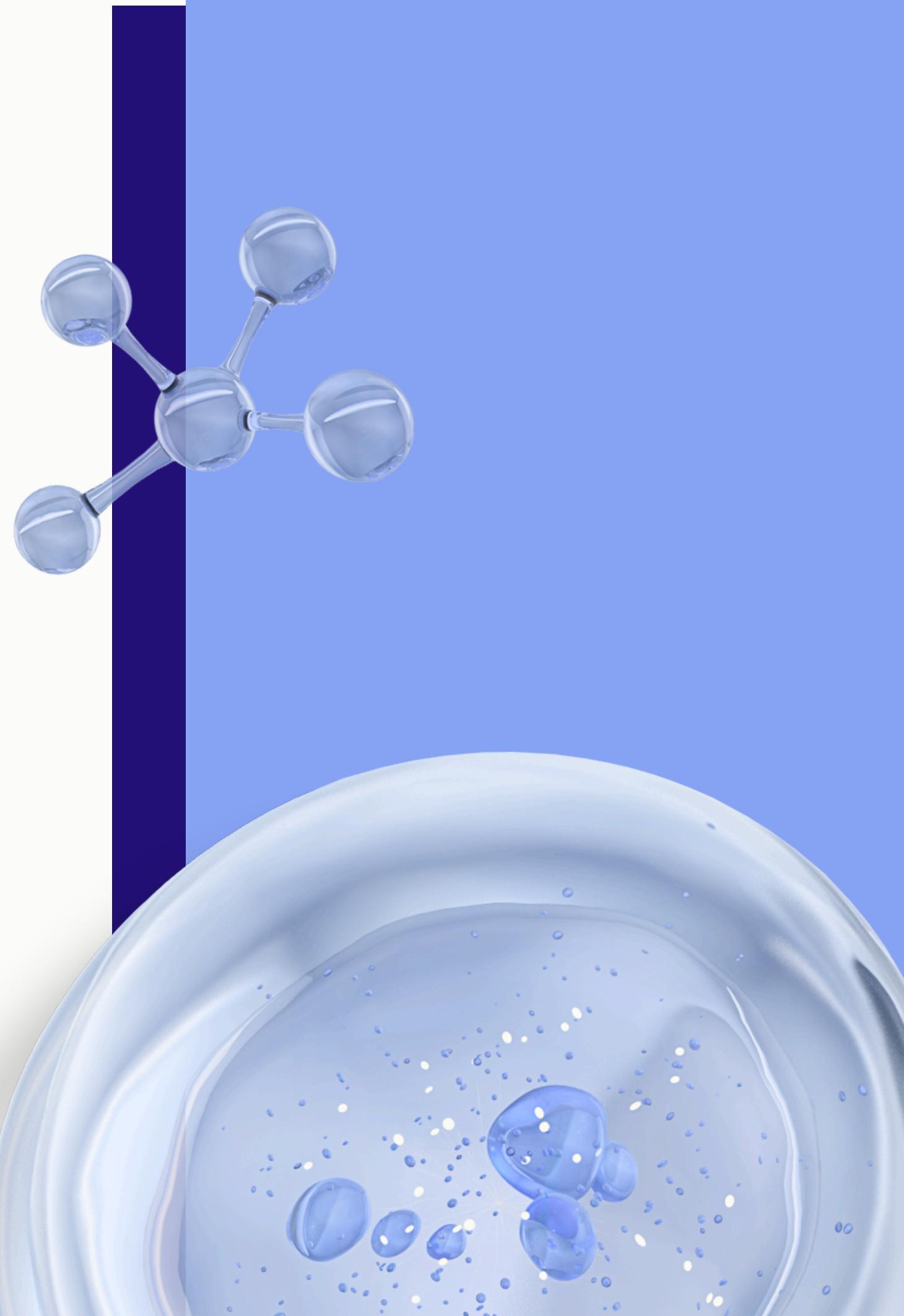
Francesco Maddaloni 0522501740
Pietro Negri 0522501367

Prof.ssa Rosalba Zizza
Prof.ssa Clelia De Felice
Prof. Rocco Zaccagnino

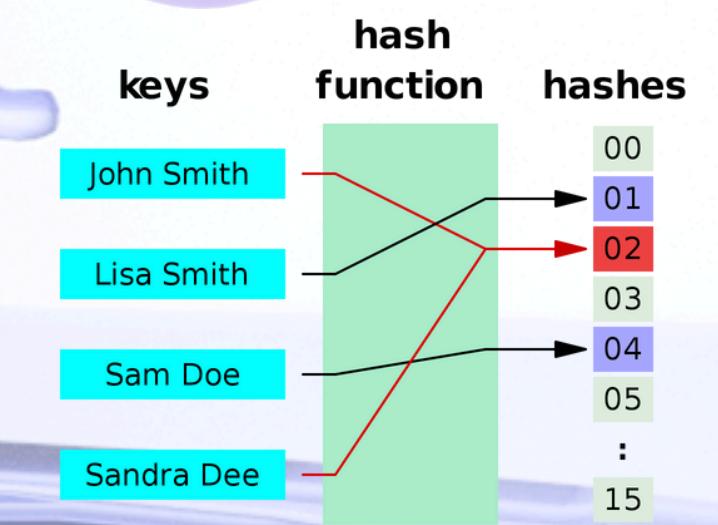


INTRODUZIONE

- Una **sequenza genomica** è una successione ordinata delle basi azotate presenti nel genoma.
- Durante l'analisi di un campione il genoma viene **sequenziato** in piccoli frammenti denominati **reads**.
- Attraverso l'assemblaggio di queste reads è possibile riottenere la sequenza genomica originale.
- Sono necessari sofisticati strumenti per effettuare questa operazione.
- Questa tesi si propone di esaminarne uno in particolare: **Abyss**.



UN PO' DI STRUMENTI



GRAFI DI DE BRUJIN

Grafo $G = (V, A)$ di un insieme di read $\{r_1, r_2, \dots, r_n\}$:

- Ad ogni nodo v_i è associato un **(K-1)-mer**
- A rappresenta l'insieme degli elementi dello **spettro di ordine k**, ovvero l'arco da v_1 a v_2 è tale che:
 - il suffisso di v_1 lungo $k-2$ è uguale al prefisso di v_2 lungo $k-2$
 - il k -mer ottenuto assemblando v_1 e v_2 è **sottostringa** in almeno una **read**
- È etichettato dall'**estensione (lunga un carattere)** di v_1 che eccede l'overlap con v_2

GRAFI DI DE BRUJIN VANTAGGI



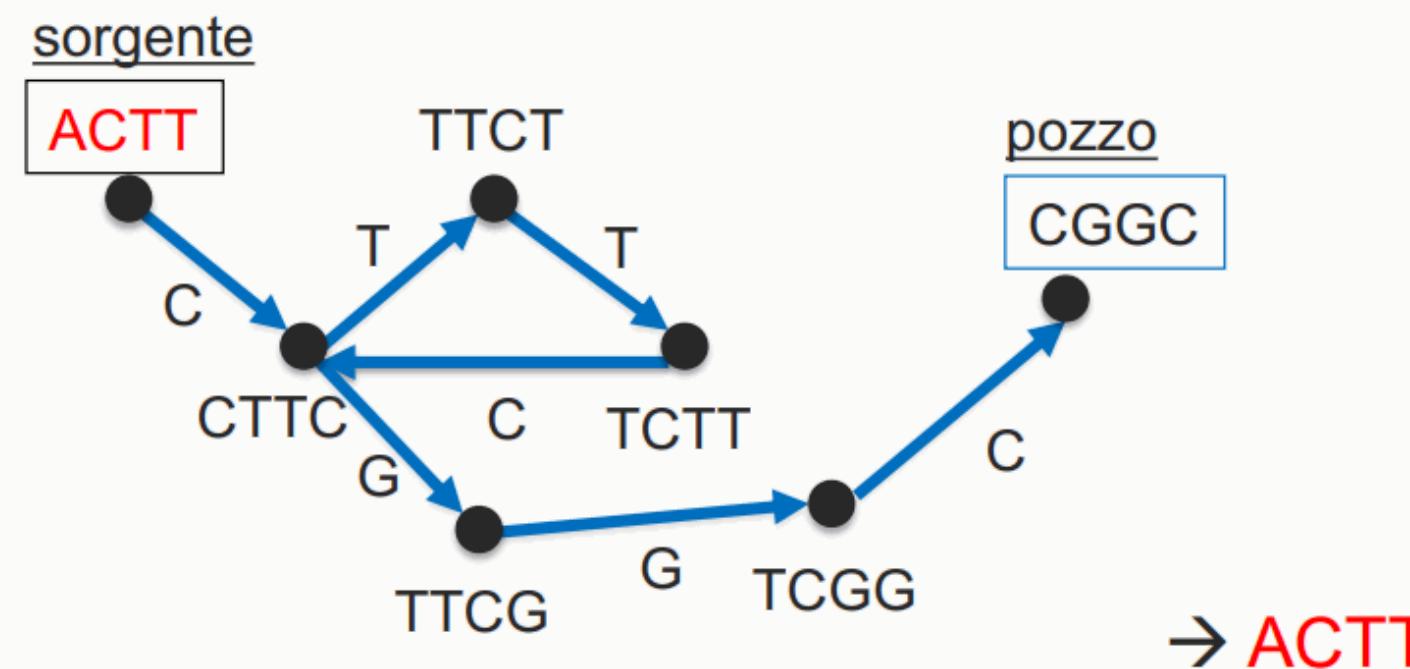
È possibile rappresentare un'**intera** sequenza genomica tramite questa struttura dati.



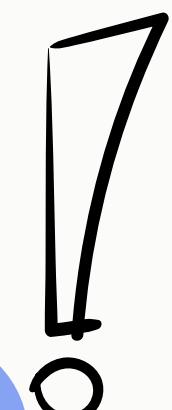
L'idea è trovare un **cammino Euleriano** che attraversi l'intero grafo



Esso attraverserà **ogni arco una e una sola volta** ricomponendo la sequenza originale attraverso i (K-1)-Mers presenti su ogni nodo



Esempio sequenza:
ACTTCTTCGGC K=5

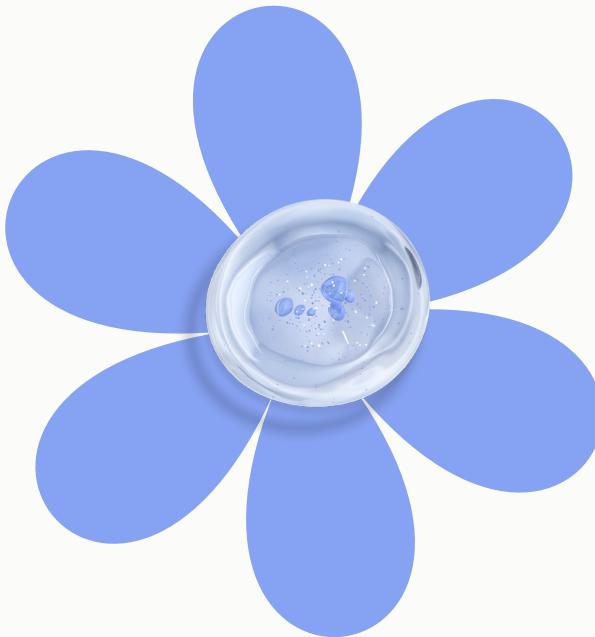


BLOOM FILTERS



Salvare una struttura come il grafo di de Brujin può essere **costoso** in termini di **memoria**.

- Si può rappresentare attraverso un **Bloom Filter**.
- Un Bloom Filter è una struttura dati **probabilistica** utilizzata per verificare in maniera efficiente l'appartenenza di un elemento a un insieme.
- Consiste di un array di cui ogni casella è uno 0 o 1.
- Sono possibili inserimenti, query e cancellazioni.

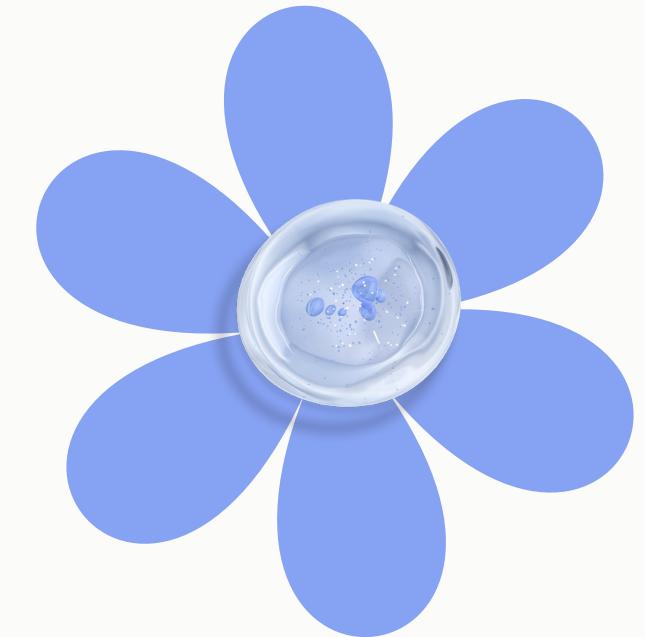


BLOOM FILTERS



Salvare una struttura come il grafo di de Bruijn può essere **costoso** in termini di **memoria**.

- Si può rappresentare attraverso un **Bloom Filter**.
- Un Bloom Filter è una struttura dati **probabilistica** utilizzata per verificare in maniera efficiente l'appartenenza di un elemento a un insieme.
- Consiste di un array di cui ogni casella è uno 0 o 1.
- Sono possibili inserimenti, query e cancellazioni.



Le cancellazioni sono da evitare!
Vediamo il perchè...

BLOOM FILTERS



Per operare con un elemento viene effettuato uno o più **hash** di quest'ultimo.

- Per l'inserimento **viene inserito un 1** negli indici della **bit signature dell'elemento**
- Per la ricerca di un elemento se tutte le posizioni della sua **bit signature** sono 1, **probabilmente** è presente
- Il risultato della ricerca **non è deterministico**. Possono esistere dei **Falsi Positivi**!



Le cancellazioni sono da evitare!
Portano falsi negativi



MINIA (R.CHIKHI, G. RIZK, 2013)



Minia è un assemblatore che basa la sua struttura sulla rappresentazione del De Bruijn Graph attraverso un Bloom Filter.



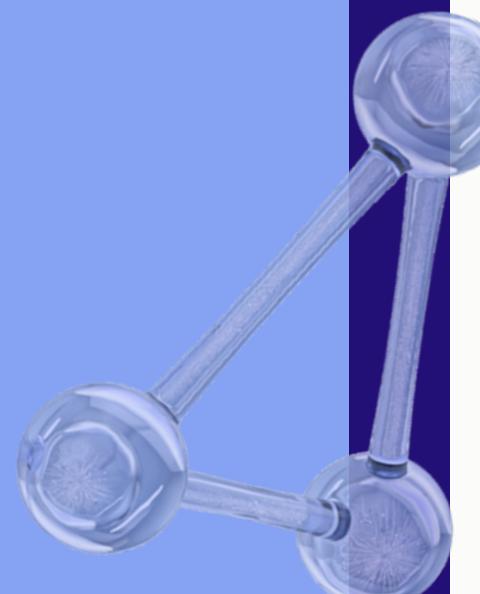
Gli archi sono dedotti implicitamente cercando nel Bloom Filter l'appartenenza di **tutte le possibili estensioni** di un K-Mer.



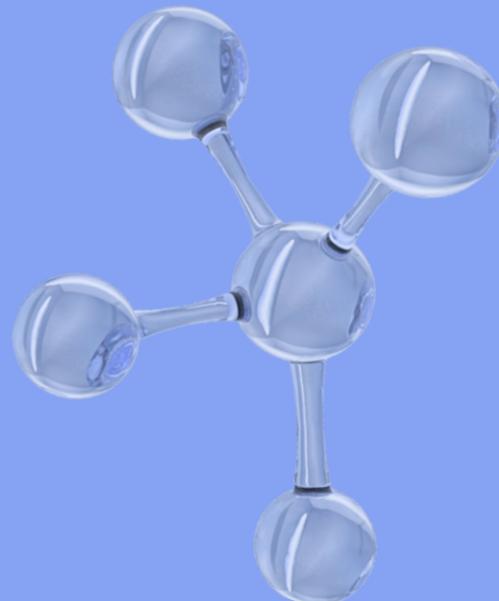
Per estensione si intende apporre al k-mer in analisi **una delle quattro possibili basi azotate (A,C,T,G)**.



Il risultato è un' **approssimazione per eccesso** del grafo originale.



MINIA (R.CHIKHI, G. RIZK, 2013)



Minia è un assemblatore che basa la sua struttura sulla rappresentazione del De Brujin Graph attraverso un Bloom Filter.



Gli archi sono dedotti implicitamente cercando nel Bloom Filter l'appartenenza di **tutte le possibili estensioni** di un K-Mer.



Per estensione si intende apporre al k-mer in analisi **una delle quattro possibili basi azotate** (A,C,T,G).



Il risultato è un' **approssimazione per eccesso** del grafo originale.



Viene chiamato De Brujin Graph probabilistico.

MINIA SOLID K-MERS



Per migliorare le prestazioni dell'assemblatore
non vengono analizzati tutti i k-mers.



Sono presi in considerazione solo i k-mers con **un più di d** occorrenze.



Questi k-mers verranno denominati **solid k-mers**.

MINIA SOLID K-MERS



Per migliorare le prestazioni dell'assemblatore non vengono analizzati tutti i k-mers.



Sono presi in considerazione solo i k-mers con **un più di d** occorrenze.



Questi k-mers verranno denominati **solid k-mers**.

7
O

Sperimentalmente d è impostato a 3.

MINIA

GESTIONE DEL FALSE BRANCHING



Minia introduce una struttura dati ausiliari denominata **cFP**.



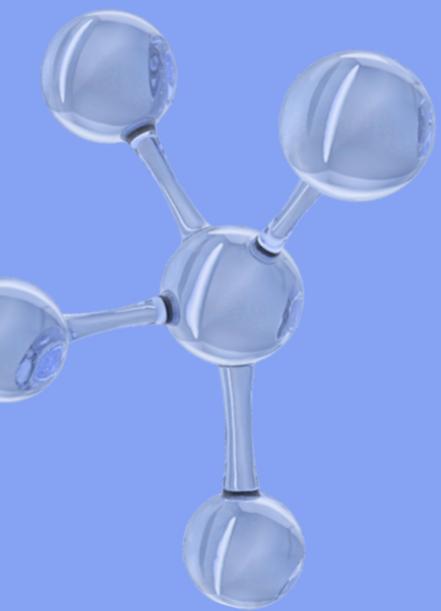
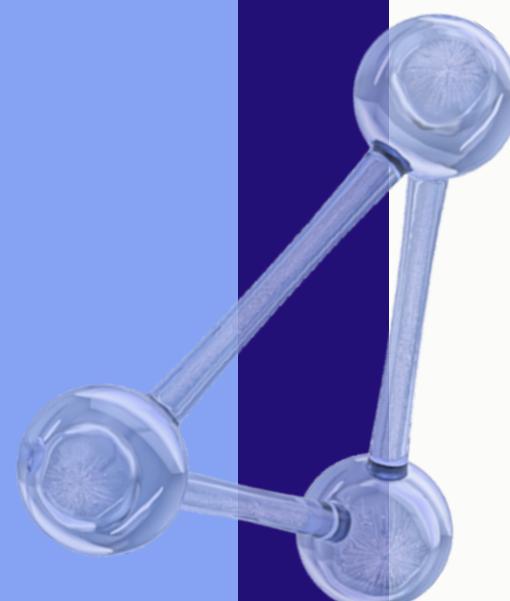
Durante il traversing tutti **i falsi positivi** sono inseriti in questa struttura.



Una **dimensione sbagliata** di cFP potrebbe far perdere i vantaggi prestazionali ottenuti fin ora.



Per **evitare sprechi di memoria** la struttura viene ridimensionata in base alle dimensioni del Bloom Filter.



MINIA

GESTIONE DEL FALSE BRANCHING



Il cFP è implementato come uno standard set.



Viene conservato su disco un insieme completo **S** che contiene tutti i k-mer



Viene chiamato **P** l'insieme che contiene gli elementi a cui il BF risponde True



Il set viene formalmente definito come:

$$cFP = P/S$$



Si costruisce iterativamente caricando un frammento alla volta di S e confrontando il risultato di S con i dati presenti nel Bloom Filter.



Un query nel Bloom Filter ora restituirà True se e solo se avremo un True dal BF e False da cFP



MINIA

GESTIONE DEL TRAVERSING



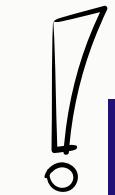
Il bloom filter non permette di **tener traccia** del traversing.



Minia introduce una struttura dati per tenere traccia delle porzioni del grafo già visitate.



Sono considerati solo i nodi con più di un arco entrante e/o uscente. I **nodi complessi**.



Potrei rivisitare nodi già visitati.



Il numero di nodi complessi è di molto inferiore al numero totale di nodi.



ABYSS 1.0 (S.D. JACKMAN, J.CHU ET AL., 2009)



Rilasciato per la prima volta nel **2009**



Tra i primi tool in grado di assemblare completamente il genoma umano su una macchina **consumer**



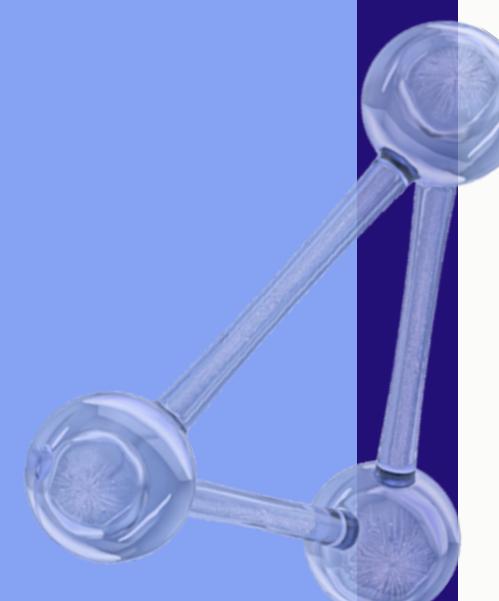
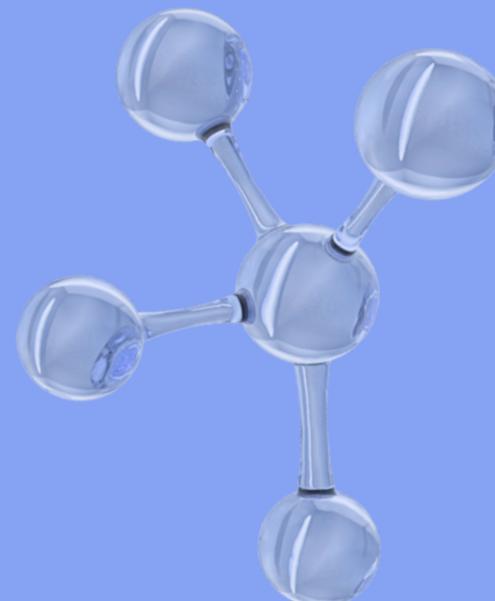
La prima versione utilizzava il **protocollo MPI** e richiedeva diversi **compute nodes**



MPI è uno standard per librerie di **message passing**, utilizzato in ambito di **parallel computing**



Consente la creazione di **cluster di macchine**



ABYSS 1.0



Organizzato come una **pipeline multistage**



3 fasi: **unitig, contig e scaffolding**



La più onerosa è **unitig**, che richiedeva memoria
nell'ordine dei TB



Il problema?

- Le strutture dati: caricava **ogni k-mer** e altri **valori aggiuntivi** in una **grandissima hash table**
- Poi costruiva il **grafo di de Bruijn** per l'assemblaggio
- Di fatto **neutralizzava** il vantaggio di usare **hardware commodity**

ABYSS 2.0 (S.D.JACKMAN, J.CHU ET AL. 2022)

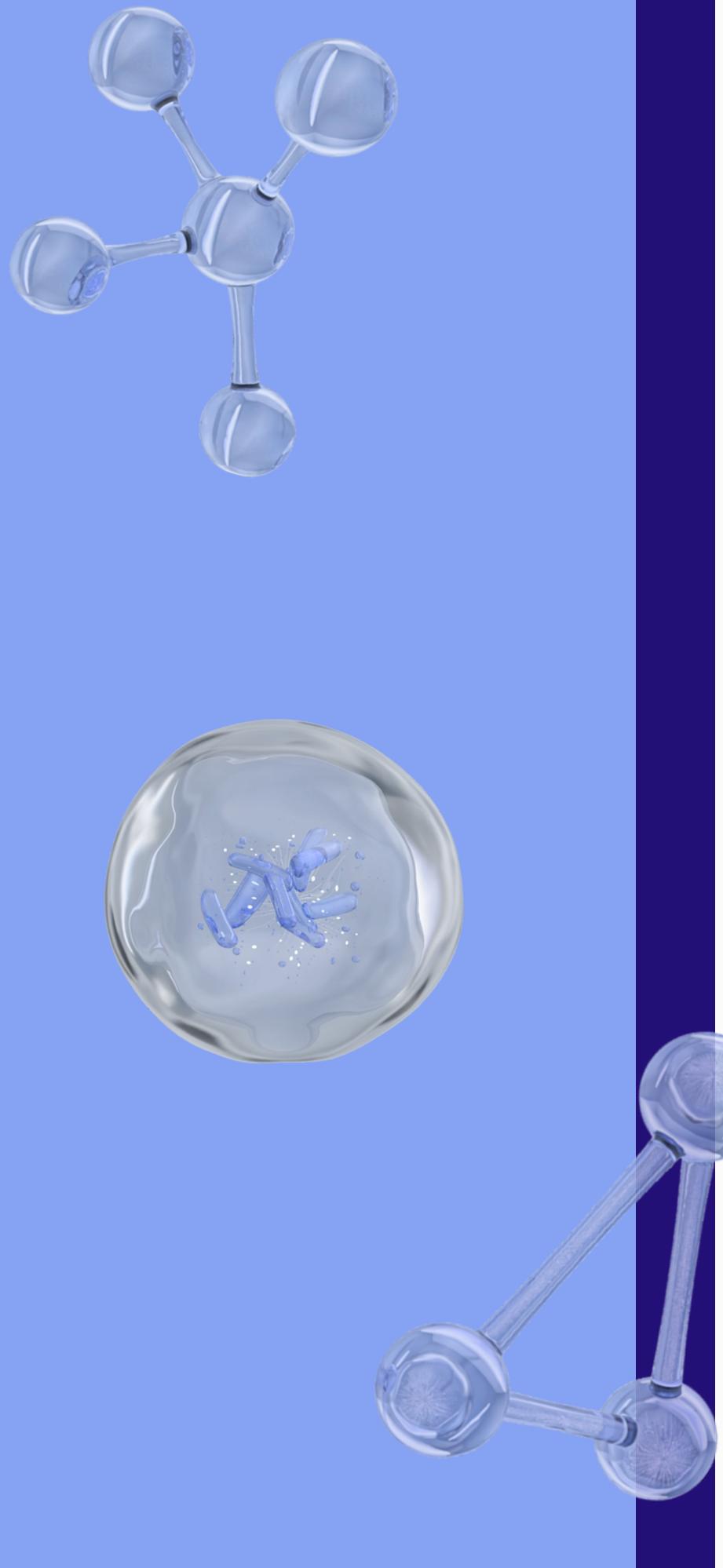
Bloom Filter per la rappresentazione del grafo, ma con alcune differenze rispetto a Minia

Non si conservano i **branching points** durante le visite del grafo, ma si estendono delle **solid reads**

Utilizzo di una combinazione di **Cascading Bloom Filters** e di meccanismi di **look-ahead** per gestire i **falsi positivi**

L'uso di una funzione hash specializzata, **ntHash**

IL FUNZIONAMENTO A GRANDI LINEE



Vengono estratti i **k-mer** dalle **reads**



Si filtrano i **k-mer** che hanno un numero di occorrenze sotto una certa soglia, utilizzando dei **Cascading Bloom Filters**



Il CBF è un array di filtri, dove ogni filtro contiene i k-mer che occorrono **una volta in più** di quello precedente



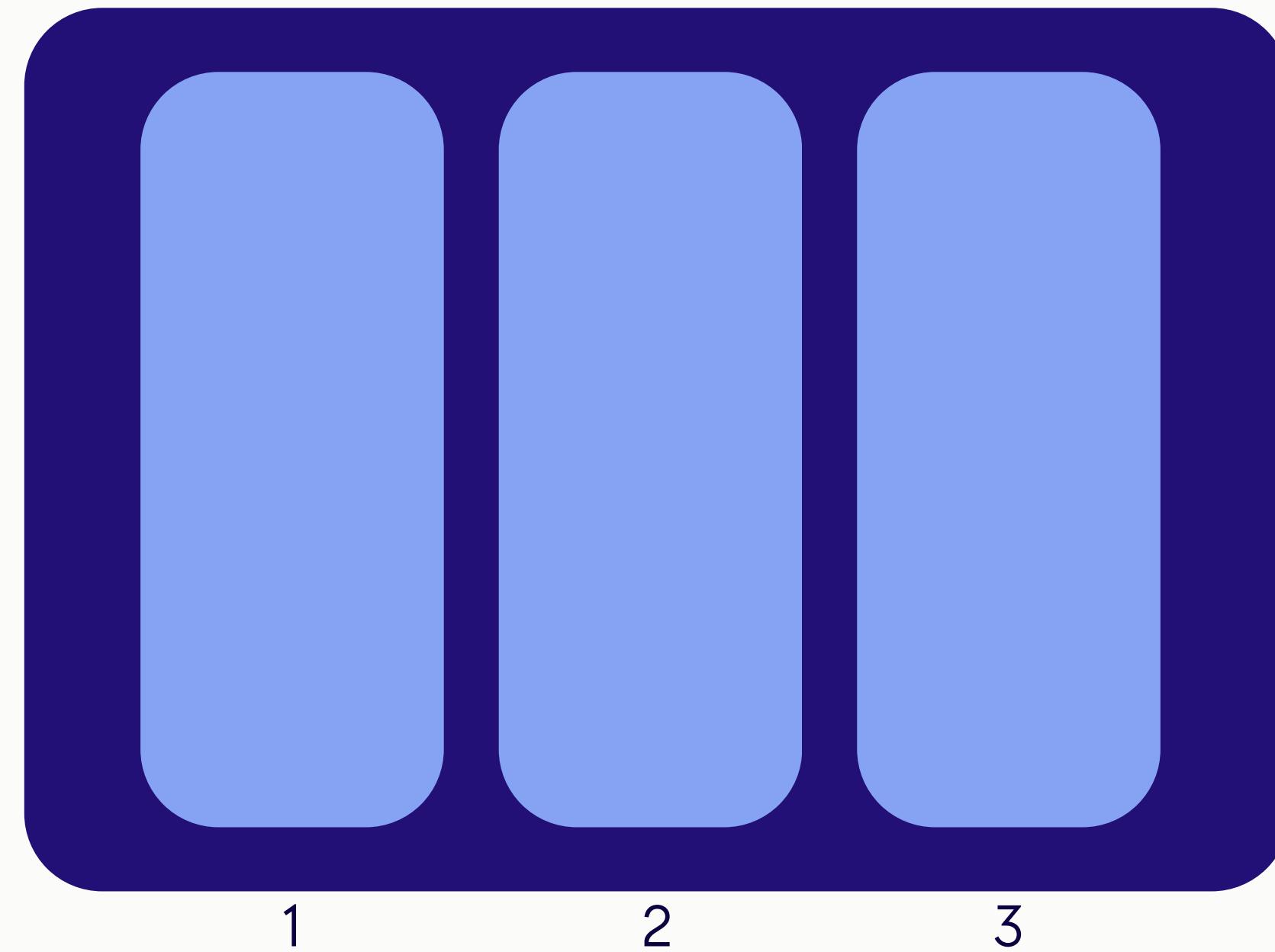
L'inserimento di un k-mer avviene cercando il **primo filtro dove non è presente**

ESEMPIO

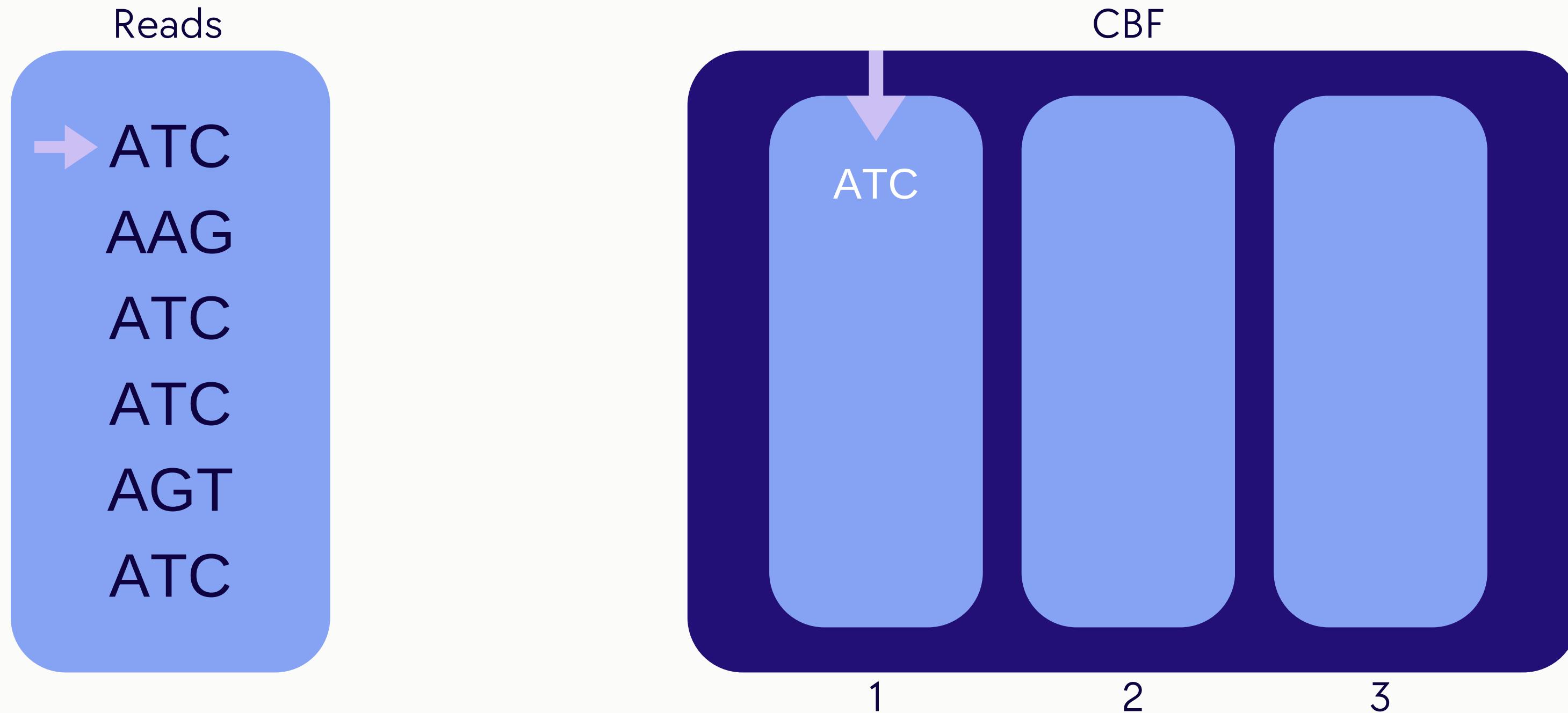
Reads

ATC
AAG
ATC
ATC
AGT
ATC

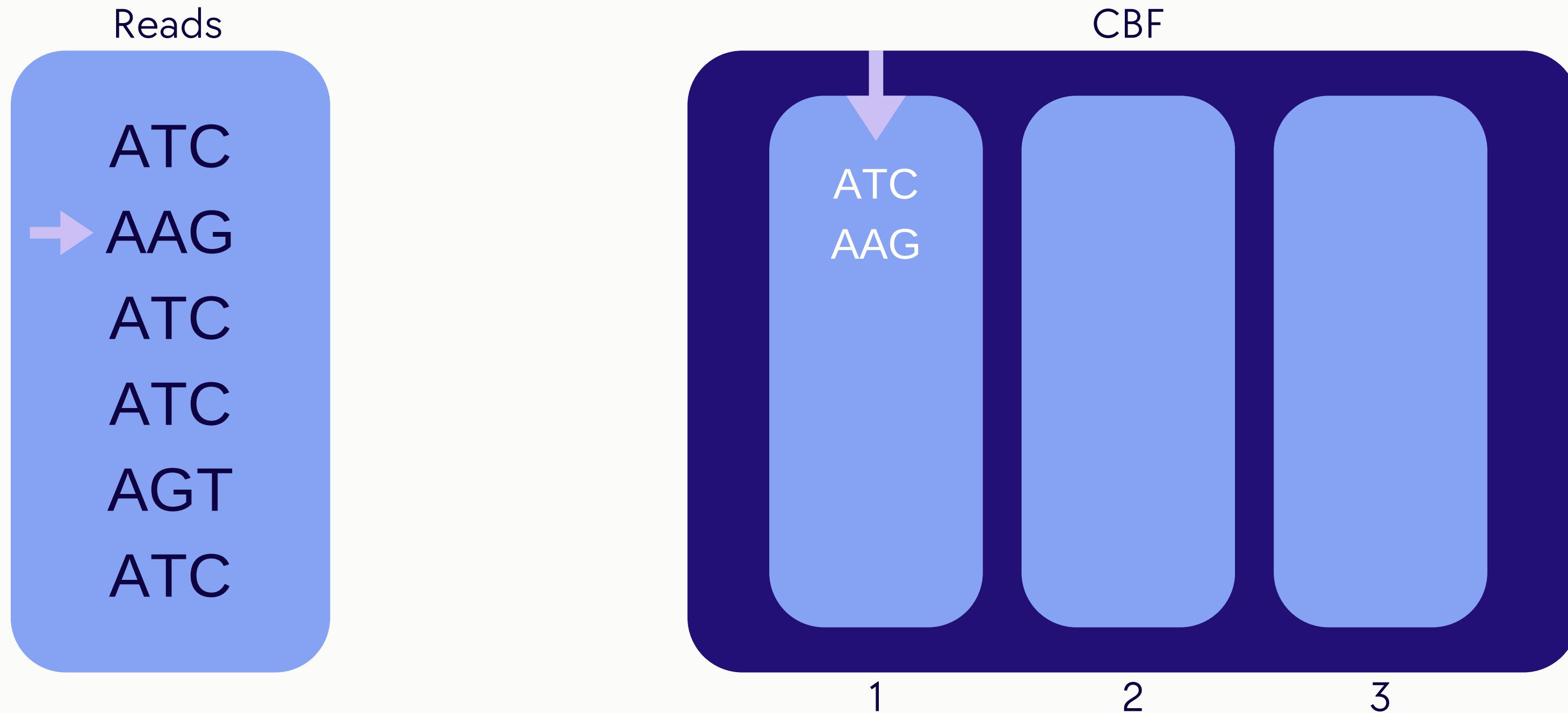
CBF



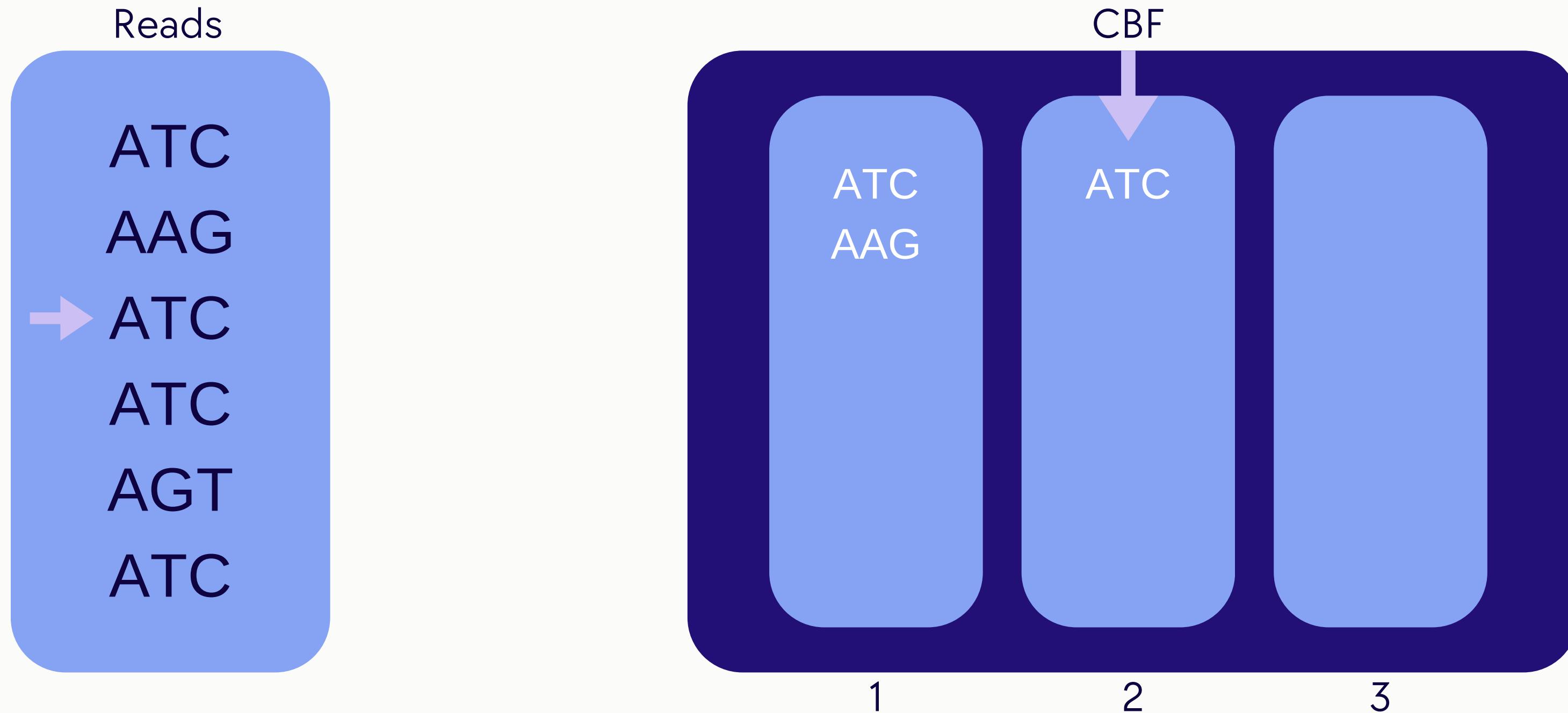
ESEMPIO



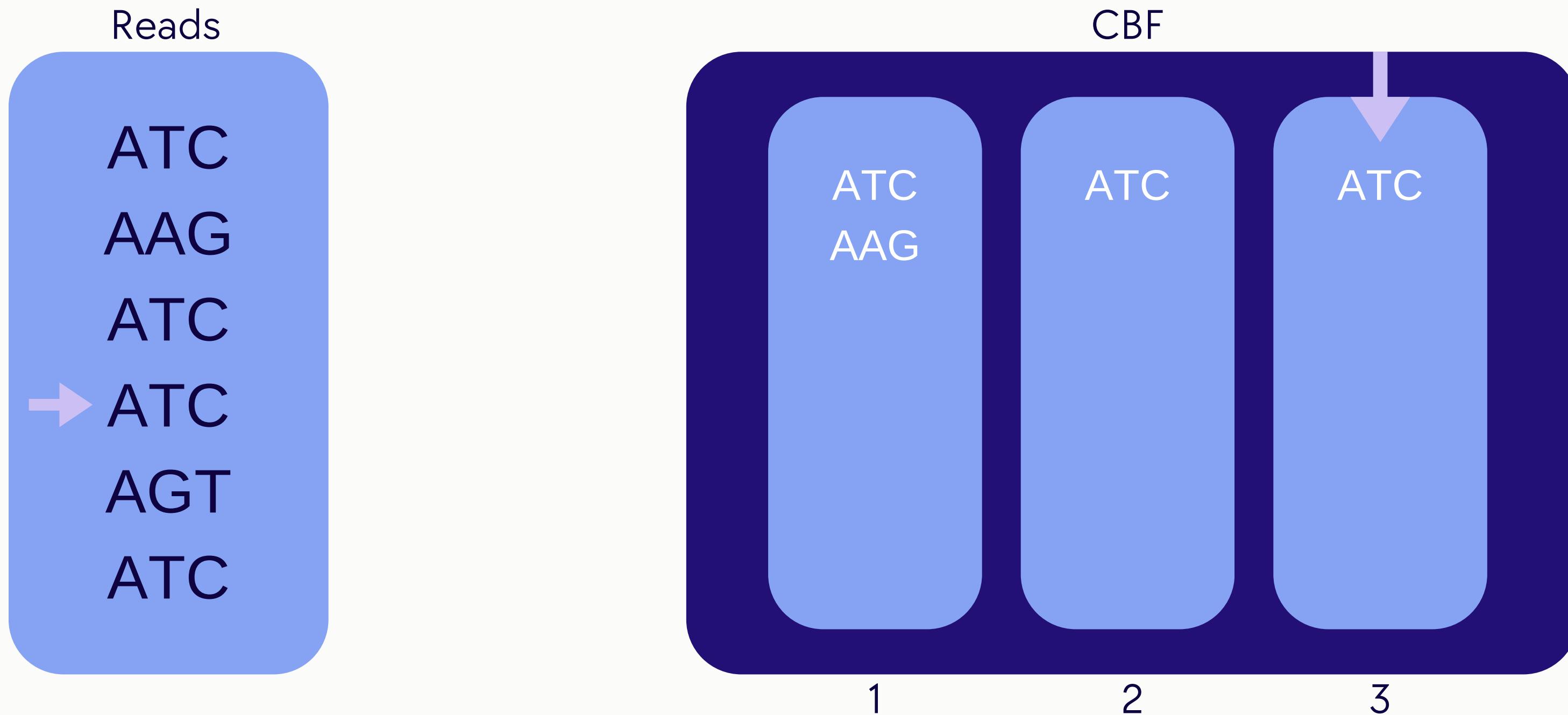
ESEMPIO



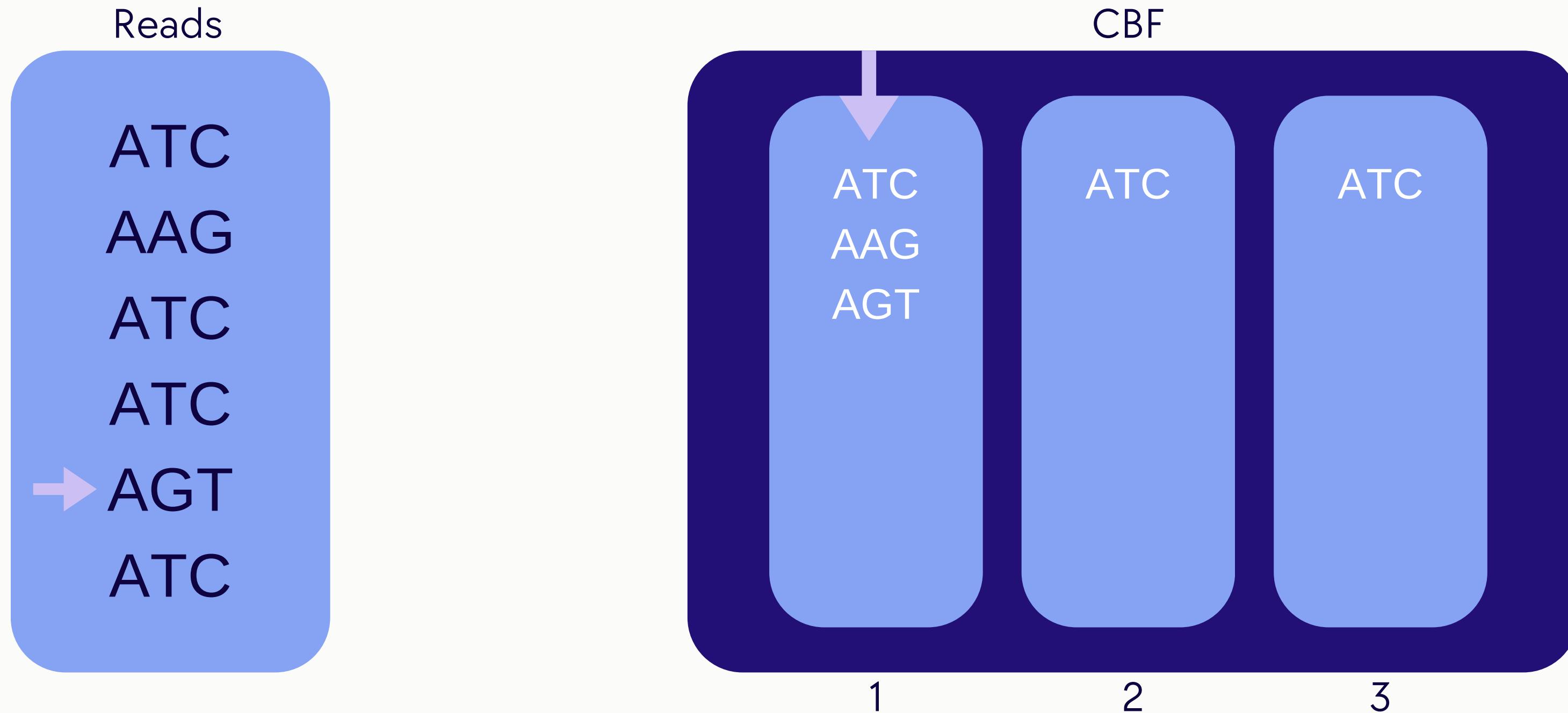
ESEMPIO



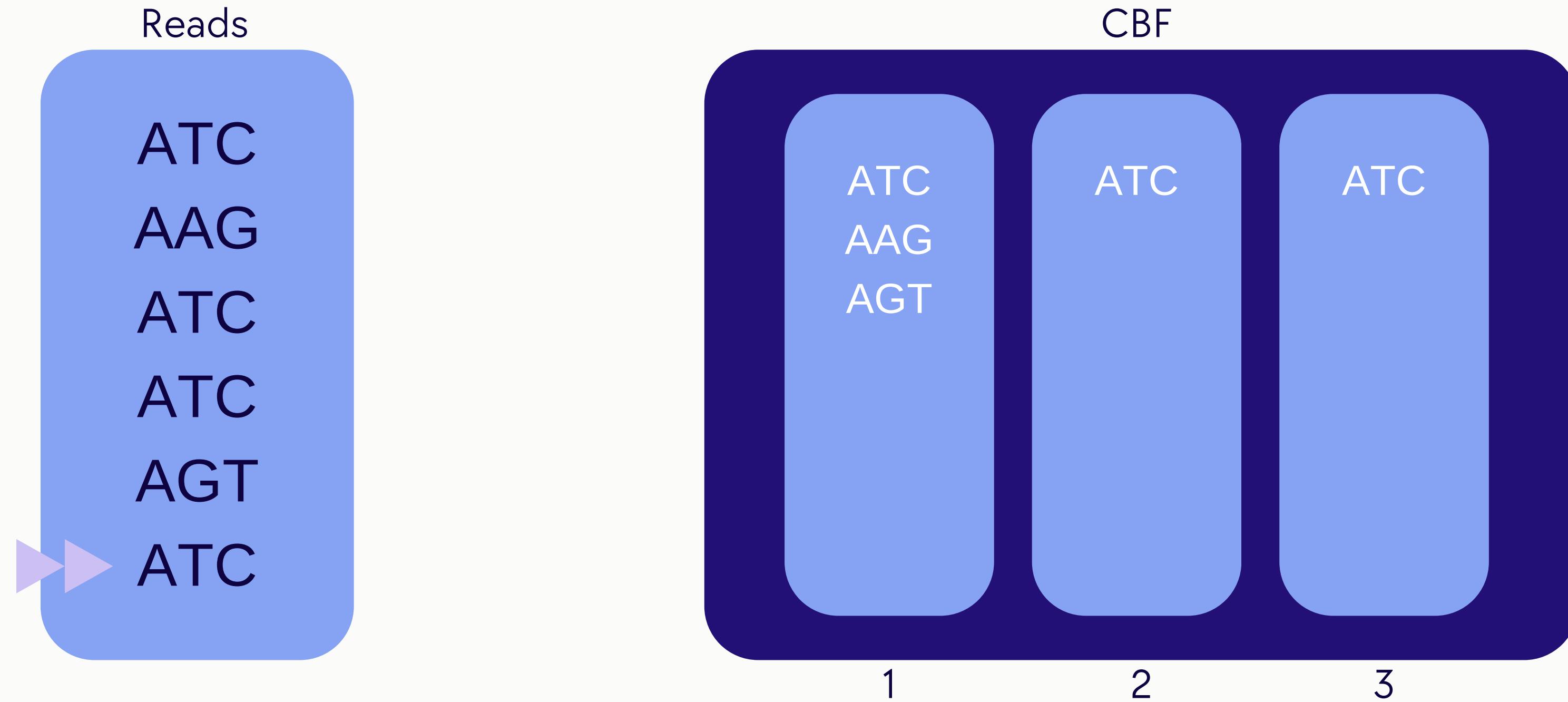
ESEMPIO



ESEMPIO



ESEMPIO



IL FUNZIONAMENTO A GRANDI LINEE

- Alla fine verranno mantenuti solo gli elementi nell'**ultimo filtro**
- Questo meccanismo consente di eliminare **gran parte dei falsi positivi**
- Bisogna scegliere bene il valore c di filtri contenuti nel CBF
- Si ottiene **sperimentalmente** in base a delle caratteristiche del genoma, tipicamente da 2 a 4
- I k-mer che superano il filtraggio vengono chiamati **solid k-mers**

IL FUNZIONAMENTO A GRANDI LINEE

- Alla fine verranno mantenuti solo gli elementi nell'**ultimo filtro**
- Questo meccanismo consente di eliminare **gran parte dei falsi positivi**
- Bisogna scegliere bene il valore c di filtri contenuti nel CBF
- Si ottiene **sperimentalmente** in base a delle caratteristiche del genoma, tipicamente da 2 a 4
- I k-mer che superano il filtraggio vengono chiamati **solid k-mers**



Sperimentalmente d è impostato a 3.

IL FUNZIONAMENTO A GRANDI LINEE

- Le read che contengono soltanto solid k-mers vengono chiamate **solid reads**
- A questo punto durante la **navigazione del grafo** queste solid reads vengono **estese**
- Si appende una delle 4 basi alla solid read, si considera l'ultimo k-mer e si **verifica la sua presenza nel filtro**
- Per **evitare ripetizioni** si tiene traccia dei k-mer già **inclusi** con un **altro BF**

LOOK AHEAD

- Sono presenti ancora alcuni **falsi positivi**, dovuti ad **errori di sequenziamento** e al **FPR** del filtro
- La maggior parte è stata rimossa dal **meccanismo dei CBF**
- La presenza dei falsi positivi porta alla creazione di **short branch**
- Più corti, non portano a nulla perché **originano da k-mer non realmente presenti nel genoma**

LOOK AHEAD

- Per mitigarli si usa un meccanismo di **look-ahead**
- A ogni branching point **si sbircia in avanti** per capire **quanto è lungo il ramo**
- Se è lungo meno di **k nodi**, viene **completamente ignorato** in fase di navigazione

NTHASH (JUSTIN CHU ET AL., 2016)

- Introdotta nel 2016, nel 2022 arriva alla seconda versione
- Una **funzione hash specializzata** per l'uso in campo **bioinformatico**
- In particolare, per essere **accoppiata a un Bloom Filter** o a una struttura dati che richieda **hash multipli**
- In grado di restituire l'hash anche delle sequenze **forward e reverse and complement**

Funzione ricorsiva

Supponendo di avere k-mer per un certo k, sequenza r di lunghezza $l > k$, è definita come segue:

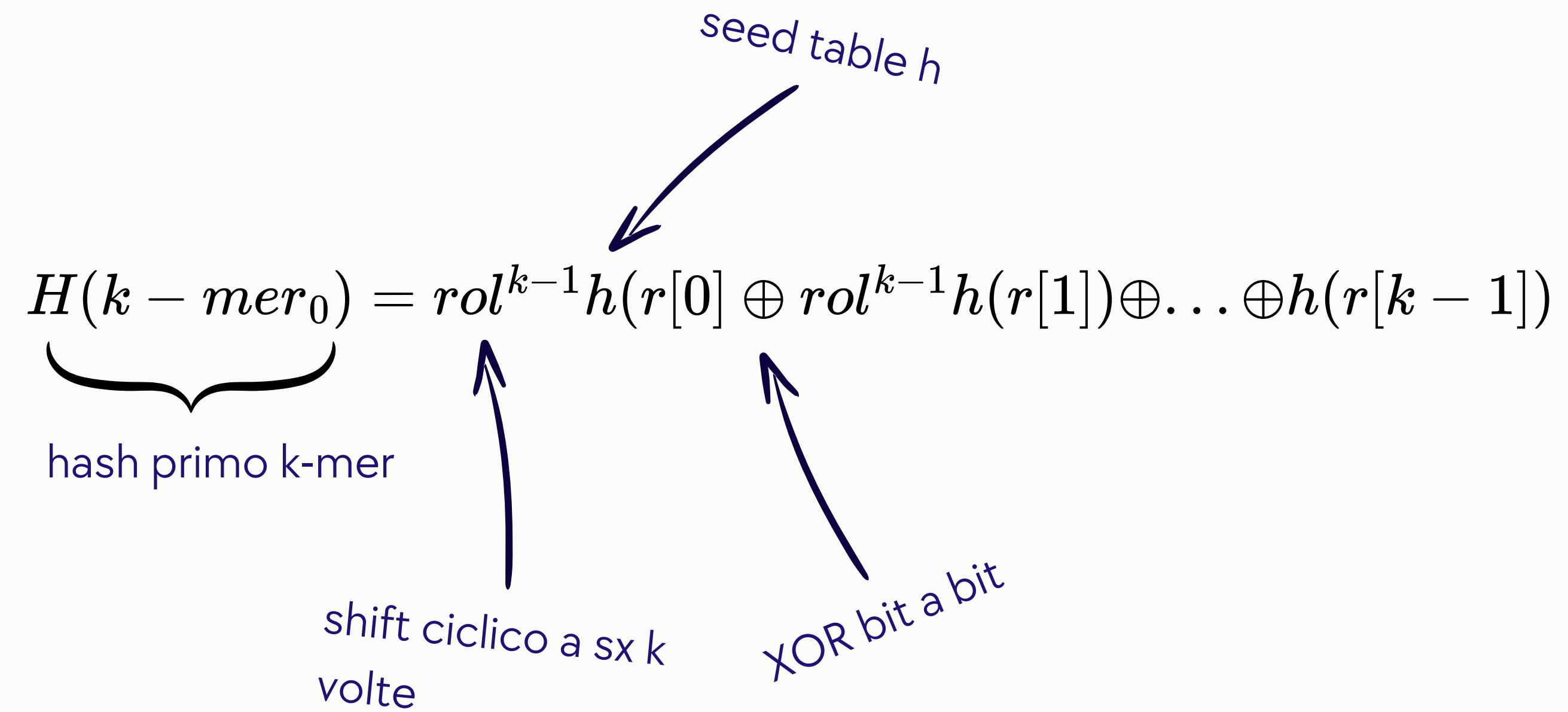
$$H(k-mer_i) = f(H(k-mer_{i-1}), r[i+k-1], r[i-1])$$

hash k-mer corrente hash k-mer precedente ultimo e primo carattere

NTHASH



La base della ricorsione è l'unica che **analizza un k-mer per intero**



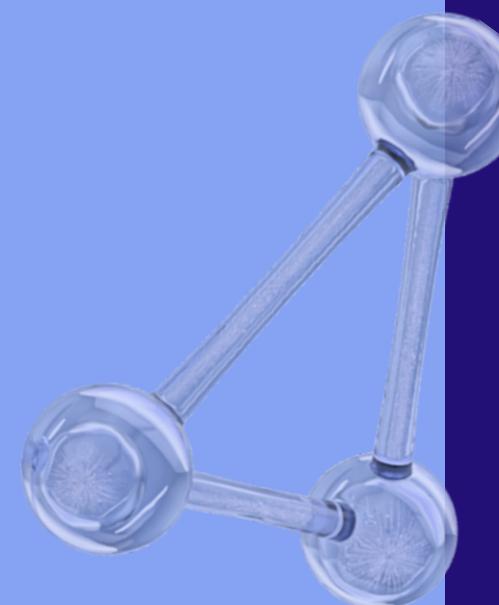


Ogni k-mer successivo viene calcolato a partire dal precedente usando la stessa combinazione di **rol** e **XOR**:

$$H(k-mer_i) = rol^{k-1}h(k-mer_{i-1}) \oplus rol^k H(r[i-1] \oplus h(r[i+k-1])$$



La seed table in particolare **associa a ogni base dell'alfabeto genetico** un valore numerico a 64 bit.



A	84
C	99
G	68
T	12

Un po' più grandi

NTHASH: COMPLESSITÀ TEMPORALE



La funzione ha una complessità temporale molto vantaggiosa

$$O(k + l)$$



Il primo step, ovvero l'hash del primo k-mer, **costa $O(k)$** assumendo costanti le operazioni di XOR bit a bit e accesso alla seed table.



Ogni step successivo viene effettuato in tempo costante, per al più **l** volte

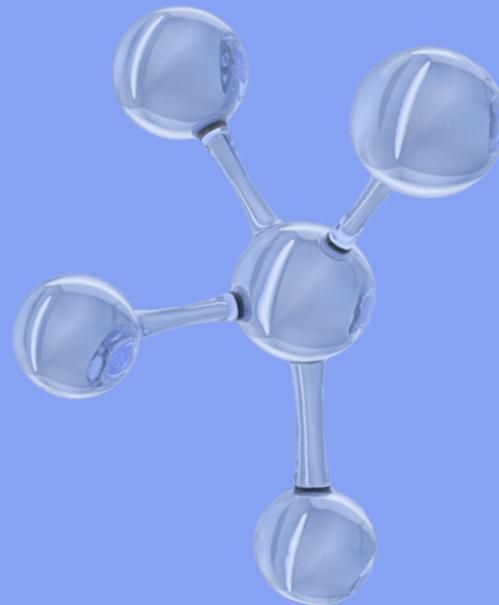


Le funzioni hash general purpose impiegano generalmente **$O(kl)$** per fare l'hash di **ogni k-mer di una sequenza**

NTHASH: L'IMPLEMENTAZIONE

- Scelto il linguaggio **C++**
- Poche ma **selettive particolarità del linguaggio** utilizzate
- Molte delle funzionalità implementabili anche in C
- Beneficia moltissimo della **gestione dinamica delle stringhe** di C++ per effettuare controlli sulla size a runtime
- Vantaggio principale è la **libreria standard di C++**, ricca di funzionalità che migliorano **l'organizzazione del codice** e la sua **portabilità**

NTHASH: L'IMPLEMENTAZIONE



- Il controllo dei limiti di rappresentazione effettuato con funzioni della libreria standard

```
static_assert(std::numeric_limits<uint64_t>::max() + 1 == 0,  
    "Integers don't overflow on this platform which is necessary for "  
    "ntHash canonical hash computation.");
```



- Funzionalità simili in C richiedono l'inclusione di altre librerie, in questo caso **limits.h**

NTHASH: L'IMPLEMENTAZIONE

- L'uso degli **oggetti** e dei **namespace** consente di differenziare diverse implementazioni della funzione
- Per esempio è disponibile una variante che non legge l'intero input, ma opera un pezzo alla volta chiamata `BlindNtHash`
- Utile per casi d'uso reali con input molto grandi
- Le varianti condividono molto codice, che risulta essere quindi organizzato meglio grazie a **all'ereditarietà** fornita dagli aspetti **object-oriented** di C++

NTHASH: L'IMPLEMENTAZIONE

```
BlindNtHash::BlindNtHash(const char* seq,
                           typedefs::NUM_HASHES_TYPE num_hashes,
                           typedefs::K_TYPE k,
                           ssize_t pos)
: seq(seq + pos, seq + pos + k)
, num_hashes(num_hashes)
, pos(pos)
, hash_arr(new uint64_t[num_hashes])
{
    if (k == 0) {
        raise_error("BlindNtHash", "k must be greater than 0");
    }
    fwd_hash = base_forward_hash(seq, k);
    rev_hash = base_reverse_hash(seq, k);
    extend_hashes(fwd_hash, rev_hash, k, num_hashes, hash_arr.get());
}
```

NTHASH2.0: (H.MOHAMADI ET AL. 2022)

- Nell'ultima versione sono stati aggiunti dei **trucchetti** per migliorare ulteriormente la performance, come **l'inlining della maggior parte delle funzioni**
- Per esempio l'operazione di **rol viene sostituita dalla srol**
- Lo shift ciclico a sx ripetuto 64 su parole a 64 bit **restituisce il valore di partenza**
- Ciò potrebbe **aumentare le collisioni** in alcuni casi
- L'approccio alternativo divide la parola in sottosequenze di lunghezze coprime con la cosiddetta **split rotation**

NTHASH: L'IMPLEMENTAZIONE

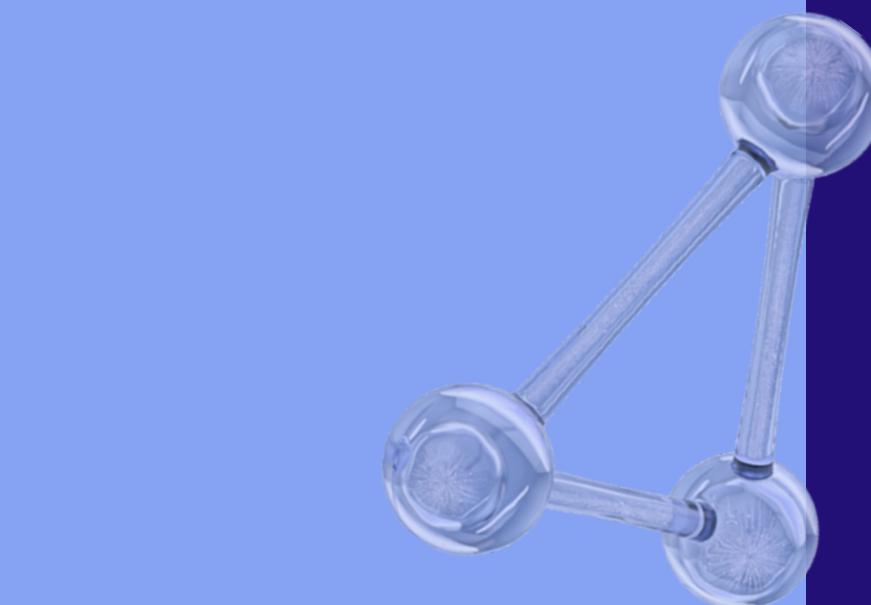
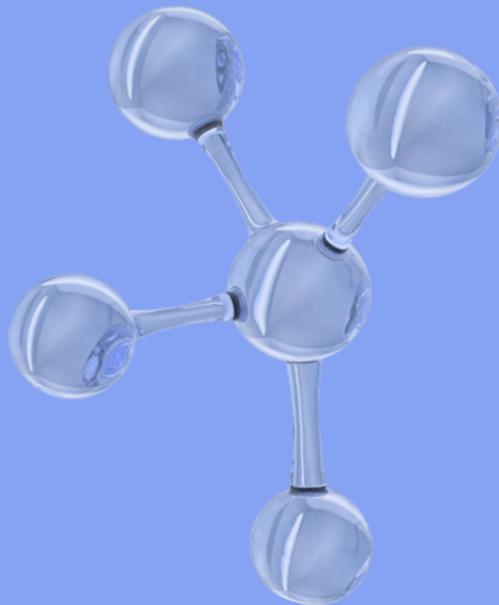
-  Nell'ultima versione sono stati aggiunti dei **trucchetti** per migliorare ulteriormente la performance
-  Per esempio l'operazione di **rol** viene sostituita dalla **srol**
-  Lo shift ciclico a sx ripetuto 64 su parole a 64 bit **restituisce il valore di partenza**
-  Ciò potrebbe **aumentare le collisioni** in alcuni casi
-  L'approccio alternativo usa la cosiddetta **split rotation**

NTHASH: SPLIT ROTATION

```
inline uint64_t srol(const uint64_t x) {
    uint64_t m = ((x & 0x8000000000000000ULL) >> 30) |
        ((x & 0x10000000ULL) >> 32);
    return ((x << 1) & 0xFFFFFFFFFFFFFFFFULL) | m;
}
```

- La split rotation divide una parola in **n sottoparole** di **lunghezza coprima**, le **ruota** e poi le **unisce**
- Nell'implementazione si sono scelte **due parole** di **lunghezza 31 e 33**
- Split in più parole restituiscono **periodicità più lunghe**

NTHASH: PRECALCOLI



Un'ulteriore ottimizzazione deriva dal fatto **che gran parte dei k-mer per k inferiori a 5 sono stati già precalcolati**



Sono infatti disponibili in delle **tabelle staticamente allocate** e utilizzate dalla funzione per **velocizzare il calcolo**

```
const uint64_t DIMER_TAB[4 * 4] = {  
    5015898201438948509U, 5225361804584821669U, 6423762225589857229U,  
    5783394398799547583U, 6894017875502584557U, 5959461383092338133U,  
    4833978511655400893U, 5364573296520205007U, 9002561594443973180U,  
    8212239310050454788U, 6941810030513055084U, 7579897184553533982U,  
    7935738758488558809U, 7149836515649299425U, 8257540373175577481U,  
    8935100007508790523U  
};  
const uint64_t TRIMER_TAB[4 * 4 * 4] = {  
    17  
    //Valori omessi per evitare confusione  
};  
const uint64_t TETRAMER_TAB[4 * 4 * 4 * 4] = {  
    6047278271377325800U, 6842100033257738704U, 5716751207778949560U, 5058261232784932554U,  
    5322212292231585944U, 4955210659836481440U, 6153481158060361672U, 6630136099103187130U,  
    7683058811908681801U, 7460089081761259377U, 8513615477720831769U, 9169618076073996395U,  
    .... // Valori omessi per evitare confusione, ma questa `e la tabella pi`u grande  
};
```

NTHASH: PRECALCOLI



Riscontrabili infatti negli step di **calcolo degli hash**, come per esempio in questo caso

```
for (unsigned i = 0; i < k - 3; i += 4) {  
    h_val = srol(h_val, 4);  
    uint8_t loc = 0;  
    loc += 64 * CONVERT_TAB[(unsigned char)seq[i]];  
    loc += 16 * CONVERT_TAB[(unsigned char)seq[i + 1]]; // NOLINT loc += 4 * CONVERT_TAB[(unsigned  
    char)seq[i + 2]];  
    loc += CONVERT_TAB[(unsigned char)seq[i + 3]];  
    h_val ^= TETRAMER_TAB[loc];  
}
```



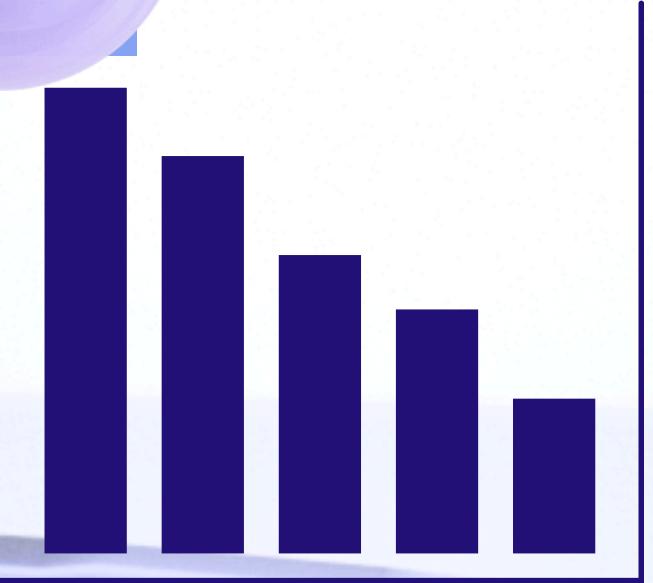
Viene ricondotto k a una combinazione di **2-mer, 3-mer e 4-mer**



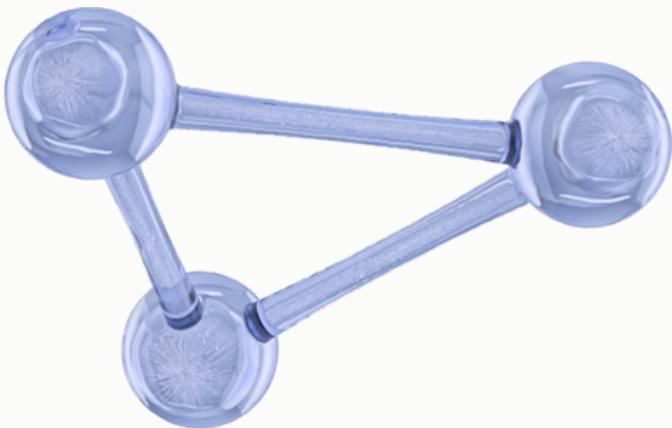
A questo punto l'hash del primo k-mer viene fatto **blocco per blocco**



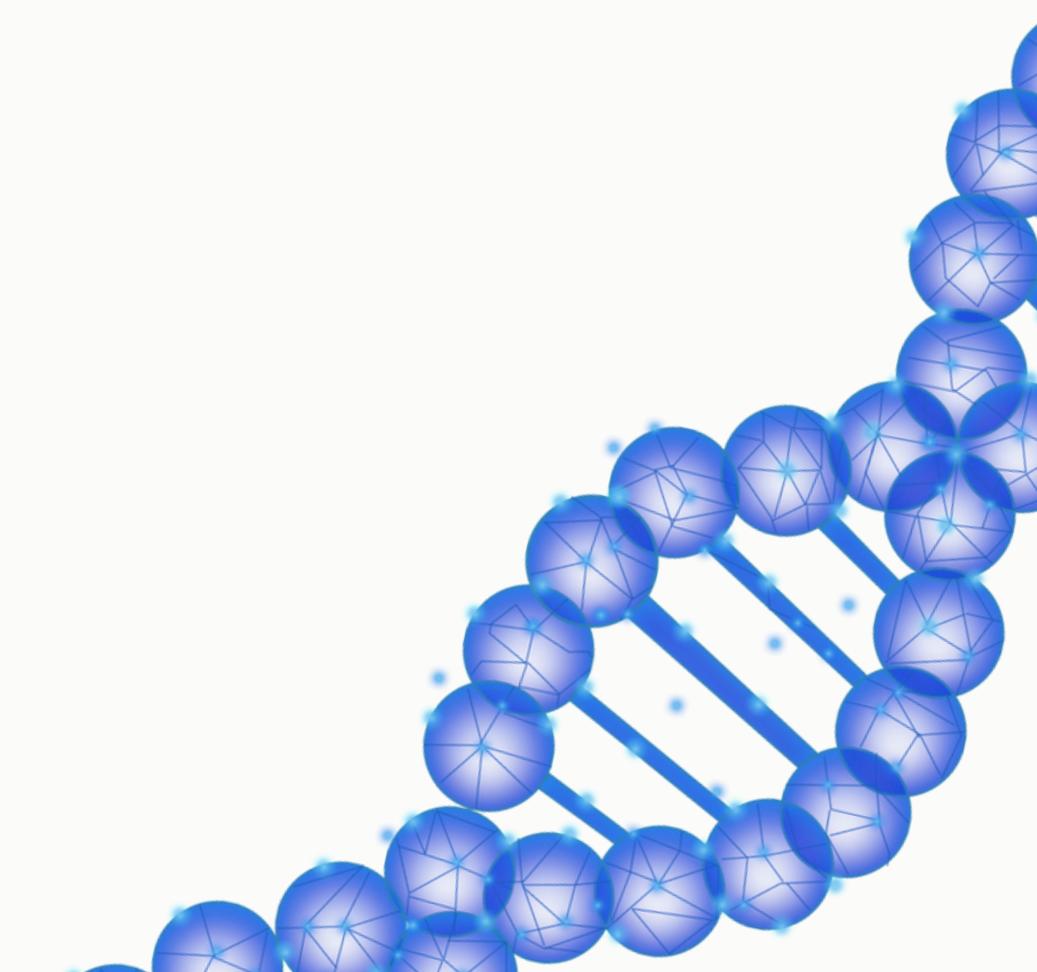
UN PO' DI TEST



KS TEST PER LA DISTRIBUZIONE CAMPIONARIA



- Effettuato il **test di Kolmogorov-Smirnov** sulla distribuzione dei valori hash generati dalle funzioni.
- Il test ha lo scopo di verificare la **somiglianza** tra due distribuzioni campionarie.
- Nel nostro caso volevamo dimostrare che i valori generati seguissero una **distribuzione uniforme**.
- Ciò avrebbe provato il corretto funzionamento delle funzioni per poter iniziare i test successivi.



KS TEST PER LA DISTRIBUZIONE CAMPIONARIA

- L'output significativo è un valore numerico denominato **p-value**.
- Esso se maggiore di 0.05 permette di non rigettare l'ipotesi iniziale. Nel nostro caso la distribuzione uniforme degli hash generati.
- Abbiamo comprovato che tutte e tre le funzioni generano valori hash con una **distribuzione uniforme**.
- Il **p-value** restituito dal test è maggiore di 0.05 per tutte e tre i test.

KS TEST PER LA DISTRIBUZIONE CAMPIONARIA

L'output significativo è un valore numerico denominato **p-value**.

Esso se maggiore di 0.05 permette di non rigettare l'ipotesi iniziale. Nel nostro caso la distribuzione uniforme degli hash generati.

Abbiamo comprovato che tutte e tre le funzioni generano valori hash con una **distribuzione uniforme**.

Il **p-value** restituito dal test è maggiore di 0.05 per tutte e tre i test.

Risultati del KS Test su NTHash2

```
data: dataNt$Values
D = 0.0056338, p-value = 0.9909
alternative hypothesis: two-sided
```

TEST DELLE PRESTAZIONI



Sono state testate le **prestazioni** di NTHash2 confrontandole con MurmurHash3 e CityHash.



L'esecuzione di MurmurHash3 e CityHash è stata adattata per funzionare sui K-mers come per NTHash2, **complessità temporale** sulla carta **maggiore**.



Imbastiti test con generazione di **1, 3 e 5 funzioni Hash** su un numero incrementale di sequenze.



I test sono stati effettuati su **sequenze sintetiche** di lunghezza fissata a **250 caratteri, con k = 50**.

- Test 1: 1 Hash per K-mer — Numero di sequenze: 500k, 1M, 2M, 4M, 8M
- Test 2: 3 Hash per K-mer — Numero di sequenze: 500k, 1M, 2M, 4M, 8M
- Test 3: 5 Hash per K-mer — Numero di sequenze: 500k, 1M, 2M, 4M, 8M

ADATTAMENTO ESECUZIONE

L'esecuzione di MurmurHash3 e CityHash è stata **adattata** per funzionare sui K-mers come per NTHash2.

MurmurHash3 e CityHash lavorano sulle **sequenze intere**.

NTHash2 genera e lavora **direttamente sui K-mers, come visto nelle slides precedenti**.

Per ogni sequenza sintetica **estraiamo tutti i k-mer**

Entrambe le funzioni genereranno l'**hash** dei k-mer

```
for(int i = 0; i < iterations; i++) {  
    // Hash all k-mers, num_hashes times for (int i = 0; i <= seqLen - k; ++i) {  
        for (int i = 0; i <= seqLen - k; ++i) {  
            // Get pointer to string buffer  
            std::string current_kmer = randomDNASequence.substr(i, k);  
            const void* keyPtr = static_cast<const void*>(current_kmer.c_str());  
            for(int j = 0; j < num_hashes; j++){  
                MurmurHash3_x64_128(keyPtr, k, 0, &hashValue);  
            }  
        }  
    }  
}
```

ADATTAMENTO ESECUZIONE

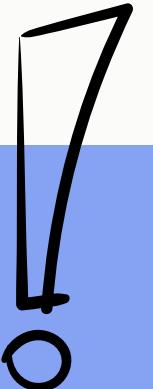
L'esecuzione di MurmurHash3 e CityHash è stata **adattata** per funzionare sui K-mers come per NTHash2.

MurmurHash3 e CityHash lavorano sulle **sequenze intere**.

NTHash2 genera e lavora **direttamente sui K-mers, come visto nelle slides precedenti**.

Per ogni sequenza sintetica **estraiamo tutti i k-mer**

Entrambe le funzioni genereranno l'**hash dei k-mer**



Il procedimento **impatta** sulle prestazioni ma era importante per effettuare test coerenti.

RISULTATI DEI TEST



Osserviamo i risultati del Test 3 con **5 Hash**.



La natura specifica di NTHash2 gli permette di ottenere **prestazioni temporali** migliori (espressi in microsecondi).

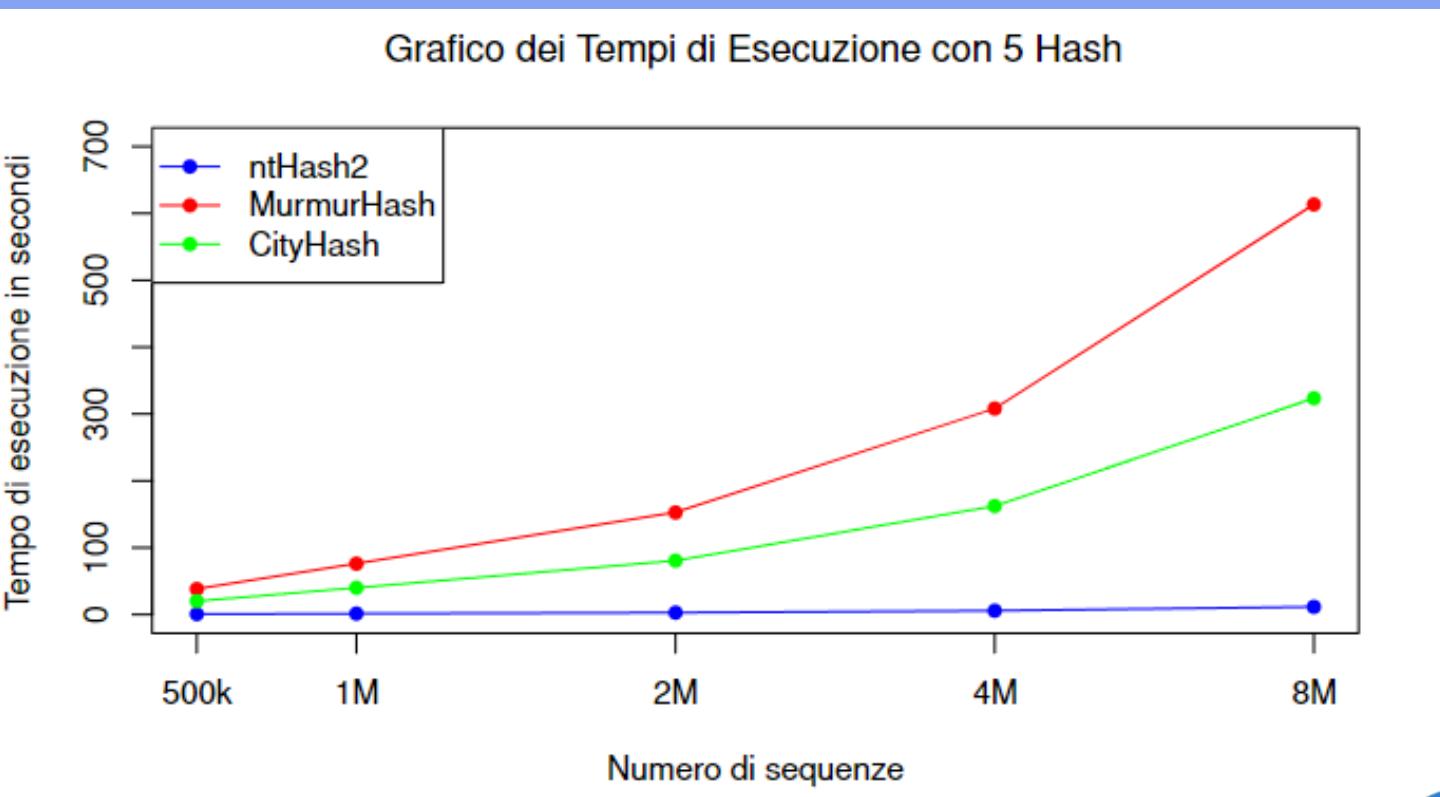


MurmurHash3 risulta la più lenta all'aumentare del numero di sequenze.



Ciò dimostra che con le giuste **ottimizzazione mirate** per uno specifico campo è possibile ottenere risultati incoraggianti.

Sequenze	ntHash2	Murmur3	CityHash128
500000	714879	38171580	20029548
1000000	1420439	76092219	40057794
2000000	2835241	152616776	80593656
4000000	5750207	308050725	162227078
8000000	11508723	613389670	323457974



BLOOM FILTER



Implementazione di un BF in C++ per **testare le performance di ntHash**



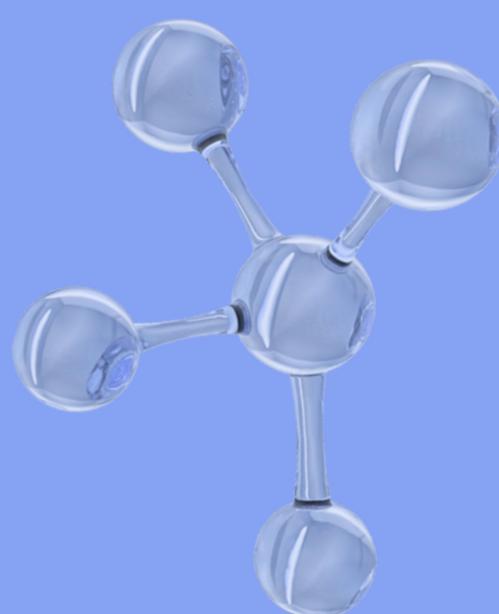
Semplice **filtro espandibile** senza operazione di **delete**



Siamo interessati a poter calcolare il **rate di falsi positivi** del filtro



Possibilità anche per **studio di altri parametri, varianti del filtro** e implementazioni di **assemblatori giocattolo** sul modello di Abyss



BLOOM FILTER



L'implementazione usa i template per poter definire genericamente cosa sia una funzione hash



Utile per dichiarare diverse varianti del filtro che usano le funzioni hash che abbiamo scelto per il confronto

```
template<typename T>
class HashFunction {
public:
    virtual size_t operator()(const T& data, size_t length) const = 0;
};
```

BLOOM FILTER

```
template<typename T>
class HashFunction {
public:
    virtual size_t operator()(const T& data, size_t length) const = 0;
};
```

- Si riscrive l'operatore di chiamata, rimanendo **generici sul tipo di dato in ingresso**
- C'è **differenza fra stringhe c++ e stringhe c**
- Ogni funzione hash specifica avrà poi una sua implementazione in cui si riscriverà l'**operatore ()** con la **logica della funzione**

BLOOM FILTER

```
template<typename T>
class NtHashFunction : public HashFunction<T> {
public:
    NtHashFunction(size_t hashes, size_t k) :
        num_hashes(hashes),
        k(k)
    {}
    size_t operator()(const T& data, size_t length) const override {
        nthash::NtHash nth(data, num_hashes, k);
        while(nth.roll()){}
        return nth.get_forward_hash();
    }
private:
    size_t num_hashes;
    size_t k;
};
```

BLOOM FILTER: IMPLEMENTAZIONE

```
template <typename HashFunc>
class Bloomfilter
{
public:

    void insert(const std::string& object)
    {
        size_t hash = hashFunc(object, object.size()) % bloomfilter_store.size();
        bloomfilter_store[hash] = true;
        ++object_count_;
    }

    void clear()
    {
        bloomfilter_store.clear();
        object_count_ = 0; // Reset the object count
    }

    bool contains(const std::string& object) const
    {
        size_t hash = hashFunc(object, object.size()) % bloomfilter_store.size();
        return bloomfilter_store[hash];
    }

    size_t object_count() const
    {
        return object_count_;
    }

    bool empty() const
    {
        return 0 == object_count();
    }

    size_t size() const
    {
        return bloomfilter_store.size();
    }

private:
    std::vector<bool> bloomfilter_store;
    size_t object_count_;
    HashFunc hashFunc;
};
```

BLOOM FILTER: IMPLEMENTAZIONE

```
private:  
    std::vector<bool> bloomfilter_store;  
    size_t object_count_;  
    HashFunc hashFunc;  
};
```



Tra le variabili private abbiamo un **vettore di booleani per segnalare le posizioni della bit signature da dover “attivare”**



Un contatore di oggetti **attualmente inseriti**



La funzione **hash generica** con cui **verrà istanziato**

BLOOM FILTER: IMPLEMENTAZIONE

```
void insert(const std::string& object)
{
    size_t hash = hashFunc(object, object.size()) % bloomfilter_store.size();
    bloomfilter_store[hash] = true;
    ++object_count_;
}
void clear()
{
    bloomfilter_store.clear();
    object_count_ = 0; // Reset the object count
}
bool contains(const std::string& object) const
{
    size_t hash = hashFunc(object, object.size()) % bloomfilter_store.size();
    return bloomfilter_store[hash];
}
size_t object_count() const
{
    return object_count_;
}
bool empty() const
{
    return 0 == object_count();
}
size_t size() const
{
    return bloomfilter_store.size();
}
```



I metodi della classe consentono di **inserire oggetti** e fare query per **vedere se sono presenti**



Tutto generico, **senza riferimenti esplicativi all'hash function utilizzata**

BLOOM FILTER: ISTANZIARE

```
Bloomfilter<CityHash>* cityFilter = new Bloomfilter<CityHash>(filter_size);
Bloomfilter<MurmurHash3>* murmurFilter = new Bloomfilter<MurmurHash3>(filter_size);
Bloomfilter<NtHashFunction>* ntFilter = new Bloomfilter<NtHashFunction>(filter_size);
```



Istanziare diventa quindi semplice, basta specificare la funzione hash che si vuole utilizzare

BLOOM FILTER: RISULTATI

Dimensione Filtro (KB)	CityHash	MurmurHash3	NtHash2
125	0.489858	0.490161	0.489921
250	0.428847	0.429241	0.429109
375	0.363858	0.364049	0.363989
500	0.311659	0.311690	0.311794
625	0.271105	0.271123	0.271084
750	0.239204	0.239201	0.239259
875	0.213800	0.213971	0.213873
1000	0.193125	0.193314	0.193229

I risultati mostrano come il **FPR del filtro decrementi molto velocemente, a 1 MB è già al 20%**

Abbiamo scelto **dimensioni di filtro molto piccole**, massimo poco meno di 1 MB, rispetto al paper

Altrimenti i FPR sarebbero stati **praticamente sempre pari a 0**

Un uso reale richiede dimensioni molto maggiori e quindi **prestazioni eccellenti**



- ntHash2 e il progetto Abyss sono **molto efficienti dal punto di vista prestazionale**
- C'è ancora molto da fare e da scoprire inerentemente al progetto, come **gli spaced seed hash**, oppure **ntCard**
- Il codice è disponibile sull'organization Github del corso e presenta **test automatizzati e replicabili**, con **istruzioni dettagliate e documentazione**
- L'implementazione del filtro e dei test **potrebbe essere riutilizzata a scopi didattici in futuro** o per lavorare **ad altri progetti che riguardano lo studio del tool**
- Per esempio espandendo il filtro con altre funzioni hash, **implementando delle varianti** oppure riusandolo per altri scopi



GRAZIE PER L'ATTENZIONE