Ling 473 Project 4
Due 11:45pm on Thursday, August 30, 2018

Bioinformatics refers the application of statistics and computer science to the management and analysis of data from the biosciences. In common with computational linguistics, the field shares the task of detecting and searching for patterns in gigantic datasets. For this project you search for DNA sequences in the 24 chromosomes of the complete hg19 GRCh37 human genome. These base pairs are represented in text form, so the problem is essentially a text processing problem.

There are 4,965 target sequences. For each of these you will identify the chromosome file and the offset within the file for all matches. This problem was selected because this search cannot be performed efficiently without special attention to the algorithm that is used. Using `grep` around five thousand times to search 2.8 gigabytes probably won't end well, even in this simplified exercise.

Instead, you will store the search targets in a **prefix trie** and make a single pass through the genome corpus. Because a position in the trie keeps track of multiple simultaneous searches, this reduces the problem from $O(NM)$ to $O(N \log M)$. Hashing cannot be used for this purpose, because the target sequences are different lengths, so you don't know how many characters from the input to use for computing a hash.

Human Genome Corpus

The following directory on patas contains 24 files, corresponding to human chromosomes 1 through 22, plus X and Y. Each file describes an ordered sequence of nucleotide bases A, T, C or G.

```
/opt/dropbox/18-19/473/project4/hg19-GRCh37/
```

These files come from `http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/hg19.2bit`. You may copy them to your home machine for local development, but ***your final code must run on patas***.

Masking

In accordance with convention for DNA data, some of the nucleotides are represented by the corresponding lower-case letter. This indicates that they are "masked," meaning that they are of low complexity and thus believed to be less important. For this project, you will do a case-insensitive search, so masked areas will be included (the target sequences are specified as all upper-case).

Unknown

There is no data available for some parts of the genome. These areas are indicated by the letter N. None of the target sequences contain N, and your trie does not need to be able to recognize N.

Loading the Target Sequences

You will develop a trie structure. Each trie node makes reference to (up to) four child nodes, one for each nucleotide. Any or all of these references can be *null*. When all four children of a node are *null*, the node is a terminal node. Each node (except the root) has exactly one parent, but nodes do not need to store a reference to their parent node, because navigation through a trie is always one-way, starting from the root. The elegance of a trie is that every node implicitly represents a unique string merely by its position.

Some trie implementations allow each node to store a "payload." One purpose of the payload is to signal whether the node represents the last character in a target string. In this case, the payload would be *null* if the trie node is not a stopping point. This sort of mechanism (for differentiating valid acceptance points) is required if the trie has to store strings that are exact prefixes of other strings that are also stored. For example, {"cat" "catamaran"}. In our application, none of the target sequences start with any of the others, so we don't need to use a payload. Instead, we will know if we are at an acceptance point (i.e. that we found a match) if the node has no child nodes.

After defining the node structure, you will instantiate a single node to represent the trie root. Next, you will populate the trie with each target sequence. The target sequences are in the following file, one per line:

```
/opt/dropbox/18-19/473/project4/targets
```

Add each target sequence to the trie by starting at the trie root and following the sequence's path of nucleotides. This is done by advancing through the target string and navigating/building the trie at the same time. For example, if you come to a child node that is missing, add it by creating a new node. When you reach the end of the target string, you should be at a node with no children—a terminal node that you just created.

If your trie nodes store a payload (not required here), you would store the payload in this node in order to mark the node as a completed match. In such a design, matching can occur at non-terminal nodes.

The target strings can be discarded now, since they are all stored in the trie implicitly (and explicitly as well, if you chose to store each string as a payload).

Searching

Now that you have a trie that represents the strings that you are searching for, you can scan the corpus in a single pass, finding all the matches. You will still need to check every character position in the corpus, but the trie allows you to check all targets at the same time and abort as soon as any match from that position becomes impossible.

You will need to keep track of two index positions in each genome file i, j and one navigation position—a pointer to a node, pn—within the trie. The "outer" index position i will always advance one character at a time. As you read from the corpus, don't forget to normalize the letter-case to whatever you stored in the trie, since we do wish to search within masked data.

For each of these outer positions, the navigation position pn will start at the trie root, and a second "inner" character position j will be initialized to the value of the outer index position i.

Now, similar to how the trie was loaded, pn will attempt to navigate through the trie, according to the characters read from the corpus, beginning from j. Of course, this time we don't modify the tree; if further navigation becomes impossible and we are not at a terminal node, there is no match, so increment the overall position i, set $j = i$ and $pn = root$, and repeat.

If you are able to reach a terminal node by matching the corpus to the trie, then you have found a match. If you didn't store the target sequence as a payload in the terminal node, it's easy enough to recover the string that you've matched: it's the substring of the input from i to j, inclusive. (Note that you don't need to build it as you go.) Record each match as described in the *Output* section below.

Performance

A major challenge will be dealing with the large files in the human genome corpus, since *j* does have to continually navigate forward and then reset (seek) backwards within a small area. If you are able to load each file into memory, this isn't a problem. Otherwise, you might consider using *file seeking* for all operations on the corpus, reading just one character at a time. This is a great solution if you can count on the operating system to buffer your read operations. Even better would be to memory-map the file you're reading, an advanced technique which might be available in your programming language or OS.

Your program will certainly be long-running. My single-threaded solution, written in C#, executes in about 4 minutes (4:09) on a 3.17GHz machine. If you are using patas/dryas, **please submit your job to condor**. You can use the 'condor-exec' shortcut for this. If execution times are prohibitive, you can report results for a smaller fraction of the corpus (5 points deduction).

Output Format

Output your results to the console (stdout). Print each **filename** on a new line (the full path can be included). All matches found in that file will follow, with each match on a single line that begins with a tab. After the tab character, print the **zero-based offset of the match** within the file (please use hex numbers). Follow this with another tab and the **sequence that was found**. Print the sequence in all caps. Your output should look like this example fragment:

```
/opt/dropbox/16-17/473/project4/hg19-GRCh37/chr1.dna
        0000C341     AAACTAACTGAATGTTAGAACCAACTCCTGATAAGTCTTGAACAAAAG
        00022767     GGGCTGGAGACTGACTTAATCACCAACAGCCAAAGGTTTTATCAATCATGCTTGCATAATAAAGCCTC
        etc...
/opt/dropbox/16-17/473/project4/hg19-GRCh37/chr10.dna
        0013F466     GGCCTGAGAGGGGGGCCCAGGCTCTCCCGGAAGACGGCCTGAGCCAGGTCCACGCTCCCCCGGAAGACGGCCTGAGA
        0013F4AA     GGCCTGAGAGGGGGGCCCAGGCTCTCCCGGAAGACGGCCTGAGCCAGGTCCACGCTCCCCCGGAAGACGGCCTGAGA
        etc...
/opt/dropbox/16-17/473/project4/hg19-GRCh37/chr11.dna
        00021443     GGGGCTGGAGACTGACTTAATCACCAACAGCCAAAGGTTTTATCAATCATGCTTGCATAATAAAGCCTC
        etc...
```

Extra Credit

For extra credit, have your program write a separate output file where the results are grouped together by sequence, rather than by file. Format the output as follows with the sequence (in all caps) on one line followed by a tab, the offset, a tab and the file name for each occurrence. Name this file `extra-credit` (note: no file extension!) and write it to the current directory.

```
TCATTCCAAGAAAAAGTCTTAGGAGTGCAGCACTTCAAAATCAGGTAATG
        028751CE    chr9.dna
        040D4F63    chr9.dna
        042383BC    chr9.dna
AAACTAACTGAATGTTAGAACCAACTCCTGATAAGTCTTGAACAAAAG
        0000C341    chr1.dna
        000165CD    chr19.dna
TTGGGCTTGGGGTACAGGAGGAGTTGGTGGGGTGCCTGTGGCCACTCCACTGCCCTCTGGGATTAGGAGGAGACAGTGGGGTCAGGACTCA
        00CC6419    chr1.dna
        00CFC296    chr1.dna
CACACACACACATATATATGTAGAGACAGGGTTTCTCCTTGGTGCCCAGATTTGGTCT
        0465C8FF    chr1.dna
CAAGATACCAATTCAAGTATGGAATTTAAGCGGTGACAAGTTAATCTAACCCTATTACAAATA
        0469AF3B    chr1.dna
```

Submission

As a courtesy to your fellow students, please use Condor when you test on patas/dryas. Include the following files in your submission:

`compile.sh`	Contains command(s) that compile your program. If you are using python or any other interpreted language that does not require compiling, then this file will be empty, or contain just the single line: `#!/bin/sh`
`run.sh`	The command(s) that run your program. Be sure to include compiled binaries in your submission so that this script will execute without first running compile.sh

condor.cmd	Condor control file, suitable for running your program as follows:
	`condor_submit condor.cmd`
	please name the stdout, stderr and log files as follows (they need to have unique names for me to run them in a batch without overwriting each other), using your netid:
	`output=`*`[netid]`*`.out`
	`error=`*`[netid]`*`.error`
	`log=`*`[netid]`*`.log`

`output`	This is the output of your program, captured from the console by executing the following command: `$ ./run.sh >output` Note that your shell script will not create the output file itself, but print to standard out.
`extra-credit`	Extra credit output (optional)
`readme.{pdf, txt}`	Your write-up of the project. Describe your approach, any problems or special features, or anything else you'd like me to review. If you could not complete some or all of the project's goals, please explain what you were able to complete.
`(source code and binary files)`	All source code and binary files (jar, a.out, etc., if any) required to run and compile your program

Gather together all the required files, making sure that, for example, any PDF or other binary files are transferred from your local machine using a binary transmission format. Then, from within the directory containing your files, issue the following command to package your files for submission.

```
tar -czf hw.tar.gz *
```

Notice that this command packages all files in the current directory; do not include any top-level directories. Upload the file to Canvas.

Grading

Correct results	25

All files present, named correctly	10
Clarity and readability of code	15
run.sh runs to completion	10
condor.cmd runs to completion	5
program efficiency	20
Write-up	15
Extra Credit	+10

Citation

Human Genome 19, GRCh37 Genome Reference Consortium Human Reference 37, Primary Assembly (GCA_000001405.1), February 2009. GRCh37 is a haploid assembly, constructed from multiple individuals. The primary assembly represents the assembled chromosomes, plus any unlocalized or unplaced sequence that represent the non-redundant, haploid assembly.