Recherche de Motifs dans un génome

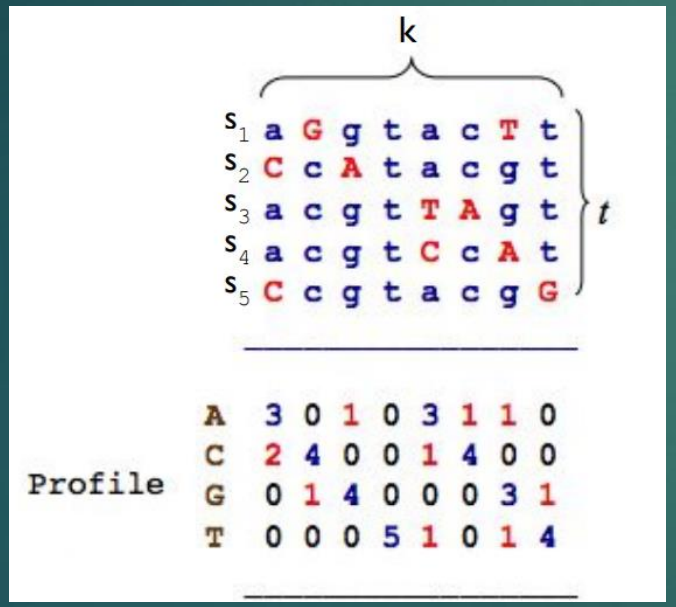
DEMEULENAERE Vivien
JEBARI Firas

Plan

- I - Détection de motifs
- II - Localisation de motifs
- III - Conclusions et résultats des recherches

I - Détection de motifs
A - Algorithmes déterministes

- Détection par force brute



- Calcul du profil d'un motif
- Calcul du score
- Récupération de la séquence consensus

I - Détection de motifs
A - Algorithmes déterministes

- Détection par force brute
  - Fiable
  - Peu efficace ($O(kn^t)$)

```
def bruteForceMotifSearch(sequences, t, n, k):
    bestscore = 0
    bestMotif=''
    for s in initialPosition ( k, n, t ):
        score_courant = score ( s, sequences )
        if score_courant > bestscore:
            bestscore = score_courant
            bestMotif = s
    return bestMotif, bestscore
```

I - Détection de motifs
A - Algorithmes déterministes

- Détection à l'aide de l'algorithme : « Median String »
  - Plus rapide
  - Irréalisable avec k trop grand ($O(4^k)$)

```
def MedianStringSearch(allkmers, sequences, t, n, k):
    bestDistance = 1000
    bestMotif = ""
    motifDist = {} #all motifs found
    for kmer in allkmers:
        dist = totalDistance ( kmer, sequences, k )
        if dist < bestDistance:
            bestDistance = dist
            bestMotif = kmer
            motifDist [ bestMotif ] = bestDistance

    return bestMotif, bestDistance, motifDist
```

I - Détection de motifs
B - Algorithmes aléatoires

- Détection à l'aide de l'algorithme : « Greedy profile motif search »

Nucléotide	0	1	2	3	4	5
A	4	7	3	0	1	0
C	1	0	4	5	3	0
T	1	1	0	0	2	7
G	2	0	1	3	2	1

Nucléotide	P(0)	P(1)	P(2)	P(3)	P(4)	P(5)
A	5/12	8/12	4/12	1/12	2/12	1/12
C	2/12	1/12	5/12	6/12	3/12	1/12
T	2/12	2/12	1/12	1/12	3/12	8/12
G	3/12	1/12	2/12	4/12	3/12	2/12

- Génération d'une matrice de probabilité à partir d'un profil.

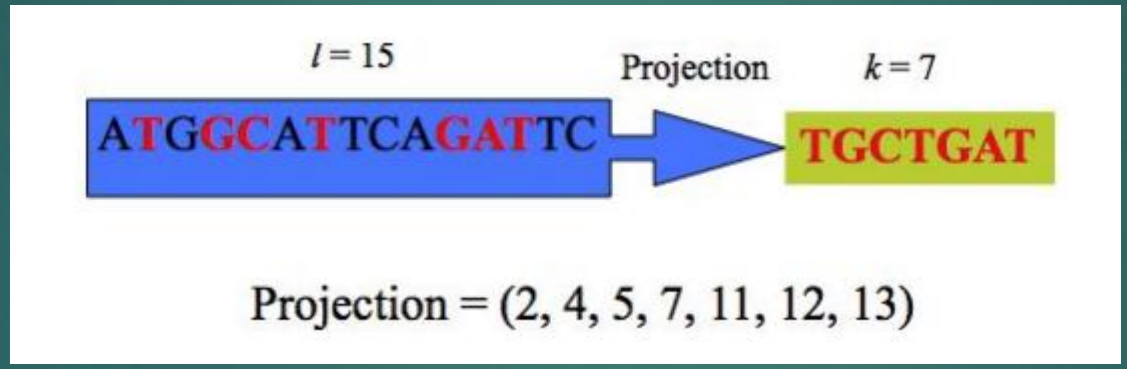
I - Détection de motifs
B - Algorithmes aléatoires

- Détection à l'aide de l'algorithme : « Median String »
  - Rapide
  - Peu fiable sur une itération

```
def GreedyProfileMotifSearch(sequences, t, n, k):
    positions = generateRandomS ( sequences, k )
    bestScore = 0
    nouv_profil = profile ( extractSeqs ( positions, sequences ), k, t )
    while getScore ( nouv_profil, k ) > bestScore:
        bestScore = getScore ( nouv_profil, k )
        profil = nouv_profil
        positions = getNewS ( PWM ( profil ) , k, sequences )
        nouv_profil = profile ( extractSeqs ( positions, sequences ), k, t )
        if changeProfile ( profil, nouv_profil ):
            break
    return positions, bestScore
```

I - Détection de motifs
B - Algorithmes aléatoires

- Détection à l'aide de l'algorithme : « Random projection »



- Génération d'une projection de taille k à partir d'une séquence

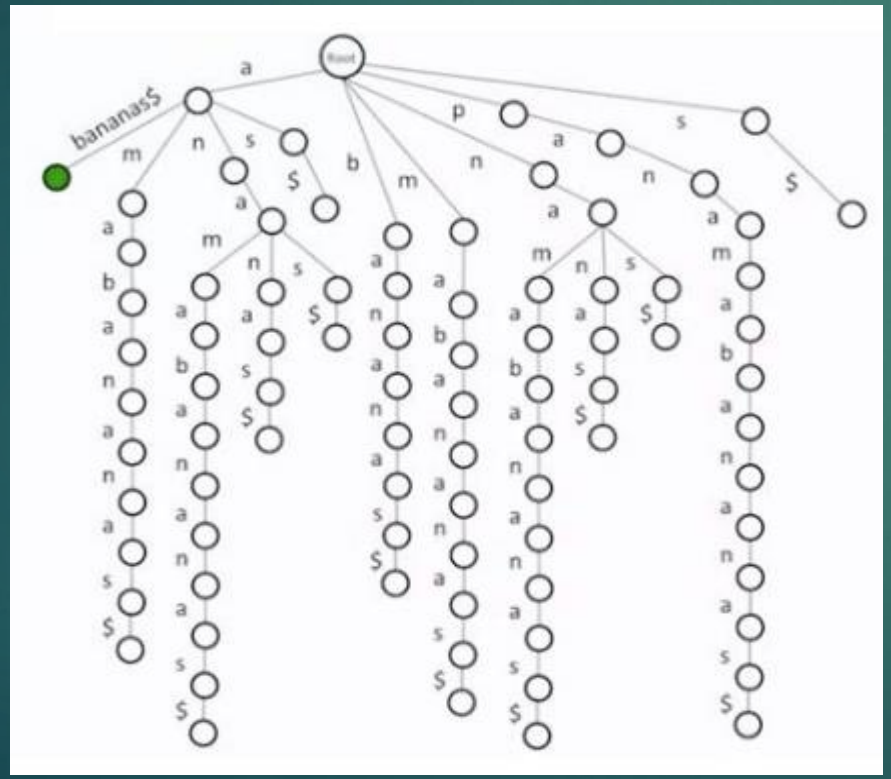
I - Détection de motifs
B - Algorithmes aléatoires

- Détection à l'aide de l'algorithme : « Random projection »
  - Rapide
  - Peu fiable sur une itération

```
def randomProjection(k, v, sequences):
    motifs  = {}; motifsSeq  = {}
    for seq in sequences:
        for i in range ( len ( seq ) - k ):
            motif = seq [ i : i + k ]
            key = generateKey ( getRandomFixePositions ( k, v ) , motif )
            motifsSeq [ key ] = motif
            if key in motifs:
                motifs [ key ] += 1
            else:
                motifs [ key ] = 1
    return motifs, motifsSeq
```

I - Détection de motifs
C - Algorithme à arbres

- Détection à l'aide de l'algorithme : « Inexact match »



- Evite les comparaisons inutiles
- Utilisation sur la séquence nucléotidique

I - Détection de motifs
C - Algorithme à arbres

- Détection à l'aide de l'algorithme : « Inexact match »
  - Rapide
  - Très coûteux en mémoire si l'arbre n'est pas compressé

```
def inexact_match(kmersV, sequences, stree, v):
    sequence = "".join ( sequences )
    nb_graines = v + 1
    taille_graine = ( int ) ( k / nb_graines )
    j = 0
    candidats = dict()
    for kmerV in kmersV:
        for i in range ( nb_graines ):
            depart = i * taille_graine
            graine = kmerV [ depart : depart + taille_graine ]
            indices_candidats = stree.find_all ( graine )
            for indice in indices_candidats:
                seq_candidate = sequence [ indice - depart : indice + k - depart ]
                if len ( seq_candidate ) == k and hamdist ( kmerV, seq_candidate ) <= v:
                    if kmerV in candidats:
                        variations = candidats [ kmerV ] [ 0 ] + [ seq_candidate ]
                        occurences = candidats [ kmerV ] [ 1 ] + 1
                        candidats [ kmerV ] = ( variations, occurences )
                    else:
                        candidats [ kmerV ] = ( [ seq_candidate ] , 1 )
    return dict ( sorted ( candidats.items(), key=lambda motif : -motif [ 1 ] [ 1 ] ) )
```

I - Détection de motifs
C - Algorithme à arbres

- Détection à l'aide de l'algorithme : « Inexact match »

Soit n la longueur de la séquence

- Complexité en calcule : $o(n^2)$
- Complexité en mémoire :
  - Sans compression : $o(\mathbf{n} * \frac{n-1}{2})$
  - Avec compression : o(n)

II - Localisation de motifs

- Algorithme de Boyer-Moore

Lettre	Valeur
T	8
E	6
A	2
M	3
S	1
*	8

- Décalage de 6 caractères selon la table

II - Localisation de motifs

- Algorithme de Boyer-Moore

```
def search(genome, motif, verbose = True ):
    badChar = badCharacter(motif)
    gs = goodSuffix(motif)
    return search_aux ( genome, motif, badChar, gs, 0, 0, verbose )

def search_aux ( genome, motif, badTable, gs, indice, comp_evitees, verbose ):
    if len ( motif ) > len ( genome ):
        return -1
    for i in range ( len ( motif ) - 1, -1, -1 ):
        if genome [ i ] not in badTable:
            return search_aux ( genome [ 1 :: ], motif, badTable, gs, indice + 1, comp_evitees, verbose )
        if genome [ i ] != motif [ i ]:
            decalage = badTable [ motif [ i ] ]
            for j in range ( min ( len ( motif ), len ( genome ) ) - 1, 0, -1 ):
                seq = genome [ i : j ]
                if seq in gs:
                    decalage = max ( decalage, gs [ seq ] )
            return search_aux ( genome [ decalage : ], motif, badTable, gs, indice + decalage, comp_evitees + decalage, verbose )
    if verbose:
        print ( "Comparaisons évitées :", comp_evitees )
    return indice
```

II - Localisation de motifs

- Algorithme utilisant l'indexation sur un mot de taille fixée

Mot	Indice
AA*	13
ACG	1
AGA	11
AGT	7
CGG	2
CTA	5
GAA	12
GCT	4
GGC	3
GTT	8
TAG	6, 10
TTA	9

- Découpage en mots de taille 3

II - Localisation de motifs

- Algorithme utilisant l'indexation sur un mot de taille fixée

```
def searchIndexTable(indexTable, k, motif, verbose = True):
    graines = seeds ( k, motif )
    indice = -1
    fin_motif = graines [ 0 ]
    comp_evitees = 0
    if fin_motif not in indexTable:
        return -1
    for candidat in indexTable [ fin_motif ]:
        indice = candidat
        comp_evitees = indice - comp_evitees
        for mot in graines [ 1 :: ]:
            indice = comparer_mots ( indexTable, mot, indice - k )
        if indice != -1:
            break
    if verbose:
        print ( "Comparaisons évitées :", comp_evitees )
    return indice
```

II - Localisation de motifs

- Algorithme utilisant une matrice de fréquences

- Modèle Nul :

$$P^{(0)}(b) = \frac{1}{L}\sum_{i=0}^{L-1}\omega_i(b)$$

- Log de vraisemblance

$$l(b_0, \dots, b_{L-1}) = \sum_{i=0}^{L-1} log_2(\frac{\omega_i(b_i)}{P^{(0)}(b_i)})$$

II - Localisation de motifs

- Algorithme utilisant une matrice de fréquences

```
def extract_indexes ( sequence, PWM, f_0, k ):
    res = []
    trouve = 0
    for i in range ( len ( sequence ) - k ):
        if trouve != 0:
            trouve -= 1
            continue
        seq = sequence [ i : i + k ]
        log=loglikehood(seq,PWM,f_0)
        if log > 0:
            res.append ( ( i, seq,log) )
            trouve = k
    return res
```

III - Conclusions et résultats des recherches

- Le motif CCAAT (hap4) revient régulièrement

- Difficultés à trouver la fonction du gène codant:
  - Motif court
  - Profil du motif introuvable