RNA-Seq Graph Builder Manual

Stefano Beretta*

March 28, 2012

RNA-Seq Graph Builder is a method to reconstruct the Splicing Graph of a gene from RNA-Seq data, without the genome information, where such a graph is a representation of the variants of alternative splicing of the gene structure.

This program predicts from NGS data the gene structure induced by the different full-length isoforms due to alternative splicing. More precisely, it analyzes RNA-Seq reads that have been sampled from the transcripts of a gene, with the goal of building a graph representation of the variants of alternative splicing corresponding to those full-length isoforms. The novelty of this method relies on the fact that it builds such a graph in absence of the genome.

1 Download and Installation

RNA-seq-Graph-Builder is implemented in $C{+}{+}$ and is currently distributed only on source form. It has been developed on Ubuntu Linux machines (v. 10.04 and 10.10) and has been tested on both 32 and 64 bit. The program also requires the $C++$ library *SEQAN* available at `http://www.seqan.de` or in Ubuntu systems it is possible to install the develop package seqan-dev by typing:

```
$ sudo apt-get install seqan-dev
```

Download

RNA-seq-Graph-Builder is developed on the AlgoLab/RNA-seq-Graph Git repository hosted by GitHub. The repository can be explored using the GitHub web interface at `https://github.com/AlgoLab/RNA-seq-Graph`.

*DISCo, Univ. di Milano-Bicocca, Milan, Italy email: beretta@disco.unimib.it

It is also possible to clone the entire repository using the following command:

```
$ git clone git://github.com/AlgoLab/RNA-seq-Graph.git
```

The source code is available directly in:

zip `https://github.com/AlgoLab/RNA-seq-Graph/zipball/v2.0.0`

tar.gz `https://github.com/AlgoLab/RNA-seq-Graph/tarball/v2.0.0`

Compilation

The program can be compiled by issuing the command at the command prompt:

```
$ make
```

2 Usage

The program takes as input a FASTA file with the RNA-seq data of a gene and returns the RNA-Seq Graph. The program is executed by typing the following command:

```
$ ./bin/build_RNA_seq_graph [options] --reads <RNA-Seq_file>
```

where the possible options are:

`-o <graphML_out_file> (Default: std output)`

`--ref_level {1-5}`

1. Standard Algorithm (Default option)
2. Add tiny blocks
3. Add linking edges
4. Add small blocks
5. Refine overlapping nodes

A summary of the available program options can be printed by invoking:

```
$ ./bin/build_RNA_seq_graph
```

without parameters, or:

```
$ ./bin/build_RNA_seq_graph --help
```

Alternatively it is possible to view the debug options by typing:

```
$ ./bin/build_RNA_seq_graph --advanced
```

An example of usage is:

```
$ ./bin/build_RNA_seq_graph --reads Raw_reads_file.fa -o Out_file
```

3 File Formats

Input: RNA-Seq Dataset

RNA-Seq file is in FASTA format (.fa, .fas or .fasta). In particular, each line of the input file describes a single read and it is composed by at least 2 rows: the first one is the header of the read (that usually starts with '>') and the second one that contains the sequence. For example:

```
>B2GA004:3:100:1143:752#0/1 /source=region /gene_id=gene /gene_strand=+
GATGAAATACTACTTCTACCATGGCCTTTCCTGGCCCCAGCTCTCTTACATTGCTGAGGACGAGAATGGGAAGAT
```

Output: RNA-Seq Graph

The program produces as output a file in txt format that contains a list of nodes and arcs of the RNA-Seq graph. It also gives as output the same graph in GDL format (`http://www.absint.com/aisee/manual/windows/node58.html`). It also print on standard output the graph in GraphML format; this latter can be redirected into a file in order to visualize or export it. By default the files are *RNA-seq-graph.txt* and *RNA-seq-graph.gdl*. For example:

```
$ ./bin/build_RNA_seq_graph --reads Raw_file.fa > RNA-seq-graph.graphml
```

If the option `-o` is specified the 3 files will have the specified name:

```
$ ./bin/build_RNA_seq_graph --reads Raw_reads_file.fa -o Out-file
```

generates *Out-file.txt*, *Out-file.gdl* and *Out-file.graphml*.

4 Program Code Description

Files

The program is written in C++ and it is organized in the following set of files:

Main : is the "main" method and contains the initial menu (with the debugging options). In this file all the procedures in the pipeline of the (standard) algorithm are called.

read_fasta : is the file containing the procedures for building the *hash table* that index the reads. The function **read_fasta()** parses the RNA-Seq reads file, creates the entry and insets them into the table.

build_chains : in this file there are all the procedures for the creation of the chains by using the unspliced reads (i.e. vertices of the graph). There are also the functions to print the built chains (used in debug mode) and to print the reads (in different ways: left/right table, unspliced/spliced/perfectly spliced reads) in the *hash table*. The function **build_unspliced_chains()** is invoked as second step of the (standard) algorithm in order to build chains by overlapping half unspliced reads. After that, the procedure **merge_unspliced_chains()** try to "fuse" chains that overlaps each other (by using the *hash table*.

join_chains : in this file there are the procedures for the creation of the links among chains by using perfectly spliced reads (i.e. arcs of the graph). The function **link_fragment_chains()** creates those links and finally **print_graph()** creates the graph in output in different formats. In this file there also the following procedures:

- **check_cutted_frags()**: try to look if the fragments cut from the chains in the linking phase can be added as graph vertices.
- **confirm_links()**: adds the "weight" to the perfectly spliced reads by summing the frequencies of the spliced reads that identify the "same junction".
- **gap_linking()**: try to add further links by considering a possible "gap" in the junction.

Data Structures

The main data structures used in the program are the two hash tables used to index the reads. In the program they are implemented as:

```
//Elements of the hash tables
struct element_table{
    table_entry* p;
    bool unspliced;
    bool half_spliced;
};

//Type of Left/Right table
typedef std::map<unsigned long long, element_table> hash_map;

//Hash tables
struct tables{
    hash_map left_map;
    hash_map right_map;
};
```

In *tables* there are the *left_map* and the *right_map* hash maps that are both "map" of the standard library C++ in which:

key is the left (resp. right) fingerprint of the entry reads

value is an *element_table* in which there is the concatenated list of read entry (`table_entry*p`) and two flags to indicate that the entry is *unspliced* or *half spliced* (i.e. perfectly spliced).

In figure 1 a graphical representation of the types pf the data structures is shown. In this data structures, the reads are added to the hash tables by creating a concatenated list of *table_entry* elements; in fact each *table_entry* element is an instance of the class:

```
class table_entry{
    private:
    //Table List
    table_entry* r_next, r_prev;
    table_entry* l_next, l_prev;
    //Fingerprints
    unsigned long long left_fingerprint, right_fingerprint;
    //Chain List
    table_entry* chain_next, chain_prev;
    //Sequence frequency
    long frequency;
}
```

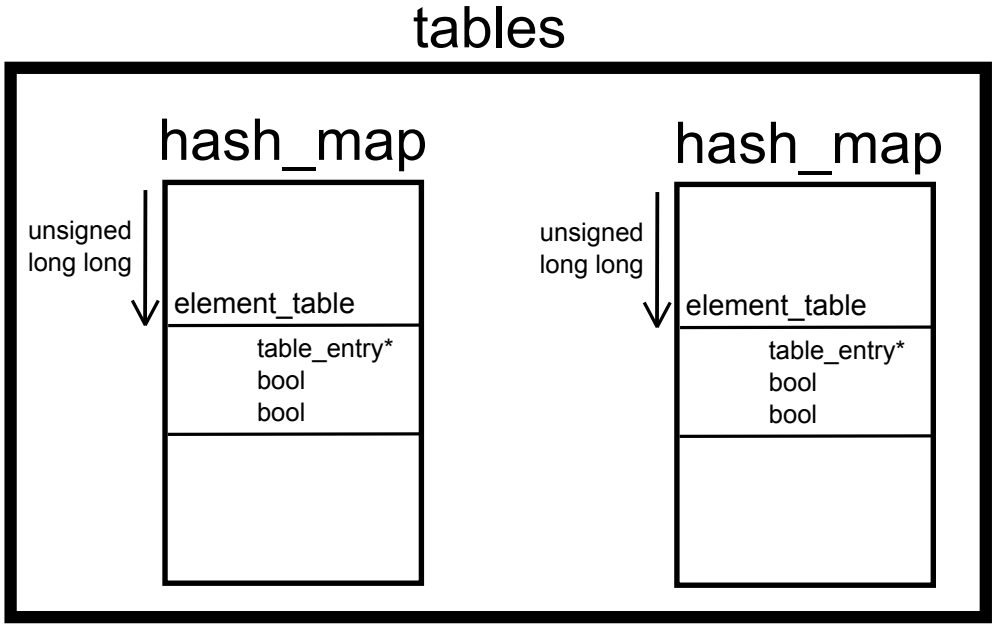

Figure 1: Types of the data structures used in the program

in which the public set/get methods are omitted.

So by using the pointers `l_next` and `l_prev` it is possible to build a bidirectional list of reads for the *left_map* table, in which all those reads share the same left fingerprints (i.e. the entry collision of the left hash table). The same *table_entry* object is also part of the bidirectional list of the right hash table and this is done by the two pointers `r_next` and `r_prev` for the reads that share the same right fingerprint. So each object is part of two list simultaneously (left and right).

The other *table_entry* pointers (`chain_next` and `chain_prev`) are the one used in the chain composition. In particular they link the current (unspliced) read to the (unspliced) one that share its left fingerprint with the right fingerprint of the current read (`chain_next`), and to the one that share its right fingerprint with the left fingerprint of the current read (`chain_prev`). The other values are the left and right fingerprints of the read and the frequency (i.e. the number of occurrences) of the read.

5 Memory Profiling

In order to monitor the memory consumption of the program we have used the library for the memory profiling `libmemusage.so`. This library can be preloaded using `LD_PRELOAD` and will intercept calls to malloc,free,realloc and various other calls. In short it will trace memory allocations for you:

```
$ LD_PRELOAD=/lib/libmemusage.so build_RNA_seq_graph --reads Raw_reads_file.fa
```

This commando will output on *std error* the memory usage of the program on the `Raw_reads_file.fa` dataset. An example of the reported output is the following:

```
Memory usage summary: heap total: 3655402386, heap peak: 183838093, stack peak: 7264
         total calls   total memory   failed calls
 malloc|   47158192     3655402386              0
realloc|          0              0              0  (nomove:0, dec:0, free:0)
 calloc|          0              0              0
   free|   47158202     3655402386
Histogram for block sizes:
    0-15           9135  <1%
   16-31          57130  <1%
   32-47        2592145   5% =====
   48-63       22528625  47% ================================================
   64-79           5686  <1%
   80-95        5897106  12% ==========
   96-111      12973065  27% ===========================
  112-127         14072  <1%
  128-143       2858573   6% ======
  144-159         53042  <1%
  160-175          3898  <1%
  176-191         10726  <1%
  192-207          1903  <1%
  208-223          8626  <1%
  224-239          2663  <1%
  240-255          6907  <1%
  256-271          1417  <1%
  272-287         32463  <1%
  288-303          2367  <1%
  304-319          4223  <1%
  320-335           736  <1%
  336-351          4181  <1%
  352-367          1567  <1%
  368-383          2363  <1%
  384-399           500  <1%
```

The *heap peak* value indicates the maximum number of Bytes used by the program, i.e. the memory requirement: in the previous example it is 183838093 Bytes, which is $\sim$ 175MB.

To use libmemusage all you have to do is to prepend `MEMUSAGE_OUTPUT=mytrace` and `LD_PRELOAD=/lib/libmemusage.so` to your application. This will instruct the library to write out a trace to the mytrace file.

This trace file can be converted to a graph using the memusagestat utility. It is not installed by most GNU distributions and can be either build from the glibc sources or from the QtWebKit performance measurement utilities. Using:

```
$ memusagestat -o output.png mytrace
```

an image with memory allocations and stack usage like the one at the end of this post will be created (i.e. `ouput.png`). The redline is the heap usage, the green one is the stack usage of the application. The x-scale is the number of allocations. The memory occupation of the previous example is reported in figure 2.

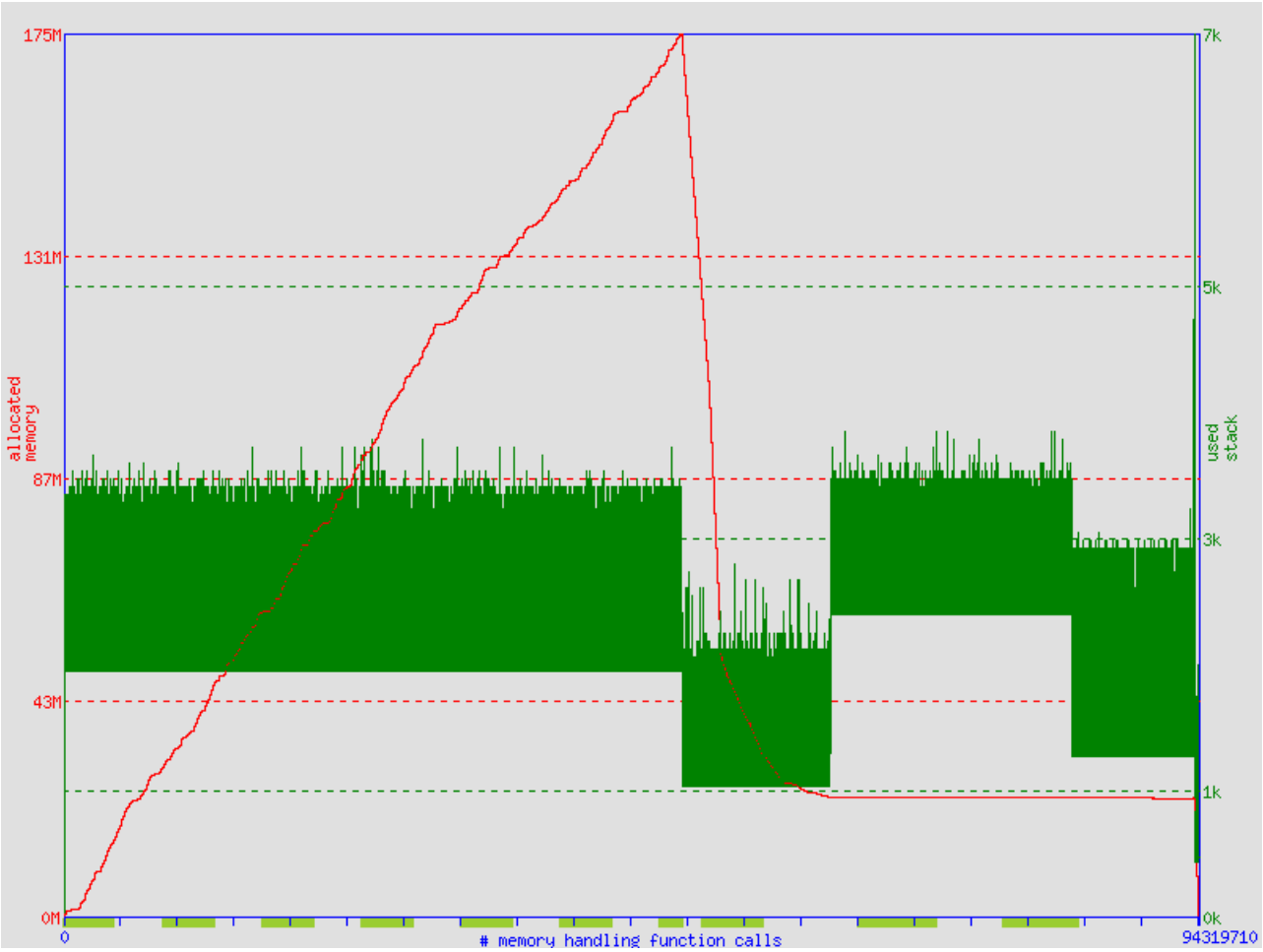

Figure 2: Example of memory usage output