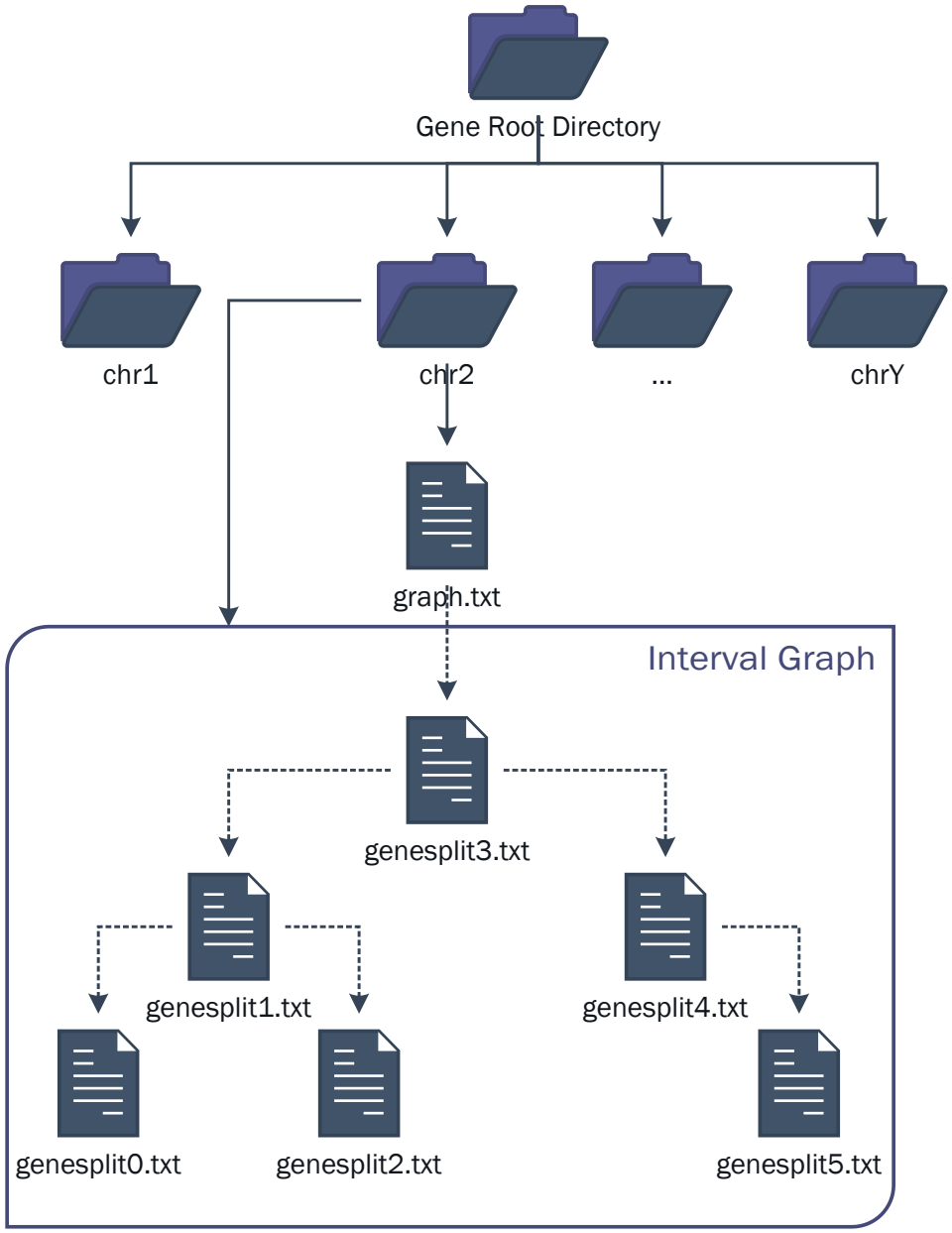
BIODISCOVERY GENOMIC DATA ORGANIZATION

By: Paul An

Graphical illustration of organization



Organization explained

Chromosome-level split

With the assumption that there are no genes that would ever cross the boundaries between chromosomes, the first step is to segment the entire genomic data into individual chromosomes. This means when a query is received, we can quickly eliminate all chromosomes that do not fall within that query range.

Graph.txt

This text file, contained in every chromosome directory, describes an adaptation of an "Interval Tree" outlined here: https://en.wikipedia.org/wiki/Interval_tree#Augmented_tree

- Tree is a Binary Search Tree organized on the lowest gene start point contained within that file
- Each node contains information on:
    - Lowest starting point of that file
    - Highest ending point of that file
    - Lowest starting point of the subtree starting at the node
    - Highest ending point of the subtree starting at the node

For querying, we perform a breadth-first-search on the graph, and if we know the sub-tree does not contain the querying range, then we can immediately rule out that branch.

Let's say a traditional search which iterates over the entire file takes $O(n)$ where $n = \#of\ entries\ in\ entire\ data$.

Our search will then amortize to a value similar to $O(Log(n) + m)$ where $n = \#\ of\ entries\ in\ entire\ data, m = \#\ of\ expected\ results$.

Of course since we still linearly search through every file that contains the range we query, this run time is only a rough estimation.

Genesplit*n*.txt

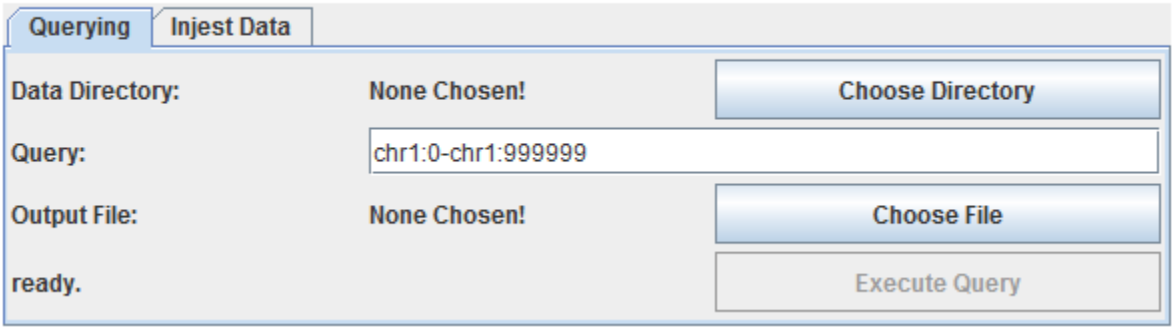
In order to assure the query meets the estimated run-time, we have to try to balance the files such that they contain manageable chunks of data. To do this, when ingesting the original data, we attempt to make it so that each file is around 10,000 entries. *(algorithm described in design details later)*

Why not extend the tree into the file?
Doing so would require additional processing and memory during ingestion. Additionally, even if we were to store entries in a pre-order fashion so that we can recover all entries, doing so would still require $O(nLog(n))$ to recover the graph using the current 'human readable' format and `BufferedReader`, as opposed to $O(n)$ of straight search.
We can consider using a rigid datasize per entry, and use `RandomAccessFile`, but I considered this out-of-scope for this project.

Interacting with the Project

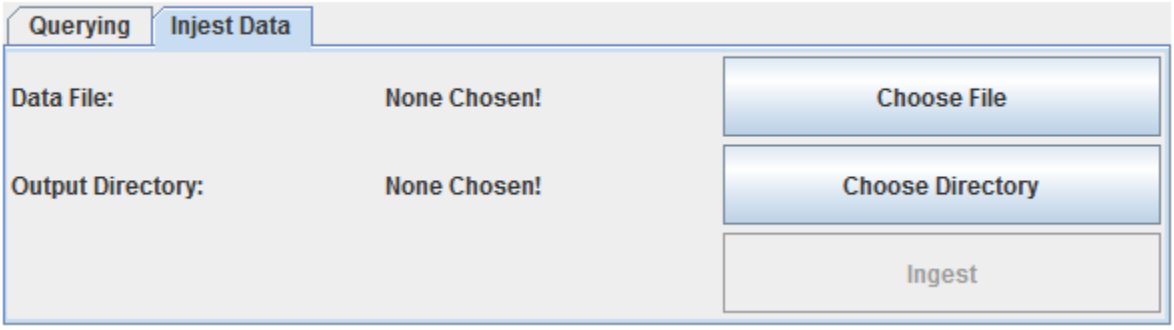
The project was programed in Eclipse. To run the UI, please refer use `BiodiscoveryInterview/src/ui/MainUIFrame.java` as the entry point.

Querying



1. Select the data directory for where the data is already organized (see next section Ingesting) Note: this selection must be a directory!
    a. Example: …\BioDiscoveryInterview\BioDiscoveryInterview\data\test2
    b. Note, once the directory is chosen, the program will automatically attempt to load the graph into memory.
2. Select the output location where the query result will be written.
    a. Example: …\BioDiscoveryInterview\BioDiscoveryInterview\result.txt
3. Enter the query string
    a. chr18:0-60000000 > chromosome 18 within 0 to 60000000
    b. chr3:5000-chr5:8000 > huge range spanning from chr3:5000 to chr5:8000
    c. chr2 > query the entirety chromosome 2
    d. chr2-chr3 > query the entire chromosome 2 to entirety of chromosome 3
    e. chr5:5000 > query chromosome 5 from 5000 to the end of that chromosome
4. Click the “Execute Query” button. Once the query is done, the result will be written to the selected output file.

Ingesting



1. Select the data file to consume and reorganize.
    a. Example: …\BioDiscoveryInterview\BioDiscoveryInterview\data\original\probes.txt
2. Select the directory in which you'd like to store the reorganized data. Note: Must be directory!
    a. Example: …\BioDiscoveryInterview\BioDiscoveryInterview\data\test2
3. Click the "Ingest" button. Once the process is done, the directory will be populated with the reorganized data for random access.

Algorithm Overview

//*n* denotes that there are some comments on that design decision in the following section.

High-level Querying Pseudocode

```
FileWriter writer = createwriter();
List<SearchUnits> units = searchTree(tree);
sort(units);//*1*
foreach(unit in units)//*2*
      execute(unit, writer);

execute(unit, writer) {
      if(unit.searchtype == EngulfNode)
            write entire split file into output file;
      else if(unit.searchtype == EngulfSubtree)
            write entire subtree's files into output file;
      else if(unit.searchtype == Search)
            open file in node, search thru file for intersections;

}
```

High-level Ingestion Pseudocode

```
//pass 1
chrhists = read all entries, create histogram of each chromosome;
//*3*

//prep
create chr folders;
foreach(chromosome in chromosomes)
      create number of files based on chrhist such that
       each file contains around 10,000 entries (adjustable in var);
      define ranges of gene points each file takes;

//pass 2
Read all entries,
Figure out which file the entry belongs to based on prev step;
Write entry into appropriate file;//*4*
```

Design decisions and details

//*1* Sorting the searching units

This step is not actually required, and potentially increases the runtime of query from $O(Log(n) + m)$ as described previously to $O(nLog(n) + m)$.

The sorting is done to ensure that the result is somewhat in order (for example, all chromosome 9 before chromosome 10, rather than a small chunk of 10, then chunk 9, then 10 again, etc.)

However, if performance is more desired than the order of result (if a computer is using the results, not humans) then we can just take this sorting line out.

//*2* Sequential read/write

In this step, multiple files are opened, read, but written to a single output file.

I considered parallelizing processing from the multiple search sources, but I believe that because we are writing onto a single file, and writing tends to be the longest operation, we would become bottlenecked on the file write, and parallelizing the read would not improve performance by much.

//*3* Using a multi-pass approach to balance the tree

This step is necessary because of a few assumption I made about the potential input data:

1. Although we are provided with a data file that has contiguous blocks of chromosome, this may not always be the case
2. Moreover, it is possible that the entries within the data file are in completely random order

My goal is to arrive at a balanced interval tree such that all nodes have around the same number of manageable entries.

One way to do so is to sort the entire data set. Using let's say a merge sort using files (since the dataset has potential to be extremely large). This would take at best $O(nLog(n))$ time, and potentially $O(n)$ temporary space, depending on implementation. I think that's not good enough.

The next best thing is to use a histogram. Processing it would be $O(n)$ time instead, and the space in-memory will be capped at the number of histogram bins, which is trivial.

However, if we knew more about the organization of the input file, for example, if the data will always be somewhat in-order, we can skip this step entirely.

//*4* Sequential read/write and opening many files

Using the same assumption as //*3*, we must keep a large amount of files open so that we can handle completely random entries. Again, if there was more understanding of the input data, we can optimize this step much better.

Parallelization was considered again, here, but note that, if for example, the entries from the input data was random, we would start seeing the hard disk needle have to move in many locations, creating large latency between small writes.