Clustering reads for DNA Data Storage Systems

Shubhransh Singhvi

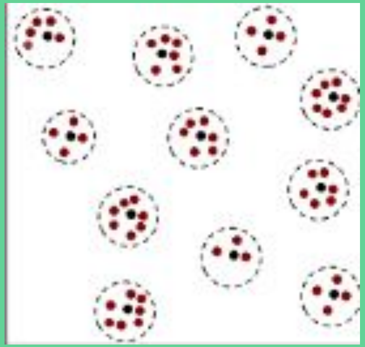
shubhranshsinghvi2001@gmail.com

Clustering billions of reads in subquadratic time complexity

DNA data storage requires a computationally intensive process to retrieve the data. In particular, a crucial step in the data retrieval pipeline involves clustering billions of strings with respect to edit distance.

Why the algo. works so efficiently for DNA storage data sets?

The algorithm converges efficiently on any dataset that satisfies certain separability properties, such as those coming from DNA storage systems. DNA storage has clusters that are well separated in Edit Distance.
The dataset is easy in the sense that the clusters are well separated. It is challenging in the sense that the size is in the magnitude of billions.



How does the algorithm work?

- At a high level, the algorithm iteratively merges clusters based on random representatives.
- Using a hashing scheme for edit distance, only a small subset of representatives are compared.
- Also a light-weight check based on a binary embedding is used to further filter pairs.
- If a pair of representatives passes these two tests, edit distance determines whether the clusters are merged.

Keywords

- Approximate Clustering:

  Trade a small loss in accuracy for a large reduction in running time

- Edit Distance:

  Metric for clustering (Computationally expensive)

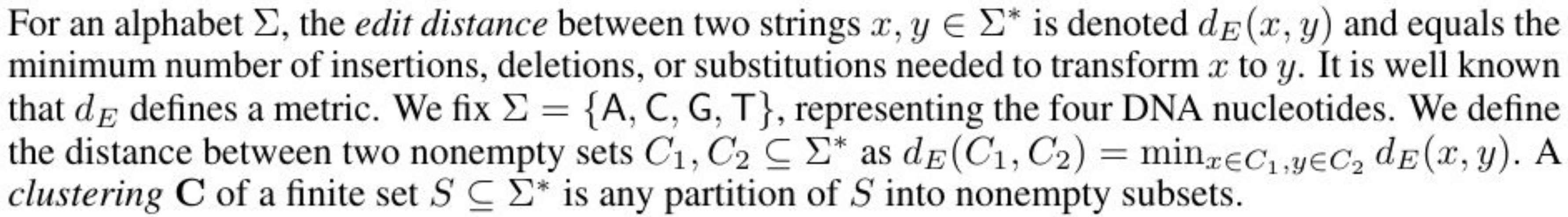
- Hashing:

  For filtering reads to avoid unnecessary computations

- q-gram distance:

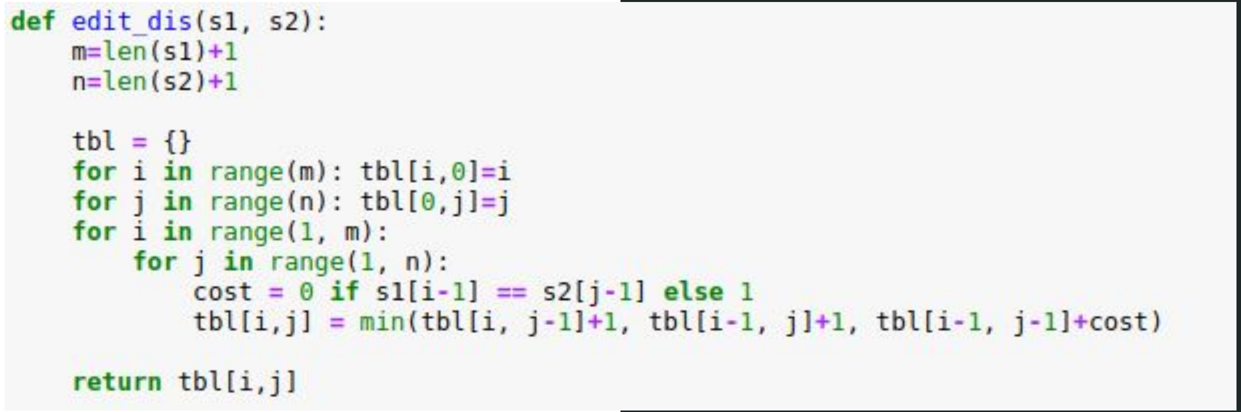
  Used to further filter reads (Computationally cheaper than edit distance)

Edit Distance and Clustering

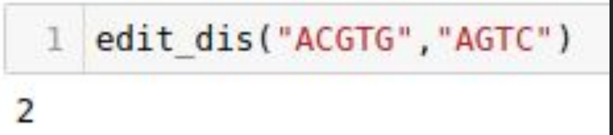


For an alphabet Σ, the *edit distance* between two strings $x, y \in \Sigma^*$ is denoted $d_E(x, y)$ and equals the minimum number of insertions, deletions, or substitutions needed to transform x to y. It is well known that d_E defines a metric. We fix $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$, representing the four DNA nucleotides. We define the distance between two nonempty sets $C_1, C_2 \subseteq \Sigma^*$ as $d_E(C_1, C_2) = \min_{x \in C_1, y \in C_2} d_E(x, y)$. A *clustering* $\mathbf{C}$ of a finite set $S \subseteq \Sigma^*$ is any partition of S into nonempty subsets.

Dynamic Programming for calculating Edit Distance between two strings

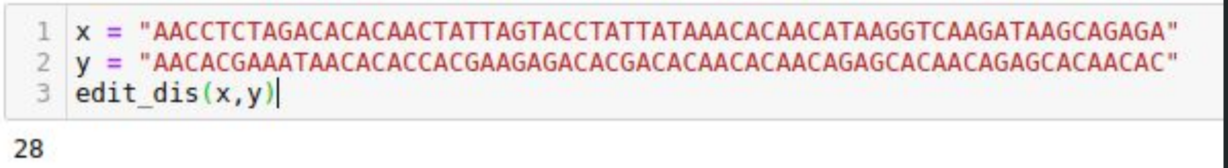


```python
def edit_dis(s1, s2):
    m=len(s1)+1
    n=len(s2)+1

    tbl = {}
    for i in range(m): tbl[i,0]=i
    for j in range(n): tbl[0,j]=j
    for i in range(1, m):
        for j in range(1, n):
            cost = 0 if s1[i-1] == s2[j-1] else 1
            tbl[i,j] = min(tbl[i, j-1]+1, tbl[i-1, j]+1, tbl[i-1, j-1]+cost)

    return tbl[i,j]
```



```python
edit_dis("ACGTG","AGTC")
```

```
2
```

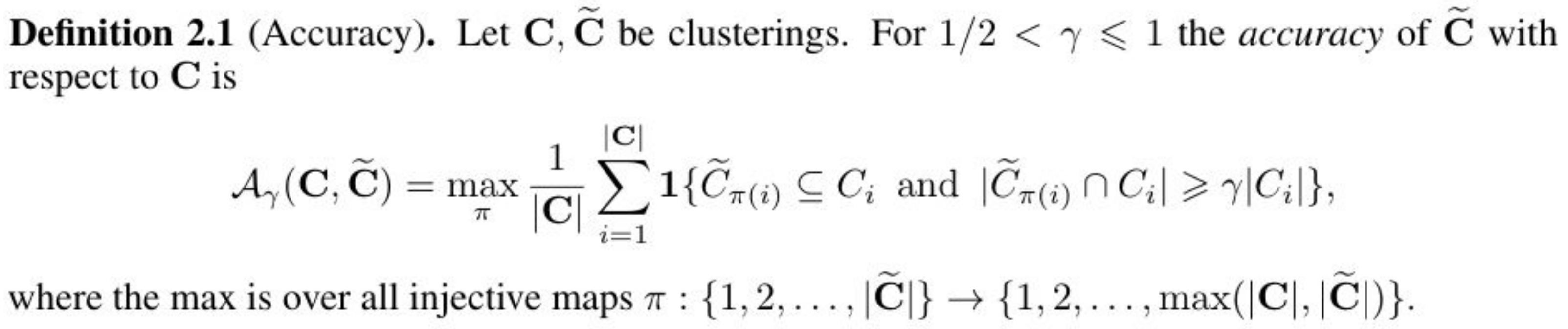


```python
x = "AACCTCTAGACACACAACTATTAGTACCTATTATAAACACAACATAAGGTCAAGATAAGCAGAGA"
y = "AACACGAAATAACACACCACGAAGAGACACGACACAACACAACAGAGCACAACAGAGCACAACAC"
edit_dis(x,y)
```

```
28
```

Accuracy

Accuracy is measured as the fraction of clusters with a majority of found members and no false positives.

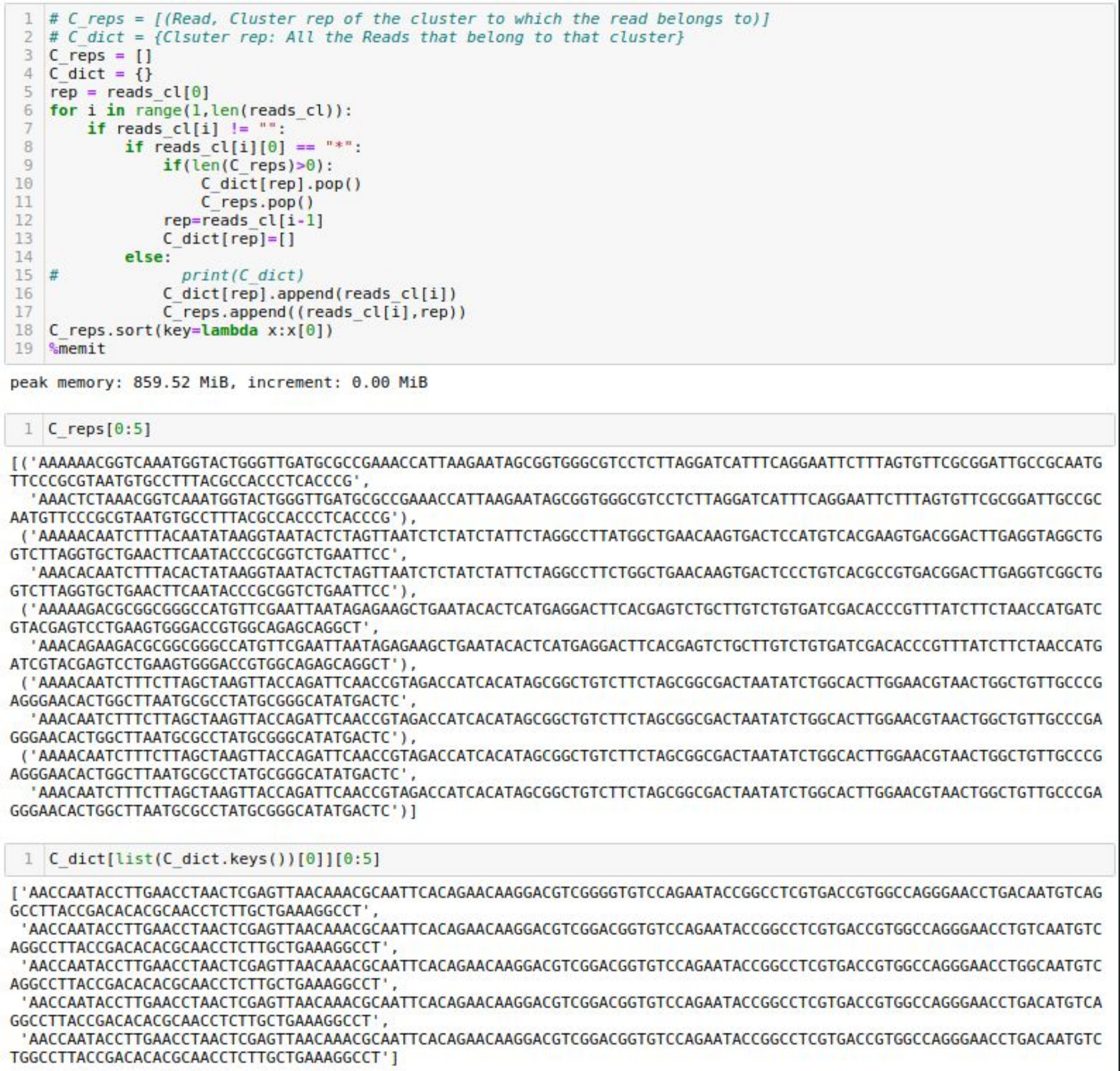


Definition 2.1 (Accuracy)**.** Let $\mathbf{C}, \widetilde{\mathbf{C}}$ be clusterings. For $1/2 < \gamma \leqslant 1$ the *accuracy* of $\widetilde{\mathbf{C}}$ with respect to $\mathbf{C}$ is

$$\mathcal{A}_\gamma(\mathbf{C}, \widetilde{\mathbf{C}}) = \max_\pi \frac{1}{|\mathbf{C}|} \sum_{i=1}^{|\mathbf{C}|} \mathbf{1}\{\widetilde{C}_{\pi(i)} \subseteq C_i \text{ and } |\widetilde{C}_{\pi(i)} \cap C_i| \geqslant \gamma |C_i|\},$$

where the max is over all injective maps $\pi : \{1, 2, \ldots, |\widetilde{\mathbf{C}}|\} \to \{1, 2, \ldots, \max(|\mathbf{C}|, |\widetilde{\mathbf{C}}|)\}$.

Experimentally, a small number of communication rounds achieve 98% accuracy on multiple real datasets, which suffices to retrieve the stored data.



```python
# C_reps = [(Read, Cluster rep of the cluster to which the read belongs to)]
# C_dict = {Clsuter rep: All the Reads that belong to that cluster}
C_reps = []
C_dict = {}
rep = reads_cl[0]
for i in range(1,len(reads_cl)):
    if reads_cl[i] != "":
        if reads_cl[i][0] == "*":
            if(len(C_reps)>0):
                C_dict[rep].pop()
                C_reps.pop()
            rep=reads_cl[i-1]
            C_dict[rep]=[]
        else:
#             print(C_dict)
            C_dict[rep].append(reads_cl[i])
            C_reps.append((reads_cl[i],rep))
C_reps.sort(key=lambda x:x[0])
%memit
```

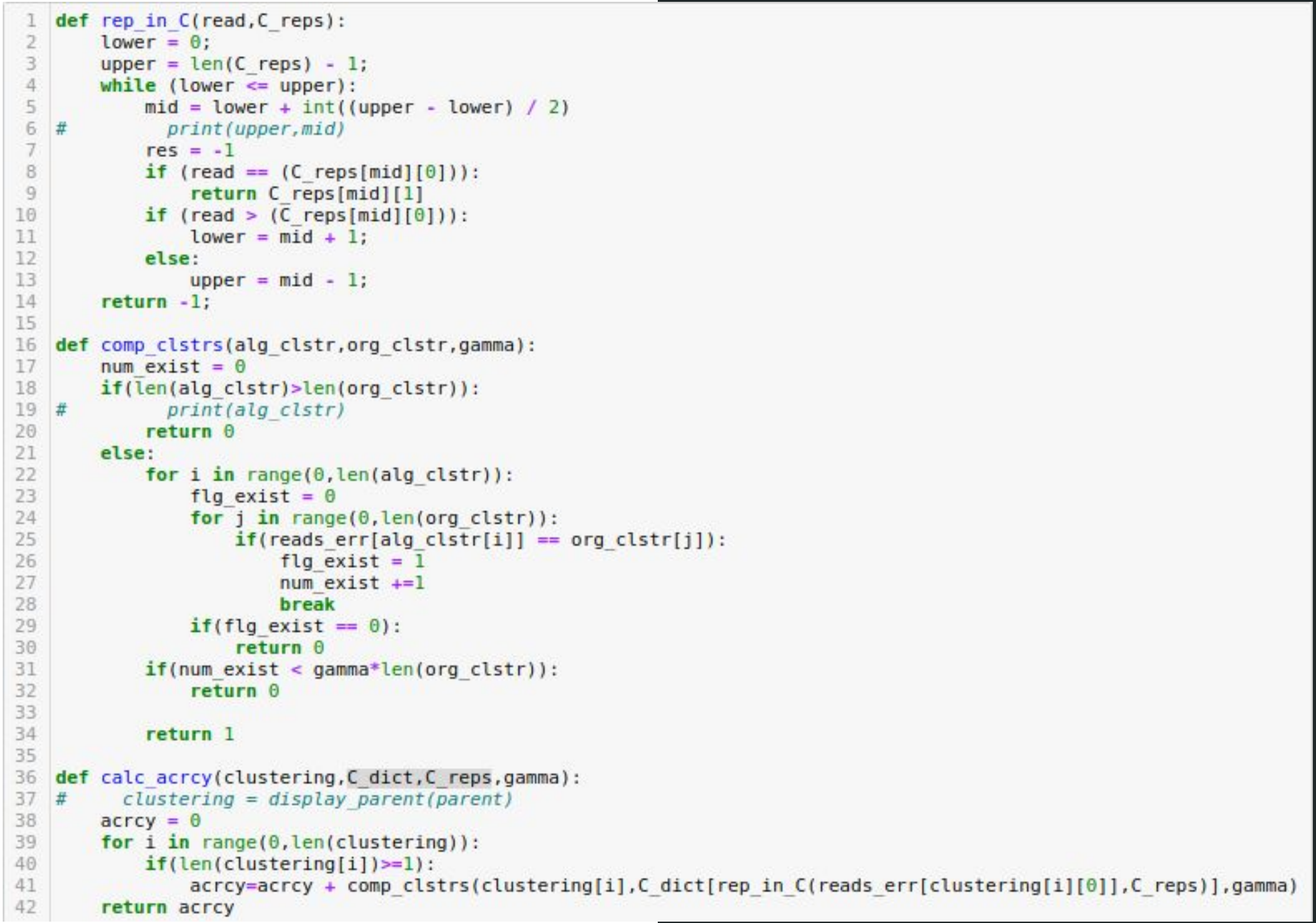
```
peak memory: 859.52 MiB, increment: 0.00 MiB
```

```python
C_reps[0:5]
```

```
[('AAAAAACGGTCAAATGGTACTGGGTTGATGCGCCGAAACCATTAAGAATAGCGGTGGGCGTCCTCTTAGGATCATTTCAGGAATTCTTTAGTGTTCGCGGATTGCCGCAATG
TTCCCGCGTAATGTGCCTTTACGCCACCCTCACCCG',
  'AAACTCTAAACGGTCAAATGGTACTGGGTTGATGCGCCGAAACCATTAAGAATAGCGGTGGGCGTCCTCTTAGGATCATTTCAGGAATTCTTTAGTGTTCGCGGATTGCCGC
AATGTTCCCGCGTAATGTGCCTTTACGCCACCCTCACCCG'),
 ('AAAAACAATCTTTACAATATAAGGTAATACTCTAGTTAATCTCTATCTATTCTAGGCCTTATGGCTGAACAAGTGACTCCATGTCACGAAGTGACGGACTTGAGGTAGGCTG
GTCTTAGGTGCTGAACTTCAATACCCGCGGTCTGAATTCC',
  'AAACACAATCTTTACACTATAAGGTAATACTCTAGTTAATCTCTATCTATTCTAGGCCTTCTGGCTGAACAAGTGACTCCCTGTCACGCCGTGACGGACTTGAGGTCGGCTG
GTCTTAGGTGCTGAACTTCAATACCCGCGGTCTGAATTCC'),
 ('AAAAAGACGCGGCGGGCCATGTTCGAATTAATAGAGAAGCTGAATACACTCATGAGGACTTCACGAGTCTGCTTGTCTGTGATCGACACCCGTTTATCTTCTAACCATGATC
GTACGAGTCCTGAAGTGGGACCGTGGCAGAGCAGGCT',
  'AAACAGAAGACGCGGCGGGCCATGTTCGAATTAATAGAGAAGCTGAATACACTCATGAGGACTTCACGAGTCTGCTTGTCTGTGATCGACACCCGTTTATCTTCTAACCATG
ATCGTACGAGTCCTGAAGTGGGACCGTGGCAGAGCAGGCT'),
 ('AAAACAATCTTTCTTAGCTAAGTTACCAGATTCAACCGTAGACCATCACATAGCGGCTGTCTTCTAGCGGCGACTAATATCTGGCACTTGGAACGTAACTGGCTGTTGCCCG
AGGGAACACTGGCTTAATGCGCCTATGCGGGCATATGACTC',
  'AAACAATCTTTCTTAGCTAAGTTACCAGATTCAACCGTAGACCATCACATAGCGGCTGTCTTCTAGCGGCGACTAATATCTGGCACTTGGAACGTAACTGGCTGTTGCCCGA
GGGAACACTGGCTTAATGCGCCTATGCGGGCATATGACTC'),
 ('AAAACAATCTTTCTTAGCTAAGTTACCAGATTCAACCGTAGACCATCACATAGCGGCTGTCTTCTAGCGGCGACTAATATCTGGCACTTGGAACGTAACTGGCTGTTGCCCG
AGGGAACACTGGCTTAATGCGCCTATGCGGGCATATGACTC',
  'AAACAATCTTTCTTAGCTAAGTTACCAGATTCAACCGTAGACCATCACATAGCGGCTGTCTTCTAGCGGCGACTAATATCTGGCACTTGGAACGTAACTGGCTGTTGCCCGA
GGGAACACTGGCTTAATGCGCCTATGCGGGCATATGACTC')]
```

```python
C_dict[list(C_dict.keys())[0]][0:5]
```

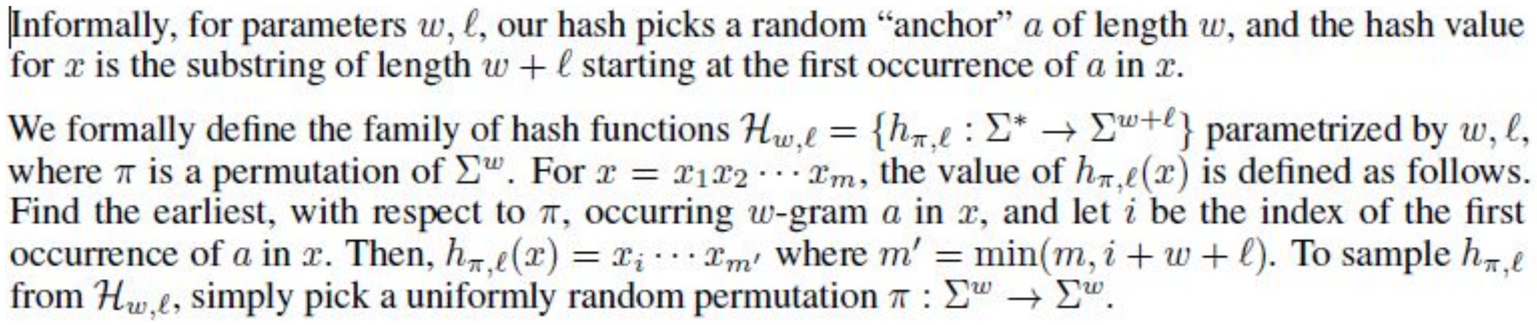
```
['AACCAATACCTTGAACCTAACTCGAGTTAACAAACGCAATTCACAGAACAAGGACGTCGGGGTGTCCAGAATACCGGCCTCGTGACCGTGGCCAGGGAACCTGACAATGTCAG
GCCTTACCGACACACGCAACCTCTTGCTGAAAGGCCT',
 'AACCAATACCTTGAACCTAACTCGAGTTAACAAACGCAATTCACAGAACAAGGACGTCGGACGGTGTCCAGAATACCGGCCTCGTGACCGTGGCCAGGGAACCTGTCAATGTC
AGGCCTTACCGACACACGCAACCTCTTGCTGAAAGGCCT',
 'AACCAATACCTTGAACCTAACTCGAGTTAACAAACGCAATTCACAGAACAAGGACGTCGGACGGTGTCCAGAATACCGGCCTCGTGACCGTGGCCAGGGAACCTGGCAATGTC
AGGCCTTACCGACACACGCAACCTCTTGCTGAAAGGCCT',
 'AACCAATACCTTGAACCTAACTCGAGTTAACAAACGCAATTCACAGAACAAGGACGTCGGACGGTGTCCAGAATACCGGCCTCGTGACCGTGGCCAGGGAACCTGACATGTCA
GGCCTTACCGACACACGCAACCTCTTGCTGAAAGGCCT',
 'AACCAATACCTTGAACCTAACTCGAGTTAACAAACGCAATTCACAGAACAAGGACGTCGGACGGTGTCCAGAATACCGGCCTCGTGACCGTGGCCAGGGAACCTGACAATGTC
TGGCCTTACCGACACACGCAACCTCTTGCTGAAAGGCCT']
```



```python
def rep_in_C(read,C_reps):
    lower = 0;
    upper = len(C_reps) - 1;
    while (lower <= upper):
        mid = lower + int((upper - lower) / 2)
#         print(upper,mid)
        res = -1
        if (read == (C_reps[mid][0])):
            return C_reps[mid][1]
        if (read > (C_reps[mid][0])):
            lower = mid + 1;
        else:
            upper = mid - 1;
    return -1;

def comp_clstrs(alg_clstr,org_clstr,gamma):
    num_exist = 0
    if(len(alg_clstr)>len(org_clstr)):
#         print(alg_clstr)
        return 0
    else:
        for i in range(0,len(alg_clstr)):
            flg_exist = 0
            for j in range(0,len(org_clstr)):
                if(reads_err[alg_clstr[i]] == org_clstr[j]):
                    flg_exist = 1
                    num_exist +=1
                    break
            if(flg_exist == 0):
                return 0
        if(num_exist < gamma*len(org_clstr)):
            return 0

        return 1

def calc_acrcy(clustering,C_dict,C_reps,gamma):
#     clustering = display_parent(parent)
    acrcy = 0
    for i in range(0,len(clustering)):
        if(len(clustering[i])>=1):
            acrcy=acrcy + comp_clstrs(clustering[i],C_dict[rep_in_C(reads_err[clustering[i][0]],C_reps)],gamma)
    return acrcy
```

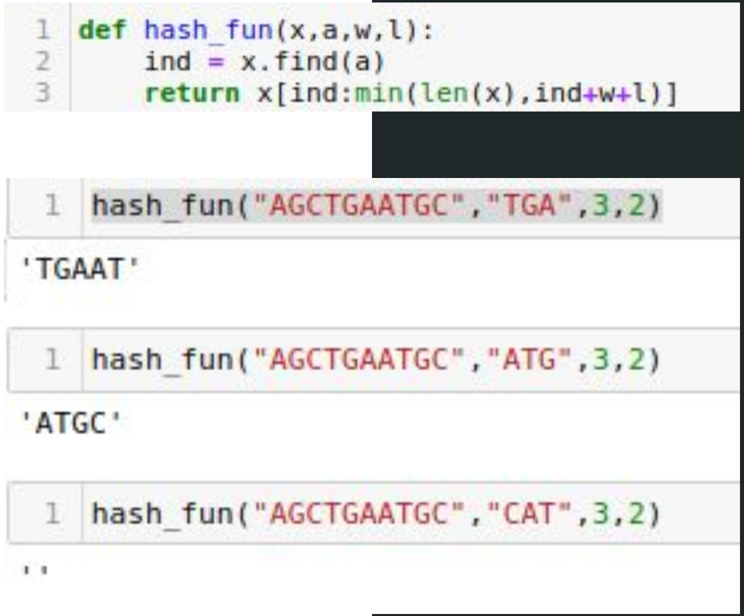
Hashing

At the core of the algorithm is a hash family that determines which pairs of representatives to compare.



Informally, for parameters w, ℓ, our hash picks a random "anchor" a of length w, and the hash value for x is the substring of length $w + \ell$ starting at the first occurrence of a in x.

We formally define the family of hash functions $\mathcal{H}_{w,\ell} = \{h_{\pi,\ell} : \Sigma^* \to \Sigma^{w+\ell}\}$ parametrized by w, ℓ, where π is a permutation of Σ^w. For $x = x_1 x_2 \cdots x_m$, the value of $h_{\pi,\ell}(x)$ is defined as follows. Find the earliest, with respect to π, occurring w-gram a in x, and let i be the index of the first occurrence of a in x. Then, $h_{\pi,\ell}(x) = x_i \cdots x_{m'}$ where $m' = \min(m, i + w + \ell)$. To sample $h_{\pi,\ell}$ from $\mathcal{H}_{w,\ell}$, simply pick a uniformly random permutation $\pi : \Sigma^w \to \Sigma^w$.



```python
def hash_fun(x,a,w,l):
    ind = x.find(a)
    return x[ind:min(len(x),ind+w+l)]
```

```python
hash_fun("AGCTGAATGC","TGA",3,2)
```

```
'TGAAT'
```

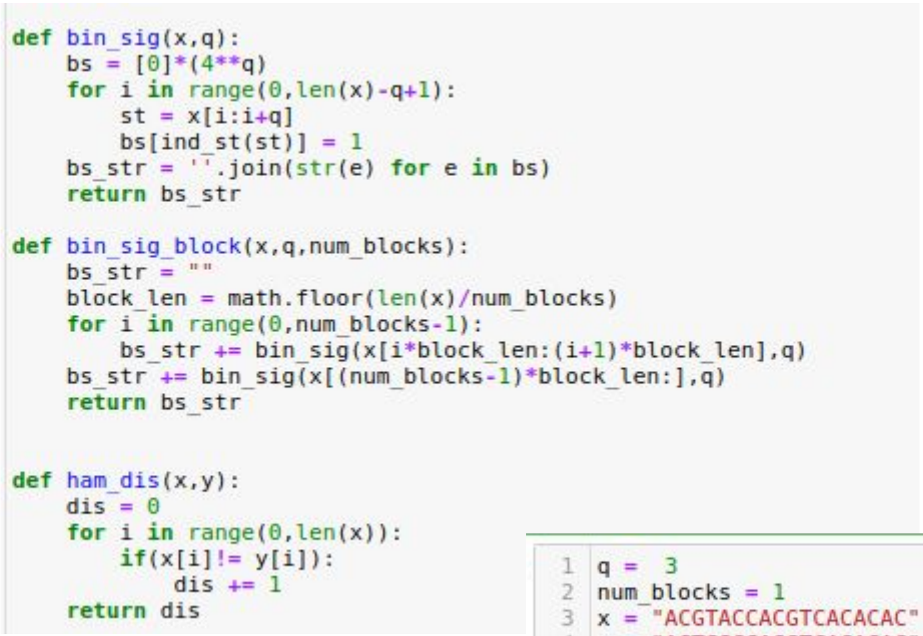
```python
hash_fun("AGCTGAATGC","ATG",3,2)
```

```
'ATGC'
```

```python
hash_fun("AGCTGAATGC","CAT",3,2)
```

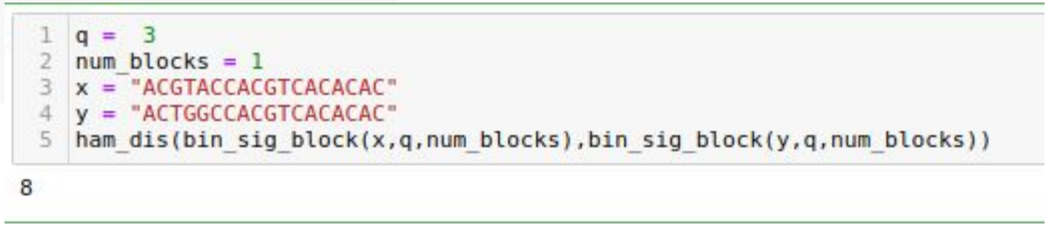
```
''
```

q-gram Distance



```python
def bin_sig(x,q):
    bs = [0]*(4**q)
    for i in range(0,len(x)-q+1):
        st = x[i:i+q]
        bs[ind_st(st)] = 1
    bs_str = ''.join(str(e) for e in bs)
    return bs_str

def bin_sig_block(x,q,num_blocks):
    bs_str = ""
    block_len = math.floor(len(x)/num_blocks)
    for i in range(0,num_blocks-1):
        bs_str += bin_sig(x[i*block_len:(i+1)*block_len],q)
    bs_str += bin_sig(x[(num_blocks-1)*block_len:],q)
    return bs_str


def ham_dis(x,y):
    dis = 0
    for i in range(0,len(x)):
        if(x[i]!= y[i]):
            dis += 1
    return dis
```

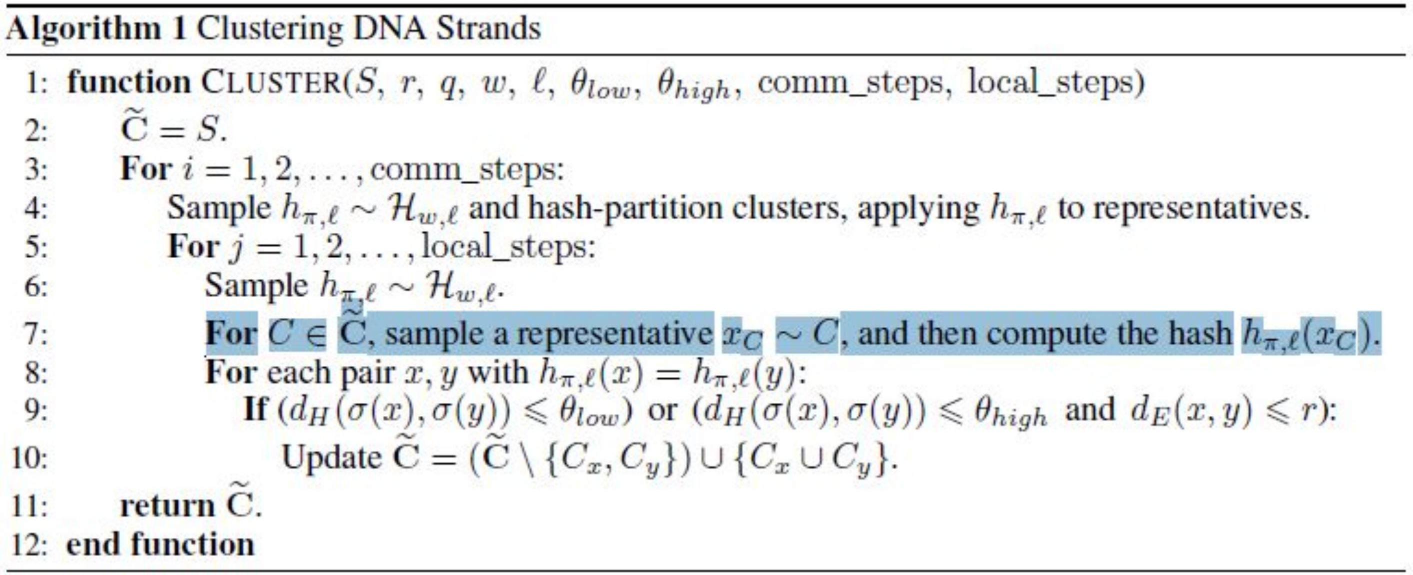
On top of hash based filtering, a cheap pre-check, based on the Hamming distance between binary signatures is used to avoid many edit distance comparisons.
The q-gram distance is an approximation for edit distance. A q-gram is simply a substring of length q, and the q-gram distance measures the number of different q-grams between two strings.



```python
q =  3
num_blocks = 1
x = "ACGTACCACGTCACACAC"
y = "ACTGGCCACGTCACACAC"
ham_dis(bin_sig_block(x,q,num_blocks),bin_sig_block(y,q,num_blocks))
```

```
8
```

Algorithm-Single Core



Algorithm 1 Clustering DNA Strands

```
function CLUSTER($S$, $r$, $q$, $w$, $\ell$, $\theta_{low}$, $\theta_{high}$, comm_steps, local_steps)
    $\widetilde{\mathrm{C}} = S$.
    For $i = 1, 2, \ldots,$ comm_steps:
        Sample $h_{\pi,\ell} \sim \mathcal{H}_{w,\ell}$ and hash-partition clusters, applying $h_{\pi,\ell}$ to representatives.
        For $j = 1, 2, \ldots,$ local_steps:
            Sample $h_{\pi,\ell} \sim \mathcal{H}_{w,\ell}$.
            For $C \in \widetilde{\mathrm{C}}$, sample a representative $x_C \sim C$, and then compute the hash $h_{\pi,\ell}(x_C)$.
            For each pair $x, y$ with $h_{\pi,\ell}(x) = h_{\pi,\ell}(y)$:
                If $(d_H(\sigma(x), \sigma(y)) \leqslant \theta_{low})$ or $(d_H(\sigma(x), \sigma(y)) \leqslant \theta_{high}$ and $d_E(x, y) \leqslant r)$:
                    Update $\widetilde{\mathrm{C}} = (\widetilde{\mathrm{C}} \setminus \{C_x, C_y\}) \cup \{C_x \cup C_y\}$.
    return $\widetilde{\mathrm{C}}$.
end function
```

5.1 Implementation and Parameter Details

For the edit distance threshold, we desire r to be just larger than the cluster diameter. With p noise, we expect the diameter to be at most $4pm$ with high probability. We conservatively estimate $p \approx 4\%$ for real data, and thus we set $r = 25$, since $4pm = 24$ for $p = 0.04$ and $m = 150$.

For the binary signatures, we observe that choosing larger q separates clusters better, but it also increases overhead, since $\sigma_q(x) \in \{0,1\}^{4^q}$ is very high-dimensional. To remedy this, we used a blocking approach. We partitioned x into blocks of 22 characters and computed σ_3 of each block, concatenating these 64-bit strings for the final signature. On synthetic data, we found that setting $\theta_{low} = 40$ and $\theta_{high} = 60$ leads to very reduced running time while sacrificing negligible accuracy.

For the hashing, we set w, ℓ to encourage collisions of close pairs and discourage collisions of far pairs. Following Theorem C.1, we set $w = \lceil \log_4(m) \rceil = 4$ and $\ell = 12$, so that $w + \ell = 16 = \log_4 n$ with $n = 2^{32}$. Since our clusters are very small, we find that we can further filter far pairs by concatenating two independent hashes to define a bucket based on this 64-bit value. Moreover, since we expect very few reads to have the same hash, instead of comparing all pairs in a hash bucket, we sort the reads based on hash value and only compare adjacent elements. For communication, we use only the first 20 bits of the hash value, and we uniformly distribute clusters based on this.

Finally, we conservatively set the number of iterations to 780 total (26 communication rounds, each with 30 local iterations) because this led to 99.9% accuracy on synthetic data (even with $\gamma = 1.0$).