General overview

1. Description of program
    a. Disk based approach
        i. Made use of the DSK algorithm
        ii. Dispatch to disk done by splitting kmers into multiple lists (folders). Further splitting done for each list into multiple sublists (txt files) where a set of kmers is stored in each sublist
        iii. Carry out kmer counting for each sublist using a memory-based approach

    b. Memory-based approach
        i. Made use of the counting bloom filter algorithm to carry out kmer counting

    c. Output
        i. Folder containing list and sublists used
        ii. Output txt file containing the kmers that occur at least q times and its respective count

    d. Memory used
        i. Uses more than 1M byte memory
        ii. Uses about $x * 10^8$ byte memory

Methods implemented

1. CountingBloomFilter class
    a. Variables initialised
        i. int m: the number of kmers in the file to be counted
        ii. int k: the number of hash functions needed
        iii. int[] CountBF: the hash table (in the form of an integer array)
        iv. int n: size of the hash table, CountBF

    b. Methods
        i. emptyBF(int m)
            1. Purpose: To create an empty counting bloom filter

            2. Algorithm
                - emptyBF() takes in 1 parameter m, which represents the number of kmers in the file
                - Initialise a false positive rate (ϵ)
                - Using the false positive rate and m, calculate
                $$n = -\frac{mln\epsilon}{(ln2)^2}$$
                - Using n and m, calculate
                $$k = \frac{n}{m}\ln 2$$
                - Initialise the hash table using n

            3. Complexity of algorithm
                a. Time complexity: $O(1)$
                    i. Constant time to calculate the variables, no matter the input m

                b. Space complexity: $O\left(-\frac{mln\epsilon}{(ln2)^2}\right) = O(n)$
                    i. Since the variables n, k are evaluated once, the space required is constant
                    ii. However, the size of CountBF depends on input m. As m increases, the size of CountBF increases, depending on the value of n calculated

ii. insert(int w, int[] CountBF)

1. Purpose: insert an element (in the form of an integer) into the counting bloom filter

2. Algorithm
   - insert() takes in 2 parameters, an integer w representing the element to be inserted and an integer array CountBF representing the hash table
   - To hash the element into the hash table, k hash functions are required
     - 1st hash function (hf0): $w\%n$
     - 2nd hash function (hf1): $w^2\%n$
     - Remaining (k-2) hash functions will take the form $(hf0 + ihf1)\%n$, from i=1 to i=k-2 [1]
   - Let v be the hash value obtained from each hash function
     - Increment the count at the v^{th} position of CountBF
       - i.e. $CountBF[v] \mathrel{+}= 1$

3. Complexity of algorithm
   a. Time complexity: $O\left(\frac{n}{m}\ln 2\right) = O(k)$
      i. For any element being inserted into the counting bloom filter, the time taken depends on the number of hash functions needed. Since k changes depending on the input m, the time required to insert an element increases as m increases.

   b. Space complexity: $O(n)$
      i. Similar to that of emptyBF()

iii. Query(int w, int[] CountBF)

1. Purpose: query the count of the element in the counting bloom filter

2. Algorithm
   - Query() takes in 2 parameters, an integer w representing the element to be inserted and an integer array CountBF representing the hash table
   - To find the positions in the hash table where the element would be hashed, use the same hash functions as in insert()
   - Initialise an integer variable, min_count to be the value of the 1st hash function (hf0) in CountBF
     - i.e. min_count = CountBF[hf0]
   - **if** 0 < CountBF[hf1] <=min_count
     - min_count = CountBF[hf1]

     end if
   - Since we have checked the value for the 1st and 2nd hash functions, initialise a counter to represent the number of hash functions that we have yet to check through (i.e. the remaining k-2 hash functions)
   - **While** (min_count>0 and counter<k-2)
     - for the remaining (k-2) hash functions, newHF
       - **if** (0<CountBF[newHF]<=min_count)
         - min_count =CountBF[newHF]
         - counter += 1
       - **else if** (CountBF[newHF] > 0 and CountBF[newHF]>min_count)
         - min_count remains the same
         - counter +=1
       - **else**
         - Element not present, min_count = 0
         - while loop breaks

       end if else

     end while
   - return min_count

3. Complexity of algorithm
   a. Time complexity: $O(k)$
      i. Similar to that of insert()

   b. Space complexity: $O(n)$
      i. Similar to that of emptyBF()

2. convertRead(String kmer)
    a. Purpose: convert a kmer into a unique value to be hashed into the counting bloom filter

    b. Algorithm
        i. Initialisation
            1. length as the length of the kmer
            2. Scores for each nucleotide base
                a. e.g. score_a = 0
            3. int_value_kmer = 0 as the unique value of the kmer
        ii. **for** i in range(0, length)
            1. Calculate the score of a nucleotide based on its score and its position in the kmer
            2. Add the score into int_value_kmer

            end for
        iii. return int_value_kmer

    c. Complexity of algorithm
        i. Let k be the length of the kmer
        ii. Time complexity: $O(k)$
            1. The time required to compute the unique value of a kmer will depend on the length of the kmer (based on the for loop). Hence, as k increases, the time required increases.

        iii. Space complexity: $O(1)$
            1. int_value_kmer takes constant space

3. canonicalForm(String str)
    a. Purpose: convert a kmer into its canonical form

    b. Algorithm
        i. Initialisation
            1. Length as the length of the kmer
            2. canonical_form as the canonical form of the input kmer
        ii. **if** str contains “n” or “N”
            1. Ignore kmer
        iii. **else**
            1. **for** loop to convert kmer into lowercase letters
                a. Store positive strand as the variable “kmer”
            end for
            2. **for** loop to get the reverse complement of kmer
                a. Store reverse complement as the variable “reverse”
            end for
            3. Find the canonical form
                a. Initialisation
                    i. score_kmer as the score of the positive strand kmer
                    ii. score_reverse as the score of the reverse complement
                b. **if** the kmer and reverse are the same
                    i. canonical form = kmer
                c. **else**
                    i. **for** i in range (0, length)
                        1. Get the ACSII score of the character at position i for kmer and reverse
                        2. Add the respective ACSII score to score_kmer and score_reverse
                        3. **if** (score_kmer<score_reverse)
                            a. kmer has a smaller value and is the canonical form
                            b. canonical_form = kmer
                            c. break for loop
                        4. **else if** (score_kmer>score_reverse)
                            a. reverse complement has a smaller value and is the canonical form
                            b. canonical_form = reverse
                            c. break for loop
                        5. else
                            a. Scores are the same (meaning that the nucleotide bases are the same so far)
                            b. continue
                        end if else
                    end for
                end if else
            end if else
        iv. return canonical_form

c. Complexity of algorithm
    i. Let k be the length of the kmer
    ii. Time complexity: $O(k)$
        1. for loop converting kmer into lower case take $O(k)$ time
        2. for loop to get the reverse complement takes $O(k)$ time
        3. for loop to calculate the kmer and reverse complement score takes $O(k)$ time
        4. total time $= O(k) + O(k) + O(k) = O(3k) = O(k)$

    iii. Space complexity: $O(k)$
        1. the string "kmer", "reverse" and "canonical form" depend on the length of the kmer. As k increases, the space required for each variable increases

References

1. Kirsch, A. and M. Mitzenmacher, *Less hashing, same performance: Building a better Bloom filter*. Random Structures & Algorithms, 2008. **33**(2): p. 187-218.