E0 259 - Assignment 5

Shankaradithyaa Venkateswaran
Sr no: 22190

October 30, 2024

1 Implementation Summary

For the code, I import the following libraries:

- numpy
- joblib
- typing

I use typing to typeset the functions to keep in line with best practices throughout the code.
Moving on I start to define a slew of functions for the assignment. The first few are meant to load the files into strings. These are:

1. readFasta : This takes in a file path as input and reads the reference string using the np.loadtxt function, which creates an array of strings.Then I skip the header and return a single string for the reference by using the join funciton.

2. readLastcol: This takes in a file path as input and reads the *chrX_last_col.txt* using the np.loadtxt function. Then I use the join function to return a single string for the last column.

3. getReads: This takes in a file path as input and reads the reads text file using np.loadtext and returns a numpy array of reads(strings).

4. getMapping: This takes in a file path as input and reads teh mapping text file using np.loadtxt and returns a numpy array of mappings(ints).

That's all for loading all the files, I call all these functions to assign their return values to global variables reference, lastCol, reads, and mapping respectively.
Moving on, I define one more function for preprocessing before the actual algorithm implementation. This function is the createMilestones functions. It takes in input of the lastCol of the Burrrows Wheeler Transform and creates a milestones array for the rank algorithm to use. The function iterates through the last column of the BWT and stores the count of the number of A, C, G, and T's at milestones. It stores the number of A, C, G, and T's above and including the milestone.

This function also creates a firstCol array, which is 4 numbers. This is because when we do select on the first column, we only need the ending point of each band of A, C, G, and Ts. So the function takes the total count of A, C, G, and Ts which are stores in the last element of the milestones arrays for each letter. Then because the first column is sorted in alphabetical order, I store the number of As first, then add the number of Cs, Gs, and T's respectively to get the other three elements. I zero index this list of 4 elements, therefore I subtract 1 from each element. This firstCol is made a numpy array and returned with the milestones arrays (which are also numpy arrays) to become global variable.
The milestone I chose was 1000, which can be changed as a global parameter, which would change the length of the milestones arrays.
Finally the last initializion for the global variables is setting the counts of matches for each exon to 0. Since we only care about the 2nd, 3rd, 4th, and 5th exons for red and green exons, I only initialize 4 counts for each.
Now I move to the actual code. Since the code is embarrasingly parralelizable, meaning the matching of one read does not affect the other, I use the joblib library and create a parallel function, which is a wrapper for my actual code. This function takes in the read index as input and then calls the actual code. I will explain my core algorithm in a little. Moving on, I call the Parallel function from the joblibs library and run the parallel wrapper for all the reads. This will update the exon counts as they are global variables. Finally I calculate the fractions using a simple function called colorBlindness, which takes in all the exon counts and returns the red exon fractions. I finally print this number to get the result
Returning to the parallel wrapper, where my actual code is taking place, I will explain top down, as that is the way I coded this assignment.
First, I make sure that the global variables are made global to the parallel function, i.e. their updates reflects in global scope.
I then call the forwardRead function which takes the read as it is and tries to match it in the reference up to 2 mismatches.
Next, I call the backwardRead function which takes the reverse complement of the read and tries to match it to the reference up to 2 mismatches.
Both these functions return a list of locations which match, this list is a numpy array, and is empty if nothing matches. Then I use the .any method to check if the lists are not empty. If they have at least one element, I call the ExonMapper function which checks if the locations are present in which of the exons.

I use ExonMapper for both the forwardRead location and the backwardRead locations and I do this for each read in the 3 million reads.
The ExonMapper function updates the global variables of the exon counts for the 2nd, 3rd, 4th, and 5th red and green exons. And as described above, after all this is done, I calculate the fractions and print the result.
Now let me explain the ExonMapper function before going into the forwardRead and backwardRead functions. The ExonMapper function takes in the locations outputted by the forwardRead and backwardRead functions. (There can be more than one place where the read is matching due to having a maximum of 2 mismatches).
I initialize the red and green variables are -1 for now. Then, for each location of the read, the ExonMapper function checks if the location is in the 2nd, 3rd, 4th, or 5th exon of the red and green exons. IT checks if the read is entirely present in a exon. It does not consider the cases where a read is present in two different red exons.
Since a read can have multiple locations, the funcion checks all of them and updates the global varibles according to the following logic. It only considers the case where the read is either just in a red exon, just in a green exon, in a matching pair of red and green exons, or not in any exons.
The rest of the helper functions
Now to get all the locations of a read I call the forwardRead and backwardReadr functions. forwardRead takes the read as is and tries to find its matches, while backwardRead takes the reverse complement of the read and tries to find its matches. I do both checks in each iteration as the read and the reverse complement of a read are treated as different reads for the pupose of matching locations.
Coming to the forwardRead function, I pass the following variables to the function:

```
reads[i], reference, firstCol, lastCol, mapping,
milestone, AMilestones, CMilestones, GMilestones, TMilestones
```

The reads[i] is the ith read and the rest of the variables are the global varibles I defined before.
The forwardRead function first calls the removeN function which changes all Ns present in the read to an A(As instructed by the assignment).
Then I call the splitReads function which takes in the read as an input and tries to split it into three equal parts using the floor divide function. I then return the three parts as a numpy array.

Finally to actually get all the locations, I call the BWTMapper function which takes in the following inputs:

```
splitReads, firstCol, lastCol, mapping,
milestone, AMilestones, CMilestones, GMilestones, TMilestones
```

The BWTMapper function checks each element in splitReads and tries to find an exact match in one of them. It iterates through the elements of splitReads and for each read, tries to match it to a row in the Burrrows Wheeler Transform. To do this, I use rank and select as defined in class.
The rank function takes the following inputs:

```
lastCol, start, end, character, milestone, milestones
```

It follows the same logic as described from the class. For a character, it finds the first and last character in between the start and end index in the lastCol. It uses the milestones array to skip to the closest start index, and then starts counting down to the first iteration of the character. For the last version of the character, it finds the closest milestone before the end index using floor division and then counts all the times the character appears till the end index.
The rank function returns these values as the ith charcter and the jth character which are then passed onto select. If rank cannot find the character within the start and end index, it returns -1.
The rank function returns this to firstLocation and lastLocation variables. If either of these is -1, then the function does not call select, instead, it breaks out of the loop for the element of splitReads to go to the next element.
The select function takes the following inputs:

```
location, character, firstCol
```

location is a little misleading. location is actualy the ith version of the character, i.e. the 75th A. The select function then finds the location of the 75th A in the firstCol. This is because the firstCol is sorted in alphabetical order and the firstCol is the ending point of each band of A, C, G, and Ts(which is zero indexed).
For a certain character, the select function starts at the end of the previous character and adds the location parameter to this to get the final location of the character.

The select function returns these locations.
These new locations are the new start and end indices and as we traverse the read, this band of indices get smaller and smaller. This happens until either there is a mismatch i.e. rank could not find a character, in which case we break; the start index becomes greater than the end index, in which case we break; or the read is completely matched, in which case we have a band of indices left. For this band, we go through for each element in the band and return store its actual location in the reference using the mapping global array that we had initialized and then along with which part of the read was exactly mapped, we return the array of elements along with the index of the element of the splitReads array.
The locations is a numpy array and this BWTMapper function returns the first instance where it finds an exact match in the elements of the splitReads array.
If there is no such element in splitReads that matches, the function returns an empty array along with -1 as a flag. After getting the locations and the index of the element in splitReads which cause the exact match I call the locationinReference fucntion, if the flag is not -1, meaning there is an exact match in the splitReads array.
The locationinReference function takes in the following inputs:

```
splitReads, locations, flag, reference
```

For each location in the list of possible lcoations, it reconstructs the read and its actual location in the reference using the splitReads array and the flag(which is the segment of the read which exactly matched). Then it checks the number of mismatches using the bruteForce function.
The bruteForce function takes in the following inputs:

```
read, readLocation, reference
```

It just checks each element of the read with each element of the refernce based on the location of the read and it counts the mismatches. It also takes care of the wraparouond if the read starts at the end of teh reference and ends at the start of the reference.
It returns the number of mismatches to the locationinReference function.
If the number of mismatches is less than 2, then the locationinReference function adds the actual read location to the list of true locations of the read. Once it does this for all possible locations for the read, it returns a numpy

array of true locations.
If none of the read locations are matches with less than or equal to 2 mismatches, then it returns an empty array.
Finally if the forwardRead function gets an empty array, it returns an empty array, else it returns the truelocations array as it is back to the parallel wrapper.
The backwardRead function does the exact same thing as the forwardRead fucntion, just that it calls an extra helper function called reverseComplement after removeN. This reverseComplement revereses the read and then changes all the characters to their complements.
The rest of the procedure is the exact same as the forwardRead function.
And that is the end of my implementation summary.

2 Results

The output which I got was:

```
redExonFraction2 = 0.45
redExonFraction3 = 0.3659090909090909
redExonFraction4 = 0.5806451612903226
redExonFraction5 = 0.4317434210526316
```

This matches the closest with configuration 1. Hence it is configuration 1 that is the main cause of color blindness.