BFAST: BLAT-like Fast Accurate Search Tool [1]

Nils Homer

Contents

List of Figures

List of Tables

Preface

This document is meant to serve as a guide for the practical use of BFAST. It includes explanations of all command-line options for each command and binary in BFAST to give an idea of basic usage. Input and output file formats are also detailed. We give many examples of use, including alignment to the whole genome, targeted genomic regions, and to the transcriptome.

This document does not try to explain the underlying algorithm or data-structures used in BFAST. For example, we will not explain the local alignment algorithms, or the underlying BFAST algorithm. Without proper understanding of the underlying algorithm, including local alignment algorithms, it is difficult to use this very flexible program knowledgeably to obtain your desired results. Therefore, reading the BFAST paper (Homer *et al.* (2009a)) is a strict prerequisite. Additionally, the local alignment paper used by BFAST for ABI SOLiD data is required for ABI SOLiD data alignment (Homer *et al.* (2009b)).

If you have anything that you would be useful to add to this guide, feel free to relay the addition to the BFAST developers. This includes but is not limited to bugs, typos, and explanations. Please see `http://genome.ucla.edu/bfast` for more details.

Enjoy!

0.1 Acknowledgements

We would like to thank Jim Kent for creating and distributing the BLAT program (Kent (2002)), which was both a motivation for this work, and was also the basis for our initial practical large scale alignment pipeline. We also thank members of the Nelson Lab: Zugen Chen, Hane Lee, Bret Harry, Jordan Mendler, Brian OConnor for input and computational infrastructure support.

The layout and organization of this book was guided by the design found in the SVN book (`http://svnbook.red-bean.com/`). We would also like to thank SVN for creating amazing version control software that manages the source code BFAST.

Chapter 1

Run BFAST now!

So impatient! If you want to run BFAST now, you have come to the right place. If you are aligning Illumina data or ABI SOLiD data to the human genome, please see section 7.1. Otherwise, then please check out the High Speed Tutorial (section 7.2).

Chapter 2

Installation

BFAST is distributed via source code. This requires the user to compile and install BFAST from source. We assume that the BFAST tarball has been downloaded and unpacked.

2.1 Prerequisites

BFAST requires the following packages to be installed:

- automake (part of GNU autotools)
- zlib-dev (ZLIB developer's libraray)
- libbz2-dev (BZIP2 developer's library)

The BZIP2 developer's library may optionally not be installed, but the `--disable-bz2` option must be used when running the configure script (see the next section).

2.2 Compilation

Enter the following commands, and note any warnings or errors. These commands are commont to linux source packages that use GNU Autotools for compilation.

```
./configure
make
make check
make install
```

If you are having problems, try regenerating the configure script (depends on `automake`):

```
sh autogen.sh
```

The installation path is set by default based on your system (see the output of `make install`). You may change the installation path by using the `--prefix=PATH` option when running the configure script, where `PATH` is the full path to the installation directory.

Chapter 3

Fundamental Concepts

3.1 What is BFAST?

The short answer is that BFAST is a tool for fast and accurate sequence alignment (it is in the name). An implementation of BFAST can be found at http://bfast.sourceforge.net. The longer answer is, well, longer.

The basic problem is as follows. We are given some reference sequence, to which we wish to compare our data. Our data are short reads (sometimes numbering in the billions) that are substrings of the reference sequence. In some cases, these short reads differ based on various edit operators that modify the read in some fashion. It is the goal of BFAST to find the location in the reference sequence that minimizes the number of edit operators used to transform the read into the reference sequence at that location. The process of concluding a read has a location in the reference sequence is called mapping or alignment.

In the case of DNA sequence alignment, these edit operators are biologically and technology motivated, including mutations/mismatches/SNPs, insertion of bases (or letters), deletion of bases (or letters), and technical errors. Typically we have some set of sequence to which we wish to compare our reads. In many cases, finding the correct location given a set of variants (such as the operators) is extremely important since we may be only interested in the variants themselves.

For example, we may be interested in mutations in Breast Cancer and therefore mapping accurately or robust to many combinations of variants is important. In other cases, the speed of mapping is important and the accuracy is not. For example, in digital gene expression analysis the goal is to estimate the number of reads map over locations in the genome as to gain an overall trend of coverage, rather than precise variant detection.

Whatever the intended application of BFAST, it can be tuned to find alignments quickly at the cost of accuracy or robustness against variants. It can also be tuned to accurately find

alignments with variants at the cost of speed. This is the power of BFAST. It is a highly generalized algorithm allowing the user to tune the algorithm to suit their specific needs.

Its implementation is ever-evolving, with new releases occurring early in development and often. BFAST is suited for use on high memory machines when used for Human Genome resequencing and is best used on a cluster of such machines. Nevertheless, it is able to be efficiently run on low memory machines (¡4GB) even for the Human Genome. It also uses multi-threaded programming to utilize the multi-core architecture of current machines. The output of BFAST is in binary format to reduce file space for large data, although we supply tools to convert these formats to text or other standard formats if so desired. Finally, we use the `C` programming language to implement BFAST to increase efficiency as well as the GNU Build System (Autotools) to handle compilation.

3.2 Program Organization

BFAST consists of two sets of programs. The most important utilities are combined into one program `bfast`. A command is specified this one binary to run each command. Auxiliary commands are separated into their own individual programs in the `butil` folder.

3.3 Work flow

The basic work flow has five steps as seen in Figure 3.1.

1. The first step is to create a reference genome from an input FASTA file that contains all the sequence to which we wish to align. The command `bfast fasta2brg` performs this task (see section 4.2).

2. The second step is to create indexes of the reference genome, which was created in the first step. The number and layout of these indexes is determined both by the user's speed and accuracy requirements. The command `bfast index` performs this task (see section 4.3).

3. The third step is to find candidate alignment locations (CALs) for each read. The expected number of CALs returned is a function of the number of indexes and the layouts chosen in the second step as well as the number offsets. The binary `bfast match` performs this task (see section 4.4).

4. The fourth step is to fully align each CAL for each read. This uses a standard local alignment algorithm (Smith and Waterman (1981)) or a custom tool for ABI SOLiD

data (Homer *et al.* (2009b)). The binary `bfast localalign` performs this task (see section 4.5).

5. The fifth and final step is to prioritize the final alignments. The user specifies criteria to select the correct alignment for each read. The criteria can be based on many factors, including uniqueness, score, or other factors. The binary `bfast postprocess` performs this task (see section 4.6).

The reference genome (Step 1) and indexes (Step 2) can be re-used between experiments where only the read data changes, not the reference genome or index layouts. Since only the reads will change, one can utilize the same reference genome and indexes created in Step 1 and Step 2 respectively. Therefore, it is advised to perform Step 1 and Step 2 such that the reference genome and indexes can be re-used, thus reducing the work flow down to three steps upon re-use. In fact, the main speed of this program comes from the idea that in general, the reference genome and associated indexes need only be built once, with the reads coming from the same reference genome (say Human) but different samples or experiments.

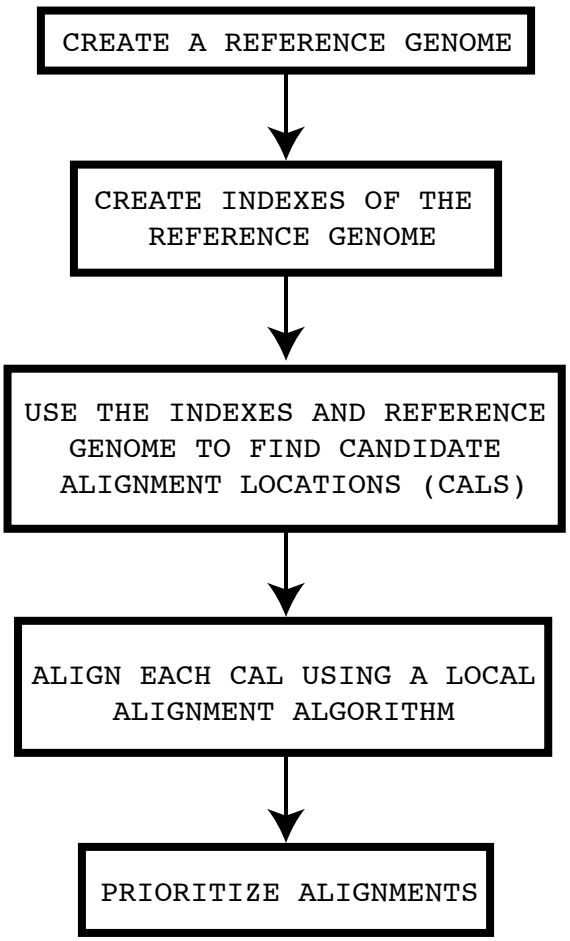

Figure 3.1: The BFAST work flow.
See section 3.3 for a description.

Chapter 4

Basic Usage

BFAST is a command-line program. It accepts many command-line options to customize and tune the alignment algorithm. The key commands are organized into one binary program called `bfast`. To access each command, we use `bfast <command>`. As seen in the section 3.3, there are five commands to be run for the entire work flow, although only three need to be used if we are aligning to a previously indexed reference.

4.1 Common Options

Some common options exist across some or all of the commands.

These options include specifying the reference genome FASTA file (
T-f), specifying the number of threads for parallel processing (`-n`), the number of reads to load at a time (`-Q`), specifying where temporary files should be stored (`-T`), specifying the encoding space (`-A`), outputting timing information (`-t`), printing program parameters (`-p`), and printing a help message (`-h`).

Other options, such as the options `-s`, `-S`, `-e`, and `-E` for specifying only a contiguous range should be considered, are shared across some of the commands but have specific implications to each command and are described in the respective command's section.

The BFAST commands `match`, `localalign`, and `postprocess` can all accept their input file from the standard input stream, facilitating the use of these BFAST commands in a pipe-and-filter model. For example, this allows the output of `match` to be piped into `localalign`, with the subsequent output be piped into `postprocess`. When the input for these commands comes from the standard input stream, no progress messages of any kind will be outputted (don't panic).

4.1.1 Usage

`-f FILENAME, --fastaFileName=FILENAME`

Specifies the file name of the FASTA reference genome (see subsection 5.1.1 for the file format). This option applies to `bfast fasta2brg`, `bfast index`, `bfast match`, `bfast localalign`, and `bfast postprocess`.

`-n INTEGER, --numThreads=INTEGER`

For `bfast index` the number of threads must be a power of two due the implementation of the index sorting algorithm (merge sort). Otherwise it is recommended that the number of threads match the number of cores or processors. This option applies to `bfast index`, `bfast match`, `bfast localalign`, and `bfast postprocess`.

`-Q INTEGER, --queueLength=INTEGER`

Specifies the number of reads to cache or load into memory at one time. This option applies to `bfast match`, `bfast localalign`, and `bfast postprocess`.

`-T DIRECTORY --tmpDir=DIRECTORY`

This option specifies the directory in which to store temporary files. For large datasets, the necessary disk space for temporary files may be large and therefore it is useful to to specify the temporary file directory. Be sure to include a trailing backslash or \. If no option is given, the temporary file directory is defaulted to the current directory. This option applies to `bfast index`, and `bfast match`.

`-A INTEGER, --space=INTEGER`

Specifies the encoding space of the alphabet. For nucleotide space, use `-A 0` (Illumina, 454, etc.). For color space, use `-A 1` (ABI SOLiD).

`-t, --timing`

This option applies to `bfast fasta2brg`, `bfast index`, `bfast match`, `bfast localalign`, and `bfast postprocess`. This option causes timing information for the execution of the program to be displayed upon successful termination.

`-p, --Parameters`

This option applies to `bfast fasta2brg`, `bfast index`, `bfast match`, `bfast localalign`, and `bfast postprocess`. This option causes the input command-line parameters to be displayed and subsequent termination of the program.

`-h, --Help`

This option applies to `bfast fasta2brg`, `bfast index`, `bfast match`, `bfast localalign`, and `bfast postprocess`. This option prints a help message.

4.2 bfast fasta2brg

fasta2brg is a command that is used to create a reference genome from FASTA file. This utility performs the first step of the work flow outlined in section 3.3. The BFAST reference genome file will be written in compressed binary format to preserve space. See subsection 5.2.1 for the file format of the BFAST reference genome file.

See section 4.1 for common options that are in use in this command.

4.2.1 Creating a Reference Genome

To create a reference genome, the required command-line option is `-f`.

We need to input a reference genome FASTA file using the `-f` option (see subsection 5.1.1 for the file format). To create a reference genome in color space, we use the option `-A 1`, otherwise we use `-A 0`.

When creating a BFAST reference genome file, the contigs will be numbered according to their order in the reference genome FASTA file (option `-f`). The numbering is one-based (begins with one). The maximum number of contigs is 2^{31} or 2147483648. The name of each contig specified in the header of the reference genome FASTA file will be also be stored. The maximum sequence length for a single contig is also 2^{31} or 2147483648. The input sequence will be assumed to be the forward strand of the genome. Only the forward strand of the genome will be stored (see subsection 5.2.1 for more details). The output will be a BFAST reference genome file (see subsection 5.2.1 for the file format).

4.3 bfast index

index is a command that is used to create the indexes of a reference genome. This utility performs the second step of the work flow outlined in section 3.3. The BFAST index file will

be written in compressed binary format to preserve space. See subsection 5.2.2 for the file format of the BFAST index file.

See section 4.1 for common options that are in use in this command.

4.3.1 Creating Indexes of a Reference Genome

To create indexes of a reference genome, the required command-line options are `-f`, `-m`, and `-w`.

We need to input a reference genome FASTA file using the `-f` option (see subsection 5.1.1 for the file format). The BFAST reference genome file must already be created using this reference genome FASTA file using `bfast fasta2brg` and will be inferred from the reference genome FASTA file file name. If we choose to create the indexes in color space (`-A 1`), a color space BFAST reference genome file must exist. The `-m` option specifies the mask or space-seed to use for this index. The `-w` option specifies the hash width (the index into the index).

The `-d` option is used to split the index into multiple parts for low-memory computation. The index will be split into 4^d parts, where d is the value for `-d` specified. Specify a value of 1 to reduce memory usage to 25%. To calculate the size of the BFAST index file before creation, see subsection 5.2.2. The `-i` option is used when we wish to create more than one index using the same reference genome FASTA file and space (`-A`).

When creating a BFAST index file, we will use all possible contig sequences from the specified BFAST reference genome file, specified using option `-r`, unless any of the options `-s`, `-S`, `-e`, `-E`, or `-x` are used. The file size of the BFAST index file can be very large for large indexes (see subsection 5.2.2 for more details), although it can be optimally split using `-d`.

The output will be a BFAST index file (see subsection 5.2.2 for the file format).

Other options are specified in subsection 4.3.2.

4.3.2 Usage

`-m STRING, --mask=STRING`

The mask or spaced seed to use. The mask is a set of zero and ones (must start and end with a one). Please see subsection 5.2.2 for more details.

`-w INT, --hashWidth=INT`

The hash width for the index. A hash is used as an index into the index but at the cost of increasing the size of the index. Please see subsection 5.2.2 for more details.

`-d INT, --depth=INT`

The depth of splitting (d). The index will be split into 4^d parts, where d is the value for `-d` specified. Use `-d 0` to not split the index. To calculate the size of the BFAST index file before creation, see subsection 5.2.2.

`-i INT, --indexNumber=INT`

Specifies this is the ith index you are creating. This is useful when multiple indexes from the same reference are to be created (in the same space).

`-R, --RepeatMasker`

Ignores lower case bases when creating the indexes. This typically corresponds to RepeatMasker sequence.

`-s INTEGER, --startContig=INTEGER`

Specifies the first contig to include when building indexes.

`-S INTEGER, --startPos=INTEGER`

Specifies the first position in the first contig to include when building indexes.

`-e INTEGER, --endContig=INTEGER`

Specifies the last contig to include when building indexes.

`-E INTEGER, --endPos=INTEGER`

Specifies the last position in the last contig to include when building indexes.

`-x FILENAME, --exonsFileName=FILENAME`

Specifies the exon-like ranges to include in the index. This option cannot be used with the `-s`, `-S`, `-e`, or `-E` options. The exon ranges must fall within bounds in the BFAST reference genome file. For the file format of the exons file, please see subsection 5.1.3.

4.4 bfast match

bfast match command takes a set of reads and searches a set of indexes to find candidate alignment locations (or CALs) for each read. This utility performs the third step in the work flow outlined in section 3.3.

The output will be in BFAST matches file format (see subsection 5.2.3 for the file format).

Also see section 4.1 for common options that are in use across some or all of the binaries.

4.4.1 Finding Candidate Alignment Locations (CALs)

To find CALs for a set of reads, the required command-line option is `-f`. We need to input a reference genome FASTA file using the `-f` option (see subsection 5.1.1 for the file format). The BFAST reference genome file must already be created using this reference genome FASTA file using `bfast fasta2brg` and will be inferred from the reference genome FASTA file file name. If the option `-A 1` is used, then both the BFAST reference genome file and the BFAST index files must have been created using the `-A 1` option.

By default, all indexes of the reference genome FASTA file will be automatically detected and used as the main indexes. The `-i` option specifies the main index numbers to use (comma separated). This corresponds to the `-i` parameter(s) used during index creation. If you wish to have a secondary set of indexes, which are used if no matches are found in the main set of indexes, use the `-I` option.

The reads by default will be read from the standard input stream. Nevertheless, a file containing the reads may be specified using the `-r` option (see subsection 5.1.2 for the file format). The output is printed in binary format to the standard output stream.

Only the forward strand of the genome is indexed (see subsection 5.2.2), so both the read and its reverse compliment will be looked-up in the index to find CALs. This can be modified by using the `-w` option, which will target a specific strand of the reference genome.

In all cases, the BFAST reference genome file and BFAST index files must match that encoding space specified by `-A`.

The `-K` and `-M` options are useful to ignore keys that return too many CALs (`-K`) and to ignore reads that in aggregate have too many CALs (`-M`). (For reference, 65% of the human genome has fewer than 100 CALs given a key size (`-k`) of 16).

If the reads file is large, a subset of reads can be specified using the `-s` and `-e` options, which helps distribute the process across a cluster. For extremely large read datasets (billions), it is recommended that the reads be split into separate files before hand.

Other options are specified in subsection 4.4.2

4.4.2 Usage

`-i STRING, --mainIndexes=STRING`

Specifies the index numbers for the main bif files (comma separated). This corresponds to the `-i` parameter(s) used during index creation. By default, all indexes of the reference genome FASTA file will be automatically detected and used as the main indexes if no main indexes are given. See subsection 5.2.2 for the file format of the BFAST index files.

For advanced users, the input can be a combination of numbers and ranges. For example, `-i 1,5-6,10` will specify that indexes `1`, `5`, `6`, and `10` will be used.

`-I STRING, --secondaryIndexes=STRING`

Specifies the index numbers for the secondary bif files (comma separated). This corresponds to the `-i` parameter(s) used during index creation. See subsection 5.2.2 for the file format of the BFAST index files. If no secondary indexes are specified, none will be used.

`-r FILENAME, --readsFileName=FILENAME`

Specifies the file containing the reads. See subsection 5.1.2 for more information on the file format of the reads file.

`-l, --loadAllIndexes`

Specifies to load all main or secondary indexes into memory. This is useful for high memory (RAM) machines.

`-j, --bz2`

Specifies that the input reads are bz2 compressed (bzip2).

`-z, --gz`

Specifies that the input reads are gz compressed (gzip).

`-o STRING, --offsets=STRING`

Specifies the offsets to use for all BFAST index files. If no offsets file is given, all possible offsets will be used. The offsets can be given as a range (i.e. `-o 0-25`), or as a comma separated list (i.e. `-o 0,1,2,3,4,5`).

For advanced users, the input can also be a combination of numbers and ranges. For example, `-i 1,5-6,10` will specify that the offsets `1`, `5`, `6`, and `10` will be used.

`-s INTEGER, --startReadNum=INTEGER`

Specifies the first read in which to process. This may be useful when distributing a large data set across a cluster.

`-e INTEGER, --endReadNum=INTEGER`

Specifies the last read in which to process. This may be useful when distributing a large data set across a cluster.

`-k INTEGER, --keySize=INTEGER`

Specifies to truncate all indexes to have the given key size. This will only be performed on indexes for which the given value is greater than the hash width and less than the original key size. This may be useful to search with greater sensitivity by reusing indexes large key sizes (See section 6.1).

`-K INTEGER, --maxKeyMatches=INTEGER`

Specifies the maximum number of matches to allow before a key is ignored. A key may return one or more CALs and therefore it may be desirable to ignore non-unique or over-represented keys. For example, a value of 100 may be useful when aligning the to Human Genome given that each index used is expected to return one CAL.

`-F FLOAT, --keyMissFraction=FLOAT`

Specifies the maximum fraction of seed positions within the read that exceed the maximum key matches limti (`-K`) for the hit to be retained. The lower the maximum to increase specificity at the cost of sensitivity.

`-M INTEGER, --maxNumMatches=INTEGER`

Specifies the maximum number of CALs to allow before we stop searching for CALS for a given read. If the limit is reached, the read will be flagged and ignored in later alignment processes. For example, a value of 500 may be useful when aligning the to Human Genome given that each index used is expected to return one CAL.

`-w INTEGER, --whichStrand=INTEGER`

Specifies to find matches on the designated strands. For both strands, use `-w 0`. For the forward strand only, use `-w 1`. For the reverse strand only, use `-w 2`.

4.5 bfast localalign

bfast localalign is a command that takes a list of Candidate Alignment Locations (CALs) for each read and performs a local alignment of each read to the reference, giving a score for the quality of the alignment. This utility performs the fourth step in the work flow outlined in section 3.3.

The output will be a BFAST aligned file (see subsection 5.2.4 for the file format). See section 4.1 for common options that are in use in this command.

4.5.1 Performing Local Alignment on Candidate Alignment Locations (CALs)

To perform local alignment of CALs, we need to input a reference genome FASTA file using the `-f` option (see subsection 5.1.1 for the file format). The BFAST reference genome file must already be created using this reference genome FASTA file using `bfast fasta2brg` and will be inferred from the reference genome FASTA file file name.

The input by default will be read from the standard input stream and must be in the same format as a BFAST matches file outputted by `bfast match`. To process a file, the `-m` option specifies a BFAST matches file outputted by `bfast match`. The output will written to the standard output stream. The output file is a BFAST aligned file, which stores the local alignments for each CAL and read in binary format (see subsection 5.2.4 for the file format).

Local alignment may be time consuming when a large number of CALs are returned. Therefore we can use the option `-M` to specify maximum number of CALs. If a read has more than the specified number, it will be ignored, and annotated as having too many CALs.

The Smith Waterman algorithm supports mismatches, indels (with affine gap penalties), and color errors. The value for the `-A` option must match the value given to the `-A` option in `bfast match`. In the case that `-A 1` is used, the algorithm will simultaneously correct for color errors. For more information on the local aligner, please see Homer *et al.* (2009b).

To align without gaps (deletions and insertions), we can use `-u`. The `-U` is used to align without considering seed constraints. With this option, bases that matched the reference during `bfast match` will be not constrained to match during `bfast localalign`.

The -s, -S, -e and -E can be used to specify only to consider CALs within a given range.

4.5.2 Usage

`-m FILENAME, --matchFileName=FILENAME`

Specifies the BFAST matches file outputted by the `match` utility. See subsection 5.2.3 for the file format.

`-x FILENAME, --scoringMatrixFileName=FILENAME`

Specifies the Scoring Matrix file used to score the alignments. Please see subsection 5.1.4 for the file format.

`-u, --ungapped`

Specifies that the local alignment will be ungapped (no deletions and insertions).

`-U, --unconstrained`

Specifies align without considering seed constraints. Without this option, bases that matched the reference during `bfast match` will be constrained to match during `bfast localalign`.

`-s INTEGER, --startReadNum=INTEGER`

Specifies the first read in which to process. This may be useful when distributing a large data set across a cluster.

`-e INTEGER, --endReadNum=INTEGER`

Specifies the last read in which to process. This may be useful when distributing a large data set across a cluster.

`-o INTEGER, --offset=INTEGER`

Specifies the number of bases before and after each CAL to include in the reference when aligning. This is not used with ungapped constrained alignment (when -u but not -U is specified). For example, a value of 10 can be used when aligning to the Human Genome, since this would allow for small insertions and deletions to be placed more accurately in the local alignment.

`-M INTEGER, --maxNumMatches=INTEGER`

Specifies to ignore reads who have more than the specified number of CALs.

`-q INTEGER, --avgMismatchQuality=INTEGER`

Specifies the average mismatch quality.

4.6 bfast postprocess

bfast postprocess is a command that takes as input a BFAST aligned file. It can convert the input file to a specified output format as well as help choose the best alignment for each read based on score or uniqueness. This utility performs the fifth step in the work flow outlined in section 3.3. Many other filters can be applied, for example with paired-end reads where we desire only alignments for which **BOTH ends align. These filters can be applied by downstream tools such as SAMtools (see http://samtools.sourceforge.net) or DNAA (see http://dnaa.sourceforge.net).**

The input by default will be read from the standard input stream and must be in the same format as a BFAST aligned file outputted by `bfast localalign`. To process a file, the `-i` option specifies a BFAST aligned file outputted by `bfast localalign`. The output will written to the standard output stream, with the output format specified by `-O`. The output file will be in the format specified by `-O` format (see subsection 5.2.4 for the file format). Optionally, the `-u` will dump all unmapped reads to a BFAST aligned file (see subsection 5.2.4 for the file format).

By default, unmapped reads will be included in the output file.

See section 4.1 for common options that are in use in this command.

4.6.1 Prioritizing Alignments

The `-a` option can be used to filter and choose the best alignment. `-a 0` will not modify the data but only convert the file to the specified output format (`-O`). Options `-a 1`, `-a 2`, and `-a 3`, will for each read select a subset of alignments from the alignment(s) found. `-a 1` will output all alignments that pass the filters. `-a 2` will output only reads that have a unique alignment regardless of score after applying the filters. `-a 3` will output only reads that have a unique best scoring alignment after applying the filters. If multiple alignments have the same best score, an alignment is not reported. `-a 4` will output only reads

that have a best scoring alignment (possibly many best scoring alignments may exist).

Paired-end reads will be scored by examining both ends to select the best paired-alignment.

For paired end reads, use `-Y 0` or `-S 0 -P 1`. For mate pair reads, use `-Y 1` or `-S 1 -P 0`.

4.6.2 Usage

`-i FILENAME, --alignedFileName=FILENAME`

Specifies the BFAST aligned file (see subsection 5.2.4 for the file format).

`-a INTEGER, --algorithm=INTEGER`

This specifies the algorithm to choose the alignment for each each end of the read after filtering. The option `-a 0` specifies that no filtering will occur. The option `-a 1` specifies that all alignments that pass the filters will be outputted. The option `-a 2` outputs only reads that have been aligned uniquely. The option `-a 3` chooses uniquely the alignment with the best score. The option `-a 4` chooses all alignments with the best score.

`-O INTEGER, --outputFormat=INTEGER`

Specifies the output format. `-O 0` specifies the output to be in BFAST aligned file format (see subsection 5.2.4 for the file format). `-O 1` specifies the output to be in BFAST sequence alignment/map file format (see https://sourceforge.net/projects/samtools/).

`-o STRING, --outputID=STRING`

Specifies output ID to prepend to the read name (BFAST sequence alignment/map file output only).

`-r STRING, --readGroupFileName=STRING`

Specifies to add the read group (@RG) line to add to the header, which is given in the specified file. Additionally, the appropriate read group (RG) tag (and LB tag if present) will be added to each read. Make sure that the line is exactly

the same as what would be printed to the SAM file, which includes the "@RG" string.

`-S INT, --strandedness=INT`

Specifies the pairing strandedness: The option `-S 0` specifies that the reads should be mapped onto the same strand. The option `-S 1` specifies that the reads should be mapped onto the opposite strand.

`-P INT, --positioning=INT`

Specifies the pairing positioning: The option `-S 0` specifies that the first read should be (5') upstream of the second read (sequencing strand). The option `-S 1` specifies that the second read should be (5') upstream of the first read (sequencing strand).

`-Y INT, --pairing=INT`

Specifies the pairing orientation: The option `-S 0` specifies that the reads are paired ends (`-S 0 -P 1`). The option `-S 1` specifies that the reads are mate pairs (`-S 1 -P 0`).

`-q INTEGER, --avgMismatchQuality=INTEGER`

Specifies the average mismatch quality (should match that value used in `bfast match`.

`-x FILENAME, --scoringMatrixFileName=FILENAME`

Specifies the Scoring Matrix file used to score the alignments. Please see subsection 5.1.4 for the file format. This file should be the file used in `bfast match`. If this option is used with ABI SOLiD data, then the `-A 1` option must be set.

`-z, --randomBest`

Specifies to choose a random best scoring alignment to break ties. This only works when used in conjunction with `-a 3`.

`-m INTEGER, --minMappingQuality=INTEGER`

Specifies to choose remove all alignments with worse mapping quality.

`-M INTEGER, --minNormalizedScore=INTEGER`

Specifies to choose remove all alignments with worse normalized alignment score. The normalized alignment score is calculated by dividing the alignment score by the read length.

`-v FLOAT, --insertSizeAVG=FLOAT`

Specifies the insert size mean (outer size) to use for pairing.

`-s FLOAT, --insertSizeStdDev=FLOAT`

Specifies the insert size standard deviation (outer size) to use for pairing.

4.7 bfast bafconvert

`bfast bafconvert` **converts BFAST aligned files to the specified output format.**

4.7.1 Usage

The usage is `bfast bafconvert [options] <files>`**. The command line options are:**

`-O`

Specifies the output type. `0` **converts a text BFAST aligned file to a binary BFAST aligned file.** `1` **converts a binary BFAST aligned file to a text BFAST aligned file.** `2` **converts a binary BFAST aligned file to a BFAST sequence alignment/map file (currently experimental, see https://sourceforge.net/projects/samtools/).**

`-f`

Specifies the reference genome FASTA file. See section 4.1 for common options for a description of this option. This option is not required for BFAST aligned file output.

`-o`

Specifies an output ID, which will be prepended to the name of each read. This option is only used for for BFAST sequence alignment/map file output only.

`-r STRING, --readGroupFileName=STRING`

Specifies to add the read group (@RG) line to add to the header, which is given in the specified file. Additionally, the appropriate read group (RG) tag (and LB tag if present) will be added to each read. Make sure that the line is exactly the same as what would be printed to the SAM file, which includes the "@RG" string.

4.8 bfast header

`header` **prints the header of a BFAST reference genome file or a BFAST index file.**

4.8.1 Usage

The usage is `bfast header [options] <files>`**. The input file can be either a BFAST reference genome file or a BFAST index file.**

4.9 bfast bmfconvert

`bfast bmfconvert` **converts a BFAST matches file from binary to text or vice versa.**

4.9.1 Usage

The usage is `bfast bmfconvert [options] <files>`**. The command line options are:**

`-O`

Specifies the output type. `0` **converts a text BFAST matches file to a binary BFAST matches file.** `1` **converts a binary BFAST matches file to a text BFAST matches file.** `2` **converts a binary BFAST matches file to a Reads FASTQ file.**

4.10 bfast brg2fasta

`bfast brg2fasta` **prints the reference genome in FASTA format.**

4.10.1 Usage

The usage is `bfast brg2fasta BFAST reference genome file`**.**

4.11 bfast easyalign

`bfast easyalign` **will run** `bfast match`**,** `bfast localalign`**, and** `bfast postprocess` **with their respective default parameters. See the respective commands for the default parameters and explanation of the command line usage.**

4.12 butil

butil is a folder containing utilities that were developed for personal use to test, debug, and compliment the BFAST program and its accompanying publication. They are included in this distribution to aid in using BFAST and to give examples of other uses for the indexes built and data generated by BFAST. There is no support or warranty for these utilities. Please use at your own risk and consult the source code if problems arise. If you find one of these utilities incredibly useful, please contact the authors/developers as to recommend the utility be supported.

To access a help message, please use the `-h` **option for all utilities.**

4.12.1 balignmentscoredistribution

Assess the alignment score distribution (histogram) for all reads with a given number of CALs. The alignment scores are binned according the given parameters.

`-i FILENAME`

The BFAST aligned file to analyze.

`-f INT`

Bins from.

`-b INT`

Bins by.

`-t INT`

Bins to.

4.12.2 balignsim

`balignsim` **generates synthetic reads given a number of variants and errors from a reference genome and tests the various local alignment algorithms.**

`-i FILENAME`

This is an input specification file. Each line contains the specification for one set of simulated reads. Each set of reads has 8 fields (all specified on one line).

1. **0: gapped 1: ungapped**
2. **0: no indel 1: deletion 2: insertion**
3. **indel length (if #2 is an indel)**
4. **include errors within insertion 0: false 1: true**
5. **# of SNPs**
6. **# of errors**
7. **read length**
8. **number of reads**

`-f FILENAME`

Specifies the reference genome FASTA file. See section 4.1 for common options for a description of this option.

`-x FILENAME`

Specifies the Scoring Matrix file used to score the alignments. Please see subsection 5.1.4 for the file format.

`-n INT`

The number of threads to use for the search.

`-A INT`

The space in which the reads should be outputted. Use `0` for nucleotide space, and `1` for color space.

4.12.3 bevalsim

`bevalsim` **parses a BFAST aligned file resulting from using reads generated by** `bgeneratereads` **to give accuracy statistics for the mapping.**

`-i FILENAME`

BFAST aligned file name to be evaluated.

`-r FILENAME`

The reads file name generated by `bgeneratereads`**.**

4.12.4 bgeneratereads

`bgeneratereads` **generates synthetic reads given a number of variants and errors from a reference genome. See the source code for the output file format.**

`-i FILENAME`

This is an input specification file. Each line contains the specification for one set of simulated reads. Each set of reads has 9 fields (all specified on one line).

1. **0: no indel 1: deletion 2: insertion**
2. **indel length (if #2 is an indel)**

3. **include errors within insertion 0: false 1: true**
4. **# of SNPs**
5. **# of errors**
6. **read length**
7. **paired end 0: true 1: false**
8. **paired end length**
9. **number of reads**

`-f FILENAME`

The reference genome FASTA file from which reads should be generated.

`-A INT`

The space in which the reads should be outputted. Use `0` for nucleotide space, and `1` for color space.

4.12.5 bindexdist

`bindexdist` **prints each unique read from the genome and the number of times it occurs, where the genome is contained in the BFAST index file.**

`-f FILENAME`

The reference genome FASTA file accompanying the BFAST index file.

`-i FILENAME`

The BFAST index file to be examined.

`-s INT`

Which strand to examine: 0 - both strands, 1 - item forward strand only, and 2 - item reverse strand only.

`-n INT`

The number of threads to use for the search.

`-T DIRECTORY`

A temporary file directory to store temporary files.

4.12.6 bindexhist

`bindexhist` **prints a histogram that counts the number of unique k-mers in the genome that occur X number of times. The k-mer chosen comes from the layout of the BFAST index file.**

`-f FILENAME`

The reference genome FASTA file accompanying the BFAST index file.

`-i FILENAME`

The BFAST index file to be examined.

`-s INT`

Which strand to examine: 0 - both strands, 1 - item forward strand only, and 2 - item reverse strand only.

`-n INT`

The number of threads to use for the search.

4.12.7 bmfmerge

`bmfmerge` **merges the results from searches from different indexes under the assumption that all the searches were performed on the same dataset. This performs the final merge step in** `bfast match` **separately such that the merge step can be separated from the search step.**

`-M INTEGER, --maxNumMatches=INTEGER`

Specifies the maximum number of CALs to allow before we stop searching for CALS for a given read. If the limit is reached, the read will be flagged and ignored in later alignment processes. For example, a value of 500 may be useful when aligning the to Human Genome given that each index used is expected to return one CAL.

4.12.8 brepeat

`brepeat` **finds all contiguous repeats in the genome specified by the index that fall within the specified unit length range and minimum contiguous length.**

`-f FILENAME`

The reference genome FASTA file accompanying the BFAST index file.

`-m INT`

The minimum unit length for a repeat.

`-M INT`

The maximum unit length for a repeat.

`-r INT`

The maximum total repeat length as a scalar multiple of the unit length.

4.12.9 btestindexes

`btestindexes` **is a utility that tests, searches for, and compares layouts for indexes against certain events, such as errors, mismatches and insertions.**

This utility can sample the space of possible indexes and the space of reads with a given set of errors and variants to find accurate index sets for use with BFAST. By specifying `-a 0`, the greedy search strategy will run. We initially seed the index set with an index with one contiguous mask. Next, we iteratively add indexes to the set as follows. We search for the best index that would increase the accuracy of the set when added. After sampling the possible space of indexes (`-s`), we choose add the best index to the set. To estimate the accuracy

of an index set, we create an accuracy profile. The accuracy profile computes the accuracy for mapping reads with a specific number of SNPs/errors and color errors (see `-M` and `-E` respectively). We prioritize color errors over SNPs, meaning when comparing the accuracy profile of two index sets, we compare the accuracy for mapping reads with `1` to the specified maximum number of color errors (`-E`) with no SNPs. We repeat the comparison with one SNP, two SNPs, up to the maximum number of SNPs (`-M`).

This utility can also be used to print the accuracy for each scenario of a read with variants and errors (`-a 1`).

`-a INT`

The algorithm to run. The option `-a 0` will search for masks. The option `-a 1` will compute the accuracy of masks read from file.

`-r INT`

Specifies the read length to examine.

`-S INT`

Specifies the number of events in our sampling space. This corresponds to the number of random reads to generate to estimate the accuracy for a specific scenario of events.

`-A INT`

Specifies the encoding space of the alphabet. For nucleotide space, use `-A 0`. For color space, use `-A 1`.

`-s INT`

Specifies the number of masks in our sampling space (for `-a 0`).

`-l INT`

Specifies the mask key size when sampling indexes (for `-a 0`).

`-w INT`

Specifies the maximum mask width when sampling indexes (for `-a 0`).

`-n INT`

Specifies the maximum index set size (or the maximum number of indexes in one set). Each index will be added greedily one at a time (for `-a 0`).

`-t INT`

Specifies the accuracy threshold that must be met for a specific scenario when comparing index set accuracy during sampling (for `-a 0`). Once the index set has reached this accuracy threshold for the given scenario, the next scenario will determine the index set selection.

`-f STRING`

Specifies the input file name for the masks (for `-a 1`). Each mask should be on a separate line.

`-I INT`

Specifies the maximum insertion length when evaluating index sets (for `-a 1`).

`-M INT`

Specifies the maximum number of mismatches. With `-A 0` this will correspond to SNPs or errors. With `-A 1` this will correspond to SNPs.

`-E INT`

Specifies the number of color errors to include (for `-A 1`).

`-p`

Prints the program parameters.

`-h`

Prints a help message.

4.13 scripts

scripts is a folder containing scripts that were developed to convert the input files from Illumina and ABI SOLiD sequencers to the BFAST FASTQ format. Additionally, we include a script to parallelize BFAST on a cluster. Currently only SGE and PBS clusters are supported.

4.13.1 bfast.submit.pl

This script will run BFAST on a SGE or PBS cluster. Please use the `-man` **option for information on how to use the script. Note that the PERL module XML Simple is required to be installed for compilation and can be found at http://search.cpan.org/dist/XML-Simple.**

`-help`

Print a brief help message and exits.

`-schema`

Print the configuration XML schema.

`-man`

Prints the manual page and exits.

`-quiet`

Do not print any submit messages.

`-config`

The XML configuration file.

4.13.2 bfast.resubmit.pl

This script is a companion script to `bfast.submit.pl`**. If any job fails (for whatever reason), this script can be used to resubmit the failed job and update any other jobs that depend on the failed job.**

`-help`

Print a brief help message and exits.

`-man`

Prints the manual page and exits.

`username`

Process all jobs in the error state from the given username.

`-jids`

Process all given job ids.

4.13.3 qseq2fastq.pl

This script will convert Illumina generated QSEQ files or Illumina generated SEQUENCE files to the BFAST FASTQ format. Please execute the script with no arguments for more information.

4.13.4 solid2fastq

This program will convert ABI SOLiD generated CSFASTA and QUAL files to the BFAST format. It will also split the input reads into chunks for parallel computation. Please execute the program with no arguments for more information. We also include a PERL version of this program for developer modification.

Chapter 5

File Formats

5.1 Input Files

These files represent the input files that are used by one or more BFAST binaries but are not generated as output by a BFAST binary. Although some files are used as input to other binaries, for example the BFAST matches file is used as input to `bfast localalign`, they are described in section 5.2. Examples of each input file is given in section 5.3.

5.1.1 Reference genome FASTA file

The reference genome FASTA file follows the familiar FASTA format used to describe one or more molecular sequences or contigs. Each contig begins with a header line, characterized by a greater-than (>) symbol at the beginning of the line. The contig's sequence is then listed beginning on a new line. The end of the contig's sequence is specified by the end of the file or a new header line for the next contig.

An example of such a file can be seen Figure 5.1. In this example, there are two contigs specified.

5.1.2 Reads FASTQ file

This file contains the reads for which we wish to align. The reads are specified in FASTQ format. The first line begins with the @ symbol. The rest of the first line will be the read name. The second line contains the sequence for the read. Currently the entire sequence must be specified one line and should be specified

$5' \rightarrow 3'$ from left-to-right. The third line will begin with the + symbol. The rest of the line can be empty or contain an arbitrary comment string. The fourth line will contain the sequence qualities.

For ABI SOLiD or color space reads, the adaptor should be included in the sequence and the colors should be encoded as $[0-4]$ with 4 signifying a unknown color. There should be one Phred-like quality score for each base in the sequence (or number of colors for ABI SOLiD data).

For paired end or multi end data, each end should be specified separately but have the same read name. They should be listed in which order they are sequenced. Paired end reads are typically on the opposite strand, with the first end having a smaller co-ordinate when mapped to the forward genomic strand. Mate pair reads are typically on the same strand, with the first end having a large co-ordinate when mapped to the forward genomic strand. Multi end, paired end, or single end data can be incorporated into the same Reads FASTQ file as long as the data follows the above rules.

Another method to specify this file is through the use of a grammar:

<fastq>	:=	<fastq>@<read name><\n><info><\n>
<fastq>@<read name>	:=	<fastq>@<read name><\n><info><\n>@<read name>
<info>	:=	<sequence><\n><comment><\n><qualities>
<read name>	:=	[^<\n>]+
<sequence>	:=	<NT sequence>
<sequence>	:=	<CS sequence>
<NT sequence>	:=	[ACGTNacgtn.]+
<CS sequence>	:=	[ACGT][01234.]+
<comment>	:=	[^<\n>]+
<qualities>	:=	[!-~]+

An example of a reads file in nucleotide space can be found in Figure 5.3. An example of a reads file in nucleotide space with paired end reads can be found in Figure 5.4. An example of a reads file in color space can be found in Figure 5.5.

5.1.3 Exons File

This Exons file specifies an exon-like structure, with each line representing an exon. Each exon has four entries specifying the start contig, start pos, end contig, and end position in that order. An example of an Exons file can be found in Figure 5.2.

5.1.4 Scoring Matrix File

The Scoring Matrix file specifies how the local aligner should score gaps in the alignment, nucleotide substitutions, and if applicable, color substitutions.

Each entry is whitespace delimited. The first two entries represent the affine gap open penalty and the affine gap extension penalty. The next two entries represent the nucleotide substitution penalties (match then mismatch). For color space alignments, the final two entries represent the color substitution penalties (match then mismatch).

An example with a file for use with nucleotide space alignment can be found in Figure 5.6. An example with a file for use with color space alignment can be found in Figure 5.7.

5.2 BFAST Files

These files are generated by the BFAST utilities. Explicit examples of these files are not given since the are specified in the source code and will (hopefully) be created through the use of BFAST.

5.2.1 BFAST reference genome file

The BFAST reference genome file stores the sequence to which we wish to align. The sequence is stored in a binary format. Each base (or color) is stored in four bits: two bits for the raw base (or color), one bit to specify if the letter was an N (or a 4), and one bit to store if the base was upper case or lower case (not applicable to a color). The BFAST reference genome file stores only the forward strand. Therefore for a genome of size G (forward strand), we can estimate the total required storage size of a BFAST reference genome file to be $G/2$ bytes.

The contigs that compose the reference genome are indexed based on the order specified in the Reference genome FASTA file (see subsection 5.1.1) along with each contig's associated name (see section 4.2).

The BFAST reference genome file will have the prefix corresponding to the reference genome FASTA file. If the BFAST reference genome file is in nucleotide space, then it will have the suffix `.nt.brg`. If the BFAST reference genome file is in color space, then it will have the suffix `.cs.brg`. Information about the BFAST reference genome file can be found by using the command `bfast header` (see section 4.8). Please see the source code for the full internal binary representation.

5.2.2 BFAST index file

The BFAST index file stores the index and hash table for the BFAST reference genome file. The index and hash table are stored in a binary format, with only the forward strand indexed.

To estimate the required storage size of an index before creation, we must know the number of contigs, hash width and genome size (forward strand). It is interesting to note that the BFAST reference genome file size does neither depend on the keysize, key width, nor mask layout. If there are more than 256 contigs in the BFAST reference genome file then each starting position indexed will require 8 bytes of storage. If the are 256 or fewer contigs in the BFAST reference genome file then each starting position indexes will require 5 bytes of storage. This representation is handled internally and is not visible to the user. Since we index a four letter alphabet, the hash with width j will require 4×4^j bytes (4 bytes per hash entry). Thus if the genome size is G (forward strand), the estimated BFAST index file required storage size is approximately $5 \times G + 4 \times 4^j$ or $8 \times G + 4 \times 4^j$ for a small number (≤ 256) or large number (> 256) contigs respectively.

If the index was created with splitting (using `-d`), then there will be 4^d separate BFAST index files. The BFAST reference genome file will have the prefix corresponding to the reference genome FASTA file. Its suffix will correspond to the index number and bin number. The index number is specified during creation. The bin number corresponds to which part out of the 4^d (see `-d`) BFAST index files. Information about the BFAST index file can be found by using the binary `header` (see section 4.8). Please see the source code for the full internal binary representation.

5.2.3 BFAST matches file

The BFAST matches file is used to store Candidate Alignment Locations (CALs) for each read processed by `match` (see section 4.4). By default, this file is stored in binary format. This file can be converted to text format for manual inspection by using the utility `bmfconvert` (see section 4.9).

To estimate the file size *a prior* is difficult. The read length, read name length, and number of CALs for each read must be known. The factor that causes the majority of the file size bloat is the average number of CALs stored per read. This can be overcome by having an upper limit on the number of CALs to store (see section 4.4).

Typically, the file extension should be `.bmf`. The file format for the text version of the BFAST matches file is as follows.

All entries are tab delimited. The first line has two entries: the @ symbol appended to the read name, and number of ends of the read. The number of subsequent lines corresponds to the number of ends in the read. For each end of the read, we have the original reads sequence, original quality values, a flag indicated whether the maximum CALs was reached, the number of CALs found (0 if the maximum was reached), and the CALs. Each CAL has three fields: the contig (1-based), position (1-based), strand, and a string representing where in the read the keys hit (condensed). Please see the source code for the full internal binary representation.

5.2.4 BFAST aligned file

The BFAST aligned file is used to store the alignments of reads to the reference genome and is created by the command `localalign` (see section 4.5). By default, this file is stored in binary format. This file can be converted to text format for manual inspection by using the utility `bafconvert` (see section 4.7).

To estimate the file size *a prior* is difficult. The read length, read name length, and number of alignments, and the length of the alignments for each read must be known. The factor that causes the majority of the file size bloat is the average number of alignments stored per read. This can be overcome by having an upper limit on the number of alignments per read to consider (see section 4.5) or by filtering the alignments (see section 4.6).

The BFAST aligned file will have the prefix `bfast.aligned.file` and the file extension `.baf`.

Please see the source code for the full internal binary representation.

5.2.5 BFAST sequence alignment/map file

BFAST is able to produce alignments in the SAM format. Some data may not be able to be represented by the SAM format, for example triple-end or quad-end data (instead of paired-end etc.)..

BFAST produces a mapping quality for each read, which depends on the `-q` parameter to `bfast postprocess` (section 4.5). If a read is unmapped, due to having no CALs, too many CALs, or being filtered (see section 4.6 for the latter), then the mapping quality is zero. If a read has one alignment, then the mapping quality is set to 255. This also indicates a mapping quality for an

alignment could not be computed accurately. If a read has a mapping quality of zero, this means that there exists another alignment that has a better alignment score. Otherwise, the mapping quality indicates the number edits away the current alignment is away from the next-best alignment in terms of alignment score. This is calculated by taking the current alignment's alignment score minus the next-best alignment's alignment score, then dividing by the alignment score of an atomic edit (a nucleotide change in nucleotide space and a color change for color space). Mapping quality should be calibrated on a experimental basis and is highly sensitive to the sensitivity settings of alignment. In general, the mapping quality is more accurately assessed under higher sensitivity scenarios.

BFAST also produces optional fields. The optional fields produced by BFAST documented in the SAM format include: RG, LB, PU, PG, AS, MQ, NM, IH, HI, MD, CS, CQ, CM, CC, and CP. Some fields are produced only when optional arguments are given to BFAST. BFAST also produces two aligner-specific optional fields: XA, and XE. XA gives the postprocessing algorithm from `bfast postprocess` if used (see section 4.6). XE gives a string indicating where color errors occurred on the forward genomic strand and has the regular expression: $([0-4])+$. If a color error occurred, then this gives the original color from the color sequence.

5.3 Example Input Files

```
>NM_006435 2
gaggaaactgttgagaaaacggaactactggggaaagggagggctcactg
agaaccatcccggtaacccgatcaccgctggtcaccatgaaccacattgt
gcaaaccttctctcctgtcaacagcggccagcctcccaactacgagatgc
tcaaggaggagcaggaagtggctatgctgggggtgccccacaaccctgct
cccccgatgtccaccgtgatccacatccgcagcgagacctccgtgcctga
ccatgtggtctggtccctgttcaacaccctcttcatgaacacctgctgcc
tgggcttcatagcattcgcgtactccgtgaagtctagggacaggaagatg
gttggcgacgtgaccggggcccaggcctatgcctccaccgccaagtgcct
gaacatctgggccctgattttgggcatcttcatgaccattctgctcatca
tcatcccagtgttggtcgtccaggcccagcgatagatcaggaggcatcat
tgaggccaggagctctgcccgtgacctgtatcccacgtactctatcttcc
attcctcgccctgccccagaggccaggagctctgcccttgacctgtatt
ccacttactccaccttccattcctcgccctgtccccacagccgagtcctg
catcagccctttatcctcacacgcttttctacaatggcattcaataaagt
gtatatgtttctggtgctgctgtgacttcaaaaaaaaa
>NM_015644 3
agtgctctcttccgccttcagtgccctgctcatcaagggtctgggtttcc
cggtcctctggcgaggatcctccaaggcgtctcacatgaaccggctcaga
aacgccaaaatctacgtggagagagctgtcaagaagaagatctttacaat
ccaaggctgctacccggtgatccggtgtctcttgcgccggaggggctggg
tggagaagaagatggtccatcgctcaggccccaccctgcgcc
tgggcttcatagcattcgcgtactccgtgaagtctagggacaggaagatg
gttggcgacgtgaccggggcccaggcctatgcctccaccgccaagtgcct
gaacatctgggccctgattttgggcatcttcatgaccattctgctcatca
tcatcccagtgttggtcgtccaggcccagcgatagatcaggaggcatcat
```

Figure 5.1: An example of a reference genome FASTA file.
See subsection 5.1.1 for a description.

```
1   891540  1   892246
1   895320  1   896847
1   897118  1   897867
1   897904  1   900541
1   1129098 1   1129929
1   1130413 1   1130935
1   1131428 1   1132152
1   1256389 1   1259906
1   2312874 1   2313457
1   2316883 1   2317370
```

Figure 5.2: An example of an Exons file
See subsection 5.1.3 for a description.

```
@4:150:844:843
GAGCGTATCGAGGCTCTAAAAAGATGTATACTAGCATTCTTCTCT
+
IIIIIIII*III3IIIIIIIIIIIIIIII,?II<1III+IIIII
@4:150:353:142
TGATTCATATCATGATGCTGGTAAACATTTTCTTTATGGTTCTCT
+
II-II.IIIIIE*%&II%&II%II?4II/8I%9I.(I((2%&6%B
@4:150:495:390
TTCGCATGTTTCTCCTTTTTTTCCCCTTCTTTCACTCTTCCTTTT
+
III4?IIIIIIDIIIIIIIII3IIIII8II7%,'2&?I%*-)II
```

Figure 5.3: An example of a reads FASTQ file in nucleotide space.
See subsection 5.1.2 for a description.

```
@4:150:276:201
TTATGCTAATTTGCATACTGACCAAGAACGTGATTACTTCATTCA
+
IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII3BII&)
@4:150:276:201
TTGTATGTTTTCATGCCTCCAAATCTTTGAGGCTTTTTTTTTTT
+
IIIIIIIIIII8IIIIIIII1III1II*>I+=/IIIIII;IIIII
@4:150:495:344
TGATTATGACCAGTGTTTCCAGTCCGTTTTTTTTTTTTTTTCT
+
IIIIIIIIIIIIIIIIII5IIIIICI?$III?I6I1%III#;
@4:150:495:344
TCTCACGTTGGCTGACGACCGCTTTGTGGCGTTTTTTATATTCT
+
IIIIIIIIIIIIII3IIIIB%<I1B)7I'IFE+I+I''C(%<&%
```

Figure 5.4: An example of a reads FASTQ file in nucleotide space with paired end reads. See subsection 5.1.2 for a description.

```
@5_20_383
T112323103300121021330102231101311010212110133112230
+
><912>4679?6,.-)/*/40=&/3=:&4309,.168'1.-...',6&*)
@5_20_1125
T120012000031030130133021211233311113000023331123 10
+
B?>7?@:8129?.:+685/93>2.+6>60,(<,)&&&%&'&*,&'/2'&(
@5_20_1365
T303201133233010301330320133303303330133230333323 13
+
0<?;?50+6:67%562925:9?,129?$+1-1$7./+1%5)&-3(',&%
@5_21_71
T103100203103003131222233113211231300220311311110 10
+
@0/@@4<.6;/'2>@??978,-''12.+1.&+%&'+'%+*0*$/%%)$)&
```

Figure 5.5: An example of a reads FASTQ file in color space. See subsection 5.1.2 for a description.

```
-175
-50
50
-150
```

Figure 5.6: An example of a Scoring Matrix file for nucleotide space.
See subsection 5.1.4 for a description.

```
-175
-50
50
-150
0
-125
```

Figure 5.7: An example of a Scoring Matrix file for color space.
See subsection 5.1.4 for a description.

Chapter 6

Advanced Topics

In this chapter we present a few different applications of BFAST, as well as how to design indexes.

6.1 How To Design Indexes

This section is meant as a brief introduction on how to design indexes. It will outline what you must ask to define indexes for your specific experiment.

We assume that you have a specific reference genome to which you wish to align, reads with a known length, an alphabet (ex. A,C,G, and T) with a known size, an intuitive feeling of what error-rate and polymorphism-rate you wish to tolerate, and the amount of time you wish to wait for BFAST to complete.

We first begin by determining the optimal key-size that corresponds to your reference genome size, alphabet size, and key length. As you may have read from the BFAST paper (Homer *et al.* (2009a)), we wish to make the lookup in an index return on average one CAL. Given a genome of size G, a key size of k, reads of length L, and an alphabet of size A, we compute the expected number of "false" random key matches F to be

$$F = (L - k + 1) \times \frac{G}{A^k}$$

This can be expressed in R code as:

```
# Calculates the number of ``false'' random key matches
# in the index given:
#   L: read length
```

```
#   k: key size
#   G: genome size (both + and -)
#   A: alphabet (DNA is 4)
F <- function(L=50, k=18, G=2*3.2*10^9, A=4)
{
return ( (L - k + 1) * (G / (A^k)) );
}
```

In this case, we test k over a varying range of values to find the smallest value of k where F is less than one. In theory this is the key size we should use. In practice, for larger genomes, the distribution of bases is non-random, for example in the Human Genome there are many stretches of long repeats. Therefore we advise you to choose a key size of $k + 2$, to further guarantee the uniqueness of the lookup to be performed. For the Human Genome, key sizes of 18 or greater will suffice.

After deciding on our key size, we move to explicitly creating the masks for the indexes. This is achieved by using the binary utility `btestindexes` (see subsection 4.12.9). This utility must be run twice, first to find a set of masks, and a second time to estimate the accuracy of those masks. We suggest using a key width (number of zeros and ones in your mask) greater than your key size (required) but also smaller than your read length, since the smaller the key width the more offsets can be used during the lookup step(Homer *et al.* (2009a)).

Nevertheless, using the `btestindexes` utility allows the user to examine various mask sets and their associated estimated accuracy against many possible error and variant combinations. We recommend that the user selects the minimal number of masks sufficient to tolerate the user's desired accuracy tolerances. The fewer masks used, the faster the alignment will be performed.

After finding a set of masks to use for alignment, the final step is to select the hash width to use. The hash accelerates the lookup by building an index of the prefixes of all possible keys in the index. In general, the hash width will take an exponential amount of space relative to the given hash width. For example, a good hash width for the Human Genome is 14, which will add approximately 1GB to the index size (see subsection 5.2.2). For smaller genomes, much smaller hash widths can be used.

Finally, both disk storage and random access memory sizes need to be considered. Based on all of the parameters above, it easy easy to calculate the required size in bytes of each index to be created. This can be found in subsection 5.2.2.

If the indexes become to large, we urge you to upgrade your machine with more random access memory memory and disk space given cost of such an up-

grade compared to the actual generation (sequencing) of the data. If all attempts at convincing the President fail, we suggest you further divide the indexes by ranges across the genome. This can be achieved by using the `-s`, `-S`, `-e`, and `-E` in `index` (see section 4.3).

6.2 Whole-Genome Alignment

Whole-Genome alignment is as simple as following the work flow presented in section 3.3.

6.3 Targeted Genomic Alignments

There are a number of ways to target specific regions within the genome, for example by specifying a subset of chromosomes or a number of contiguous regions.

The first method is to use `index` and to specify an Exons file (see section 4.3). The second method is to use command line options to limit the starting contig and position, and ending contig and position.

Applications of this type of index creation included targeted pull-down methods, where it is known only a certain set of regions will be sequenced.

6.3.1 Using `index` and exon list

To target specific regions of a larger reference genome, we can specify a Exons file when creating the indexes in `index` using the `-x` option (see section 4.3). This will limit the locations indexed to just those specified in the Exons file. Subsequently, options that limit the number of CALs returned by a key look up or in total for a read (see section 4.4 and section 4.5) will only be relative to this reduced index.

6.3.2 Using command-line options to specify one contiguous range

The command line options `-s`, `-S`, `-e`, and `-E` can be used to only consider one contiguous range within the BFAST reference genome file. This specified during the index creation (see section 4.3), during the local alignment step (see section 4.5), or when prioritizing alignments (see section 4.6). The step at which these options are specified will affect the resulting output.

If specified during the index creation step, the BFAST index file will only contain the sequence from that region. Thus, only CALs within this range will be found. If we are limiting the number of CALs returned by a key or in total for a read (see section 4.4 and section 4.5), then only CALs within the range will count towards these limits.

If specified during the local alignment step, only CALs that fall within the specified range will produce alignments. If we are limiting the number of CALs returned by a key or in total for a read (see section 4.4 and section 4.5), then all CALs that are possible in the indexes will count towards these limits. For example, if the index is of the whole genome, but we are interested in one chromosome, then if the CAL limits are used the limits are imposed according to CALs found in the whole genome, not the specified region. This may be useful if we want to flag reads that have high homology to a larger region or genome, but to have the alignments only be outputted within a specified range. Furthermore, since the local alignment step is typically the most expensive step for computation, ignoring alignments outside a certain range will reduce the number of local alignments needed.

If specified when prioritizing alignments, only alignments within the specified range will be outputted. This is similar to limiting the alignment range using `localalign` but is useful when a BFAST aligned file has been created with alignments to the full reference genome and we wish to only report alignments within a contiguous region.

6.4 Transcriptome Alignment

In some cases a contiguous reference genome is not the desired reference sequence. Examples include alignment to the transcriptome, including different transcript of genes, splice variants, or isoforms. This type of alignment can be easily handled by BFAST. We refer to each transcript, splice variant, isoform, or contigous sequence as a contig.

The sequence each possible transcripts should be given as independent contig in the Reference genome FASTA file when creating the BFAST reference genome file (see section 4.2). This will ensure that each transcript will be indexed separately and reported separately. Each step of the work flow (see section 3.3) should proceed as normal.

The contigs will be given an index number based on the order specified in the Reference genome FASTA file as well as outputting their name as defined in

the Reference genome FASTA file. In this manner the ID of the contig can be recovered. In a BFAST matches file (see subsection 5.2.3) generated by `match` (see section 4.4) only the index number is given for compactness. In a BFAST aligned file (see subsection 5.2.4) generated by `localalign` (see section 4.5) either the index number or original contig name can be used.

Other local alignment algorithms to support spliced alignments are currently under development and could be produced during local alignment.

6.5 Bisulfite Treated or Methylation Alignment

Bisulfite sequencing is an interesting experiment whereby we wish to know the methylation status of certain bases. We assume that in the sequence data some of the C bases have been converted to T by bisulfite treatment. In this case, we wish to align the sequence data to a reference genome, tolerating the fact that a fair number of mismatches when aligned will come from the fact that Cs have been converted to Ts.

BFAST can support this type of alignment. In brief, we will convert all Cs to Ts in the reference genome, convert all Cs to Ts in the sequence reads to be aligned (and annotate where those conversions were made), align the converted reads to the reference genome, then finally convert the reads back to their original state using the annotations.

Suppose we have 25 contigs representing the 25 chromosomes of the Human Genome. We convert each strand of each chromosome by changing every C to a T, for a total of 50 final methylated contigs (this must be done independently by the user). We use this "converted" reference genome as input when creating a BFAST reference genome file.

Next we convert every C to a T in each read in our input Reads FASTQ file (this must be done independently by the user). We can either annotate where each conversion occurred, or just store the original read. Either way, the annotation or the original read can be appended to the read name, since this will be kept throughout by BFAST.

After converting the reference sequence and the input reads, we run BFAST using the standard work flow (see section 3.3) with two exceptions. The first exception is that in `match` we wish to use the option `-w 1` so that we only match to the forward strand of each contig (see section 4.4). The reason for this is that we will index each strand separately. The second exception is that we have reduced our alphabet size from four (A, C, G, and T) to three (A, G, and T).

The reduced alphabet size must be taken into consideration when deciding on the masks for our indexes, since our genome complexity has been reduced. This should lead the user to use a larger key size to combat this reduced complexity.

After alignment using BFAST is complete, we simply convert back the read to its original state (this must be done independently by the user) thereby giving us the locations where there are Ts in the reference and Cs (or Ts) in the read.

6.6 Color Space Alignment

The work flow for color space has five steps as seen in Figure 6.1 similar in fashion to the one described in section 3.3.

1. In the first step we build two reference genomes: a nucleotide space genome using the option `-A 0` in `fasta2brg`, and a color space genome using the option `-A 1` (see section 4.2).

2. In the second step we create the indexes in color space by using the color space reference genome built in the first step and the option `-A 1` (see section 4.3).

3. In the third step we search for CALs using the color space indexes created in the second step, using the color space reference genome built in first step, and by using the option `-A 1` (see section 4.4).

4. In the fourth step we perform local alignment using the nucleotide space reference genome built in the fist step and by using the option `-A 1` (see section 4.5).

5. In the fifth step we prioritize the local alignments as was previously described in section 3.3.

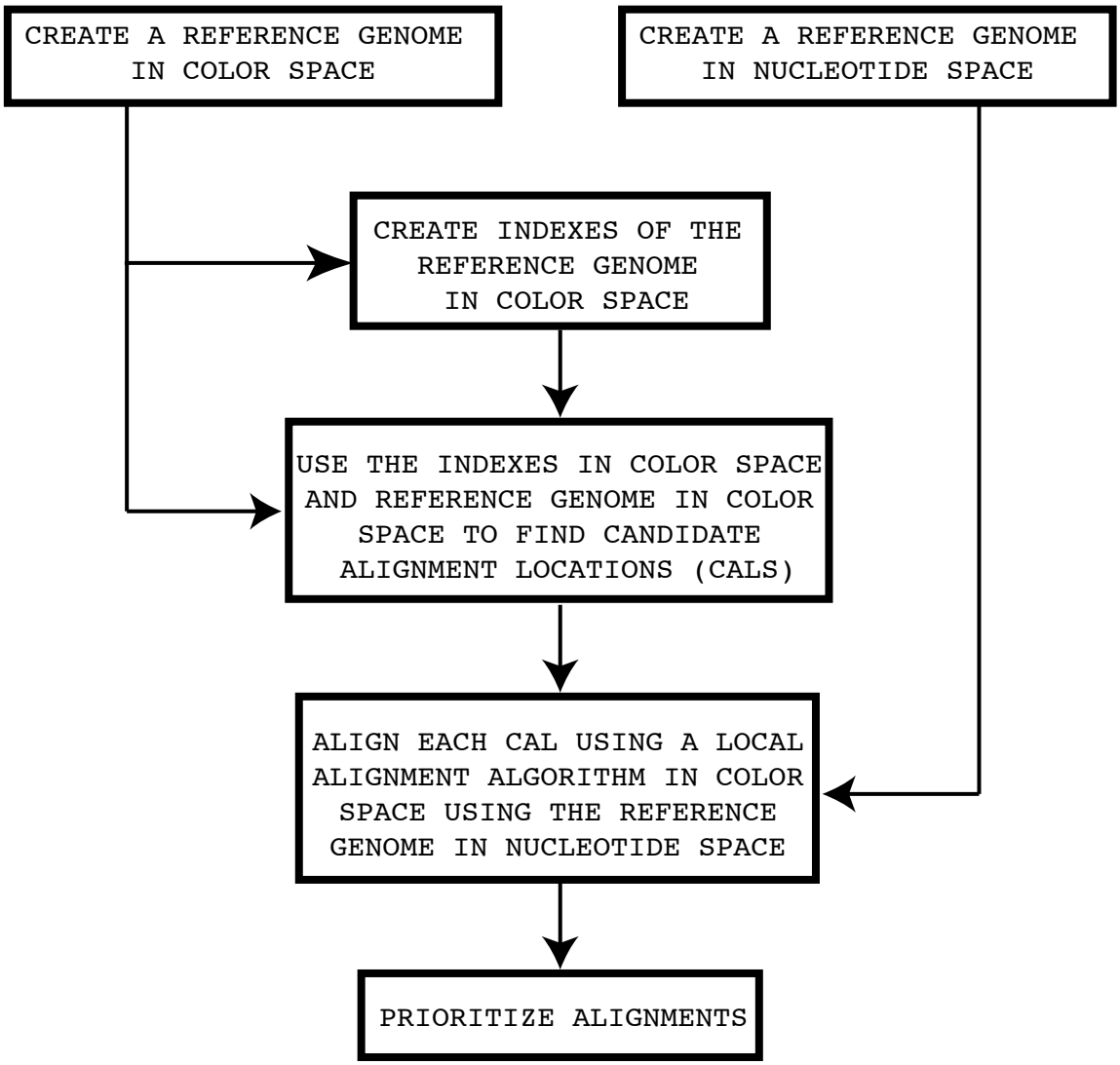

Figure 6.1: The BFAST work flow for color space alignment. See section 6.6 for a description.

Chapter 7

Appendix

7.1 Human Genome Alignment Recommended Settings

We assume that you have the full hg18 reference in the FASTA format in a file called hg18.fa. We detail our recommended commands to align sensitively to the human genome, at the cost of speed, outputting in the SAM format (see http://samtools.sourceforge.net). Further filtering, especially filtering on mapping quality, alignment score, and alignment quality, should be also be performed. Please use the `-n` option for multi-threaded alignment where possible. For Illumina data see subsection 7.1.1. For ABI SOLiD data see subsection 7.1.2

When mapping paired end or paired end data, the orientation must be specified in `bfast postprocess`. For paired end reads, use `-Y 0` or `-S 0 -P 1`. For mate pair reads, use `-Y 1` or `-S 1 -P 0`.

7.1.1 Illumina

We assume your reads are in Illumina QSEQ format (the files that end with _qseq.txt) and we wish to align lane $< N >$. Note, we align one lane all at one time but splitting the converted reads file allows for parallelism.

We suggest a hash width of 14, although this should be reduced if you are splitting the indexes for low-memory computation.

For reads less than > 40bp, the masks for the main indexes should be:

```
11111111111111111
11101000111000111010001101111
11110100110111101010101111
11111111111111001111
11110111011001010011111111
11110111000101010000010101110111
10110011010111101001100100010111
11101100101000010001011001110011111
11110111111111111111
11011111100010110111101101
```

For reads greater than or equal to 40bp, the main indexes should be:

```
111111111111111111111
111110111011101010010101101111
101111010110100101100001101000111111
101110011010011001001111101010001011111
111110110111011111011111111
111111100101001000101111101110111
111101011100101000101011010101111111
111101101011011001100000101101001011101
111101101000100011010110010110011010011
111101001011011010111001011011011
```

Given the above, the commands we should execute are:

Convert the reads:

```
$perl bfast-0.6.5b/scripts/qseq2fastq.pl -q s_<N>
```

Convert the reference:

```
$bfast-0.6.5b/bfast/bfast fasta2brg -f hg18.fa
```

Create the indexes:

```
$bfast-0.6.5b/bfast index -f hg18.fa -m <mask> -w 14 -i <index number>
```

Search the indexes:

```
$bfast-0.6.5b/bfast match -f hg18.fa -r reads.s_<N>.fastq > bfast.matches.file.s_<N>.bmf
```

Perform local alignment:

```
$bfast-0.6.5b/bfast localalign -f hg18.fa -m bfast.matches.file.s_<N>.bmf > bfast.aligned.file.s_<N>.baf
```

Filter alignments:

```
$bfast-0.6.5b/bfast postprocess -f hg18.fa -i bfast.aligned.file.s_<N>.baf
> bfast.reported.file.s_<N>.sam
```

7.1.2 ABI SOLiD

We assume your reads are at least 50bp in length. We will split the input into $10,000,000$ read pieces for parallel computation.

We suggest a hash width of 14, although this should be reduced if you are splitting the indexes for low-memory computation.

The masks for the main indexes should be:

```
1111111111111111111111
1111101001111100111111111111
1011111101100110001111100011111
1111111100101111000001100011111011
111111110001111110011111111
1111101101001100001100011001111111
11111111111100111011111111
1110110000111111111001111011111
1110110001011010011100101111101111
1111110010001100010111001100011000111111
```

Given the above files, the commands we should execute are:

Convert the reads:

```
$bfast-0.6.5b/scripts/solid2fastq -n 10000000 -o reads *.csfasta *.qual
```

Convert the reference (nucleotide space):

```
$bfast-0.6.5b/bfast fasta2brg -f hg18.fa
```

Convert the reference (color space):

```
$bfast-0.6.5b/bfast fasta2brg -f hg18.fa -A 1
```

Create the indexes:

```
$bfast-0.6.5b/bfast index -f hg18.fa -m <mask> -w 14 -i <index number> -A 1
```

Search the indexes:

```
$bfast-0.6.5b/bfast match -f hg18.fa -A 1 -r reads.<N>.fastq > bfast.matches.file.hg18.<N>.bmf
```

Perform local alignment:

```
$bfast-0.6.5b/bfast localalign -f hg18.fa -m bfast.matches.file.hg18.<N>.bmf -A 1
> bfast.aligned.file.hg18.<N>.baf
```

Filter alignments:

```
$bfast-0.6.5b/bfast postprocess -f hg18.fa -i bfast.aligned.file.hg18.<N>.baf -A 1
> bfast.reported.file.hg18.<N>.sam
```

Note that for parallel computation, execute `bfast match`, `bfast localalign`, and `bfast postprocess` for each converted input file created (replace $< N >$ with the input file number). Also, since color space local alignment may be slower than the match step, we can use the `-s` and `-e` options in `bfast localalign` to further parallelize the local alignment.

7.2 High-Speed Tutorial

Not for the faint of heart, most details will be omitted.

Your best bet is to follow the work flow in section 3.3 (or section 6.6 for color space). A quick list of relevant sections are as follows:

- **Step 1: subsection 4.2.1.**
- **Step 2: subsection 4.3.1.**
- **Step 3: subsection 4.4.1.**
- **Step 4: subsection 4.5.1.**
- **Step 5: subsection 4.6.1.**

7.3 Copyright

GNU GENERAL PUBLIC LICENSE Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Lesser General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually

obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable

form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations

under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation,

write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

¡one line to give the program's name and a brief idea of what it does.¿ Copy-

right (C) ¡year¿ ¡name of author¿

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

¡signature of Ty Coon¿, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License.

Bibliography

Homer, N., Merriman, B., and Nelson, S. F. (2009a). BFAST: An Alignment Tool for Large Scale Genome Resequencing. *PLoS ONE*, 4, e7767.

Homer, N., Merriman, B., and Nelson, S. (2009b). Local alignment of two-base encoded dna sequence. *BMC Bioinformatics*, 10(1), 175.

Kent, W. (2002). BLAT–the BLAST-like alignment tool. *Genome Res.*, 12, 656–664.

Smith, T. and Waterman, M. (1981). Identification of common molecular subsequences. *J. Mol. Biol.*, 147, 195–197.