Split Sequence Bloom Tree v1.0 User Guide

Brad Solomon and Carl Kingsford

July 9, 2018

1. Synopsis

SSBT is a program that will allow you to index a set of short-read sequencing experiments and then query them quickly for a given sequence. The first step to use SSBT is to create such an index. This is done via the "hashes", "count", "build", and "compress" commands. Then you can query that index for any sequence to find the files in which that sequence likely appears.

If you use SSBT, please cite:

Brad Solomon and Carl Kingsford. Improved Search of Large Transcriptomic Sequencing Databases Using Split Sequence Bloom Trees. Journal of Computational Biology. 2018. http://doi.org/10.1089/cmb.2017.0265

2. Analysis Pipeline Overview

To build and query a dataset using SSBT, you should follow one of two general pipelines. The first is the original SSBT method introduced at RECOMB 2017. It uses a greedy insertion schema which should be sufficient for most trees of size < 100 TBases. The general pipeline works as follows:

1. You first initialize the hash functions by running "hashes" command.
2. Convert each fasta file in the database to a bloom filter bit vector using the "count" command.
3. Compile a list of all the bit vectors generated and build the SSBT for that set using the "build" command.
4. Compress the bit vectors that make up the entire tree using the built in "compress" command.
5. Save your input queries as a line-separated text file and run the "query" command with the desired threshold and using the compressed SSBT file.

More details are given below.

If you are working on a larger dataset or are interested in producing a smaller overall index with a significantly shorter construction time at the cost of additional pre-processing steps, you should use the SSBT re-implementation of the SBT-ALSO (Sun Chen, Harris Robert S., Chikhi Rayan, and Medvedev Paul.Journal of Computational Biology. 2018. http://doi.org/10.1089/cmb.2017.0258). Our version uses a minhash schema to identify global pairwise similarity. This modified pipeline works as follows:

1. You first initialize the hash functions by running "hashes" command.
2. Convert each fasta file in the database to a bloom filter bit vector using the "count" command.
3. **Construct a minhash sketch for each bloom filter using the "minhash" command.**
4. **Construct an "instructions file" from the "instruct_from_minhash" command.**
5. **Build the SSBT using the "build_from_instruct" command.**

6. Compress the bit vectors that make up the entire tree using the built in "compress" command.
7. Save your input queries as a line-separated text file and run the "query" command with the desired threshold and using the compressed SSBT file.

More details are given below.

3. Analysis Pipeline Details

3.1 RECOMB 2017 Construction Schema

You first initialize the hash functions by running "hashes" command. The parameter `hashfile` is the hashfile output by the "hashes" command, the parameter `-k` sets the kmer index size (default 20), and the parameter `nb_hashes` sets the number of hash functions to use. A typical value for this is 1:

```
bt hashes [-k 20] hashfile nb_hashes
```

Next, you count the kmers in each of your input fastq/fasta files using the "count" subcommand; this will produce a bloom filter (with the name filter_out.bf.bv) for the given uncompressed file fasta_in.fasta :

```
bt count [--cutoff 3] [--threads 16] hashfile bf_size fasta_in.fasta filter_out.bf.bv
```

Note that the output from the count command is a Sequence Bloom Filter! Conversion to a Split Sequence Bloom Filter is as easy as renaming the extension to ".sim.bf.bv". To use SSBT, you MUST convert your bloom filters. Although we recommend using the default values for cutoff and counting threads, it is possible to adjust the minimum count required for a k-mer to be added to the bloom filter with the `--cutoff` parameter (the default is to include any element with a count of 3 or greater) or to adjust the number of threads used by the Jellyfish library with the `--threads` parameter. The parameter `hashfile` is output from the previous "hashes" command and the `bf_size` gives the bloom filter size. Although `bf_size` can be set arbitrarily large, we suggest setting the bloom filter size to the total count of unique kmers being inserted into the SSBT. We have provided a Jellyfish script (get_bfsize.sh) which takes in a list of fasta.gz files and compiles the total count of unique canonical kmers by their frequency in the total file set. Regardless of the tool used or number selected, it must be the SAME for all files that you are putting into the same SSBT. If you have lots of files on which to build a SSBT, you should write a little script that uses the above command to create a bloom filter for each. Something like:

```
for f in *.fasta ; do
    bt count hashfile bf_size $f `basename $f .fasta`.bf.bv
done
```

Now, create a list of the `.bf.bv` files that you just created as text file with one filename per line (using, say, `ls *.bf.bv > listoffiles.txt`) and call the "build" subcommand:

```
bt build [--sim-type 0] hashfile filterlistfile bloomtreefile
```

Here, `filterlistfile` is a list of the `.bf.bv` files. `sim-type` is an advanced command that changes the rule for where the files are inserted into the tree. After building (which may take some time), `bloomtreefile` will contain the data about the tree, and there will be many more `.bf.bv` files created.

Finally, you must compress the tree to query it:

```
bt compress bloomtreefile compressedbloomtreefile
```

This will create a compressed version of the tree (compressedbloomtreefile) which you can then query.

You can issue a query by putting 1 query sequence per line in a query text file and using the "query" subcommand:

```
bt query [--max-filters 1] [-t 0.8] [--leaf-only 0]
         [--weighted weightfile] bloomtreefile queryfile outfile
```

Here, `-t` gives the sensitivity threshold: the number of kmers that must be present for a query to be found. Values closer to 1 reduce the number of false positives. You can also weight the individual kmers in the query file by providing a complete array of weights for each kmer in a space separated weightfile.

3.2 Alternative Minhash Construction

The alternative minhash schema diverges slightly from the original pipeline in the construction step. The primary divergence is the construction of a minhash sketch for each bloom filter. This is done through the "minhash" command as follows:

```
bt minhash hashfile bloomfilter.bf.bv
```

The minhash sketch will have the same filename as "bloomfilter.bf.bv" but with the extension ".minhash". It will automatically be written to the same directory as the bloom filter.

Once you have a minhash for each bloom filter in your dataset, you can run the "instruct_from_minhash" command as follows:

```
bt instruct_from_minhash hashfile minhash_list out_instruction out_index
```

Given the hashfile and a list of minhashes, the "instruct_from_minhash" command will construct a list of instructions (out_instruction) and the output index that the instruction set will build (out_index). The list of instructions are human readable and indicate the exact order of merges necessary to produce an SSBT from the input set. This construction schema will write each file to disk at most twice – once to construct the internal node as a parent for two leaves and once to construct a parent node from a node. The output index is equivalent to the bloomtreefile from the "build" command; however in order to use this out_index as an SSBT, you have to build a tree from this set of instructions as follows:

```
bt build_from_instruct hashfile instruction_file
```

This will parse the instruction command to produce the SSBT index on the bloomfilters matching the minhash sketches. For this command to work properly, the bloom filters must be ".sim.bf.bv" extensions and located in the same directory as the minhash sketches. If they are not located in the same directory, you can manually adjust the instruction file to provide the proper path.

4. Command Descriptions

4.1 Hashes

bt hashes [-k 20] hashfile nb_hashes

- **k** is an optional parameter that sets the k-mer size used in every step of the SSBT pipeline
- **hashfile** is the location of the file being written
- **nb_hashes** is an integer that sets the number of hashes generated for the bloom filters

Usage:

To build a set of conserved hash functions for the bloom filters, use a command like:

bt hashes myhashfile.hh 1

This will write a file 'myhashfile.hh' which stores the necessary information for the Jellyfish library's bloom filter functions.

4.2 Count

bt count [–cutoff 3] [–threads 16] hashfile bf_size fasta_in filter_out.bf.bv

- **cutoff** is an optional parameter that sets the minimum count required required for a unique k-mer to be added to the bloom filter

- **threads** is an optional parameter that sets the number of threads the Jellyfish library uses when counting k-mers
- **hashfile** is the location of the hashfile written using the "hashes" function
- **bf_size** is the number of expected k-mers in the bloom filter.
- **fasta_in** is the location of the input fasta being counted
- **filter_out.bf.bv** is the location of the bloom filter being written

Usage:
To convert a fasta short-read file to a SSBT bit vector, use a command like:

bt count myhashfile.hh 2000000000 SRR001.fasta SRR0001.bf.bv

This will count and hash the k-mers associated with 'SRR001.fasta' using the settings defined by 'myhashfile.hh' and store all k-mers in 'SRR0001.bf.bv'

A note on setting the bf_size: As every filter must have a uniform size, bf_size should be set to an approximate count of the unique k-mers in your complete data set. This avoids saturation in the highest levels of the SSBT and is the extra space is largely factored out through the compression step. If space is a concern, it is also possible to set this value to be the size of the largest leaf filters, as overall accuracy is only affected by the false positive rate of the leaf filters. This will however greatly increase the run-time of SSBT queries.

4.3 Build

bt build [–sim-type 2] hashfile filterlistfile bloomtreefile

- **sim-type** is an option that defines the similarity metric used. (0) uses the default Hamming distance between the union of bit vector's similarity and remainder filters while (2) first counts the total number of matching 1's in the similarity filter and then uses Hamming distance on the remainder filter when there is no similarity between vectors.
- **hashfile** is the location of the hashfile written using the "hashes" function
- **filterlistfile** is the location of a plaintext file containing the paths to all the bit vectors generated by the "count" function
- **bloomtreefile** is the location of the SSBT structure file being written

Usage:
To build the bloomtree from a list of SSBT bit vectors, use a command like:

bt build myhashfile.hh mybitvectorlist.txt mySSBT.bloomtree

This will build the SSBT through single-threaded insertions of each element in 'mybitvectorlist.txt' and write the union filters to the same directory as the leaves. Once the tree is completely built, the edge-relationships that define the tree will be saved to 'mySSBT.bloomtree'.

4.4 Compress

bt compress bloomtreefile compressedbloomtreefile

- **bloomtreefile** is the location of the SSBT structure file written by the "build" function
- **compressedbloomtreefile** is the location of the [compressed] SSBT structure file being written

Usage:
To compress the bloomtree from bit vectors to rrr compressed vectors, use a command like:

bt compress mySSBT.bloomtree myCompressedSSBT.bloomtree

This will compress every file in the original SSBT and write a new bloomtree using the same edge-relationships but the rrr compressed files.

4.5 Query

bt query [–max-filters 100] [-t 0.8] [–leaf-only 0] [–weighted weightfile] bloomtreefile queryfile outfile

- **max-filters** is an option that defines the total number of filters that can be loaded at one time into memory. The default is set to 100 filters in memory.
- **threshold (t)** is a float between 0 and 1 that defines the proportion of query k-mers that must be present in any bloom filter to define a "hit". The default value assumes a valid hit contains 80% of exact-matching k-mers.
- **bloomtreefile** is the location of the SSBT structure file written by the "build" function or the compressed SSBT structure file written by the "compressed" function. Using the "compressed" file results in a substantially faster query time.
- **queryfile** is the location of a text file containing line-separated full-length sequences.
- **outfile** is the location of the [compressed] SSBT structure file being written

Usage:
To query the SSBT for an arbitrary set of sequences, use a command like:

bt query -t 0.8 mySSBT.bloomtree myQueryFile.txt myOutFile.txt

This will batch query the bloom tree encoded by 'mySSBT.bloomtree' for every line-separated sequence in 'myQueryFile.txt' at a query k-mer threshold of 0.8. If your query of interest is a housekeeping gene or is known to be expressed in the majority of files, it may be beneficial to set the 'leaf_only' option to 1 and ignore the tree structure by querying only the tree leaves.

4.6 Sim

bt sim [–sim-type 1] hashfile bvfile1 bvfile2

- **sim-type** is an option that defines the similarity metric used. (0) uses the default Hamming distance between two bit vectors while (1) uses a Jaccard index metric.
- **hashfile** is the location of the hashfile written using the "hashes" function
- **bvfile1, bvfile2** are any combination of two SSBT bit vectors constructed using the "count" function.

Usage:
To test the raw bit similarity between two bloom filters encoded by the SSBT, use a command like:

bt sim hashfile.hh SRR0001.bf.bv SRR0002.bf.bv

This will return the similarity using either the default Hamming (0) or Jaccard (1) metrics. We recommend using the Jaccard coefficient for most standard operations. The SSBT has several other similarity metrics, including a similarity coefficient which prioritizes any shared 1-valued bits in the similarity vector and breaks ties by comparing similarity in the remainder vector. This unnamed similarity is sim-type (2).

4.7 Minhash

bt minhash hashfile bvfile

- **hashfile** is the location of the hashfile written using the "hashes" function
- **bvfile** is any SSBT bit vector constructed using the "count" function.

Usage:
To construct a minhash sketch for a bit vector, use a command like:

bt minhash hashfile.hh SRR0001.bf.bv

The minhash sketch is internally set to use 100 hashes and will always write with the same name and file location (but ".minhash" extension) as the bit vector file. The current implementation supports up to 275 hashes. The SSBT minhash uses an integer to integer xorshift hash (George Marsaglia. Xorshift RNGs. 2003.) using the index of each 1-valued bit in a bloom filter for each hash.

4.8 Instruct_from_minhash

bt instruct_from_minhash hashfile minhash_list out_instruction out_index

- **hashfile** is the location of the hashfile written using the "hashes" function
- **minhash_list** is line-separated text-file containing the full path of a set of minhash sketches.
- **out_instruction** is the absolute path (or filename) for the output list of instructions needed to build a global pairwise construction SSBT using the smallest number of writes to disk.
- **out_index** is the equivalent file for the SSBT "build" command's bloomtreefile. It records the SSBT structure that can be constructed from the instruction set.

Usage:
To build an instruction file from a set of minhash sketches, use a command like:

bt instruct_from_minhash hashfile.hh minhash_list.txt mh_build_instructions.txt ssbt_out.txt

This will produce the instructions necessary for construction and provide the output file needed post-construction. **The ssbt_out.txt file produced by this process will not work without constructing the tree from the instruction set (see Build from Instruct).**

4.9 Build_from_instruct

bt build_from_instruct hashfile instruction_file

- **hashfile** is the location of the hashfile written using the "hashes" function
- **instruction_file** is the instruction set produced by "instruct_from_minhash" and saved to "out_instruction".

Usage:
To build an SSBT from an instruction file, use a command like:

bt build_from_instruct hashfile.hh mh_build_instructions.txt

This will produce the uncompressed SSBT defined by the instruction's out_index. At this point, you can treat the output file identically to one produced through the base build.