FastAnnotation: A Generalized FASTA File Format Parser

Jay Rao

December 2019

1 Introduction

Sequence data files are widely used in the biological sciences in areas ranging from genetics to bioinformatics. The primary file format most widely used to store and process sequence information is the FASTA format, originally formulated in 1985 by Pearson and Lipman [1]. One of the attractions of the FASTA format is the simplicity of its structure, which is composed of two alternating parts, *headers* followed by *sequences*. One of the biggest issues with the file format comes from the fact that the header section lacks any standardized structure which has led to the rise of many different FASTA header formats. FASTA headers, while parsed easily by the human eye, are not trivial to parse programmatically. This inflexibility creates difficulties when interpreting and annotating sequences. For this reason, annotations are commonly stored in extraneous files and databases apart from the sequence data, which often creates an inconvenient disconnect between related sources of information. While FASTA is one of the most popular formats in bioinformatics, its loose definition has led to the creation of specialized file parsers suited only for the specific tools they are defined for. Such specializations might require filters to rewrite and reformat header data for use by other subsequent tools in the sequence processing pipeline.

In this paper we present a simple Python parsing wrapper and propose an extension to the general FASTA format which creates a backward compatible and extendable FASTA file format that can be operated on using a single standardized parser. The implications of this extension involve the ability to freely edit and update sequence annotations without disturbing existing tool-chains and data pipelines. We include certain backwards compatibility features with the general parser to allow for the reading of currently existing valid FASTA files, with the option to extend them using our proposed new extensions. We do this through the introduction of basic JSON syntax in the headers of each sequence that allows for annotations to be defined using basic `<key>:<value>` pairs instead of context-specific keywords and delimiters as is currently the common practice.

2 Background

Before proposing any new extension to FASTA, we first provide an overview of the current definition of FASTA files and other associated file formats that have arisen since its introduction.

2.1 FASTA

The FASTA file format is defined as having alternating header and sequence lines, with header lines differentiated from sequence lines through the use of a single `>` character as their first character. Any number of sequence lines can follow a header. The sequence is terminated when

either the end-of-file or the next header line is reached. Sequence lines are a series of characters from a multiple standardized alphabets. For DNA nucleotide sequences the alphabet is `A C G T`. For RNA sequences `A C G U`, For protein sequences 28 characters are used to represent the common amino acids, stop codes, and gaps [2]. Additional sequence alphabets have also be defined, such as the IUPAC notation, to represent multiple alternative nucleotides at a given position [3]. In contrast, the format of FASTA headers, is unstructured, with the exception header requirements introduced by a specific query package or analysis package.

2.2 Diversity of FASTA formats

The first sample is a FASTA file that uses spaces as header delimiters. The file below describes two sequences and uses the basic nucleotide alphabet. Each header line is delimited with a `>` and is always followed by the sequence it describes. [2]

Listing 1: Space delimited FASTA

```
> name genus species
ACCGCGATCGATTATCGTGTCATGACGT
> name2 genus2 species2
AGGCGGTCCATGTGACGTGTGACACGTACGAGTA
```

Next is a sample sequence defined using an amino acid alphabet that uses a | character as a header delimiter instead of spaces.

Listing 2: Abridged Sample from Zhang Lab at UMich [2]

```
>gi|186681228|ref|YP_001864424.1| phycoerythrobilin:ferredoxin
MNSERSDVTLYQPFLDYAIAYMRSRLDLEPYPIPTGFESNSAVVGKGKNQEEVVTTSYAFQTAKLRQIRA
AHVQGGNSLQVLNFVIFPHLNYDLPFFGADLVTLPGGHLIALDMQPLFRDDSAYQAKYTEPILPIFHAHQ
QHLSWGGDFPEEAQPFFSPAFLWTRPQETAVVETQVFAAFKDYLKAYLDFVEQAEAVTDSQNLVAIKQAQ
LRYLRYRAEKDPARGMFKRFYGAEWTEEYIHGFLFDLERKLTVVK
```

Most basic FASTA parsers assume on a properly formatted header, and may not be able to properly parse both files. Without the existence of a standard format parser, a lot of pre-processing and rewriting is required to coerce incompatible FASTA formats through different tools. The introduction of a general purpose parser would reduce the developer overhead of bioinformaticists who wish to create a pipeline of tools where components expect FASTA headers in different formats. In this paper we aim to provide a general, extensible format as well as provide a Python library for parsing and adding annotations to FASTA files.

2.3 Other Popular File Formats

Many other sequence related file formats exist and are widely used. It is useful to see how these file formats have evolved and the particular FASTA-related problems that they address.

2.3.1 FASTQ

FASTQ is a common FASTA extension that provides quality scores in addition to the basic sequence and header data that FASTA provides. Interestingly, in the paper that introduced the FASTQ format the authors Cock et al. noted that "over time [FASTA] has evolved by consesus; however, in the absence of an explicit standard some parsers will fail..."[4]. The FASTQ format uses a scoring metric called PHRED which scores each single nucleotide to indicate how accurate the read was for the given base. This scoring is useful because the process of reading a sequence of genomic data is subject to noise and small variations can cause bases to be misread as another.

FASTQ files are structured similarly to FASTA files with each entry made up of 3-4 lines. Each entry must have a header that is delineated with a `@` character. After the header comes the sequence which is defined in the same way as FASTA. Finally comes the quality line which uses a contiguous range of ASCII characters to represent a range of allowed scores. Internally these ASCII characters are converted back into numbers representing the score for the base at the same position in the string. An optional line may exist between the sequence and score data that simply repeats the header line but is delineated with a `+` character instead of a `@` [4].

Here is a example of a FASTQ file.

Listing 3: Sample FASTQ file. Taken from supplemental data from [5]

```
@E00170:310:HNLK2CCXX:1:1101:20953:3296 1:N:0:CCGTCCCG
TATGAGTAAATATGTCATGAATTGGAGGCCAGGCTTATCTAGACA
+
,AFFFKKAA,,FFKK,F<FKFKKK<<KFAFF<(,,,,AF,AFKKK
@E00170:310:HNLK2CCXX:1:1101:6329:3313 1:N:0:CCGTCCCG
TTTTCTTTTTAGCAAATAACCGAAGATAAGCTTAAAACATTTTCT
+
AA,AFKKFFKKKKKKKKKAKKKF<FKKKAKFF,,AF<7FK7AFKKK
```

Interestingly, even FASTQ couldn't protect itself from a fragmented standardization as a second popular standard for FASTQ, developed by Solexa Inc [4] and was eventually bought by Illumina. The two standards are almost identical except for how the quality scores are calculated and the range of ASCII characters used for quality strings. This difference requires FASTQ processing pipelines to either detect the format or that the user supply (via a flag or commandline argument) the PHRED format used to accurately read both files types. [4]

2.3.2 SAM

SAM stands for Sequence Alignment/Map and is a file format that is widely used for the storing of sequence alignment data. At a high level, sequence alignments specify matching substrings from a given sequence to some provided "reference" sequence. This is useful for understanding the similarities of multiple potentially different genomic sequences.

SAM files are most commonly used to align FASTQ sequences to FASTA sequences. The FASTA sequences are referenced by the name given in their headers, and an integer offset into their sequences that best match a FASTQ sequence specified for each line.

SAM files are delimited with tab characters and consist of a header and alignment section. The optional header section, must always precede any alignment lines if it exists, and should be labeled as a header line with the `@` character. [6, p.1] The alignment lines are much more strict and have 11 required tab-delimited fields that must exist with optional extra fields available.

[6, p.5] Below is a sample SAM file.

Listing 4: Sample from p.2 of SAM tools file specification [6]. The FASTQ sequence *r0001* aligns to the FASTA sequence *ref* at character 30 (1-based) with 8 matches followed by 2 extra inserted bases followed by 4 matches, etc. The second line describes an alignment of FASTQ sequence *r002* to *ref*.

```
@HD VN:1.6 SO:coordinate
@SQ SN:ref LN:45
r001 99 ref 7 30 8M2I4M1D3M = 37 39 TTAGATAAAGGATACTG *
r002 0 ref 9 30 3S6M1P1I4M * 0 0 AAAAGATAAGGATA *
r003 0 ref 9 30 5S6M * 0 0 GCCTAAGCTAA * SA:Z:ref,29,-,6H5M,17,0;
r004 0 ref 16 30 6M14N5M * 0 0 ATAGCTTCAGC *
r003 2064 ref 29 17 6H5M * 0 0 TAGGC * SA:Z:ref,9,+,5S6M,30,1;
r001 147 ref 37 30 9M = 7 -39 CAGCGGCAT * NM:i:1
```

SAM files define a header format, which simplifies writing a parser, but the headers are not extensible, thus adding extra information to the alignment lines would once again require writing specialized parsers to handle handle new features while maintaining the ability to process earlier SAM formats. This will once again make pipelining through multiple tools difficult for specialized SAM files and will require some level of pre-processing or stream rewriting.

2.3.3 VCF

VCF stands for Variant Call Format and is another popular sequence annotation file format, like SAM, that does not store actual sequence data, but instead stores annotation information that is relative to a provided reference sequence. It is primarily used to store genomic variation information, as the name suggests, which just defines how sequences differ from a reference sequence through a variety of metrics.

This file format slightly differs from the previous file formats we have explored in that it expects a number of meta-descriptor lines that provide information about the format of the file. These lines are delimited by two successive **#** characters at the start of the line. These meta lines describe things like the contents of each data line and the format and order of particular values in each line.

After the meta-lines comes a single header line that is delineated with a single **#** character. The header line is a constant tab-delimited string consisting of 8 values. The data lines follow directly after the header line and are tab-delimited as well.

Here is an abridged sample of a VCF File.

Listing 5: Edited Sample from p.1 of VCF Spec [7]

```
##fileformat=VCFv4.2
##fileDate=20090805
##source=myImputationProgramV3.1
##reference=file:///seq/references/1000GenomesPilot-NCBI36.fasta
.
.
.
##phasing=partial
##INFO=<ID=NS,Number=1,Type=Integer,Description="Number of Samples With Data">
##INFO=<ID=DP,Number=1,Type=Integer,Description="Total Depth">
.
.
.
##FILTER=<ID=q10,Description="Quality below 10">
##FILTER=<ID=s50,Description="Less than 50% of samples have data">
##FORMAT=<ID=GT,Number=1,Type=String,Description="Genotype">
##FORMAT=<ID=GQ,Number=1,Type=Integer,Description="Genotype Quality">
##FORMAT=<ID=DP,Number=1,Type=Integer,Description="Read Depth">
##FORMAT=<ID=HQ,Number=2,Type=Integer,Description="Haplotype Quality">
#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT NA00001 NA00002 NA00003
20 17330 . T A 3 q10 NS=3;DP=11;AF=0.017 GT:GQ:DP:HQ 0|0:49:3:58,50 0|1:3:5:65,3
```

The VCF file format starts to move in the right direction by defining the meta headers that indicate the layout of the data lines and what they represent in the file. However, VCF chooses to do this in a radically different way than our proposed extension to FASTA. VCF uses a basic schema model that is similar to those used in basic Database programs. By strictly defining header information at the top of the file, you allow the file to be highly customizable but do not tightly couple the sequence with the annotation data in the file. In the end, a separate FASTA file with the related sequence data is still required for VCF files to be of any use, and our extension aims to completely join the two disparate sources of information into one general file type.

Now that we have completed our brief introduction of popular genomic file formats we will introduce our proposed extension to FASTA and demonstrate how it simplifies the reading and writing of complex sequence data.

3 Extension of FASTA

3.1 Our Goals

Our primary goal is to provide a general computer-parsable FASTA header format, that allows annotation information and genomic sequences to be stored jointly instead of in separate external files and databases. Our extension is built off of the flexibility of JSON and therefore does not require any of the overhead of files like VCF that define a file-specific schema that describes the subsequent columns and their data types per row. Our extension operates on a principle that is similar to a document model for a NoSQL database, wherein entries are in the form of JSON objects and each entry is unique, meaning that the `<key>:<value>` pairs do not need to exist in every object across the file and can instead be highly customized for the specific sequence that they annotate.

Another key goal is to provide this functionality in a relatively compact way that maintains a level of human-readability while still being efficiently computer-parsable. To make the file computer-parsable and still human-readable we rely heavily on the JSON object as an easily de-serializable data-structure, but also take this one step further with the introduction of lazy loading and evaluation into the FastAnnotation parsing class. At a high level, this is done by selectively not de-serializing JSON objects that we find in the headers of the FASTA file, and instead defer the de-serialization of the objects until they are requested. This saves us time during the initial loading of the FASTA file and prevents unnecessary computation in the case where certain headers are never accessed.

Finally we wish to make this format backwards compatible and general enough to be used effectively in bioinformatics pipelines that currently exist. Backwards compatability means that existing FASTA files should be able to be parsed using the FastAnnotation parser and can also be updated and re-written in the updated FASTA format we define. We also hope that our updated extension is utilized in existing pipelines to simplify the analysis process that often requires external annotation files and databases to be provided.

Before moving into an in-depth explanation of the extension to the FASTA format and how the general Parser works, we will first provide some basic background on the structure that enables all of this - JSON.

3.2 JSON

Our extension of FASTA hinges on the usage of JSON. JSON stands for Javascript Object Notation and is a standard representation used for data in a variety of domains. The convenience of JSON comes from its flexibility and its ease-of-use which allows us to roughly maintain the current version of FASTA's human-readability while still making editing and updating files an easier process. JSON is defined through the use of `<key>:<value>` pairs. These `<key>:<value>` pairs are delimited from each other with commas with the whole set of `<key>:<value>` pairs being encased within `{}` [8].

Listing 6: Sample valid JSON

```
1 {
2     "key1" : [1,2,3],
3     "key2" : "value2",
4     "key3" : {
5                  "subkey1" : "subvalue1",
6                  "subkey2" : ["a", "b", "c"]
7              }
8 }
```

In the sample in Listing 6 we see a number of key properties of JSON that make it useful for our problem domain. On line 2, 3, and 4 you can see three keys defined with three different value types corresponding to the keys, with the first being a list, the second being a string, and the third being another JSON object with its own set of keys and values. We will use these features to store annotation values for sequences in the FASTA file with keys being the descriptions of the annotations and the values being the value of the annotation. We will now dive into how the parser is set up and how it can be used to read, write, and extend the FASTA format without requiring a new parser for each new annotation addition.

3.3 The Extension

The extension we propose is a simple addition to the existing FASTA standard. In a normal FASTA file you have headers that can have multiple values that are delineated using some constant delimiter. In our format we add a JSON object to the end of the header. From there, we define an API that can be used to add annotations to that JSON object without modifying the existing delimited section at the front of the header. This allows for arbitrary annotations to be added after a FASTA file has already been created without causing the parser to immediately fail after the update.

As a small example of the power of the extension let's walk through a toy sample that shows the before and after of writing to a FASTA file.

In Listing 1 we can see a sample FASTA file that is delimited by spaces. Let's assume that we want to add a `chromosome` annotation to the headers of each of the files. Through the use of the parser and its write functionalities we can transform Listing 1 into something that looks like Listing 7.

Listing 7: Space delimited FASTA with added annotation

```
> name genus species {"chromosome":13}
ACCGCGATCGATTATCGTGTCATGACGT
> name2 genus2 species2 {"chromosome":14}
AGGCGGTCCATGTGACGTGTGACACGTACGAGTA
```

In Listing 7 there is now an additional JSON object that was added to the header file that was delimited using the same delimiter as the parser used when parsing the file. On the next read of this file, the parser does not have to be re-written to accommodate the new annotation and will simply read the JSON and internally convert it into the appropriate `<key>:<value>` pair. While this was just a sample of the expected usage of the parser and the extended file format, we will more rigorously examine the syntax of the parser and its ability to update FASTA files in the next section.

3.4 How the API works

3.4.1 Loading FASTA

The basic syntax used to load an arbitrary FASTA file can be seen in Listing 8 and consists of 1 required parameter and 2 optional parameters with provided default values.

Listing 8: Sample FASTA load

```
1 fasta = "filename.fa"
2 delimiter = ";"
3 keyIndex = 1
4 sequences = Parser(fasta, keyIndex=keyIndex, delimiter=delimiter)
```

The `fasta` variable on line 1 refers to the absolute path to the FASTA file you want to load. The absolute path to the FASTA file is the only required parameter for the parser. On line 2 we define the delimiter that is used to separate the individual components in each header line in the file. This will default to | if it is not provided. On line 3 we provide the `keyIndex` which defines the index in the delimited header where the unique primary key is located. The `keyIndex` will default to the value 0, which corresponds to the string that directly follows the `>` character in each header line.

For example, in Listing 9 the `keyIndex` value would be set to 1 as the index of the primary key on each line is 1, keeping in mind that we use zero-based indexing.

Listing 9: Primary key index example

```
1 > common1 primary_key1 common2 common3
2 ACGTGTGTAC
3 > common1 primary_key2 common3 common4
4 AGCTGGGTGTG
```

In this example we would also pass `delimiter = " "` to the parser since the default value for the delimiter parameter is | and is not valid for this FASTA example.

3.4.2 Working with the Parser Object

Picking up from the code defined in Listing 8 and the file defined in Listing 9 we will now go over how to access and edit the data stored within the `sequences` variable.

Listing 10: Basic data manipulation

```
1 sequences = Parser(fasta, keyIndex=keyIndex, delimiter=delimiter)
2 # Get a list of keys
3 keys = sequences.keys()
4 # Print each key and value
5 for key in keys:
6     print(key, sequences[key])
```

Running the code in Listing 10 should produce the output found in Listing 11. At a high level, we load the parser object instance into the sequences variable. From there we call a pre-defined API function called `.keys()` which provides us a list of existing keys in the current parser object. Finally we iterate through the keys using a basic `for...in` loop, and print the values stored in the Parser object. The sample output can be seen in Listing 11

Listing 11: Output from Listing 10

```
1 primary_key1 {"delimited":["common1", "common2", "common3"], "seq":"ACGTGTGTAC"}
2 primary_key2 {"delimited":["common1", "common3", "common4"], "seq":"AGCTGGGTGTG"}
```

Note that the structure of the dictionary output includes a special `"delimited"` key that contains a list of the remaining non-JSON delineated values in each given header. The actual structure of the parser object is that of a nested dictionary, with each primary key pointing to a dictionary that defines its annotations. We will now go over how to edit these dictionaries to add annotations and write out the edited structures into a valid FASTA file.

3.4.3 Writing FASTA

Picking up from where Listing 10 leaves off, we see in Listing 12 a sample of adding an annotation to one of the primary keys and writing the output back to a FASTA file.

Listing 12: Edit and write to file

```
# Add a new key:value pair indicating sequence edit
sequences["primary_key1"]["seqEdit"] = True
# Update the sequence
sequences["primary_key1"]["seq"] = "A"
# Write to file
outfile = "output.fa"
sequences.write(outfile)
```

The resulting FASTA file should look like Listing 13

Listing 13: Resulting FASTA from Listing 12

```
> common1 primary_key1 common2 common3 {"seqEdit":True}
A
> common1 primary_key2 common3 common4
AGCTGGGTGTG
```

In Listing 13 you can see how the written FASTA file now has an additional annotation stored in a JSON object for the first header, as well as an updated sequence value on the next line. It is important to note that the JSON object was added using the same delimiter as is specified during the initialization of the Parser object. This newly created file can once again be read by the Parser class, in the same way as the original file, and the resulting object will reflect the added annotation and sequence changes. We strictly follow the JSON standard for acceptable key and value types which defines only strings as the only valid key type while values have much more variety including lists, booleans, objects, numbers, and strings.[8]

3.4.4 Recommended Usage

Our primary use case for this parser is to read and iterate through sequence data with their corresponding annotations. It is important to note that adding new annotations to the sequence data is a computationally expensive process in order to maintain data integrity. In our previous examples for writing, in Listing 12, we provided basic syntax information on how to write assuming you were fully aware of the dynamics of your particular FASTA file.

In large and complex FASTA files with a variety of headers, like in Figure 1, updating the annotations and understanding which keys are protected and already in use is important to ensure backwards compatibility and safety. Let's assume a hypothetical situation where we want to add an annotation called "src" to each of the headers in the edges FASTA file defined in Figure 1, which tells us the lab from which the sequence data was sampled from. However, the edge FASTA file already has an annotation with the key "src" in some, if not all, of its headers. Overwriting this value could lead to a catastrophic loss of data and break a variety of pipelines and tools in which the file is currently being used.

Listing 14: Annotation writing example with safety checks

```
1  # Assuming we use the Edgefile from Figure 1
2  fasta = "EdgeFile_Chr1.fa"
3  delimiter = ";"
4  sequences = Parser(fasta, delimiter=delimiter)
5
6  # Checking for existing "src" annotation
7  keyDict = sequences.keyDict()
8
9  # keyDict returns a dictionary of existing keys and values
10 print("src" in keyDict) # Returns True meaning the "src" key already exists
11
12 # See what values look like for existing "src"
13 print(sequences.findInstances("src", limit=10)) # returns first 10 values
14
15 # Select new key: "lab" and do another test for availability
16 print("lab" in keyDict) # Returns False meaning "lab" key is unused
17
18 # Add annotation using "lab" key using method described previously
19 for element in sequences:
20     sequences[element]["lab"] = <val>
21
22 # Write out to file
23 sequences.write("update_with_lab.fa")
```

The FastAnnotation library provides helper methods that aid in the selection of appropriate and unused new keys. We will walk through a basic example in Listing 14. We start by loading the edges FASTA file found in Figure 1 (lines 1-4). Next we use the FastAnnotation library method `keyDict()` to get a dictionary of existing keys with their corresponding value types as their values (line 7). For example, an entry in this resulting dictionary for the edges FASTA file might be `{"src": "str"}` where `"src"` is the existing key value and `"str"` is its corresponding type in the dictionary. It is important to note that this is a computationally expensive step. As part of our parser we choose to de-serialize existing JSON headers at the last possible moment to speed up the loading and storing process, but when the `keyDict()` method is run, you incur the overhead of de-serializing all JSON in your parser that has yet to be de-serialized. Therefore, this method should only be used when you are planning on updating and adding new annotations to your FASTA file.

Now that we have the existing keys in hand, we can perform basic `in` checks to see if a potential new key already exists. We do this on line 16 and notice that `"src"` already exists, and as such a new key must be chosen. In case you are still curious about what the values for the existing `"src"` key look like we provide another helper method `findInstances(key, limit)` which takes in the key you are interested in as a parameter and the number of results you are interested in seeing and outputs a list of sample values found in the parser object (line 13). Note that our parser expects each independent key to have only 1 type of value associated with it and for clients to not overload a single key with multiple value types. Finally we decide on another key name and perform the same checks we performed for `"src"` and find that the key is available (line 16). We are now free to update our parser object as we did in our basic write example and write the new object back out to a FASTA file (lines 19-23).

4 Relevance to Genome Graphs and Pangenomes

A Pangenome, as defined by Marschall in his paper "Computational pan-genomics: status, promises, and challenges", is "any collection of genomic sequences to be analyzed jointly or to be used as a reference" [9][p.1].

You may notice that the definition of a pangenome does not define how these sequences are jointly analyzed and instead simply defines a new standard for analysis. One increasingly common implementation of a pangeome comes in the form of Genome Graphs.

4.0.1 Genome Graphs

A Genome graph is a representation of one or more genomes in the form of a graph with nodes and edges. We initially developed the FastAnnotation parser as a tool for constructing a genome graph implementation of a pangenome. In the Graphical Genome project [10] (currently under review) we define a genome graph that leverages the FastAnnotation parser and format to allow for the same generalized parser to read and write the same FASTA files through various iterations, annotation additions, and deletions in the development process. The Graphical Genome project uses nodes to represent conserved sequence among genomes and edges to represent divergence in their sequences. There exists 2 separate FASTA files for each chromosome which defines the nodes and the edges respectively. In Listing 1 you can see a fragment of a edge file. Node files are defined in the same way, but with slightly different annotations and naming conventions.

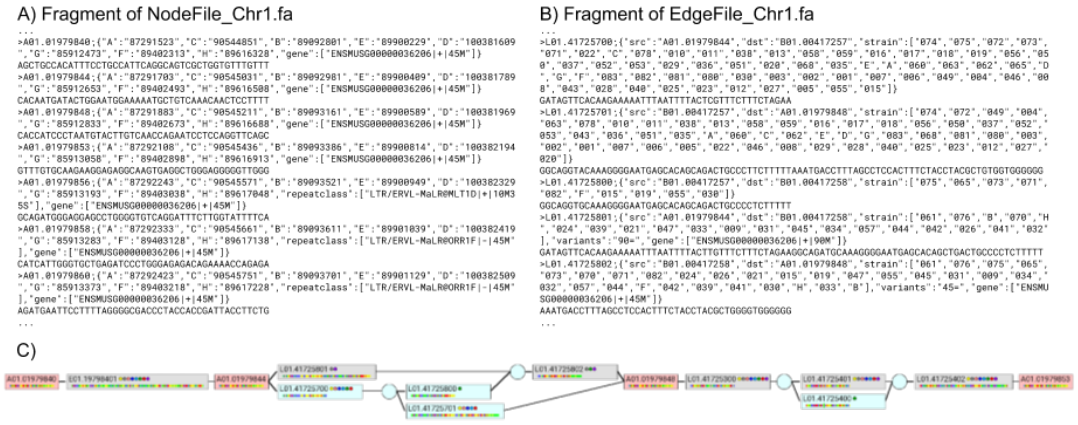

Figure 1: Graphical Genome Node/Edge FASTA example with visualization. Figure from currently under review Graphical Genome paper[10]. In section A and B we define a node and edge FASTA file using our new FASTA file extensions. Each header line has a single element in the delimited section which defines the primary key, followed by a JSON object defining the annotations for that specific sequence data. Section C shows a basic visualization of the node and edge files from section A and B with red rectangles/blue circles representing nodes and the blue/gray rectangles representing edges.

Each header line in Figure 1 is made up of the primary key followed by the annotation JSON. The annotations `src` and `dst` refer to the source and destination nodes that flank the given edge which provides information about the topology of the graph, while the other annotations provide biologically relevant information [10]. In the JSON object you can also see the `"gene"` annotation that defines alignment as an annotation that would normally be defined in a SAM file.

The `"variant"` annotation provides equivalent information from a VCF file [10]. We are able to merge these two disparate sources of information into the headers of the FASTA file to provide more utility with just the single FASTA file. Throughout the development of the genome, there were many times that edges and nodes needed to be moved, added, or deleted and without a generalized parser and file format this would have been impossible to efficiently do.

4.0.2 Relevance to FASTA extension

In order to do effective joint analysis on multiple sequences a significant amount of annotation information is required, and Su Hang, et al use an identical FASTA file format and parser to effectively store and update these annotations. They create 2 individual FASTA files for each chromosome, one for Nodes and one for Edges. Each file has headers that are `;` delimited with the primary key immediately following the `>` and the remaining annotations all existing within a JSON object [10].

5 Conclusion

In this paper we have defined a generalized, extendable parser for FASTA files that allow for the addition of new annotations and updates to existing FASTA files without breaking the parsing ability. We provide a basic API through which all of these actions may be completed with relatively low overhead and developer cost. We highlight how this idea has wide reaching relevance, especially in the field of Genome Graphs and Pangenomes where annotations are exceedingly important to comparing different sequences against each other in a single structure.

In the future we hope to extend this work to more file formats, the most obvious of which being FASTQ files, which are almost identical to FASTA files and store additional information in the form of quality scores. Our hope is that through eventual adoption, this generalized parser will allow for the easy creation of complex pipelines which modify and edit FASTA files freely, and are able to do so without worrying about future pipeline parsing errors.

References

[1] William R Pearson and David J Lipman. Improved tools for biological sequence comparison. Proceedings of the National Academy of Sciences, 85(8):2444–2448, 1988.

[2] Zhang Lab. What is fasta? Technical report, University of Michigan, https://zhanglab.ccmb.med.umich.edu/FASTA/.

[3] IUPAC nucleotide code. `https://www.bioinformatics.org/sms/iupac.html`.

[4] P Cock, C Fields, N Goto, M Heuer, and P Rice. The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. Nucleic acids research, 38(6):1767–1771, 2010.

[5] Anuj Srivastava, Andrew P. Morgan, Maya L. Najarian, Vishal Kumar Sarsani, J. Sebastian Sigmon, John R. Shorter, Anwica Kashfeen, Rachel C. McMullan, Lucy H. Williams, Paola Giusti-Rodríguez, Martin T. Ferris, Patrick Sullivan, Pablo Hock, Darla R. Miller, Timothy A. Bell, Leonard McMillan, Gary A. Churchill, and Fernando Pardo-Manuel de Villena. Genomes of the mouse collaborative cross. Genetics, 206(2):537–556, 2017.

[6] The SAM/BAM Format Specification Working Group. Sequence alignment/map format specification. Technical report, The SAM/BAM Format Specification Working Group, http://samtools.github.io/hts-specs/SAMv1.pdf, 2019.

[7] The SAM/BAM Format Specification Working Group. The variant call format (vcf) version 4.2 specification. Technical report, The SAM/BAM Format Specification Working Group, https://samtools.github.io/hts-specs/VCFv4.2.pdf, 2019.

[8] json.org. Introducing json. Technical report, json.org, https://json.org.

[9] T Marschall. Computational pan-genomics: status, promises and challenges. Briefings in Bioinformatics, 19(1):118–135, 2016.

[10] H Su, J Rao, Z Chen, M Najarian, J Shorter, F Manuel de Villena, and L McMillan. The collaborative cross graphical genome. July 2019.

A Appendix

The FastAnnotation repository contains all relevant code and documentation for the software defined in this paper.
FastAnnotation Repository: `https://github.com/jay1723/fastannotation`
The Graphical Genome repository contains all the documentation and code for the CCGG Graphical Genome project where my first version of the parser was implemented.
Graphical Genome Repository: `https://github.com/jay1723/GraphicalGenome`