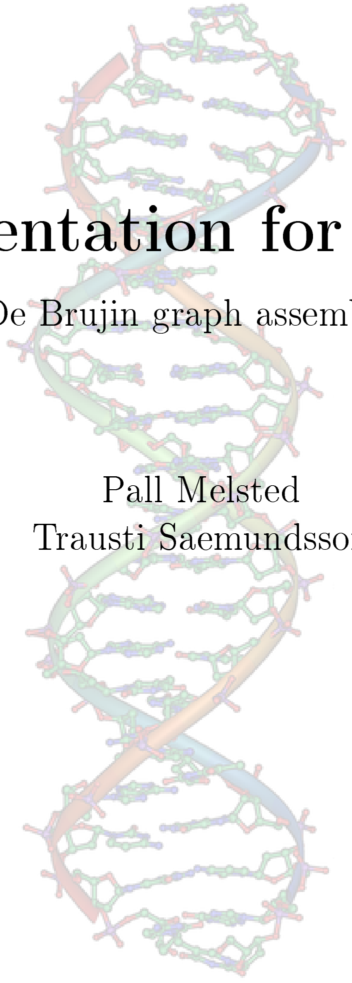

Documentation for BFGraph

A memory efficient De Brujin graph assembler using Bloom Filters

Pall Melsted
Trausti Saemundsson

BFGraph Documentation August 15, 2012

Questions and bug reports: pmelsted@gmail.com

Contents

Cover photo is from Wikipedia[10]

Chapter 1

Introduction

This is a documentation for the program BFGraph. BFGraph is a De Brujin graph assembler. Currently it makes the pregraph but does not simplify it like SOAPdenovo[8] and Velvet[9]. It makes the pregraph by using Bloom Filters in contrast to most other assemblers, that use hash tables. This saves a lot of memory and the result is independent of the false positive rate of the Bloom Filter.

A Bloom Filter is quite similar to a hash table. The difference is that if a Bloom Filter is queried for a specific key, it answer correctly if the key is not stored but answers incorrectly according to the false positive rate if the key is stored within it.

The first phase of BFGraph makes the pregraph according to a Bloom Filter but then fixes the graph afterwards which makes it independent of the probabilistic nature of the Bloom Filter.

The memory usage of BFGraph is a lot lower than that of most other assemblers, and it is compared in the paper about this program.

Chapter 2

Definitions

kmer[12]: String of A,C,G,T which has length k (k is often equal to 31)
backward-kmer: Add a base to the beginning of a kmer and skip the last character
forward-kmer: Add a base to the end of a kmer and skip the first character
one kmer is a neighbor of another kmer: Either one kmer is a backward-kmer of the other or a forward-kmer
twin: Interchange A <-> T and C <-> G in a string and then reverse it
contig: String of A,C,G,T which has length greater or equal to the kmer size
self-looped contig: The first and the last kmer of the contig are neighbors.
hairpinned contig: Either the last kmer of the contig is a neighbor of the twin of its forward-kmer or the first kmer of the contig is a neighbor of the twin of its backward-kmer. A contig can be hairpinned both forward and backward but it is rare.
read: A sequence of A,C,G,T or N from a sequencing machine[11], we only use kmers that have no 'N's.

Chapter 3

Dependencies

The program uses few programs made by others and are included in the program directory:

- **sparsehash**[4] is inside the directory google, a memory efficient hash table made by Google used in this program for storing common kmers.
- **libdivide**[7] in the file *libdivide.h* is used for fast integer division.
- **kseq**[1] in the file *kseq.h* is used for fasta/fastq file reading.
- **MurmurHash**[3] in the files *hash.cpp* and *hash.hpp* is used for hash functions in the Bloom Filter.

Chapter 4

Usage

First the program has to be compiled. Run `‘make‘` to do that. The directory *example* contains two small read files in fastq format: *tinyread_1.fq* and *tinyread_2.fq*.

Here follows a guide on how to run the program on these two files with kmer-size 31 (parameter **-k**). The script *example.sh* does the same as described below.

4.1 Part I: Filter the reads (FilterReads)

This command filters the reads and saves the result to a new file, *example/output/tiny.bf* (it will contain a Bloom Filter).

```
$ ./BFGraph filter example/tinyread_*.fq -k 31 -o example/output/tiny.bf -n 8000 -N 4000 -v
```

The parameter **-n** is an upper bound of the number of kmers from the read files and the parameter **-N** is an upper bound of the number of *different* kmers from the read files. The parameter **-v** is for verbose mode.

(a) Values for the parameters -N and -n

On the TODO list for BFGraph is to write a program to estimate these numbers from the reads. But until it has been written the user has to estimate them by himself.

The first number is easy to calculate by hand. The files *tinyread_1.fq* and *tinyread_2.fq* have in total 2000 reads of length 70. Since the kmer-size is 31, we will get 40 kmers from every read ($70 - 31 + 1 = 40$). Thus the number of kmers from the files is: $40 \cdot 2000 = 8000$.

NOTE: These number do not have to be accurate, but the program runs faster if they are close to correct values.

We can expect that an average kmer will be seen at least twice so 4000 is not a bad value for **-N**.

(b) Optional parameters

(b).1 -v

The parameter **-v** enables verbose mode. It is disabled by default.

(b).2 -t

The parameter **-t** sets the number of threads. The default value is 1.

(b).3 -c

The parameter **-c** sets the chunk-size for reads, i.e. how many reads are split between all the threads in every iteration. The default value is 20000.

(b).4 -b

The parameter **-b** sets the number of bits to use in the first Bloom Filter. The default value is 4.

(b).5 -B

The parameter **-b** sets the number of bits to use in the second Bloom Filter. The default value is 8.

(b).6 -s

The parameter **-s** is the seed to use for randomization. The default value is time based.

4.2 Part II: Create the contigs (BuildContigs)

This phase of the program reads the file *example/output/tiny.bf*, which is a Bloom Filter, and creates contigs from the kmers. The program must be run with the same kmer-size as the Bloom Filter file was created with, in this case 31. This command creates the contigs in one thread and saves the results into files with the prefix: *example/output/tiny*.

```
$ ./BFGraph contigs example/tinyread_*.fq -k 31 -f example/output/tiny.bf -o example/output/tiny -v
```

(a) Optional parameters

(a).1 -v

The parameter **-v** enables verbose mode. It is disabled by default.

(a).2 -t

The parameter **-t** sets the number of threads. The default value is 1.

(a).3 -c

The parameter **-c** sets the chunk-size for reads, i.e. how many reads are split between all the threads in every iteration. The default value is 1000 (Note that the default values are different for filter and contigs).

(a).4 -s

The parameter **-s** is the stride. Every stride-th kmer in a contig is mapped to the contig. The default value is the kmer size.

4.3 Extra: Visualize the De Brujin graph

For small read files the Python program *make_graph.py* can create a **.dot** file with Graphviz[5] to visualize the De Brujin graph.
If you ran the commands above you can now run this program with the prefix from Part II above.

```
$ ./make_graph.py example/output/tiny
```

This creates the file *example/output/tiny.dot*.

This file can be read a native **.dot** file reader like ZGRViewer[13] or converted to **.PNG** if Graphviz is installed with the following command:

```
$ dot -Tpng example/output/tiny.dot -o example/output/tiny.png
```

Chapter 5

Outline of the algorithm

5.1 FilterReads

(a) Short description

Filter out erroneous kmers.

(b) Detailed description

Go through every kmer in every read (we skip kmers that contain 'N's). If a kmer is not in the first Bloom Filter, insert it. Else check if it is in the second Bloom Filter, if not insert it there.

The size of the first Bloom Filter is controlled by the parameter **-n** and the size of the second Bloom Filter is controlled by the parameter **-N**.

The first Bloom Filter is for keeping track of all the kmers but the second Bloom Filter is only for keeping track of kmers that are seen more than once.

The parameter **-n** should be close to the number of all distinct kmers in the reads but **-N** should be close to the number of distinct kmers that are seen more than once. Thus the **-n** parameter should always be bigger than **-N**.

The first Bloom Filter works like a sieve to filter out the erroneus kmers.

After filtering all the kmers in all the reads the second Bloom Filter will contain every kmer that occurs more than once and some more kmers in direct ratio to the false positive rate, which is controlled by the parameter -N. Those extra kmers do not pose a problem because they are handled in BuildContigs.

5.2 BuildContigs

(a) Short description

Here we make the pregraph using the second Bloom Filter created in FilterReads.
Detailed description:

(b) Detailed description

Here we iterate again through every kmer in every read. We skip kmers not in the second Bloom Filter (and of course kmers containing any 'N's). Other kmers are most likely seen more than once and we use the second Bloom Filter to create a contig surrounding those kmers.

The process is as follows.
If a kmer is not a part of an already created contig see (a) else see (b):

(a)
- Create a contig which starts by being just the kmer itself.
- There are always four possible forward-kmers from every kmer (add A,C,G or T). If only one of those forward-kmers is in the second Bloom Filter we add this character to the contig and then repeat this process with the corresponding forward-kmer.
- If some forward-kmer is the same as the original kmer we stop increasing the contig (self-loop).
- If a kmer is a neighbor of the twin of its forward-kmer we stop increasing the contig (hairpin).
- We do the same repetitively for all possible backward-kmers.
- If a kmer is a neighbor of the twin of its backward-kmer we stop increasing the contig (hairpin).

Every contig is created along with data structure to keep track of how many times each kmer within it occurs (coverage of kmers). Initially every kmer has zero coverage. When the contig has been created we increase the coverage of the kmers that came from the read by one.

Since the contigs are created with respect to the (second) Bloom Filter the read does not have to intersect with all of the contig. Hence the beginning or the end might be partially zero covered.

(b) Increase the coverage of all kmers from the read that intersect with the already created contig.

When we have finished iterating through all the kmers from every read we have a lot of contigs. But some of these contigs might have kmers with zero or one coverage, which means that the Bloom Filters lied to us. Then we split the contigs one those kmers.

There might also be contigs that can be joined together but the Bloom Filter told us that there were two or more connections. We join all such contigs.

We now save all contigs along with unique ids to a file. We also save all connections between contigs to a file. It is easy to see that those contigs are independent on the probabilistic nature of the Bloom Filters since we split and join the contigs afterwards.

Chapter 6

Structure of the program

The file *BFGraph.cpp* is compiled into the executable file *BFGraph* when ‘make‘ is run. *BFGraph* runs the correct functions based on the input.

When the command ’filter’ is given, the method *FilterReads* is called, (implemented in *FilterReads.cpp*).
When the command ’contigs’ is given, the method *BuildContigs* is called, (implemented in *BuildContigs.cpp*).
A detailed call graph is shown on Figure 6.1.

The program flow is quite similar for those two commands. Parameters are first validated and an appropriate error message is given on any error. If all the parameters are valid, either *FilterReads_Normal* or *BuildContigs_Normal* is called depending on the command name. Both those methods go through all the reads (in a while loop) in as many threads as given by the parameter **-t**.

The OpenMP[6] library is used for parallel programming in BFGraph.

6.1 FilterReads.cpp

(a) FilterReads_Normal

This methodcreates two Bloom Filters (by using the class *BloomFilter*), one big based on the parameter **-n** and one small based on the paramter **-N**.

Then a while loop goes through all the reads in parallel. Each kmer in every read is taken and checked if is already in the first Bloom Filter, if so it is put into the second Bloom Filter. If it is not in the first Bloom Filter it is put there.

When this has finished the second Bloom Filter contains all the kmers that were seen at least twice, but it contains more kmers whose count is affected by the false positive rate of the Bloom Filter, which is controlled by **-n**. Thus if a kmer is in the second Bloom Filter it was "probably" seen twice. This Bloom Filter is then saved to a file provided by the parameter **-o**.

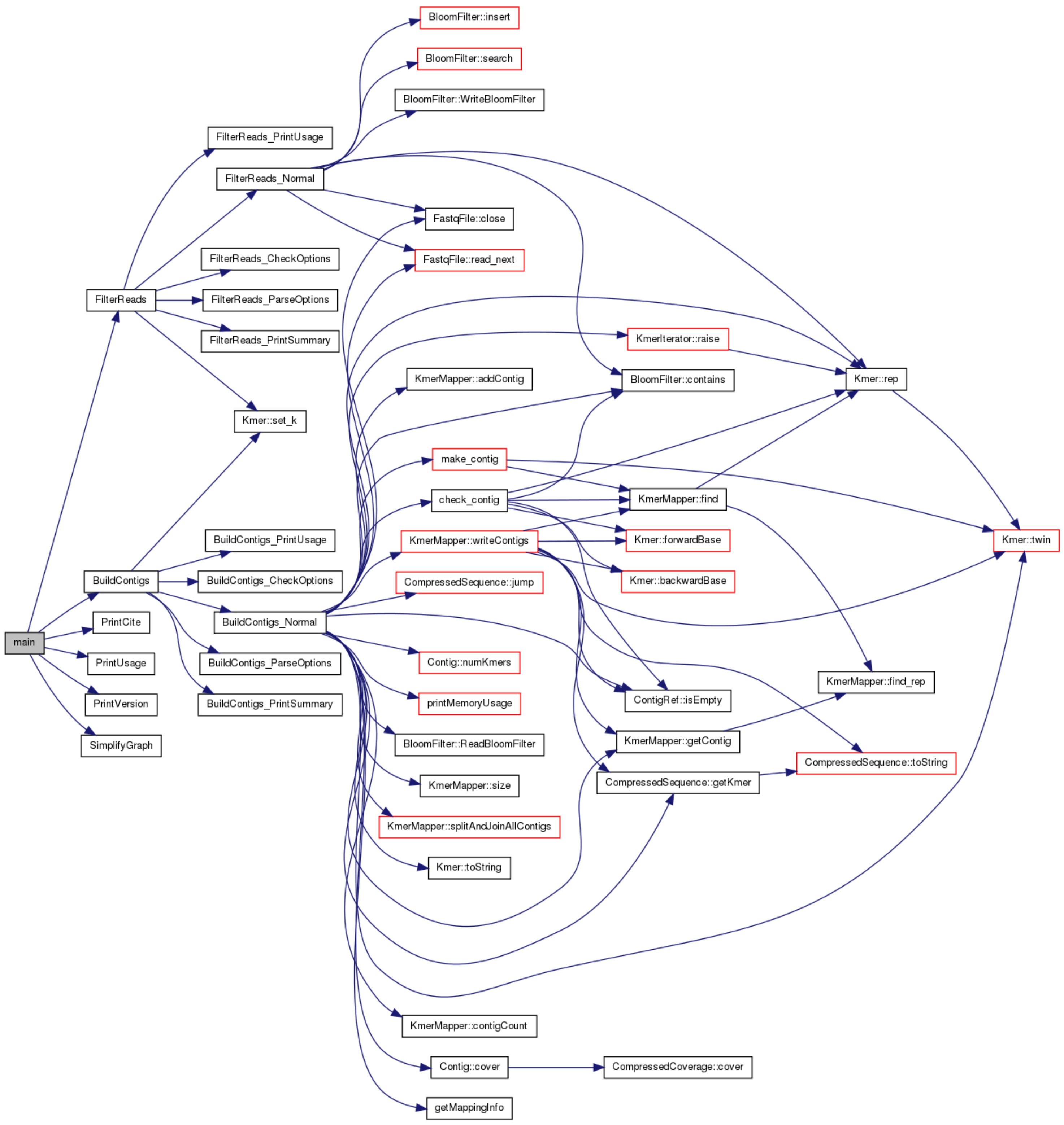

Figure 6.1: The call graph from the main function in BFGraph.cpp. Generated by Doxygen[2]

6.2 BuildContigs.cpp

(a) BuildContigs_Normal

This method reads the Bloom Filter, with the parameter **-f**, that was created with *FilterReads.* It also creates an instance of *KmerMapper* which uses the *sparse_hash_map* from Google to store where kmers are located within contigs. We say that a kmer 'maps' to a contig if the kmer or its twin is a part of the contig.

Now a while loop goes through all the reads in parallel. Each kmer in every read is taken and the Bloom Filter is asked whether it contains this kmer or not. If not nothing is done, but if the Bloom Filter contains it a few things are done.
First the method *check_contig* is called to check if the kmer is inside an already created contig. If so the coverage of this kmer is increased in that contig (the method *getMappingInfo* is used to get the correct location inside the contig, implemented in *ContigMethods.cpp*).
Else if the kmer does not map to a contig the method *make_contig* is called, implemented in *ContigMethods.cpp.* This method calls the method *find_contig_forward* which uses the Bloom Filter to travel from the original kmer, implemented in *FindContig.cpp.* It travels as far as possible but stops if there is not exactly one possibility forward or backward. It also stops if a self-loop is found and it also stops if the resulting contig will be hairpinned.

Chapter 7

Files

7.1 Unused files

Every file in BFGraph's base directory is used by the program except the following files:

- *CountBF.hpp*
- *CountBF.cpp*
- *DumpBF.hpp*
- *DumpBF.cpp*
- *KmerIntPair.hpp*
- *KmerIntPair.hpp*
- *bloom_filter.hpp*
- *BlockedBloomFilter.hpp*

7.2 Used files

The following classes reside in files by a similar name except ContigRef, it is located in *KmerMapper.hpp* and FastqFile is located in *fastq.hpp*. After each class name is a short description of its purpose.

(a) Classes

- *BloomFilter*: Store a Bloom Filter in memory, write the Bloom Filter to a file, read a Bloom Filter from a file.
- *CompressedCoverage*: Store the coverage of each kmer in a contig with as few bits as possible.
- *CompressedSequence*: Store the bases in a contig with as few bits as possible.
- *Contig*: Store bases in a contig with CompressedSequence and store its coverage with CompressedSequence
- *ContigRef*: Store a mapping location inside a contig or store a pointer to a Contig instance.
- *FastqFile*: Open multiple fasta/fastq files.

- *Kmer*: Store the bases in a kmer with as few bits as possible, give the twin kmer, give the forward or backward kmer.
- *KmerIterator*: Iterate through kmers in a read.
- *KmerMapper*: Store where kmers map to contigs, make new contigs, map contigs, join or split contigs while preserving the mapping and coverage info.

(b) Structs

- *FilterReads_ProgramOptions*: Store parameter values from the user for the subcommand 'filter'
- *BuildContigs_ProgramOptions*: Store parameter values from the user for the subcommand 'contigs'
- *CheckContig*: Store results from the fuction *check_ contig*
- *FindContig*: Store results from the fuction *find_ contig_forward*
- *MakeContig*: Store results from the fuction *make_ contig*
- *NewContig*: Store contig information, used in *BuildContigs_ Normal*.
- *KmerHash*: Calculate the hash of a Kmer instance.

Chapter 8

Important functions

8.1 In *KmerMapper.cpp*

- *mapContig*: Map a contig
- *addContig*: Add a contig to the map
- *joinTwoContigs*: Join two contigs and preserve their mapping and coverage info
- *joinAllContigs*: Call joinTwoContigs for all contigs that can be joined
- *splitAllContigs*: Split all contigs that have low coverage, make new contigs with correct mapping and coverage info.

8.2 In *Kmer.cpp*

- *forwardBase*: Puts a character after the end of a kmer and returns the resulting kmer
- *backwardBase*: Puts a character before the beginning of a kmer and returns the resulting kmer
- *twin*: Returns the twin of the given kmer.
- *rep*: Returns the alphabetically smaller, the kmer or the twin of the kmer

8.3 In *CompressedCoverage.cpp*

- *cover*: Increase coverage of kmers

8.4 In *CompressedSequence.cpp*

- *jump*: Compare a read string to a compressed dna sequence and return how long the continuous intersection is.

8.5 In *FindContig.cpp*

- *find_contig_forward*: Call the method forwardBase of a given kmer successively while there is only one resulting kmer in a given Bloom Filter. Prevents self-loops and hairpinned contigs.

8.6 In *ContigMethods.cpp*

- *getMappingInfo*: Gets the correct location of a kmer inside a contig.
- *check_contig*: Checks whether a kmer maps to a contig or not, and stores the results in an instance of *CheckContig*.
- *make_contig*: Creates a contig around a kmer with help from the function *find_contig_forward*.

Chapter 9

Final words

The code has been tested quite extensively both on many different read inputs and there are custom tests in the directory **tests**. We still appreciate all bug reports to pmelsted@gmail.com.

Thanks for reading!

Bibliography

[1] attractivechaos. *Generic stream buffer plus a FASTA_Q parser.* [Online; accessed 10-August-2012]. 2012. URL: https://github.com/attractivechaos/klib/blob/master/kseq.h.

[2] Doxygen. *Generate documentation from source code.* [Online; accessed 14-August-2012]. 2012. URL: http://www.stack.nl/~dimitri/doxygen/.

[3] Google. *smhasher - Test your hash functions.* [Online; accessed 14-August-2012]. 2012. URL: http://code.google.com/p/smhasher/.

[4] Google. *sparsehash - An extremely memory-efficient hash_map implementation.* [Online; accessed 10-August-2012]. 2012. URL: http://code.google.com/p/sparsehash/.

[5] Graphviz. *Graphviz - Graph Visualization Software.* [Online; accessed 13-August-2012]. 2012. URL: http://www.graphviz.org/.

[6] OpenMP. *The OpenMP API specification for parallel programming.* [Online; accessed 13-August-2012]. 2012. URL: http://openmp.org/wp/.

[7] ridiculous_fish. *libdivide is an open source library for optimizing integer division.* [Online; accessed 10-August-2012]. 2012. URL: http://libdivide.com/.

[8] SEQanswers - SEQwiki. *SOAPdenovo - SEQwiki.* [Online; accessed 10-August-2012]. 2012. URL: http://seqanswers.com/wiki/SOAPdenovo.

[9] SEQanswers - SEQwiki. *Velvet - SEQwiki.* [Online; accessed 10-August-2012]. 2012. URL: http://seqanswers.com/wiki/Velvet.

[10] Michael Ströck. *An overview of the structure of DNA.* [Online; accessed 9-August-2012]. 2006. URL: http://commons.wikimedia.org/wiki/File:DNA_Overview2.png.

[11] Wikipedia. *DNA sequencing — Wikipedia, The Free Encyclopedia.* [Online; accessed 14-August-2012]. 2012. URL: http://en.wikipedia.org/wiki/DNA_sequencing.

[12] Wikipedia. *k-mer — Wikipedia, The Free Encyclopedia.* [Online; accessed 9-August-2012]. 2012. URL: http://en.wikipedia.org/wiki/K-mer.

[13] ZGRViewer. *ZGRViewer, a GraphViz/DOT Viewer.* [Online; accessed 13-August-2012]. 2012. URL: http://zvtm.sourceforge.net/zgrviewer.html.