KAnalyze Manual

v2.0.0

July 7, 2016

Contents

Chapter 1

Introduction

1.1 About K-mers

K-mers are short nucleotide sequences of a fixed length, k, and they are extracted from longer sequences. The first k-mer is the first k bases of the sequence. The second k-mer is k bases starting on the *second* base of the sequence. This process is repeated until the last base of the sequence appears in a k-mer.

Figure 1.1 shows an example of converting a short sequence to k-mers with $k = 4$. In reality, k-mers are much larger, such as 31 or 48.

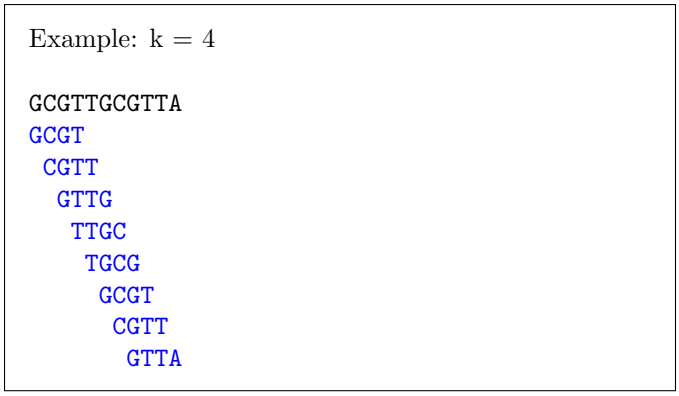

Figure 1.1: An example of extracting k-mers from a short sequence ($k = 4$). The short blue sequences are the k-mers extracted from the sequence in black.

Transforming sequences to k-mers is often used it implement genomic analysis algorithms that do not rely on sequence read alignments, or *mapping-free* approaches, and these algorithms are often many times faster than their alignment counterparts.

1.2 About KAnalyze

KAnalyze is a toolkit for transforming and handling sequence data as k-mers.

Its *command-line user interface* (CLUI) is designed to be both user-friendly and easily integrated into scripted pipelines. All of KAnalyze is build around an *application programming interface* (API) that can be plugged directly into programs. Additionally, there is a modular interface that allows it to be extended without changing the KAnalyze source code.

1.2.1 Components and Modules

Internally, KAnalyze is composed of *components* that perform specific tasks. For example, there is a component to turn sequence reads into k-mers, a component to write k-mers to output, and many other components that can rearrange and transform k-mers and data within the pipeline.

A *module* links components in a defined way to transform input to output. For example, A module might count k-mers in sequence data, or stream k-mers out in order. The same components, such as the component that turns sequences into k-mers, can be used for both.

This structure makes KAnalyze powerful for several reasons. These components can be run in different threads and operate in parallel, and multiple components can be allocated at one step of the pipeline. Also, the pipeline can be easily altered, which makes it possible to filter or transform data by inserting a new component into the pipeline. Of course, all these parts are reusable, and so entirely new pipelines can be developed with minimal effort.

1.3 System Requirements

Any computer with a Java 7 Java Virtual Machine (JVM) should be able to run KAnalyze, including Linux, MacOS, and Windows.

Minimum system requirements:

- Java 7
- 3 GB RAM

1.4 Command-Line Interface

The *command-line user interface* (CLUI) built into KAnalyze is designed to be user-friendly. There is extensive help built into the interface, and error messages are designed to be as descriptive as possible. It is also designed to integrate well into scripted pipelines with consistent semantics and well-defined return codes.

1.4.1 Basic Usage

There are three modules built into KAnalyze: one to count k-mers ("count"), one to stream k-mers in order ("stream"), and one to retrieve information for citing KAnalyze ("cite"). Usage for both is covered in this section, but general examples will use the count module.

All modules are built into the KAnalyze JAR file, and they can be run by executing the JAR directly. A script for each is also packaged with the software to make it easier to run them, and each script is the name of the module (count, stream, and cite).

If the JAR file is executed, the first argument must be the name of the module to be run. The only other valid value is the help option, `-h`. After the name of the module, all other arguments are given to the module, and these are the same arguments that would be used if the module script was used to run it.

Examples:

```
java -jar kanalyze.jar cite ref
```

```
cite ref
```

If the program is used in a Linux or UNIX environment, it may be necessary to place `./` before a command in the local directory that is not installed in a directory listed by the `PATH` environment variable.

```
./cite ref
```

1.4.2 Example Usage

A small file, `test.fa`, is included with the KAnalyze package. This file can be used to run quick examples of KAnalyze.

```
count -k 8 test.kc test.fa
```

The same command run directly by Java:

```
java -Xmx3G -jar kanalyze.jar count -k 8 test.kc test.fa
```

1.4.3 Getting Help

Extensive module help is built into the command-line interface, and it is accessed with the `-h` option or its long form, `--help`.

If KAnalyze is executed with the `java` command, then `-h` must come after the module name.

```
count -h
```

Help Topics

The `-h` will list all command-line options by default, but it can also be used to access help on special topics. Each topic is accessed by giving an argument to the help option, `-h`*topic_name* or `--help=`*topic_name*. The list of available topics is itself a topic, and it can be accessed with `-htopics` or `--help=topics`.

```
count -htopics
```

```
count --help=topics
```

For example, the *options* topic discusses the semantics of command-line options, including short options, long options, arguments, and optional arguments.

```
count -hoptions
```

```
count --help=options
```

1.4.4 Java Command

In most cases, KAnalyze modules should be run using one of the scripts packaged with it. When Java is executed, the maximum allowed memory is set, and this varies by system and Java installation. If the Java command is run and the maximum memory is not specified, then it may be allocated significantly less than it needs. The module scripts set the memory to a reasonable value, and so using them is recommended.

With command-line options, KAnalyze can be tuned to handle large amounts of data efficiently, and it will need more memory. In this case, the module script should not be used, and Java should be executed directly with the `-Xmx` option.

For example, to run the count module with 12 GB of data:

```
java -Xmx12G -jar kanalyze.jar count count_opts
```

Chapter 2

Count Module

2.1 About

The count module finds the frequency of each unique k-mer in all sequences.

Using default values, this module can process data of any size using 3 GB of memory. If k-mer sizes larger than 100 are used, then more memory may need to be allocated to it (Section 1.4.4).

2.2 Command-Line Options

2.2.1 Input and Output

Option	Argument	Description	Default
`--charset`	CHARSET	Character set encoding of input files. This option specifies the character set of all files following it. The default, "UTF-8", properly handles ASCII files, which is a safe assumption for most files. Latin-1 files with values greater than 127 will not be properly parsed.	UTF-8
`-c` `--countfilter`	FILTER_SPEC	Add a filter for k-mers by count. The argument to this option is a filter name followed by a colon (:) and any arguments to the filter. The built-in filter, "kmercount", can be specified with a count threshold. For example, "kmercount:2" and "2" will filter k-mers with a count less than 2.	
`-f` `--format`	FORMAT	Set the input sequence format type. This option determines how the format files are read. This option may be set multiple times when reading files with different formats. Use "count -hreader" to get a full list of readers.	auto
`--input`	SEQUENCE	Inputs a sequence directly from the command line. If the format is "auto", which is the default if `-f` was not given, the sequence string is treated as a raw sequence. Otherwise, the format of this content is determined by `-f`.	
`--kmerfilter`	FILTER_SPEC	Specify a k-mer filter. The argument to this option is a filter name followed by a colon (:) and any arguments to the filter. The built-in k-mer filter name is "kmerfile", and so "kmer-file:filterfile" will read k-mers from "filterfile", and only k-mers found in that file will be counted.	
`-k` `--ksize`	KSIZE	Set the k-mer size. May be any integer value.	31
`-o` `--out`	FILE	Set the output file name.	count.kc

-m --outfmt	FORMAT	Set the format of the output file. See `count -hwriter` for a list of available output formats.	seq
--quality	SEQ_FILTER	This option is an alias for `--seqfilter`.	
-r --reverse	<MODE>	Reverse complement k-mers as they are generated. Valid modes are "duplicate", "single", "canonical", and "forceduplicate". For "duplicate", the default mode if no mode is specified, original k-mers and their reverse complements are counted unless a k-mer is self-complementary. Mode "single" counts only reverse complements. Mode "canonical" counts either the original k-mer or its reverse complement, whichever comes first in ASCII sort order. Mode "forceduplicate" is like "duplicate", except it double-counts self-complementary k-mers. When specifying a mode, do not include a space after `-r` (e.g. `-rcanonical`).	single
-R --noreverse		Disable reverse complementing k-mers as they are counted.	TRUE
--seqfilter	SEQ_FILTER	Filter sequences as they are read and before k-mers are extracted from them. Some sequence readers can filter or alter reads at runtime. The most common filter is a quality filter where low-quality bases are removed. The filter specification is a filter name followed by a colon (:) and arguments to the filter. If a filter name is not specified, then the "sanger" quality filter is assumed. For example, "sanger:10" and "10" will filter k-mers where any base's quality score less than 10. The sequence filter specification is set for all files appearing on the command-line after this option. To turn off filtering once it has been set, files following "–noseqfilter" will have no filter specification.	
--noseqfilter		Turn off sequence filtering for all files following this option. If `--seqfilter` or `--quality` was specified, this option disables sequence filtering. These options together make it possible to specify filtering for some files and disable filtering for others.	TRUE
--stdin		Read sequences from standard input. The format is determined by how `-f` (`--format`) is set on the commmand line before this option is encountered. The character set and sequence filter are also set by `--charset` and `--seqfilter` (or `--quality`). If the format is "auto" (or not set), then "raw" is assumed.	
--nostdin		Do not read sequences from standard input. This option reverses `--stdin`.	TRUE
--stdout		Write output to standard output instead of a file. Unless redirected, this output is written to the the screen.	

2.2.2 Performance Tuning

Option	Argument	Description	Default
--dumpseg		Dump segment files to disk.	TRUE
--nodumpseg		Keep all segments in memory and do not offload them to disk. When enough memory is available, this can speed analysis significantly. Using this option will likely cause out of memory errors.	
-d --kmerthreads	THREADS	Set the number of threads to use for the pipeline k-mer step. This step converts sequence reads into k-mers.	1
--kmerbatchsize	BATCH_SIZE	Set the size of k-mer batches. This is the number of k-mers or k-mers with counts that are passed between components in a single batch.	10000
--kmercachesize	CACHE_SIZE	Set the number of k-mer and k-mer count batches to be cached while components are not using them.	500
--kmerqueuesize	QUEUE_SIZE	Set the size of queues containing k-mer and k-mer count batches as they are passed between companents.	150

-g --segsize	SEG_SIZE	Size of k-mer segments stored in memory. This many k-mers are accumulated into an array in memory. When the array is full, it is sorted, written to disk, and cleared for the next set of k-mers. This is a performance tuning parameter that does not need to be set for the majority of data sets. If a value is chosen that is higher than the default, then more than 2GB of memory will be required. See the KAnalyze manual for more information on performance tuning. This number may be in decimal, hexadecimal, or octal. An optional multiplier (k, m, or g) may included after the number. For the maximum value, use keyword "max". See the KAnalyze manual usage documentation for more information on acceptable formats.	26843546
-l --splitthreads	THREADS	Set the number of threads to use for the pipeline split step of the count pipeline.	1
--segqueuesize	QUEUE_SIZE	Set the size of queues containing segments as they are passed between the count split and merge companents.	20
--seqbatchsize	BATCH_SIZE	Set the size of sequence batches. This is the number of sequence bases that are passed between components in a single batch.	10000
--seqcachesize	CACHE_SIZE	Set the number of sequence batches to be cached while components are not using them.	250
--seqsourcequeuesize	QUEUE_SIZE	Set the size of queues containing sequence sources waiting for the reader component.	50
-t --threads		Set the number of concurrent threads KAnalyze should use.	4

2.2.3 Other

Option	Argument	Description	Default
-h --help	<TOPIC>	Get command-line help. If "TOPIC" is specified, then help on a specific topic is shown. Try --help=topics (or -htopics) for a list of topics.	
-p --property	KEY=VALUE	Set a key/value property. These properties can alter the behavior of components that are dynamically loaded into KAnalyze.	
-s --splitonly		Read files, split k-mers into sorted segments, but do not merge and do not delete the segment files. This option is useful for creating split segments on a distributed system, but merging must take place on the same machine.	
-S --nosplitonly		Perform splitting and merging in the same operation.	TRUE
-v --verbose		Output extra information.	
-V --noverbose		Disable extra information output.	TRUE
-x --autodelete		Automatically delete segment (temporary) files generated by the split component after merging.	TRUE
-X --noautodelete		Do not automatically delete segment files.	
--lib	LIB	Load a library packaged in a JAR file. Custom components, such as output writers or input readers, can be defined externally and imported as a library.	
--liburl	URL	Load a library from its URL. Custom components, such as output writers or input readers, can be defined externally and imported as a library.	
--minmask	MASK	While determining the minimizer of a k-mer, sub-k-mers are XORed with this mask while comparing them, but the minimizer is not XORed (it is still a sub-k-mer of the original k-mer). This option can be used to break up large minimizer groups due to low-complexity k-mers when minimizers are used. A value of 0 disables the mask.	0

`--minsize`	SIZE	Size of minimizers or 0 to disable processing by minimizers.	
`--temploc`	<LOCATION>	The location where segments are offloaded. This argument must be a directory or the location for a new directory. Parent directories will be created as needed. If not set, temporary files are placed in the same directory as the output file or the current directory if there is no output file.	
`--notemploc`		Use the output file directory as the location for temporary files.	TRUE
`--trace`		Report a stack trace with errors and warnings. This is a debugging option, and it should not normally be turned on.	
`--notrace`		Do not report a stack trace with errors and warnings.	TRUE

2.2.4 Examples

`java -jar kanalyze.jar count -k 8 -o counts.kc -f fasta test.fa`
Reads sequences from test.fa, counts all 8-mers, and writes k-mer counts to counts.kc.

`java -jar kanalyze.jar count -k 8 -o counts.kc -m hex -f fasta test.fa`
Reads sequences from test.fa, counts all 8-mers, and writes k-mer counts to counts.kc as hexadecimal strings.

2.3 Pipeline Structure

The count module counts all occurrences of each unique k-mer in the input sequence data and outputs a list sorted by k-mer. Like all modules, it is implemented as a pipeline of components, and each component performs a specific task. Figure 2.1 shows the basic structure of this module.

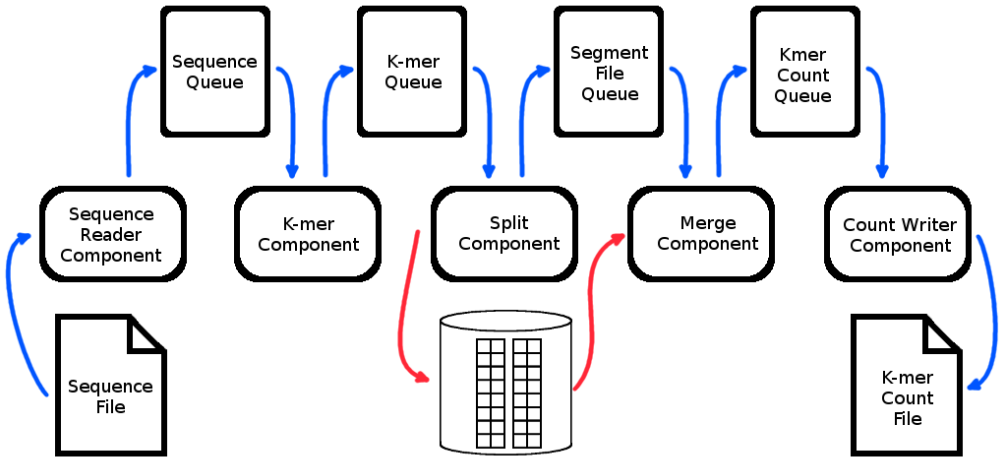

Figure 2.1: Structure of the components in the count pipeline. Data is read from one or more sequence files by a reader component, then it is transformed to k-mers. A memory buffer ("segment") is filled, sorted, counted, and offloaded to disk. When all data has been written to a segment file, they are merged to produce the final output.

Components are connected through memory queues. To avoid excessive synchronization overhead, batches of elements are passed through these queues. After a batch is consumed, it is reused instead of destroyed and reallocated.

The split and merge components are responsible for transforming the stream of k-mers into k-mer counts. The split component accumulates k-mers in a memory queue, which KAnalyze refers to as a segment *segment*. The segment is then sorted, the k-mers are counted, and the result is offloaded to a segment file *segment file*. Once all k-mers are in a segment file, the merge component merges them into the final k-mer counts.

Some of the options directly affect the pipeline structure. For example, filtering k-mers before counting inserts a filter step after the k-mer component. Filtering by k-mer count inserts a filter after the merge component. Changing the number of threads dedicated to a component creates multiple components to handle that step.

2.4 Performance Tuning

Tuning the performance of the count module requires a basic understanding of the count module structure (Section 2.3), a list of performance tuning options (Section 3.2.2), and typically, more memory than the default 3 GB (Section 1.4.4). An understanding of the system resources is also necessary, such as the number of CPU cores and amount of memory.

The majority of the CPU time in the count module will be spent sorting the segments in memory. The sorting algorithm used is a dual-pivot quicksort that is distributed over multiple threads, and the number of threads allocated is set by `--threads`. Increasing the threads to a number at or close to the number of CPU cores available should maximize sorting performance.

Increasing the segment size (`--segsize`) can improve performance by reducing the number of segments that need to be merged. The optimal size will be machine dependent. This value cannot exceed the maximum value of a 32-bit 2's complement integer ($2^31 - 1$).

If the segment size does not fill memory, then more than one split component may be allocated (`--splitthreads`). While one split component is sorting the segment, another can be reading k-mers. Setting this to a value higher than 2 may not increase the performance.

The size of batches passed among components, the size of the queues that store those batches, and the size of caches that hold batches as they are recycled can also be tuned. This may not have a noticeable impact on performance, but it may help tune KAnalyze to run in a very memory-restricted environment.

Chapter 3

Stream Module

3.1 About

The stream module outputs k-mers in the order they are found in the input sequences.

3.2 Command-Line Options

3.2.1 Input and Output

Option	Argument	Description	Default
`-f` `--format`	FORMAT	Set the input sequence format type. This option determines how the format files are read. This option may be set multiple times when reading files with different formats. Use "count -hreader" to get a full list of readers.	auto
`-k` `--ksize`	KSIZE	Set the k-mer size. May be any integer value.	31
`-m` `--outfmt`	FORMAT	Set the format of the output file. See `count -hwriter` for a list of available output formats.	seq
`--charset`	CHARSET	Character set encoding of input files. This option specifies the character set of all files following it. The default, "UTF-8", properly handles ASCII files, which is a safe assumption for most files. Latin-1 files with values greater than 127 will not be properly parsed.	UTF-8
`--index`		Write location of k-mer in the sequence with each k-mer. The first k-mer of the first sequence is "1".	
`--noindex`		Write only the k-mer and not the k-mer location in the sequence.	TRUE
`--seqfilter`	SEQ_FILTER	Filter sequences as they are read and before k-mers are extracted from them. Some sequence readers can filter or alter reads at runtime. The most common filter is a quality filter where low-quality bases are removed. The filter specification is a filter name followed by a colon (:) and arguments to the filter. If a filter name is not specified, then the "sanger" quality filter is assumed. For example, "sanger:10" and "10" will filter k-mers where any base's quality score less than 10. The sequence filter specification is set for all files appearing on the command-line after this option. To turn off filtering once it has been set, files following "–noseqfilter" will have no filter specification.	

`--noseqfilter`		Turn off sequence filtering for all files following this option. If `--seqfilter` or `--quality` was specified, this option disables sequence filtering. These options together make it possible to specify filtering for some files and disable filtering for others.	TRUE
`--stdin`		Read sequences from standard input. The format is determined by how `-f` (`--format`) is set on the commmand line before this option is encountered. The character set and sequence filter are also set by `--charset` and `--seqfilter` (or `--quality`). If the format is "auto" (or not set), then "raw" is assumed.	
`--nostdin`		Do not read sequences from standard input. This option reverses `--stdin`.	TRUE
`--stdout`		Write output to standard output instead of a file. Unless redirected, this output is written to the the screen.	

3.2.2 Performance Tuning

Option	Argument	Description	Default
`--kmerbatchsize`	BATCH_SIZE	Set the size of k-mer batches. This is the number of k-mers or k-mers with counts that are passed between components in a single batch.	10000
`--kmercachesize`	CACHE_SIZE	Set the number of k-mer and k-mer count batches to be cached while components are not using them.	500
`--segqueuesize`	QUEUE_SIZE	Set the size of queues containing segments as they are passed between the count split and merge compenents.	20
`--seqsourcequeuesize`	QUEUE_SIZE	Set the size of queues containing sequence sources waiting for the reader component.	50

3.2.3 Other

Option	Argument	Description	Default
`-h` `--help`	<TOPIC>	Get command-line help. If "TOPIC" is specified, then help on a specific topic is shown. Try `--help=topics` (or `-htopics`) for a list of topics.	
`-p` `--property`	KEY=VALUE	Set a key/value property. These properties can alter the behavior of components that are dynamically loaded into KAnalyze.	
`-v` `--verbose`		Output extra information.	
`-V` `--noverbose`		Disable extra information output.	TRUE
`--lib`	LIB	Load a library packaged in a JAR file. Custom components, such as output writers or input readers, can be defined externally and imported as a library.	
`--liburl`	URL	Load a library from its URL. Custom components, such as output writers or input readers, can be defined externally and imported as a library.	

3.2.4 Example Usage

`java -jar kanalyze.jar stream -k 8 -o stream.km test.fa`
Converts test.fa to 8-mers and writes each 8-mer to stream.km

`java -jar kanalyze.jar stream -k 8 -o stream.km --index test.fa`
Converts test.fa to 8-mers and writes each 8-mer to stream.km with a location for each k-mer

Chapter 4

Cite Module

4.1 About

Outputs citation information for KAnalyze.

4.2 Usage

Run `cite ref` for short reference text and `cite bibtex` for a bibtex entry. The default is "ref".

4.3 Reference Text

```
Audano, P., & Vannberg, F. (2014). KAnalyze: a fast versatile pipelined K-mer toolkit.
Bioinformatics (Oxford, England), 30(14), 20702. doi:10.1093/bioinformatics/btu152
```

Chapter 5

Supplementary Information

5.1 Nucleotide Characters

KAnalyze recognizes the IUPAC codes[1] as valid nucleotide characters. To this set, it adds "X" for any base and "U" for RNA sequences. The full set is "ACGTRYKMSWBDHVNUXacgtrykmswbdhvnux.-", and KAnalyze will complain about input sequences that contain characters other than these.

5.2 Input File Formats

5.2.1 AUTO

The default file format is "AUTO", and KAnalyze will attempt to guess the input format based on the file name.

FASTA files are recognized by file names that end with ".fasta", ".fa", ".fna", or ".fas". FASTQ files end with ".fastq", ".fq", or ".fnq". If any of these file names end with ".gz" after the format, they are read as a GZIP compressed version of that file (e.g. ".fastq.gz").

Custom readers can be loaded at runtime, and the reader may tell KAnalyze what file name patterns it recognizes. Therefore, AUTO will work to resolve file types for these custom readers.

5.2.2 FASTA

FASTA files are compatible with the FASTA format acceptable by NCBI BLAST[2]. The only exception is that the reader permits blank lines within the sequence record. A single-line description beginning with '>' precedes each sequence record. The sequence record may be on a single line or split over any number of lines, and there are no restrictions on the line length.

5.2.3 FASTAGZ

A GZIP compressed FASTA file. The uncompressed file contents must follow the FASTA format defined in Section 5.2.2.

[1] http://www.bioinformatics.org/sms/iupac.html
[2] http://www.ncbi.nlm.nih.gov/BLAST/blastcgihelp.shtml

5.2.4 FASTQ

FASTQ files must be compatible with the format published in Nucleic Acid Reseach[3]. The first line of a record begins with '@'. The sequence may be split over multiple lines. A line beginning with '+' will separate the sequence from the quality scores. Quality scores are read line-by-line until the number of quality score characters is the same as the number of sequence characters read for any given record.

5.2.5 FASTQGZ

A GZIP compressed FASTQ file. The uncompressed file contents must follow the FASTQ format defined in Section 5.2.4.

5.2.6 RAW

Raw sequence files are the simplest record type. A sequence may be split over any number of lines. Blank lines separate records.

5.3 Kmer Output Formats

There are several output formats built into KAnalyze, and new ones may be loaded at runtime.

5.3.1 Sequence Format

K-mers are written as a string of A, C, G, and T. Each k-mer has k letters. Note that if RNA sequences were read, U will be converted to T when sequences are output in this format.

5.3.2 Decimal Format

K-mers are output using the base-10 decimal representation of k-mers.

5.3.3 Hexadecimal Format

Tab delimited file of k-mers and counts using the base-16 decimal representation of k-mers.

5.3.4 FASTA Format

FASTA representation of k-mers native to early versions of Jellyfish (Marçais and Kingsford, 2011). Each FASTA record is a single k-mer, and the description line is the k-mer count or index.

5.3.5 Indexed K-mer Count

An *indexed k-mer count* (IKC) file allows a record of k-mers to be rapidly searched without loading the whole file into memory, which is necessary for a hash table. It accomplishes this because it can group k-mers by minimizers.

A minimizer is found by taking a k-mer and its reverse-complement and finding all sub-k-mers of it. Typically, the sub-k-mer size is 15. If all the sub-k-mers are sorted, the first one in the sort order is defined as the minimizer for that k-mer. The next k-mer in a sequence will share all but 1 base with the previous k-mer, and therefore, the minimizer is likely the same. Minimizers group k-mers together that likely occur together within a sequence. This fact can be

[3] http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2847217/

exploited by memory-mapped files, which load pages of a file into memory as needed. Therefore, when searching for k-mers sequentially in a sequence, this structure can query k-mer records almost as fast as storing them in memory, like hash table, except only pages of the file are loaded as needed, and when a page is no longer needed, its memory is freed.

Figure 5.1 gives the structure of an IKC file.

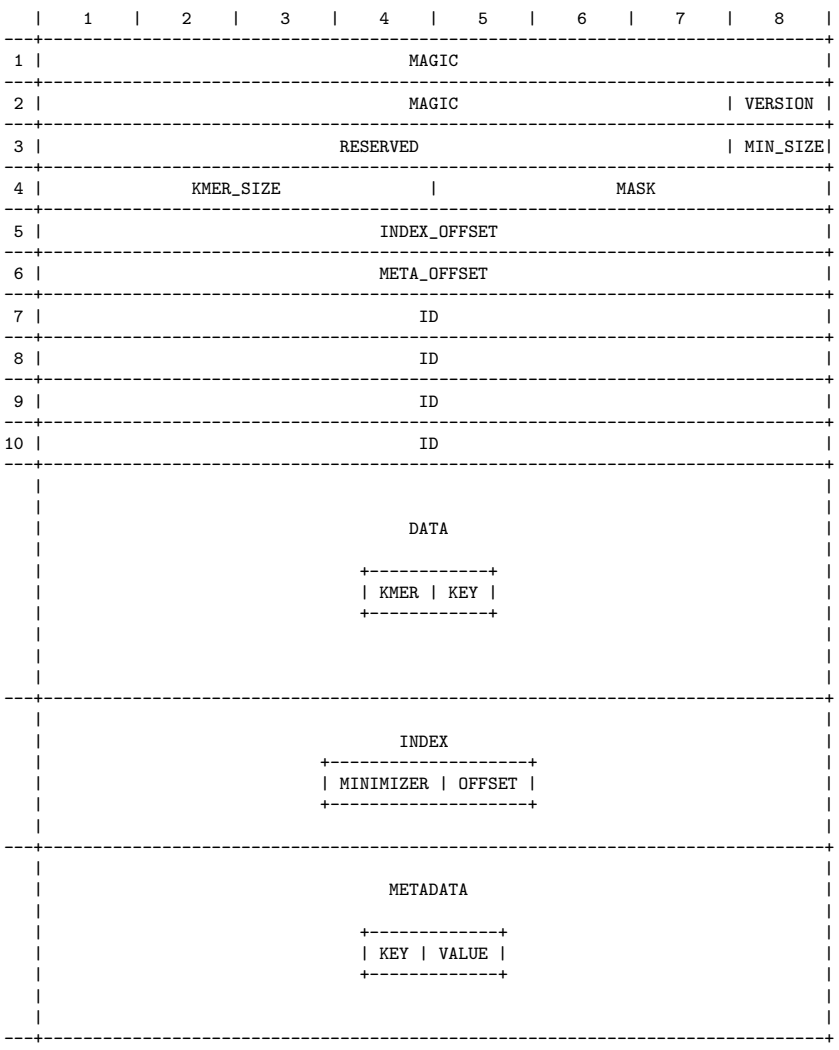

Figure 5.1: Structure of an IKC file.

MAGIC (15 bytes) The NULL terminated string "Idx_Kmer_Count" (UTF-8/ASCII enceded). This is invariant and must be the first 15 bytes in every indexed k-mer count file, otherwise, a reader should throw an error immediately. The string magic number makes it easy to inspect the file (hex editor, strings utility, etc.) and determine what it is.

VERSION (1 byte) Version of this file as a two's complement integer. For current implementation, this is always 1. The value must be positive.

RESERVED (7 bytes) Bytes reserved for future expansion without re-arranging the header. It is placed in this location to allow an expansion of MIN_SIZE in case larger minimizers are ever supported. If future versions support flags in the header, they should be located at the lower end of this range.

MIN_SIZE (1 byte) Size of the minimized k-mers as a two's complement integer. As of file version 1, this field must be between 1 and 15 (inclusive).

KMER_SIZE (4 bytes) Size of k-mers as a two's complement integer. This field must not be negative.

MASK (4 bytes) Minimizer mask. All potential minimizers of a k-mer (sub-k-mers of the k-mer and its reverse

complement) are XORed with this mask, and the lesser of all is chosen. The minimizer itself is not XORed, but the XORed form of it is used to select the minimizer. This can be useful for breaking up large minimizer groups resulting from k-mers with low complexity.

INDEX_OFFSET (8 bytes) Offset of the file where the index section begins as a two's complement integer. See "INDEX" below.

METADATA_OFFSET (8 bytes) Offset of the file where the metadata section begins. See "METADATA" below.

DATA A collection of k-mer/key records sorted by minimizer and then by k-mer. This sort order preserves grouping by minimizers, and all k-mers in a minimizer are sorted for fast searching (i.e. binary search). Each k-mer/key record starts with the bytes of a k-mer, which varies by k-mer size. For example, k-mers of 13, 14, 15, and 16 require 4 bytes to encode (4 bases per byte). The key is normally the k-mer count, but it can be the key for other data, such as a phylogenetic association (see "METADATA"). This key is always a 4-byte two's complement integer, and it may not be negative.

INDEX A sorted list of index records associating a minimizer with a file offset that points to the data section where the minimizer group begins. Each of these records consists of the minimizer (4 byte two's complement integer) and an offset (8 byte two's complement integer). Neither the minimizer nor the offset may be negative.

METADATA (optional)] Normally, the keys in the data section are counts associated with k-mers, however, the data section may associate k-mers with other metadata. If this section is present, it implies that the k-mers are associated with metadata, and data associated with each key is defined in this section. Note that the format of the data section is not altered, but the presence of this section changes how it may be interpreted. Each record in this field consists of a key (4 byte two's complement integer) and a NULL-terminated UTF-8 encoded string. For each k-mer in the DATA section with key K, the METADATA for that k-mer is encoded in this section with key K. Note that each key may only be associated with one metadata string, but this string may take any form (for example, a list of attributes). This section should be sorted by key. If a key appears more than once, the last one takes precedence and the others should be discarded.

5.4 Command Line Return Codes

The KAnalyze user interface always returns a well-defined code. When executed from a script environment, this return code can easily be checked to see if KAnalyze completed normally or not. Each return code is defined in this section.

In the following list of return codes, the numeric code is listed. For API users, the constant defined in `edu.gatech.kanalyze.Const` is also listed.

0 (ERR_NONE) The program terminated normally. All k-mers were successfully processed without error, or the help option was invoked.

1 (ERR_USAGE) Command line arguments were incomplete, improperly formatted, or required arguments were missing.

2 (ERR_IO) An I/O (input/output) error occurred reading or writing data. This error is normally returned for file I/O errors.

3 (ERR_SECURITY) A security error, such as permissions denied, occurred.

4 (ERR_FILENOTFOUND) A required or specified file was not found.

5 (ERR_DATAFORMAT) Data in an input file is improperly formatted.

6 (ERR_ANALYSIS) Some un-recoverable error occurred during analysis.

7 (ERR_INTERRUPTED) A program thread was interrupted while it was running. This would normally be returned if the program is terminated before it completes.

8 (ERR_LIMITS) Some pre-set limit was exceeded.

9 (ERR_EXECTERM) Execution was terminated prematurely for some unknown reason. This occurs when a step in a pipeline terminates unexpectedly.

98 (ERR_ABORT) The program or a process was terminated before it completed. When the action is requested or expected, this should be returned in lieu of ERR_INTERRUPTED even if interrupting threads was necessary.

99 (ERR_SYSTEM) Some serious unrecoverable system error occurred. These errors are almost certainly program bugs. Some other unrecoverable errors, such as running out of memory, could return this code.

5.5 Building From Source

The source is distributed with an Apache Ant build file with multiple targets. To build these, Apache Ant must be installed.

To run a target, enter `ant <target>`. Multiple targets may be supplied, for example `ant clean package`.

The most commonly used targets are "clean", "compile", and "package". The clean removes all temporary files. It is not often necessary, but it should be executed before building a package for testing to ensure no artifacts are left behind from previous builds. The compile target compiles the class files and does nothing else. The package target generates the JAR file and distributable packages.

The "doc.javadoc" target generates the Javadoc pages from KAnalyze source comments. See Section 5.5.1 for more information about these pages. "doc.manual" generates this manual as a PDF file.

The "test" target executes unit tests packaged with KAnalyze. See Section 5.6 for more information on unit testing.

Other targets are called by the targets described above and are not often called directly.

For a full list of targets and brief descriptions, enter `ant -projecthelp`.

5.5.1 Building Javadoc Pages

Javadoc is a very powerful tool for documenting APIs written in Java. Section 6.1 introduces the concept and the KAnalyze rules governing these comments.

Javadoc comments begin with "/**", and end with "*/". In KAnalyze, these comments appear before every class, method, and field. The comment starts with a description of what the element does. For methods, it documents parameters and the conditions under which all exceptions are thrown. Full documentation for writing these comments can be found online[4].

The ant build system (Section 5.5) generates web pages from these comments. The "doc.javadoc" target builds two sets of pages. One, the API documentation, documents all elements available on the API. This target is intended for developers who are extending or using the API. The other, FULL documentation, documents everything including private members that are not available to the API. This is meant for KAnalyze maintenance programmers.

After running the javadoc Ant target, the API documentation can be access by loading "build/doc/javadoc/api/index.html" in a browser (relative to the project root). The FULL documentation can be access by loading "build/doc/javadoc/full/index.html" in a browser (relative to the project root).

5.6 Running Unit Tests

KAnalyze is distributed with a set of unit tests. As changes are made to the system, existing components can be quickly re-tested. These tests are intended to be thorough. For example, the unit tests for the k-mer component tests all valid k-mer sizes. Some of the test cases use up to 4GB of memory.

Tests are easily executed with "ant test". This target compiles the test classes and executes all of them. Note that test classes are not compiled by the "compile" target and are not distributed with compiled packages.

[4] http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Chapter 6

API

All of KAnalyze is designed around an API (Application Programming Interface) that can be driven from other programs. The CLI (Command Line Interface) itself is a simple application that calls the API to carry out tasks. Since KAnalyze is written in Java, the API is implemented as a set of Java classes with well documented interfaces.

6.1 Javadoc Documentation

The KAnalyze API is fully documented with Javadoc comments. Every class and class member (method and field), regardless of scope, has a Javadoc comment. All method arguments, return values, and exceptions are contained in the method's Javadoc comment. For any method arguments that are objects, the comment must disclose what happens when `null` values are received for that field (either by the argument's comment, or the comment on the exception if one is thrown). All exceptions, whether or not they are runtime exceptions, must be documented along with the conditions that cause them to be thrown. Assertion errors are not documented. Any deviations from these rules should be reported as a bug.

Javadoc comments can be converted into a set of HTML pages that document the code. See Section 5.5.1 for details.

6.2 Project Organization

The KAnalyze project is organized into several packages, which are all sub-packages of `edu.gatech.kanalyze`.

The main packages are:

edu.gatech.kanalyze.batch Data is passed from one pipeline to another through batches of elements, and the classes in this package implement the batch structures as well as batch management code.

edu.gatech.kanalyze.comp Components that are used to implement in-memory pipelines. Most pipeline components take input from one component, process it, and output to another component. Linking them together forms a pipeline in memory where each pipeline has its own thread. The bulk of the work performed by KAnalyze is done by these components.

edu.gatech.kanalyze.concurrent Classes for managing concurrent processes. Most of this code is built into other components, but this package contains an interface for items that can be stopped.

edu.gatech.kanalyze.condition Components of a built-in messaging system. Since the API may be used as part of a larger system, any messages are inserted into this system, the API client can subscribe to it, and so errors and warnings may be handled appropriately.

edu.gatech.kanalyze.ifs General-purpose interfaces that may be used by many parts of the system.

edu.gatech.kanalyze.io IO classes used by multiple parts of the KAnalyze system.

edu.gatech.kanalyze.module Modules form the major commands that are run on the command line. Modules load and link components in a pre-defined way. They can be thought of as complete pipelines, and the command line interface runs these modules.

edu.gatech.kanalyze.test JUnit test packages. To run tests, see section 5.6.

edu.gatech.kanalyze.util General purpose utility classes run by many parts of the KAnalyze system.

6.3 Components

Components are the pipeline elements that are connected together to form a complete module. They are designed to read and write from synchronized queues. Each component instance should run in its own thread, and downstream components can process data from upstream components as it becomes available. The pipeline architecture is the key to the flexibility of KAnalyze.

Reading and writing individual elements from synchronized queues as they are passed from one element to another is inefficient. The function calls are synchronized, so repetitive locking and unlocking destroys performance. To address this limitation, most components pass batches of elements through the queues. As a component generates data, it saves it to a batch object with a fixed capacity. When the batch object is full, it is passed to the queue and a new batch is retrieved. The component that consumes a batch sends it back to a batch cache to avoid the overhead of deallocating and reallocating memory for batch objects.

All components extend `edu.gatech.kanalyze.comp.KAnalyzeComponent`. Component objects are created with their constructor and run with the parent class's `run()` method. Most components take two syncrhonized queues in their constructor (one for input, and one for output), and when `run()` is called, it begins reading from the input queue and writing to the output queue.

Input and output queues are parameterized instances of `edu.gatech.kanalyze.util.BoundedQueue<T>`. When reading from an empty queue, a component will wait for data to be added to the queue. When there is no more data, pipeline management code will shutdown the queue, and when a shutdown queue becomes empty, it returns `null` to indicate that there is no more data to be read.

6.3.1 Count Merge Component

Input source: `edu.gatech.kanalyze.util.BoundedQueue<Segment>`

Output target: `edu.gatech.kanalyze.util.BoundedQueue<KmerCountBatch>`

Class: `edu.gatech.kanalyze.comp.count.CountMergeComponent`

Complements the count split component (Section 6.3.2). Segment files created by the split component are merged and final k-mer counts are sent to the output queue.

6.3.2 Count Split Component

Input source: `edu.gatech.kanalyze.util.BoundedQueue<KmerBatch>`

Output target: `edu.gatech.kanalyze.util.BoundedQueue<Segment>`

Class: `edu.gatech.kanalyze.comp.count.CountSplitComponent`

Complements the count merge component (Section 6.3.1). K-mers are received from the input queue and accumulated into a memory buffer until it is full. The memory buffer is sorted, and k-mer counts are written to a file on disk ("segment file"). The file location is sent on the output queue for the merge component. The memory buffer is cleared, and it repeats the process until all k-mers have been written to segment files.

6.3.3 Discard Component

Input source: `edu.gatech.kanalyze.util.BoundedQueue<? extends Batch> queue`

Output target: None

Class: `edu.gatech.kanalyze.comp.discard.DiscardComponent<E extends Batch>`

Discards batches. This is used to cap dead-ends in a pipeline and prevent dead-locks. The pipeline component can monitor this component like any other to determine when to terminate a step (Section 6.3.7).

6.3.4 Filter Component

Input source: `edu.gatech.kanalyze.util.BoundedQueue<T>`

Output target: `edu.gatech.kanalyze.util.BoundedQueue<T>`

Class: `edu.gatech.kanalyze.comp.filter.FilterComponent<T>`

Applies filters to elements. A filter loads a separate class to implement the filter logic.

6.3.5 K-mer

Input source: `edu.gatech.kanalyze.util.BoundedQueue<SequenceBatch>`

Output target: `edu.gatech.kanalyze.util.BoundedQueue<KmerBatch>`

Class: `edu.gatech.kanalyze.comp.kmer.KmerComponent`

Processes sequence reads from the input queue, k-merizes them, and writes k-mers to the output queue.

6.3.6 K-mer Writer

Input source: `edu.gatech.kanalyze.util.BoundedQueue<T extends KmerBatch>`

Output target: N/A

Class: `edu.gatech.kanalyze.comp.kmerwriter.KmerWriterComponent<T extends KmerBatch>`

Reads k-mers from batches and writes them to output. Classes that extend `KmerWriter` handle where to send the output and how it should be formatted.

6.3.7 Pipeline Component

Input source: N/A

Output target: N/A

Class: `edu.gatech.kanalyze.comp.pipeline.PipelineComponent`

Manages a pipeline of components. This is unlike other modules because it does not read from or write to synchronized queues. Instead, it handles all the logic needed to execute a pipeline, such as starting components, shutting down queues when components complete, and terminating all components when any part of the pipeline fails. The reason this is implemented as a component is so that pipelines can be inserted into other pipelines. For example, k-mer counting could be part of a much larger pipeline, and since the count module is a pipeline, none of the configuration steps performed by the count module need to be duplicated.

6.3.8 List Read Component

Input source: `java.util.List<? extends E>`

Output target: `edu.gatech.kanalyze.util.BoundedQueue.BoundedQueue<E>`

Class: `edu.gatech.kanalyze.comp.queuelist.ListReadComponent<E>`

Read from a list and insert elements into a queue.

6.3.9 List Write Component

Input source: `edu.gatech.kanalyze.util.BoundedQueue.BoundedQueue<E>`

Output target: `private final List<E>`

Class: `edu.gatech.kanalyze.comp.queuelist.ListWriteComponent<E>`

Read from a queue and insert elements into a list.

6.3.10 Queue Merge Component

Input source: `edu.gatech.kanalyze.util.BoundedQueue.BoundedQueue<T>`

Output target: A priority list of `edu.gatech.kanalyze.util.BoundedQueue.BoundedQueue<E>`

Class: `edu.gatech.kanalyze.comp.queuemerge.QueueMergeComponent<T>`

Reads from multiple queues and transfers batches to a single queue. This can be used to merge branches of a pipeline, such as input from different types of sources.

6.3.11 Reverse Complement

Input source: `edu.gatech.kanalyze.util.BoundedQueue<KmerBatch>`

Output target: `edu.gatech.kanalyze.util.BoundedQueue<KmerBatch>`

Class: `edu.gatech.kanalyze.comp.rcompl.RevComplComponent`

Reverse-complements k-mers from the input queue and writes to the output target. There are four modes (enum `edu.gatech.comp.rcompl.RevComplMode`).

Modes:

`RevComplMode.DUPLICATE` Outputs the original k-mer and its reverse complement. If a k-mer is its own reverse complement, it is only output once. Most applications requiring reverse-complement will use this mode.

`RevComplMode.FORCEDUPLICATE` Outputs the original k-mer and its reverse complement. If a k-mer is its own reverse complement, it is output twice (once for the original k-mer, and once for the reverse-complement).

`RevComplMode.LESSER` Takes the reverse complement and outputs the lesser of the original or reverse complement where lesser is defined by the sort order of k-mers. This mode may be useful for storing a set of k-mers and their reverse complements in about half the space required to store both.

`RevComplMode.SINGLE` Outputs the reverse complement of k-mers. The original k-mer is discarded.

6.3.12 Reader

Input source: `edu.gatech.kmer.util.BoundedQueue<SequenceSource>`

Output target: `edu.gatech.kmer.util.BoundedQueue<SequenceBatch>`

Class: `edu.gatech.kanalyze.comp.reader.ReaderComponent`

Reads a queue of sequence sources, extracts sequence strings, and writes sequence read objects to the output queue.

Classes that extend `edu.gatech.kanalyze.comp.reader.SequenceReader` process the data, and this component provides a framework for handling the sequence data as it flows through the desired reader.

Filters in package `edu.gatech.kanalyze.comp.readfilter` may be applied to the sequence data by this component. For example, to quality filter FASTQ files.

6.4 Utility Classes

KAnalyze has several utility classes for shared functionality. Most of the classes are used by multiple components or modules.

All of these classes are found in package `edu.gatech.kanalyze.util`

6.4.1 Argument Parser

The `argparse` sub-package contains an argument parsing system built on GNU Getopt for Java. Each option has a class associated with in, and that class contains everything needed to manage the option. It associates a short option, long option, default value, help text, code to process the option, and code to be run when a module using this system is initialized (so that default values can be set). This associates all the option code in one place so that it is easier to manage. For instance, if an option is added, a separate method to return help text does not need to be updated because it is all managed together. The option processing system uses these classes to generate the help text.

In addition to option processing, this system contains a mechanism for help topics. These help topics can describe anything, such as command-line semantics or output formats.

6.4.2 K-mer Utilities

The k-mer utility classes are found in the `kmer sub-package`. All utilities must extend `KmerUtil`. This utility class contains low-level functions for manipulating k-mers, such as string conversions, reverse complementing, computing a minimizer, and comparing k-mers.

The sort order of k-mers can be defined by custom implementations of a k-mer utility. No other part of the KAnalyze codebase compares k-mers; it is all done with calls to a `KmerUtil` instance.

6.4.3 BoundedQueue

`BoundedQueue` is a thread-safe queue of a fixed size. This class is parameterized, so it can queue objects of any type.

`put()` adds elements to the queue, and it will block until the queue is not full. `take()` gets elements from the queue, and it will block on an empty queue until an element is available.

The queue is stopped by calling `shutdown()`. A shutdown queue will not accept new elements, but it will continue to return elements already contained within it until they are depleted. Once depleted, `take()` returns `null` to signal that all data has been processed. If a queue is shutdown while threads are waiting on `take()`, they are woken up, and `null` is returned to them.

Chapter 7

License

7.1 Documetation

This document is licensed under the GNU Free Documentation License 1.3 or later. The file "COPYING.DOC" contains the text of this license. If you did not receive the file in the source distribution, please contact us or the GNU website for a copy.

7.2 KAnalyze Software

The KAnalyze software is licensed under the GNU Lesser General Public License version 3 or later. The file "COPYING" contains the text of the GNU GPL, and the file "COPYING.LESSER" contains the GNU LGPL extension to the GPL. If you did not receive the file in the source distribution, please contact us or the GNU website for a copy.

Acronyms

API Application Programming Interface. 3

CLUI Command-Line User Interface. 3, 4

IKC Indexed K-mer Count. 15

JAR Java ARchive. 4

Glossary

component A small unit of code in KAnalyze that performs a simple well-defined task and is typically connected to other components to form a *module*. 4

indexed k-mer count A file format that can be rapidly searched for k-mers without storing k-mer records in memory. 15

mapping-free An approach to analyzing sequence data that does not involve "mapping" (aligning) sequence reads to a reference. 3

module A collection of *components* that are connected in some meaningful way to transform data from raw input to final output. 4

segment A memory queue used by the count module to accumulate k-mers before counting and offloading to disk. 9

segment file A segment of k-mers and counts that has been offloaded to a file. 9

Index

Bibliography

Marçais, G. and Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, **27**(6), 764–770.