sequencer, v. 1.15: Simulate Shotgun Sequencing

Bernhard Haubold
Max-Planck-Institute for Evolutionary Biology

October 27, 2017

Introduction

The program sequencer simulates shotgun sequencing. It takes as input a set of sequences and produces as output a collection of random reads. These can then be used, for example, to test assembly programs.

Getting Started

The sequencer software was written in standard C on a UNIX computer. It is operated from the command line. To start using the software, carry out the following steps:

- Unpack program

  ```
  tar -xvzf sequencer_xxx.tgz
  ```

  where xxx denotes the current version.

- Change into newly created directory

  ```
  cd Sequencer_xxx
  ```

- Print program options

  ```
  ./sequencer -h
  ```

- Test program

  ```
  ./sequencer phage_lambda.fasta
  ```

Listing: sequencer.c

```
/***** sequencer.c ************************************
 * Description: Program for simulating shotgun sequencing.
 * Author: Bernhard Haubold, haubold@evolbio.mpg.de
 * Date: Tue Jun 22 17:30:49 2004.
 * License: GNU General Public
 ******************************************************/
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>
```

```c
#include <math.h>
#include <unistd.h>
#include "sequence_data.h"
#include "eprintf.h"
#include "interface.h"
#include "ran.h"

void
run_sequencer (int fd, char *input_file, long *read_id, Args *args);
void process_read(FILE *fp, char *seq, long seqLen, long seqPos, int
    readLen, double err, int *dic, char *dicStr, int circular, int **
    profiles);
int
main (int argc, char *argv[])
{
  Args *args;                   /* arguments */
  char *version;                /* program version */
  int i, fd;
  long read_id;
  int idum, err;
  FILE *fpra;

  version = "1.15";
  args = get_args (argc, argv);

  if (args->h == 1)
    {
      print_usage ();
      return 0;
    }
  else if (args->v)
    {
      print_version (version);
      return 0;
    }
  /* seed for random number generation */
  if (args->s != 0)
    idum = args->s;
  else if ((fpra = fopen ("random_seed.dat", "r")) != NULL)
    {
      if((err = fscanf (fpra, "%d", &idum)) != 0)
        idum = -time(NULL);
      fclose (fpra);
    }
  else
    idum = -time (NULL);
  init_genrand (idum);

  read_id = 0;
  if (args->num_input_files == 0)
    {
      fd = 0;
      run_sequencer (fd, "stdin", &read_id, args);
    }
```

```
    else
      {
66      for(i = 0; i < args->num_input_files; i++)
          {
            if ((fd = open (args->input_files[i], O_RDONLY, 0)) <= 0)
              eprintf ("ERROR [sequencer]: cannot open file %s for reading\n"
                  , args->input_files[i]);
            run_sequencer (fd, args->input_files[i], &read_id, args);
71          close(fd);
          }
      }
    if (args->s == 0)
      {
76      fpra = efopen ("random_seed.dat", "w");
        fprintf (fpra, "%d\n", (int) genrand_int32 ());
        fclose (fpra);
      }

81  free (args);
    free (progname ());
    return 0;
  }

86 /* process_read: process the nucleotide sequence of a sequencing read
       defined by
   * seq: template sequence
   * seqLen: length of the template sequence
   * seqPos: position of the read on the tempate sequence
   * readLen: length of the sequencing read
91 * err: sequencing error per nucleotide
   * dic: map nucleotides to integers
   * dicStr: map integers to nucleotides
   * circular: 1|0 to indicate whethe or not the template sequence is
       circular
   * profiles: if this is NULL, print nucleotides, else, count up profile
96 */
  void process_read(FILE *fp, char *seq, long seqLen, long seqPos, int
      readLen, double err, int *dic, char *dicStr, int circular, int **
      profiles){
    long i, n, nuc1, nuc2;

    n = seqPos + readLen <= seqLen ? readLen : seqLen - seqPos;
101 for(i=0;i<n;i++){
      nuc1 = dic[(int) seq[seqPos + i]];
      nuc2 = nuc1;
      /* sequencing error */
      if (genrand_real1 () <= err)
106     while ((nuc2 = (int)(genrand_real1 () * 4)) == nuc1)
          ;
      if(profiles)
        profiles[seqPos + i][nuc2]++;
      else
111     fprintf (fp, "%c", dicStr[nuc2]);
    }
```

```
    if(circular){
      n = readLen - n;
      for(i=0;i<n;i++){
        nuc1 = dic[(int) seq[i]];
        nuc2 = nuc1;
        if (genrand_real1 () <= err)
          while ((nuc2 = (int)(genrand_real1 () * 4)) == nuc1)
            ;
        if(profiles)
          profiles[i][nuc2]++;
        else
          fprintf(fp,"%c", dicStr[nuc2]);
      }
    }
    if(!profiles)
      fprintf(fp,"\n");
  }

  /*
   * run_sequencer: This function is still rather confused. I find it very
     hard to integrate paired-end
   * reads in the current structure while still keeping the option to
     generate profiles instead of
   * sequencing reads. The following steps need to be disentangled:
   *   - Get fragment to sequence; this should perhaps be done explicityly:
     input sequence and output fragment as character array
   *   - Determine length of first read and either
   *       output read with error
   *     or
   *       count nucleotides for profiles with error
   *   - If reads are paired
   *       determine length of second read and either
   *         output read with error
   *       or
   *         count nucleotides for profiles with error
   */
  void
  run_sequencer (int fd, char *input_file, long *read_id, Args *args)
  {
    Sequence *seq;                /* sequence data */
    long residues;
    long sum, coverage, p, pf, printed_len, i, j;
    long seqi, seq_ind, max_res;
    long frag_len, read_len;
    long *num_res;
    int **profiles = NULL, *dic = NULL;
    long start, end;
    FILE *fout;
    char *sequence[2], strand;
    char **forward, **reverse;
    char *dic_str = "ACGTN";

    seq = NULL;
    seq = read_fasta (fd);
```

```
    fout = stdout;

166 /* generate conveniently accessible forward & reverse strands */
    forward = (char **) emalloc (seq->num_seq * sizeof (char *));
    reverse = (char **) emalloc (seq->num_seq * sizeof (char *));
    num_res = (long *) emalloc (seq->num_seq * sizeof (long));
    forward[0] = seq->seq;
171 num_res[0] = seq->borders[0];
    max_res = num_res[0];
    for(i = 1; i < seq->num_seq; i++){
        forward[i] = seq->seq + seq->borders[i - 1] + 1;
        num_res[i] = seq->borders[i] - seq->borders[i - 1] - 1;
176     if (num_res[i] > max_res)
          max_res = num_res[i];
        forward[i][num_res[i]] = '\0';
    }
    dic = get_restricted_dna_dictionary (dic);
181 if (args->p)
      profiles = initialize_profiles (profiles, max_res);
    else
      profiles = NULL;
    for (i = 0; i < seq->num_seq; i++)
186   reverse[i] = revcomp_string (forward[i]);
    coverage = args->c * seq->num_nuc;
    if(args->a)                   /* dealing with an alignment? */
      coverage /= seq->num_seq;
    sum = 0;
191
    sum = 0;
    seq_ind = 0;
    while (sum < coverage) {
        /* choose input sequence */
196     seq_ind = genrand_real1 () * seq->num_seq;
        sequence[0] = forward[seq_ind];
        sequence[1] = reverse[seq_ind];
        residues = num_res[seq_ind];
        (*read_id)++;
201     if(args->S) /* use fixed position */
          p = args->S - 1;
        else        /* generate random position */
          p = (int) (genrand_real1 () * residues);
        pf = p;
206     /* get random read length */
        read_len = args->R + (rand_Normal()*args->D) + 0.5;
        /* get random fragment length */
        frag_len = args->l + (rand_Normal()*args->d) + 0.5;
        if(read_len > frag_len)
211       read_len = frag_len;
        /* determine strand */
        if (!args->f){
          if (genrand_real1 () > 0.5){
            seqi = 1;
216         p = residues - p - 1;
          }else
```

```
      seqi = 0;
    }else
      seqi = 0;
    /* write sequence to output */
    i = 0;
    if (args->r || read_len <= residues - p + 1) /* circular or linear
        without edge*/
      printed_len = read_len;
    else
      printed_len = residues - p;                 /* linear with edge */
    if (seqi)
      strand = 'r';
    else
      strand = 'f';
    for (j = 0; j < strlen(seq->headers[i]); j++){
      if(seq->headers[i][j] == ' '){
        seq->headers[i][j] = '\0';
        break;
      }
    }
    if (!args->p && printed_len){
      if(seqi){
        end = pf + 1;
        start = pf - printed_len + 2;
      }else{
        start = pf + 1;
        end = pf + printed_len;
      }
      fprintf (fout, ">Read#%d file=%s sequence=%s strand=%c start=%ld
          end=%ld",
               (int)*read_id, input_file, seq->headers[seq_ind] + 1,
                  strand, start, end);
      if(args->P)
        fprintf(fout," mate=1\n");
      else
        fprintf(fout, "\n");
    }
    sum += printed_len;
    process_read(fout, sequence[seqi], residues, p, read_len, args->E,
        dic, dic_str, args->r, profiles);
    if(args->P){
      /* get new random read length */
      read_len = args->R + (rand_Normal()*args->D) + 0.5;
      p += frag_len - 1;
      if(args->r)          /* circular genome */
        p %= residues;
      else{               /* linear genome */
        if(p >= residues){
          if(residues - read_len < p - frag_len) /* does the read fit on
              the remaining fragment? */
            read_len = frag_len + residues - p - 1;
          p = residues-1;
        }
      }
```

```
        if(!args->f){ /* change strand */
          seqi = (seqi == 0) ? 1 : 0;
          if(seqi)
            pf = p;
          p = residues - p - 1;
          if(!seqi)
            pf = p;
        }
        if (seqi)
          strand = 'r';
        else
          strand = 'f';
        if (args->r || read_len <= residues - p + 1)
          printed_len = read_len;
        else
          printed_len = residues - p;
        if (!args->p && printed_len){
          if(seqi){
            end = pf + 1;
            start = pf - printed_len + 2;
          }else{
            start = pf + 1;
            end = pf + printed_len;
          }
          fprintf (fout, ">Read#%d file=%s sequence=%s strand=%c start=%ld
              end=%ld mate=2\n",
                   (int)*read_id, input_file, seq->headers[seq_ind] + 1,
                      strand, start, end);
          process_read(fout, sequence[seqi], residues, p, read_len, args->E
              , dic, dic_str, args->r, profiles);
        }
        sum += printed_len;
      }
    }
  if (args->p){
    if(args->C)
      print_profiles_single_contig (fout, profiles, max_res, args->m);
    else
      print_profiles (fout, profiles, max_res, args->m);
  }
}
```

Change History

- v. 1.5 (August 2008)
  - allow multiple input sequences
  - set for read length distribution $\sigma = \sqrt{\mu}$ (previously: $\sigma = \mu \times 0.1$)
- v. 1.6 (April 18, 2009)
  - added more information to the header lines of the reads
  - added the option to print out profiles instead of raw sequences
  - added `-f` option to sample from forward strand only
  - added `-p` option to print out profiles

  - added `-m` option to specify minimum coverage of valid profile
  - added sequencing error and corresponding `-E` option
  - added random seed handling via file `randomSeed.dat`
- v. 1.7 (May 19, 2009)
  - `-p` option now implies `-f`, which is not a user option any more
- v. 1.15 (October 27, 2017)
  - Reverted from configure/make system to my home-grown style in order to get in line with the other programs published as part of "Bioinformatics for Evolutionary Biologists, A Problems Approach".