Kovarik Bioinformatics Testing for NuProbe Scientist I

Introduction

This project is the Bioinformatics Testing for NuProbe Scientist I for Dennis Kovarik. This project was completed on 7/12/2021.

PCR is a procedure where scientists amplify target sequences to perform further study. One reason they my want amplification is if they wanted to align these sequences to determine their homology and/or similarity to each other. Software such as Bowtie2 will attempt to align amplicons, but at times it will fail to align some sequences. This non-specific amplification is bad because it can suppress the on-target amplification, which would increase non-uniformity and increase costs. Because of this, it is important to try to remove the primers that caused to amplification of these unaligned amplicons. Doing this would help produce better results from PCR amplification by promoting uniformity and decreasing costs.

This project attempts to do just that! Given the results from the Bowtie2 software and a list of primers, this software attempts to identify the primer pairs that led to the non-specific amplification. Some of the assumptions made in the process are as follows:

1. There is no adapter, meaning that the forward and reverse primers start and the very ends of the sequence.

2. Further, a complete match between the primers and sequence where required before being considered a match.

Now in reality, some mismatches between the primer and target sequence may be tolerated, but by enforcing a complete match between the primer pairs and the sequence, we do to things that are beneficial.

1. We limit the number of False Positive matches between the primers and thee sequence. The less mismatches we have, the less likely a primer will be falsely identified as binding to a sequence.
2. By enforcing a complete match, we can do a binary search for a match, which would save time.

By doing this, this software can help researchers identify primers that might be negatively affecting their PCR results.

Setup

This project was developed for the Ubuntu 20.04 OS.

Dependences

- Git
- Python 3.8
- pysam
- plotly
- NumPy
- openpyxl
- progress
- tqdm
- Pandas
- openyxl

Cloning This Repo with HTTPS

To download this repository on your device, you must clone this repo using either HTTPS or SSH. The easiest way to clone this repository on your local device is through HTTPS. If you SDK allows you to clone a repo through HTTPS, then do so. Otherwise, you can do it directly on the command prompt. To do so, open up the command prompt and move to the desired directory. Then simply run the following command and enter you credentials.

```
git clone https://github.com/denkovarik/Kovarik-BI-Testing-for-NuProbe-
Scientist-I.git
```

After the repo has been cloned on your device, move into the Kovarik-Technical-Interview directory from the command line.

```
cd Kovarik-BI-Testing-for-NuProbe-Scientist-I
```

Cloning This Repo with SSH

You can also clone this repo using SSH. Follow the instructions below to clone the repo using SSH. Please note that if you have already cloned the repo using HTTPS, then you can skip to the 'Install Dependencies' step.

Generate an SSH Key Pair

In order to clone this repository, you need to add your public SSH key to this repo. If you don't have one, then you would need to generate one. How to Generate SSH key in Windows 10? Easy Methods!! should help you generate an SSH key pair.

Add Your Public SSH Key to GitHub

Once you have an SSH Key Pair generated, you need to add your public SSH key to GitHub. Follow How to view your SSH keys in Linux, macOS, and Windows to access you public key. Then follow Adding a new SSH key to your GitHub account to add your public SSH key to GitHub.

Clone the Repository

If your SDK allows for it, then clone this repository through you SDK. Otherwise, open up the command prompt, move the the directory of your choice, then run the following command.

```
git git@github.com:denkovarik/Kovarik-BI-Testing-for-NuProbe-Scientist-I.git
```

After the repo has been cloned on your device, move into the Bioinformatics-Tools directory from the command line.

```
cd Kovarik-BI-Testing-for-NuProbe-Scientist-I
```

Install Dependencies

```
./setup/setup.sh
```

Usage

This program can be run without any command line arguemtns. As a result the program will run on the example files.

```
python find_unaligned_amplicon_primers.py
```

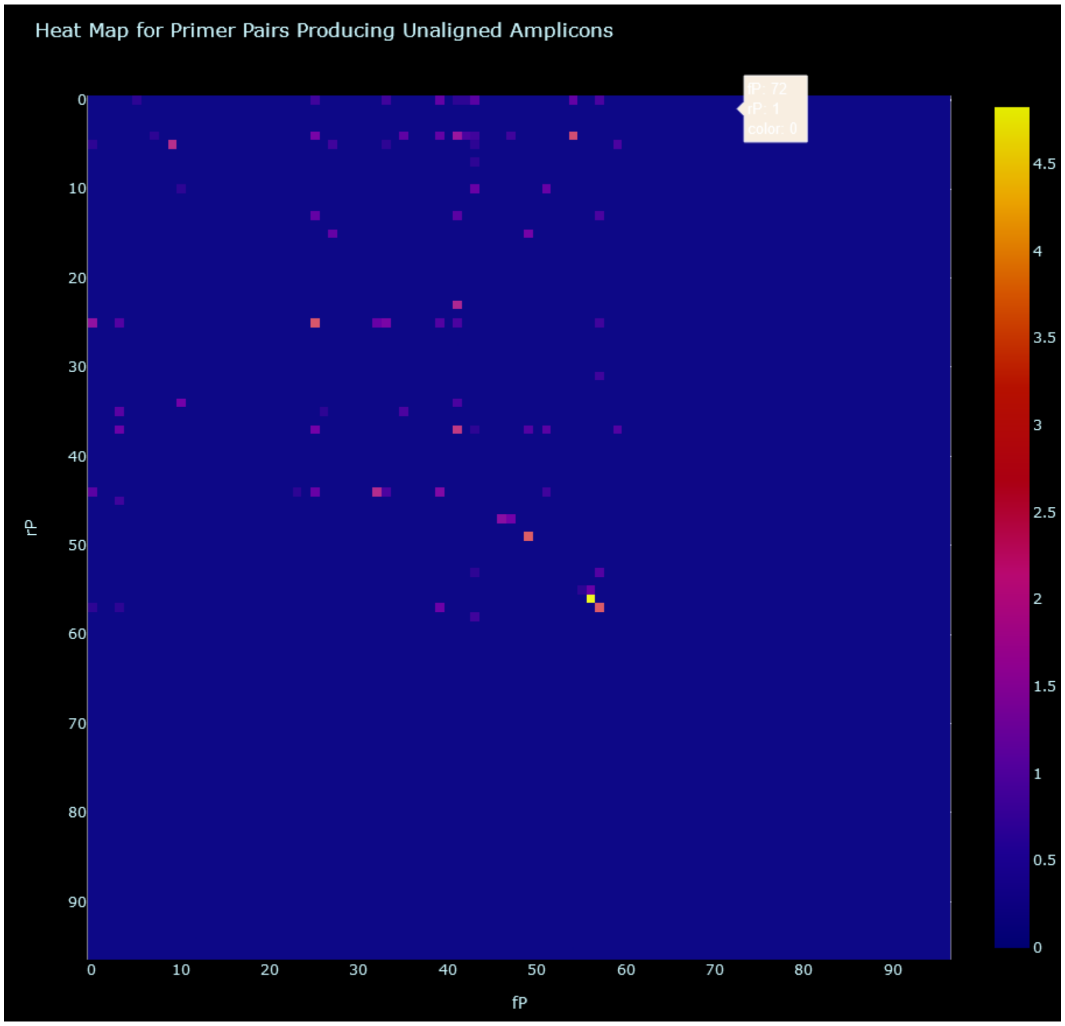
Command Line Arguments Supported

```
find_unaligned_amplicon_primers.py --bowtie2_rslt_path <filepath to Bowtie2 results>
                                    --primers_path <filepath to.xlsx file holding the primers>
                                   --output_NS_pairs <path to NS Pairs excel file>
                                   --output_NS_list <path to NS List excel fil
```

```
--heat_map <path to the heat map>
```

Output from Program

As a result of running this program, 2 output excel files and a heat map will be produced. By default, they output files can be found in the my_output/, but this can also be changed. The file output_NS_list.xlsx is simply a list of the unique sequences along with the forward, reverse primers, and some stats. The file output_NS_pairs.xlsx lists the unique primer pairs that make them up along with more states. Finally, a heat map is generated to give a visual representation of the results and provide insight into which primer pairs actually match with a sequence.



Description of Algorithms

The different python libraries where mainly used to store and manipulate the data from the excel files. Classes were made to emulate the functionalities of sequences and primers. These classes where also used in conjunction with python dictionaries to group sequence by unique sequence. As stated previously, complete match between the primers and sequence where required before being considered a match. This was decided mainly to avoid the risk of falsely matching primers to unaligned sequences. Further, enforcing a strict match between the two allowed us to optimize the search for matching primers. This was done by having a binary search search of the matches among the list of primers given.

Program Structure

find_unaligned_amplicon_primers.py: This is the main file which is the start of the program

utils.ph

```
def create_heat_map(pairs):
    """
    Creates a heat map to visualize the primer pairs.

    :param pairs: The primer pairs to map.
    """



def create_primer_pairs(amplicons):
    """
    Creates a list of sorted Primer_Pairs objects.

    :param amplicons: The amplicons to include
    :param pairs: A sorted list of Primer_Pairs objects
    """
```

```
def print_counts(counts):
    """
    Prints results of the counts for the number of primers matched to sequences.


    :param counts: Dictionary of counts


def print_usage():
    """
    Prints the usage statement to the screen.
    """


def read_primers(primers_filepath):
    """
    Reads primers from primers.xlsx


    :param filepath: The path to the primers excel file
    """
    # Read in primers for excel file
    primers_df = pd.read_excel(primers_filepath)
    # Dictionary to contain primers




def update_counts(counts, fP, rP, unique):
    """
    Increments counts for matching the primer to sequences.
```

```
def write_output_NS_list(amplicons, total, args):
    """
    Writes the primer pairs to an excel file 'output_NS_list.xlsx'.


    :param amplicons: A list of amplicons sorted in descending order.
    :param total: The total number of amplicons
    """



def write_output_NS_Pairs(pairs, args):
    """
    Writes the primer pairs to an excel file.


    :params pairs: List of sorted Primer_Pairs
    :params path: Filepath for excel file.
    """
```

Amplicon.py

```
"""
    Class that is used to represent an Amplicon.
    """
    def __init__(self, seq, fP=None, rP=None):
        """
        Initializes an instance of the Amplicon class.


        :param self: An instance of the Amplicon class.
        :param seq: The nucleotide sequence for the amplicon
        :param fP: The forward primer for the amplicon
        :param rP: The reverse primer for the amplicon
        """
        # Set the sequence for the amplicon

    def __eq__(self, other):
        """
        Overloaded equality operator.

        :param self: Instance of the Amplicon class.
        :param other: The other Amplicon to compare to.
        """

    def __gt__(self, other):
        """
        The overloaded greater than operator.

        :param self: Instance of the Amplicon class.
        :param other: The other Amplicon to compare to.
        """

    def __lt__(self, other):
        """
        The overloaded less than operator.

        :param self: Instance of the Amplicon class.
        :param other: The other Amplicon to compare to.
        """



    def match_primers(self, fps, rps):
            """
            Determines which forward and reverse primers bind to this instance of
            the amplicon.

            :param self: This instance of the Amplicon class
            :param fps: A list of forward primers
            :param rps: A list of reverse primers
            """

        def __str__(self):
            """
            Overloaded to string method for the class.

            :param self: An instance of the Amplicon class.
            """
```

Forward_Primer.py

```
"""
    Class that is used to represent a Forward_Primer.
    """

    def binds_to(self, seq):
        """
        Determines if this instance of the Forward_Primer class can bind
        to the Sequence object 'seq'.

        :param self: An instance of the Primer class
        :param seq: The sequence to try to bind to as a Sequence object
        """
```

```
def __eq__(self, other):
    """
    The overloaded equality operator for the class.

    :param self: An instance of the Forward_Primer class.
    :param other: Another instance of the Forward_Primer class to compare
                this instance to.
    """

def __lt__(self, other):
    """
    The overloaded less than operator for the class.

    :param self: An instance of the Forward_Primer class.
    :param other: Another instance of the Forward_Primer class to compare.
    """

def __getitem__(self, index):
    """
    Overloaded function for indexing the Sequence class.

    :param self: An instance of the Forward_Primer class
    :param index: The index of the sequence to access
    """

def __len__(self):
    """
    Overloaded function for the sequence length for the Forward_Primer class.
```

```
        :param self: An instance of the Forward_Primer class
        """

    @staticmethod
    def sort(fps):
        """
        Sorts a list of forward primers.

        :param self: An instance of the Forward_Primer class
        :param fps: A list of Forward_Primer objects
        """

    def __str__(self):
        """
        The overloaded to string method for the class.

        :param self: An instance of the Forward_Primer class
        """
Primer
    """
    Class that is used to represent a Primer.
    """
    @ abstractmethod
    def binds_to(self, seq):
        """
        Abstract method that determines if this instance of the Primer can bind
        to the Sequence object 'seq'.

        :param self: An instance of the Primer class
        :param seq: The sequence to try to bind to as a Sequence object
        """
Primer_Pair.py
    """
    Class to represent a primer pair.
    """
    def __init__(self, fP, rP, count):
        """
        Inits an instance of the Primer_Pair class.

        :param self: An instance of the Primer_Pair class.
        :param fP: The forward primer in the pair
        :param rP: The reverse primer in the class
        """

    def __eq__(self, other):
        """
        The overloaded equality operator.

        :param self: An instance of the class
        :param other: The other instance to compare to.
        """

    def __gt__(self, other):
        """
        The overloaded greater than operator for the class.

        :param self: An instance of the Primer_Pair class.
        :param other: The other primer pair to compare to.
        :return: True if self.count > other.count
        :return: False otherwise
        """

    def __hash__(self):
        """
        Overloaded hash function for the Primer_Pair class.

        :param self: An instance of the Primer_Pair class
        :return: The hash for the class instance
        """

    def __lt__(self, other):
        """
        The overloaded less than operator for the class.

        :param self: An instance of the Primer_Pair class.
        :param other: The other primer pair to compare to.
        :return: True if self.count < other.count
        :return: False otherwise
        """



    def __str__(self):
        """
        The overloaded to string function for the class.

        :param self: An instance of the Primer_Pair class.
        :return: The string representation of the instance
        """
Reverse_Primer.py
    """
    Class that is used to represent a Reverse_Primer.
    """

    def binds_to(self, seq):
        """
        Determines if this instance of the Forward_Primer class can bind
        to the Sequence object 'seq'.

        :param self: An instance of the Primer class
        :param seq: The sequence to try to bind to as a Sequence object
        """

    def __eq__(self, other):
        """
        The overloaded equality operator for the class.

        :param self: An instance of the Reverse_Primer class.
        :param other: Another instance of the Reverse_Primer class to compare
                    this instance to.
        ""”

    def __getitem__(self, index):
        """
        Overloaded function for indexing the Sequence class.

        :param self: An instance of the Reverse_Primer class
    :param index: The index of the sequence to access
    """

def __len__(self):
    """
    Overloaded function for the sequence length for the Reverse_Primer class.

    :param self: An instance of the Reverse_Primer class
    "””



def __gt__(self, other):
    """
    The overloaded greater than operator for the class.

    :param self: An instance of the Reverse_Primer class.
    :param other: Another instance of the Reverse_Primer class to compare.
    """



def __lt__(self, other):
    """
    The overloaded less than operator for the class.

    :param self: An instance of the Reverse_Primer class.
    :param other: Another instance of the Reverse_Primer class to compare.
    """

def make_reverse_comp(self):
        """
        Constructs the reverse compliment of the sequence for the Reverse
        Primer.

        :param self: An instance of the Reverse_Primer class
        """



    def __str__(self):
        """
        The overloaded to string method for the class.

        :param self: An instance of the Reverse_Primer class
        """

Sequence.py:
    """
    Class that is used to represent a Sequence.
    """
    class Nucleotide():
        """
        Inner class to provide useful functionality in regards to nucleotides.
        This includes identifying which symbols are valid nucleotides whether
        to symbols for nucleotides are compliments of each other.
        """
        # Set to indicated the valid nucleotide symbols
        valid_symbols = set(("A","C","G","T","U","a","c","g","t","u"))
        # Dictionary to identify nucleotide complements. Should be faster than
        # if statements
```

```
compl = {
        "A" : set(("t","T","u","U")),
        "C" : set(("g","G")),
        "G" : set(("c","C")),
        "T" : set(("a","A")),
        "U" : set(("a","A")),
        "a" : set(("t","T","u","U")),
        "c" : set(("g","G")),
        "g" : set(("c","C")),
        "t" : set(("a","A")),
        "u" : set(("a","A")),
    }
# Dictionary used for constructing complimentary sequences
get_compl =    {
            "A" : "T",
            "a" : "t",
            "T" : "A",
            "t" : "a",
            "G" : "C",
            "g" : "c",
            "C" : "G",
            "c" : "g",
          }
# Dictionary for identifying the number of hydrogen bonds between a
# given nucleotide and its compliment
h_bonds =   {
          "A" : 2,
          "a" : 2,
          "T" : 2,
            "t" : 2,
            "U" : 2,
            "u" : 2,
            "G" : 3,
            "g" : 3,
            "C" : 3,
            "c" : 3,
          }
    pass



def __init__(self, seq):
    """
    Initializes an instance of the Sequence class.

    :param self: An instance of the Sequence class.
    :param seq: A sequence of nucleotides as a list of Nucleotides.
    """

def __eq__(self, other):
    """
    Overloaded equality operator for the class

    :param self: An instance of the Nucleotide class.
    :param other: An other nucleotide being compared
    :return: Bool indicating if this seq is equal to other
    """

def __getitem__(self, index):
    """
    Overloaded function for indexing the Sequence class.

    :param self: An instance of the Sequence class
    :param index: The index of the sequence to access
    :return: The corresponding base at 'index'
    """

def __hash__(self):
    """
    Overloaded hash function for the Sequence class.

    :param self: An instance of the Sequence class
    :return: The has for the class instance
    ""”


def __len__(self):
    """
    Overloaded function for the sequence length for the Sequence class.

    :param self: An instance of the Sequence class
    :return: The length of the sequence as an int
    """

def __lt__(self, other):
    """
    The overloaded less than operator.
    :param self: An instance of the Sequence class.
    :param other: Another instance of the Sequence class to compare to.
    :return: Boolean indicating whether this sequence is less than the
            other.
    """



def primer_binary_srch(self, primers):
    """
    Searches a list of forward primers for one that binds to this instance
    of the sequence class. The search is completed via binary search, and
    a fP that binds to a search is one in which every base in the forward
    primer matches the corresponding base in the sequence.

    :param self: An instance of the Sequence class
    :param primers: A list of primers to match to
    :return: The Forward_Primer that binds to the sequence
    """

def primer_srch(self, primers):
    """
    Searches a list of forward primers for one that binds to this instance
    of the sequence class. The search is completed by brute force, and a fP
    that binds to a search is one in which every base in the forward primer
    matches the corresponding base in the sequence.

    :param self: An instance of the Sequence class
    :param primers: A list of primers to match to
    :return: The Forward_Primer that binds to the sequence
    """



def __str__(self):
    """
    Overloaded function to convert the instance of the class to a string.


    :param self: An instance of the Sequence class
    :return: The string representation of the class.
    """
```