KAnalyze Manual
v0.9.7

Contents

1 Introduction

1.1 About K-mers

K-mers are short nucleotide sequences of a fixed length, k. For example, short sequences of length 8 are 8-mers.

Converting sequences to k-mers is an iterative process. To get the first k-mer, start at the first base and read the first k characters. The second k-mer starts at the second base, and so on until the end of the last k-mer reaches the end of the sequence.

Analyzing sequence data sets as k-mer sets has many applications in bioinformatics. Some sequence assembly and gene annotation algorithms rely on k-mers.

1.2 About KAnalyze

KAnalyze converts nucleotide (DNA and RNA) sequences to k-mers. KAnalyze evaluates 5 different bases, A, C, G, T, and U. All other letters are skipped and omitted from any k-mer. It can be run as a command line utility, or integrated into pipelines through its command line interface or programming interface (API).

This program was built for speed, accuracy, and longevity. It can compete with the fastest k-mer programs available, it has been extensively tested, and the source code is built to survive updates.

KAnalyze is written in Java, and it requires Java 7 to run. Any machine with a Java 7 JVM (Java Virtual Machine) should be able to run KAnalyze. We have tested it on Linux, Windows, and MacOS.

KAnalyze can currently generate k-mers of size 1 to 31. To optimize for speed, k-mers are processed numerically, which creates this 31-mer size restriction.

2 Command Line Usage

KAnalyze has several modules, and each module executes a different task. Each module is a complete pipeline that takes raw input and writes the final results. For example, the *count* module reads sequence files and writes a sorted tab-delimited file of all k-mers and their counts. This section covers how to run each module.

Java 1.7 (aka Java 7) must be installed in order to run any commands in this section. Run `java -version` to see your version of Java. If you do not see "java 1.7.0" or a later version, Java must be updated before continuing.

2.1 Running Modules

KAnalayze is packaged with a program to run each module. The Windows package comes with exe files, and the Linux package comes with Bash scripts. These programs setup the default maximum memory for the program and runs the module.

For example, the count module can be run by typing 'count' from the command line. On Linux, you may need to begin the command with './'[1] ('./count' in this case).

2.2 Java Command

In most cases, using the module programs described in Section 2.1 is adequate. Sometimes, it's necessary to run the jar file directly.

The current built-in modules allocate 2GB memory. Some advanced parameters, such as multi-threading components or increasing queue sizes, requires more than 2GB of memory. In these cases, the module programs will not work and the jar file must be run directly.

General form:
`java -jar -Xmx2G kanalyze.jar <module> [<module arguments>]`

This allocates 2GB of RAM (-Xmx2G), which is required by the built-in modules. More memory can be allowed by increasing 2G.

After the `<module>` argument, the rest of the command line is the same as above.

2.3 Count Module

The count module counts k-mers from all input files. This module is suitable for large sequence data sets. This module uses a fixed amount of memory to accumulate k-mers. When that memory is full, k-mers are sorted and k-mer counts are written to a file. The memory is refilled with the next set of k-mers, and the process repeats. After all k-mers have been written to files, each file is opened and the k-mer counts are merged and written to the output file. This module is designed to use about 2GB of memory.

2.3.1 Usage

```
count [-d <k-mer threads>] [-f <format>] [-g <segsize>] [-k <ksize>]
      [-l <split threads>] [-m <outfmt>] [-o <out>] [-p <key=value>]
      [-r <reverse>] [-r] [-R] [-s] [-S] [-t <temploc>] [-v] [-V] [-x] [-X]
count -h

-d --kmerthreads [default = 1]
        Set the number of threads to use for the pipeline k-mer step.
```

[1] It depends on how your PATH environment variable is setup. If the current directory (.) is in your path, then './' is not necessary.

```
-f --format [default = raw]
        Set the input sequence format type. This option determines how the
        format files are read. Valid examples include "raw", "fasta",
        "fastq", "fastagz", and "fastqgz". This option may be set multiple
        times when reading multiple files with different formats.

-g --segsize [default = 65536000]
        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.

-h --help
        Print help and exit.

--input
        Inputs a sequence directly from the command line. The argument to this
        option is treated the same as the contents of a file. The format is determined
        by how -f (--format) is set on the command line before this option is
        encountered.

-k --ksize [default = 31, max = 31]
        K-mer size. This determines the size of k-mers extracted from
        input sequences.

-l --splitthreads [default = 1]
        Set the number of threads to use for the pipeline split step.

-m --outfmt [default = seq]
        Sets the format of the output file. Several output formats are built
        into KAnalyze, including seq, hex, dec, and fasta.

-o --out [default = count.kc]
        Output file name for the file writer.

-p --property
        Set a key/value pair for the modules.

-r --reverse
        Reverse complement k-mers as they are generated. This option has an
        optional mode argument. Valid modes are "duplicate", "single", and
        "canonical". In the default mode, duplicate, the original k-mers and
        their reverse complements are counted. Mode single counts only reverse
        complements. Mode canonical counts either the original k-mer or its
        complement, whichever comes first in ASCII sort order. When specifying a
         mode, do not include a space after '-r' (e.g. -rcanonical).

-R --noreverse [default]
        Disable reverse complementing k-mers as they are counted.

-s --splitonly
```

```
        Read files, split k-mers into sorted segments, but do not merge. This
        option is useful for creating split segments on a distributed system,
        but merging must take place on the same machine. For more information on
        splitting and merging in separate steps, see the count module usage
        information in the KAnalyze manual.

-S --nosplitonly [default]
        Perform splitting and merging in the same operation. This option
        negates -s

--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.

--stdout
        Write output to standard output instead of a file. Unless redirected, this output
        is written to the the screen.

-t --temploc [default = Output file directory]
        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.

-v --verbose
        Output extra information after job is complete.

-V --noverbose [default]
        Disable extra information output.

-x --autodelete [default]
        Automatically delete segment (temporary) files generated by the split
        component after merging. In -s (--splitonly) mode, segment files are not
        deleted. Segment files input on the command line are also not
        automatically deleted.

-X --noautodelete
        Do not automatically delete segment files on the command line.
```

2.3.2 Example Usage

`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.

2.3.3 Properties

This module recognizes the following properties (-p)

Property	Default	Description
comp.count.autodelete	true	Automatically delete segment files generated by the split component. This does not affect segment files input on the command line. This parameter is automatically set by -x and -X
comp.count.segsize	98304000	Count split component writes segments of this many k-mers. Increasing this parameter requires additional memory. This parameter is automatically set by -g
comp.kmer.batchsize	100000	The size of k-mer batches passed to the next component after k-merizing reads.
kanalyze.outfmt	seq	K-mer output format. Valid values are "seq", "dec", "hex", and "fasta". Setting this property has the same effect as the -m (–outfmt) option.

2.3.4 Segment Size Format

The segment size argument (-s or --segsize) is an integer number represented in decimal, hexadecimal, or octal and optionally followed by a multiplier. Acceptable numbers begin with an optional sign (+ or -) followed by a decimal, hexadecimal, or octal representation of a number. Hexadecimal numbers must begin with "0x", "0X", or "#". Octal numbers must begin with a zero.

Valid multipliers are:

- k or kb: Binary kilo (1,024)
- kd: Decimal kilo (1,000)
- m or mb: Binary mega ($1{,}024^2$ or 1,048,576)
- md: Decimal mega (1000000)
- g or gb: Binary giga (1024^3 or 1,073,741,824)
- gd: Decimal giga (1,000,000,000)

The keyword "max" may be used as the whole argument. The maximum value is 2,147,483,647[2].

No part of this number string is case sensitive (integer or multiplier).

2.3.5 Count Performance Tuning

The count module has three performance tuning parameters. They are -d (--kmerthreads), -l (--splitthreads), and -s (--segsize). Use of these parameters will not affect the final results, but it will affect resource usage (memory, CPU, and I/O) the amount of time required complete the task. The default values for these parameters work on most systems and is suitable for most data sets. When performance tuning KAnalyze, it is recommended to do a default run on an anticipated data set first, then try the tuning parameters and compare the time required to complete the task.

KAnalyze operates in several steps. They are:

1. Extract sequence reads from source files.
2. Convert sequence reads to k-mers.
3. Split k-mers into sorted segments.
4. Merge all sorted segments.
5. Write sorted k-mer counts to disk.

The -d (--kmerthreads) parameter controls how many threads are used to convert sequence reads to k-mers. The conversion algorithm is very fast, and this is often not main performance bottle-neck. Increasing the number of threads may reduce the amount of time the split component waits for k-mers.

[2] Java's array size limit.

If the number of split threads is increased (see below), then increasing this parameter may improve performance.

The -l (--splitthreads) parameter controls how many threads are used to accumulate and write sorted k-mer segments to disk. The split component takes k-mers from the k-mer component, stores them in memory, and when the amount of memory allocated to k-mers is full, it writes the sorted segment to disk. The time required for this component to accumulate k-mers, sort them, and perform the required disk I/O can stop the KAnalyze process while it completes. Having two or more split components may increase performance so one can be accumulating an sorting while another is writing. Too many threads may have a negative performance impact by writing to multiple files at once or by excessive context switching (as active threads change).

The -s (--segsize) parameter controls how many k-mers are accumulated in memory by the split component before they are sorted and offloaded to disk. The parameter argument is the size of a k-mer array. When that array is full, it is sorted and written. The actual memory taken by this array is this argument multiplied by 8 (k-mers are long integers and require 8 bytes each). The maximum value ("-s max") will allocate 16GB of memory to the array. On very large data sets, the default segment size may result in an excessive number of files, which slows down the merging process. In that case, increasing this value may improve performance. If the memory usage is not adequately increased with this parameter, excessive GC (Garbage Collection) activity will degrade performance or terminate the operation.

Increasing any of these parameters from their default values requires additional memory above the 2GB default limit. See Section 2.2 for more information on how to run KAnalyze with additional memory allocated for the JVM (Java Virtual Machine). When the memory limitation is reached, "Out of Memory" or "GC overhead limit exceeded" errors occur. Note that "GC" in this error message refers to "Garbage Collection" and not the nucleotide bases "G" and "C"[3].

2.4 Stream Module

The stream module converts sequences to k-mers and dumps them to a file. This is most likely used to generate k-mers for another program to analyze.

2.4.1 Usage

```
stream [-f <format>] [-k <kmer size>] [-m <outfmt>] [-o <output>]
       [-p <key=value>] -v -V
stream -h

-f --format [default = raw]
        Set the input sequence format type. This option determines how the
        format files are read. Valid examples include "raw", "fasta",
        "fastq", "fastagz", and "fastqgz". This option may be set multiple
        times when reading multiple files with different formats.

-h --help
        Print help and exit.

--input
        Inputs a sequence directly from the command line. The argument to this
        option is treated the same as the contents of a file. The format is determined
        by how -f (--format) is set on the command line before this option is
        encountered.

-k --ksize [default = 31, max = 31]
        K-mer size. This determines the size of k-mers extracted from
        input sequences.
```

[3]Unfortunately, this is a JVM-generated error message we have no control over.

```
-m --outfmt [default = seq]
        Sets the format of the output file. Several output formats are built
        into KAnalyze, including seq, hex, dec, and fasta.

-o --out
        Output file name for the file writer.

-p --property
        Set a property in the form "key=value".

--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.

--stdout
        Write output to standard output instead of a file. Unless redirected, this output
        is written to the the screen.

-v --verbose
        Output extra information after job is complete.

-V --noverbose [default]
        Disable extra information output.
```

2.4.2 Example Usage

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

2.4.3 Properties

This module recognizes the following properties (-p)

Property	Default	Description
comp.count.deletemp	true	Count merge component deletes temporary files after completing.
comp.count.segsize	98304000	Count split component writes segments of this many k-mers. Increasing this parameter requires additional memory.
kanalyze.outfmt	seq	K-mer output format. Valid values are “seq”, “dec”, “hex”, and “fasta”.
comp.kmer.batchsize	100000	The size of k-mer batches passed to the next component after k-merizing reads.

2.5 Cite Module

Outputs information on citing KAnalyze.

2.5.1 Usage

```
cite [ref|bibtex]
cite -h

-h --help
        Print help and exit.
```

```
Outputs information on citing this software in papers and publications.

Valid citation types are: ref, bibtex
```

2.5.2 Example Usage

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

3 API

KAnalyze is not only a command line program, it is also 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

KAnalyze is a Java program, so the API is implemented as a set of Java classes with well documented interfaces. Any Java program can directly call the API classes. Other programs can use technologies such as JNI (Java Native Interfaces) to drive the API from native code, such as C or C++.

Lastly, the command line itself can be used to drive the program from a script or batch file. Because KAnalyze always terminates with a well-defined return code, scripts can easily execute the CLI and check for errors. For a list of the return codes, see section 6.3.

The remainder of this section outlines the structure and capabilities of the API. It is assumed that a reader of this section has at least a fundamental understanding of Java.

3.1 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 program bug.

Javadoc comments can be converted into a set of HTML pages that document the code. In fact, the Java API itself uses Javadoc, so most Java programmers are familiar with the format of the HTML pages. The KAnalyze build system will create the Javadoc pages for two different levels: api and full. The API level contains only public and protected members, while the full documentation contains everything. For a programmer interested in using the API or creating a custom component, the API level documentation is probably sufficient. The full documentation is intended for maintenance programmers.

For more information on building Javadoc HTML pages from source, see section 6.4.1.

3.2 Project Organization

The KAnalyze project is organized into several packages, which are all sub-packages of `edu.gatech.kanalyze`. The major packages are discussed in this section. Most of these major packages also have sub-packages.

The main packages are:

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.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. The command line interface runs these modules.

edu.gatech.kanalyze.test JUnit test packages. To run tests, see section 6.5.

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

3.3 Components

Components are the pipeline elements that are connected together. They are designed to read and write from synchronized queues. Each component instance should run in its own thread. Because of their parallel nature, downstream components can process data from upstream components as it becomes available. The pipeline architecture is the key to the flexibility and speed 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 an array of some fixed size. When the array is full, it is passed to the queue and a new array is created for the next batch. If the last batch is not full, it is copied to an array of an appropriate size so that there are not empty elements in the batch.

For a concrete example, consider the count module. It reads from source sequence files, k-merizes sequences, counts each instance of unique k-mers, and outputs the counts to a file. To count large data sets, a split component writes several "segments" of sorted k-mer counts to disk, and a merge component merges segments. This module loads and links the read, kmer, count split component, count merge component, and countfilewriter components. Data then flows from the source file to the destination file through this pipeline. The module itself does little more than create the components and link them.

All components extend `edu.gatech.kanalyze.comp.PipelineComponent`. 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>`. To signal the end of the stream, a `null` value is written to the stream. All components must re-write the `null` value back to the stream after reading it to ensure that all threads get the EOF signal.

The folowing sections outline the flow of information through each component. Any component specific properties, i.e. the one set with the `-p` command line option, are also documented.

3.3.1 Count Merge Component

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

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

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

Complements the count split component (Section 3.3.2). Segment files created by the split component are merged and final k-mer counts are sent to the output queue.

Once `null` is read from the input queue, it is written back to the input queue (to ensure all threads get the signal).

Component properties:

comp.count.deltemp If set to the string "true" (not case sensitive), the segment files are deleted after they are read.

Global properties: None

3.3.2 Count Split Component

Input source: `edu.gatech.kanalyze.util.BoundedQueue<long[]>`

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

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

Complements the count merge component (Section 3.3.1). 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 receives the next batch of k-mers. The process is repeated until all k-mers are read.

Once `null` is read from the input queue, it is written back to the input queue (to ensure all threads get the signal).

Component properties:

comp.count.segsize Number of k-mers accumulated in the memory buffer before they are sorted and dumped to a segment file. This value must not be less than `CountSplitComponent.MIN_SEGMENT_SIZE` or greater than `CountSplitComponent.MAX_SEGMENT_SIZE`.

Global properties: None

3.3.3 Count File Writer

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

Output target: `String` (File Name)

Class: `edu.gatech.kanalyze.comp.countfilewriter.CountFileWriterComponent`

Reads k-mer counts from an input queue and writes to a file with the specified file name.

Once `null` is read from the input queue, it is written back to the input queue (to ensure all threads get the signal), and the output file is closed.

Component properites: None

Global properties:

kanalyze.outfmt Controls the output format of k-mers. Values should be `in`, `hex`, or `seq` (default = `seq`).

3.3.4 File Writer

Input source: `edu.gatech.kanalyze.util.BoundedQueue<long[]>`

Output target: `String` (File Name)

Class: `edu.gatech.kanalyze.comp.filewriter.FileWriterComponent`

Reads k-mers from the input queue and writes to a file. The file name is the output target. This component is useful for streaming k-mers into a file where counting or coalescing them is not required.

Once `null` is read from the input queue, it is written back to the input queue (to ensure all threads get the signal), and the output file is closed.

Component properites: None

Global properties:

kanalyze.outfmt Controls the output format of k-mers. Values should be `in`, `hex`, or `seq` (default = `seq`).

3.3.5 K-mer

Input source: `edu.gatech.kanalyze.util.BoundedQueue<SequenceRead[]>`

Output target: `edu.gatech.kanalyze.util.BoundedQueue<long[]>`

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. Both the input queue and the output queue work in batches of items.

Once `null` is read from the input queue, it is written back to the input queue (to ensure all threads get the signal), and `null` is propogated to the output queue.

Component properites: None

Global properties: None

3.3.6 Reverse Complement

Input source: `edu.gatech.kanalyze.util.BoundedQueue<long[]>`

Output target: `edu.gatech.kanalyze.util.BoundedQueue<long[]>`

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

Reverse-complements k-mers from the input queue and writes to the output target. There are three modes (enum `edu.gatech.comp.rcompl.RevComplMode`), and one must be supplied to the constructor. If `null`, the default mode, `RevComplMode.DUPLICATE` is used.

Modes are as follows:

RevComplMode.DUPLICATE Outputs the original k-mer and its reverse complement. Most applications requiring reverse-complement will use this mode.

RevComplMode.LESSER Takes the reverse complement and outputs the lesser of the original or reverse complement (lesser defined by ASCII sort order of k-mers or the numeric sort order of numeric k-mers, which are the same). 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.

Once `null` is read from the input queue, it is written back to the input queue (to ensure all threads get the signal), and `null` is propogated to the output queue.

Component properites: None

Global properties: None

3.3.7 Reader

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

Output target: `edu.gatech.kmer.util.BoundedQueue<SequenceRead[]>`

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.

Each `SequenceSource` object contains information about the sequence source (usually a file) and the declared type (e.g. FASTA or FASTQ). The type information is used to load the appropriate source reader. Source readers classes are located and objects instantiated using Java reflections. For more information about writing custom sequence readers for new file formats, see section 4.3.

Once `null` is read from the input queue, it is written back to the input queue (to ensure all threads get the signal), and `null` is propogated to the output queue.

Component properites: None

Global properties: None

3.4 Utility Classes

KAnalyze has several utility classes for shared functionality. Most of the classes are used by multiple components or modules. These classes may also be useful for other applications.

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

3.4.1 BoundedQueue

`BoundedQueue` is a thread-safe queue of a fixed size. If the size is not set when the queue is created, the default size, `edu.gatech.kanalyze.Constants.DEFAULT_QUEUE_SIZE` is used. 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.

Queues of this type are used to pass elements from one pipeline component to the next. It is designed to be simple and fast.

3.4.2 KmerUtil

`KmerUtil` is a collection of k-mer methods used in several parts of the program. It performs tasks such as checking for a valid k size and converting k-mers among different representations.

3.4.3 LongBoundedQueue

`LongBoundedQueue` was originally used for passing k-mers from one component to the next. Now that k-mers are passed in batches and batches are array objects, this class is no longer used in the main program. Since parts of the unit test code in `edu.gatech.kanalyze.test` still use it, and since it could be useful for some API, it has been left in the project. It may be removed in future revisions.

Besides passing long integers instead of classes, `LongBoundedQueue` is identical to `BoundedQueue`.

3.4.4 LongCounter

`LongCounter` is a hash-table based k-mer counter where the hash key is the k-mer and the value is the number of times the k-mer was added to the counter.

This class is very similar to `java.util.HashTable` in the standard Java API. In fact, the hash function is identical. However, because this implementation is not as general and because it places some restrictions on the bucket array (must be powers of 2), this implementation is generally twice as fast as Java `HashTable`.

This class is not currently used by any of the components packaged with KAnalyze, but it may be useful for future components or extensions outside of the main distribution.

3.4.5 LongHashSet

`LongHashSet` is a hash-table based set. It is used to keep track of which k-mers were encountered when the number of times each k-mer was found is not important.

Like `LongCounter`, this class is based on an optimized algorithm that outperforms implementation, `java.util.HashSet`.

This class is not currently used by any of the components packaged with KAnalyze, but it may be useful for future components or extensions outside of the main distribution.

3.4.6 LongIterator

`LongIterator` is an iterator interface that returns long integers instead of objects. For iterators over large k-mer sets, this iterator avoids the auto-boxing and auto-unboxing overhead of using the standard Java iterator interface. Normally, data structures that use this iterator also have an iterator method for retrieving a standard Java iterator.

This class is not currently used by any of the components packaged with KAnalyze, but it may be useful for future components or extensions outside of the main distribution.

3.4.7 StringUtil

`StringUtil` is a collection of string parsing utilities for parsing actions commonly found in KAnalyze. Currently, it only has the `toNameCase()` method.

`toNameCase()` is used by any class that uses a string to locate a class through the Java reflections API. For example, the sequence reader takes a string such as "fasta", which is not case sensitive, and uses it to find a class named "FastaReader". This method converts "fasta" to "Fasta".

4 Extending KAnalyze

WARNING: Loading code from an unknown source or code that is not stable is a significant security and stability risk. Use caution when extending KAnalyze or accepting 3rd party modules.

KAnalyze was designed to be extensible. As part of this design, new features can be added to KAnalyze *without modifying existing code*. For example, to write a new module, sequence reader, or action for the filter component, no part of the existing system has know about the feature before the feature is loaded.

Only certain parts of the system are loaded dynamically. For example, the module name on the command line is used to locate and load the module class. The subsections below describe the dynamic classes.

Dynamic loading is facilitated by the Java Reflection API. The Java Classloader is given the class name and asked to load it. If the class can be found, it is checked to make sure it fulfills other requirements required for instantiating and running it. KAnalyze requires dynamic classes to have a specific constructor signature and to extend a specific superclass so that it has a way of instantiating an object and calling methods on an otherwise unknown class.

Additional libraries can be specified at runtime, and the classloader will search them when loading dynamic classes. The boostrap classloader, which is the default classloader, is always consulted first. Therefore, it is not possible to override classes built into KAnalyze by loading an external library.

Loading and executing arbitrary code presents a significant security risk and should not be done without caution. If you are loading external libraries, make sure you know and trust where it came from. That external library can execute anything, which means it can read files, modify files, and even install malicious software on the machine. Furthermore, KAnalyze does not require special permissions to run, so it should never be given superuser access.

4.1 Conventions

In the following set of requirements, *xxx* and *Xxx* is the name of the module in lower case and camel-case, respectively. Capitalization is important.

For example, if the module name is "supercool", then *xxx* is "supercool" and *Xxx* is "Supercool". Note that even though this module name is two words together ("super" and "cool"), only capitalize the first letter.

Be aware of how your code affects the system. Use proper exception handling, and never call `System.exit()`.

4.2 Modules

4.2.1 Requirements

1. Module class: `edu.gatech.kanalyze.module.xxx.XxxModule`
2. Module extends `edu.gatech.kanalyze.module.KAnalyzeModule`
3. Module has a public default constructor with no arguments.
4. Module class is not abstract.

4.2.2 Initialization

The KAnalyze main method is `edu.gatech.kanalyze.KAnalyzeModule.main()`. This method reads any command line arguments that appear before the module name. Once the module name is reached, it stops processing the command line options. It then uses the module name to build the fully qualified class name, `edu.gatech.kanalyze.module.xxx.XxxModule`. The class is then loaded and instantiated with the default constructor.

Any libraries loaded with the `--lib` or `--liburl` arguments, which appeared before the module name on the command line, are passed to the instantiated module. This allows the module to use the same libraries to load other dynamic components. This occurs before `configure()` is called.

All command line arguments up to and including the module name are removed from the argument array and used to call `configure()` on the module object. This allows `configure()` to process the remaining arguments as if it were the main method. This method returns `true` if the module should be run in GUI mode, and `false` if it should run in CLI mode.

If the module is run in CLI mode (`configure()` returned `false`), then `exec()` is called followed by `postExec()`. `postExec()` is always called even if `exec()` throws an exception. `exec()` must be defined by the module's implementation. `postExec()` is an empty method that can optionally be overridden.

If the module is run in GUI mode (`configure()` returned `true`), then `runGui()` is called. Neither `exec()` nor `postExec()` are called in this case. The default implementation of `runGui()` simply throws `UnsupportedOperationException`, so override it to launch a GUI if the module has one. If the module does not have a GUI, then never return `true` from (`configure()`, and do not override this method. It is up to (`configure()` to decide when to launch the GUI. For example, the count module launches the GUI if there are no command line arguments to the module or an explicit command line option is used to enter GUI mode.

4.2.3 Error Handling

Module implementations should never call `System.exit()` directly or indirectly. If no errors or special conditions occur that stop the module from completing, `exec()` or `runGui()` should simply return. To report errors, use any one of the `error()` or `warn()` method defined in `AbstractConditionGenerator`, which `KAnalyzeModule` extends (via `KAnalyzeRunnable`).

This uses the error handling system described in Section 5.1. `error()` and `warn()` generate conditions that are fed into this system. By default, `KAnalyzeModule` creates an instance of `KAnalyzeStreamConditionListener` and uses it to handle conditions. In response to erro conditions, this listener reports an error to `System.err`, calls `abort()` on the module, terminates the Java virtual machine (JVM) by calling `System.exit()` with the return code specified by the condition. Therefore, the module may use `error()` to take down the system. This default assumes that the module is being run directly from the command line, which is usually the case.

If the module is not run from the command line, then the default condition listener must be removed by calling `clearDefaultConditionListener()` and replaced with a condition listener that reports errors and warnings in a meaningful way. The custom condition listener can be installed by calling `addListener()`. Failing to do this will take down the JVM and terminate the program that started the module.

4.2.4 Conventions

If a help option is invoked on the command line, `configure()` should print the help message and set a flag so that `exec()` knows to return immediately without trying to run. The help option should be invoked with `-h` or `--help`, and it should not require an argument. It is acceptable to create a help option with optional arguments to output specific details as requested by the user.

`--lib` and `--liburl` may be implemented in the module's command line options, which will allow this argument to appear on either side of the module name on the command line. The only requirement is that libraries that must be loaded to find the module itself must be specified before the module name. Since these libraries are shared with the module, additional dynamic classes can be loaded from them. The module may clear it's libraries with `clearLoaders()`.

4.3 Sequence Readers

Module requirements:

1. Reader class: `edu.gatech.kanalyze.comp.reader.xxx.XxxSequenceReader`

2. Reader extends `edu.gatech.kanalyze.comp.reader.SequenceReader`
3. Reader has a constructor with the signature `(edu.gatech.kanalyze.util.BoundedQueue<edu.gatech.kanalyze.c int, Properties)`
4. Reader class is not abstract.

In the following description, class names without a fully qualified package are found in package `edu.gatech.kanalyze.comp`

The reader must implement the `read()` method from the parent class, `SequenceReader`. The read method takes a single argument, which is a `SequenceSource` object. The reader then calls `getInputStream()` from the source object and reads it until there is no more data to retrieve.

Each sequence string is then written to the next element of a batch array already setup by the parent class, `readBatch[]`, while incrementing the batch counter, `batchSize`, until the batch counter is equal to the size of the batch array (the batch is full). The reader then calls `flush()`, which is implemented by the parent class. `flush()` will write the batch to the pipeline, create a new batch array, and reset the counter. To summarize, the reader writes sequences to a queue until it is full, then it calls `flush`, and it repeats the process until all sequences have been read.

The parent class has a protected variable, `splitLength`. If sequence reads are longer than this, they should be split into multiple reads. The last k - 1 elements from the end of the current split must be copied to the beginning of the next split to preserve correct k-mer generation. For fully assembled sequences, this prevents KAnalyze from memory starvation by attempting to input large reads at once.

Sequence readers should encapsulate the `read()` method in a try/finally that always calls `flush()` before it terminates. Failing to do so may result in loss of reads data.

5 Implementation Details

This section contains important implementation details for those wishing to extend or maintain KAnalyze.

5.1 Condition Reporting System

TODO: Fill in this section.

6 Supplementary Information

6.1 Input File Formats

6.1.1 FASTA

FASTA files are compatible with the FASTA format acceptable by NCBI BLAST[4]. The only exception is that the reader permits blank lines within the sequence because all blank lines are ignored by the parser.

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. There are no restrictions on the line length.

FASTA files are not checked by the reader. The sequence string may contain any characters as long a line does not begin with '¿'. The k-mer component that translates sequences skips over all unrecognized characters. Checking sequences while reading is omitted for performance reasons.

Standard line break characters are recognized: LF (Linux/Unix/MacOS), or CRLF (Windows). The file may or may not end with a newline character.

6.1.2 FASTAGZ

A GZIP compressed FASTA file. The uncompressed file contents must follow the FASTA format defined in Section 6.1.1.

6.1.3 FASTQ

FASTQ files must be compatible with the format published in Nucleic Acid Reseach[5]. 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. The quality score characters are discarded (this reader does not enforce proper scores).

This reader permits blank lines between records, but not within records.

Standard line break characters are recognized: LF (Linux/Unix/MacOS), or CRLF (Windows). The file may or may not end with a newline character.

6.1.4 FASTQGZ

A GZIP compressed FASTQ file. The uncompressed file contents must follow the FASTQ format defined in Section 6.1.3.

6.1.5 RAW

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

Standard line break characters are recognized: LF (Linux/Unix/MacOS), or CRLF (Windows). The file may or may not end with a newline character.

[4] http://www.ncbi.nlm.nih.gov/BLAST/blastcgihelp.shtml
[5] http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2847217/

6.2 Kmer Output Formats

K-mers may be output in one of several formats. The most natural is the sequence format, which a sequence of k bases, each A, C, T, or G. Note that if RNA sequences are read, U is converted to T on output. This is because T and U are represented the same internally.

Two other representations, decimal and hexadecimal, take a little more explanation. Internally, k-mers are represented as integers. Converting the sequences to integers makes processing and storing the k-mers fast and efficient.

To understand the k-mer nucletide sequence to integer conversion, the nucleotide sequence can be thought of a base-4 numbering system where A is 0, C is 1, G is 2, and T is 3. Each base can be conveniently represented as a two-bit number where A = 00, C = 01, G = 10, and T = 11. A k-mer sequence is then a sequence of these letters. For example, the 8-mer CCAGTCGT is 0101001011011011.

Note that k-mers are given a natural order where lower numbers come before higher numbers. For 8-mers, the first possible k-mer is AAAAAAAA, and the last possible k-mer is TTTTTTTT. This numeric ordering can be taken advantage of in many ways.

Java's long integer is a 64-bit 2's complement integer. Each k-mer is is two bits in length. In order to preserve the natural ordering, we chose not to include negative numbers, which is any 2's complement integer that begins with 1. Excluding the first bit, which is always 0, a 64-bit integer can represent any k-mer size 1 to 31. Therefore, the maximum k-mer size is currently 31.

Binary numbers are conveniently represented in hexadecimal. K-mers can also be written as hexadecimal integers, which affords them a compact representation. For example, the same 8-mer as above, CCAGTCGT, can be represented as 0x52DB.

The global property, *kanalyze.outfmt*, sets one of the following output options. If the property is not set, the default sequence option is set. For convenience, most modules have a shorter command line option to set this property.

6.2.1 Sequence Format

Tab delimited file of k-mers and counts. 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.

6.2.2 Decimal Format

Tab delimited file of k-mers and counts using the base-10 decimal representation of k-mers. For more information about how k-mer sequences are represented as integers, see the above section 6.2.

6.2.3 Hexadecimal Format

Tab delimited file of k-mers and counts using the base-16 decimal representation of k-mers. For more information about how k-mer sequences are represented as integers, see the above section 6.2.

6.2.4 FASTA Format

FASTA representation of k-mers counts. Each FASTA record is a single k-mer and its count. The description line is the k-mer count and the sequence is the k-mer sequence.

6.3 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.Constants` 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.

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.

6.4 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 6.4.1 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 6.5 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`.

6.4.1 Building Javadoc Pages

Javadoc is a very powerful tool for documenting APIs written in Java. Section 3.1 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[6].

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

The ant build system (Section 6.4) 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/inde in a browser (relative to the project root).

6.5 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 utility class tests use up to 8GB 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.

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.