`lzd`, v. 0.6: Lempel-Ziv Decomposition

Bernhard Haubold
Max-Planck-Institute for Evolutionary Biology, Plön, Germany

July 4, 2019

1 Introduction

The Lempel-Ziv decomposition (Ziv and Lempel, 1977) divides a string into repeating subunits. More formally, let S be a string, then starting at the first position of S, $S[1]$, look for the longest prefix of $S[1..]$ that is repeated somewhere to the left of $S[1]$. If there is no repeat, as in the case of the first character, the character itself is the factor. The search for the next factor would start at $S[2]$, and so on. For example, let $S = $ `CCCTCTGCGA`, the decomposition is

```
C.CC.T.CT.G.C.G.A
```

You can think of these factors as being "left"-repeats, as the repeat starting at $S[i]$ is always located to the left of $S[i]$, while the remainder of the string to the right-hand side is ignored.

The Lempel-Ziv decomposition is central to data compression algorithms and the program `lzd` simply serves as a didactic tool to generate the decomposition. This is done following the algorithm by (Crochemore et al., 2008) and a table like the one shown in their Figure 1 can also be generated by `lzd`.

2 Getting Started

`lzd` was written in C on a computer running Linux and should work on any standard UNIX system. However, please contact me at `haubold@evolbio.mpg.de` if you have any problems with the program.

- Unpack the program

  ```
  tar -xvzf lzd_XXX.tgz
  ```

  where `XXX` indicates the version.

- Change into the newly created directory

  ```
  cd Lzd_XXX
  ```

  and list its contents

  ```
  ls
  ```

- Generate `lzd`

  ```
  make
  ```

- List its options

  ```
  ./lzd -h
  ```

- The input string for `lzd` needs to be in FASTA format, which is widely used in bioinformatics and consists of a header line beginning with >, followed by the text on an arbitrary number of non-empty lines. For an example, take a look at the example sequence discussed above:

  ```
  cat Data/test.fasta
  >TestSeq
  CCCTCTGCGA
  ```

  To factorize it, type

  ```
  ./lzd ./Data/test.fasta
  C.CC.T.CT.G.C.G.A
  ```

- To factorize the example string used by (Crochemore et al., 2008), type

  ```
  ./lzd ./Data/algPaper.fasta
  a.b.b.a.abb.baa.ab.ab
  ```

- To also reproduce Figure 1 by (Crochemore et al., 2008), use the `-t` option:

  ```
  ./lzd -t ./Data/algPaper.fasta
  i w[i] sa[i] lcp[i] lpf[i]
  0 a 8 0 0
  1 b 9 2 0
  2 b 3 0 1
  3 a 12 1 1
  4 a 10 1 3
  5 b 0 0 2
  6 b 4 3 4
  7 b 13 0 3
  8 a 7 0 2
  9 a 2 0 3
  10 a 11 2 2
  11 b 6 1 2
  12 a 1 0 2
  13 b 5 2 1
  a.b.b.a.abb.baa.ab.ab
  ```

3 Listings

The following listings document central parts of `lzd`.

3.1 The Driver Program: `lzd.c`

```
/***** lzd.c **************************************
 * Description:
 * Author: Bernhard Haubold, haubold@evolbio.mpg.de
 * Date: Wed Jul 29 16:03:05 2015
 **************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include "interface.h"
```

```
#include "eprintf.h"
#include "sequenceData.h"
#include "complexity.h"

void scanFile(int fd, Args *args){
Sequence *seq;
char *origSeq;
  int i;
  long origLen;
  LempelZivFact *lzf;

  seq = readFasta(fd);
  origLen = seq->len;
  origSeq = seq->seq;
  for(i=0;i<seq->numSeq;i++){
    if(i){
      seq->len = seq->borders[i]-seq->borders[i-1]-1;
      seq->seq += (seq->borders[i]+1);
      if(args->m)
        lzf = mlComplexity(seq);
      else
        lzf = lzComplexity(seq);
      seq->seq -= (seq->borders[i]+1);
    }else{
      seq->len = seq->borders[i];
      if(args->m)
        lzf = mlComplexity(seq);
      else
        lzf = lzComplexity(seq);
    }
    lzf->str = origSeq;
    lzf->strLen = origLen;
    if(args->n){
      printf("#␣n\tn/site\n");
      printf("%ld\t%g\n",lzf->n,(double)lzf->n/(double)lzf->strLen);
    }else
      printLzDecomp(lzf,args);
    seq->seq = origSeq;
    freeLempelZivFact(lzf);
  }
  seq->len = origLen;
  freeSequence(seq);
  freeEsa();
}

int main(int argc, char *argv[]){
  int i, fd;
  char *version;
  Args *args;

  version = "0.6";
  setprogname2("lzd");
  args = getArgs(argc, argv);
  if(args->v)
```

```
    printSplash(version);
  if(args->h || args->e)
    printUsage(version);
  if(args->numInputFiles == 0){
    fd = 0;
    scanFile(fd, args);
  }else{
    for(i=0;i<args->numInputFiles;i++){
      fd = open(args->inputFiles[i],0);
      scanFile(fd, args);
      close(fd);
    }
  }
  free(args);
  free(progname());
  return 0;
}
```

3.2 Calculating the Enhanced Suffix Array: `esa.c`

```
/***** esa.c **************************************
 * Description: Enhanced Suffix Array.
 * Reference: Abouelhoda, Kurtz, and Ohlebusch
 *   (2002). The enhanced suffix array and its
 *   applications to genome analysis. Proceedings
 *   of the Second Workshop on Algorithms in
 *   Bioinformatics, Springer Verlag, Lectore Notes
 *   in Compter Science.
 * Author: Bernhard Haubold, haubold@evolbio.mpg.de
 * Date: Mon Jul 15 11:11:19 2013
 **************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <divsufsort.h>
#include <string.h>
#include "eprintf.h"
#include "esa.h"

Esa *globalEsa;
long *globalIsa;

long *getSa(Sequence *seq){
  long i, n, *sa2;
  sauchar_t *t;
  saidx_t *sa1;

  n = seq->len;
  t = (sauchar_t *)seq->seq;
  sa1 = (saidx_t *)emalloc((size_t)n * sizeof(saidx_t));
  if(divsufsort(t,sa1,(saidx_t)n) != 0){
    printf("ERROR[esa]:␣suffix␣sorting␣failed.\n");
    exit(-1);
  }
  sa2 = (long *)emalloc(n*sizeof(long));
  for(i=0;i<n;i++)
    sa2[i] = (long)sa1[i];
  free(sa1);
  return sa2;
}

/* getLcp: compute LCP array using the algorithm in Figure 3
 *   of Kasai et al (2001). Linear-time longest-common-prefix
 *   computation in suffix arrays and its applications. LNCS 2089
 *   p. 191-192.
 */
long *getLcp(long *sa, Sequence *seq){
  long i, j, h, n, *rank, *lcp;
  char *t;

  n = seq->len;
  t = seq->seq;
  rank = (long *)emalloc(n*sizeof(long));
  lcp = (long *)emalloc(n*sizeof(long));
  for(i=0;i<n;i++)
    rank[sa[i]] = i;
  h = 0;
  lcp[0] = 0;
  for(i=0;i<n;i++){
    if(rank[i] > 0){
      j = sa[rank[i]-1];
      while(t[i+h] == t[j+h]){
        h++;
      }
      lcp[rank[i]] = h;
      if(h>0)
        h--;
    }
  }
  globalIsa = rank;
  return lcp;
}

Esa *getEsa(Sequence *seq){
  Esa *esa;

  esa = (Esa *)emalloc(sizeof(Esa));
  esa->sa = getSa(seq);
  esa->lcp = getLcp(esa->sa,seq);
  esa->isa = globalIsa;
  esa->n = seq->len;

  globalEsa = esa;

  return esa;
}

void freeEsa(){
  free(globalEsa->sa);
  free(globalEsa->lcp);
  free(globalEsa->isa);

  free(globalEsa);
}
```

3.3 Lampel-Ziv Factorization: `factor.c`

```
/***** factor.c *********************************
 * Description: Compute the longest previous factor
 *   array using a suffix array and a longest
 *   common prefix array.
 * Reference: Crochemore, M., Ilie, L. and Smyth,
 *   W. F. (2008). A simple algorithm for com-
 *   puting the Lempel Ziv factorization. In:
 *   Data Compression Conference, p. 482-488.
 *   Computing longest previous factor in linear
 *   time and applications.
 * Author: Bernhard Haubold, haubold@evolbio.mpg.de
 * Date: Mon Jul 15 10:29:09 2013
 **************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "factor.h"
#include "stack.h"
#include "eprintf.h"
#include "esa.h"
#include "interface.h"

long *globalSa;
long *globalLcp = NULL;
long *globalLpf;
long *globalIsa = NULL;

long minimum(long a, long b){
  if(a < b)
    return a;
  else
    return b;
}

long maximum(long a, long b){
  if(a > b)
    return a;
  else
    return b;
}

void initGlobalLcp(Esa *esa) {
  int n = esa->n;
  globalLcp = (long *)emalloc(n * sizeof(long));
  for(int i=0; i < n; i++)
    globalLcp[i] = esa->lcp[i];
```

```
}

/*
 * computeLpf: Compute longest previous factor
 * Reference: M. Crochemore, L. Ilie, W.F. Smyth.
 *   A simple algorithm for computing the Lempel-Ziv
 *   factorization, in: J.A. Storer, M.W. Marcellini
 *   (Eds.), 18th Data Compression Conference, IEEE
 *   Computer Society, Los Alamitos, CA, 2008,
 *   pp. 482-488.
 */
long *computeLpf(Esa *esa){
  long i, n;
  long *lpf, *sa, *lcp;

  n = esa->n;
  esa->lcp = erealloc(esa->lcp,(n+1)*sizeof(long));
  esa->sa = erealloc(esa->sa,(n+1)*sizeof(long));
  lpf = (long *)emalloc((n+1) * sizeof(long));

  initGlobalLcp(esa);

  sa = esa->sa;
  lcp = esa->lcp;
  globalSa = sa;
  sa[n] = -1;
  lcp[n] = 0;
  lpf[n] = 0;
  stackInit(1);
  stackPush(0);

  for(i=1;i<=n;i++){
    while(!stackEmpty() &&
          (sa[i] < sa[stackTop()] ||
           (sa[i] > sa[stackTop()] && lcp[i] <= lcp[stackTop()]))){
      if(sa[i] < sa[stackTop()]){
        lpf[sa[stackTop()]] = maximum(lcp[stackTop()],lcp[i]);
        lcp[i] = minimum(lcp[stackTop()],lcp[i]);
      }else
        lpf[sa[stackTop()]] = lcp[stackTop()];
      stackPop();
    }
    if(i < n)
      stackPush(i);
  }
  freeStack();

  return lpf;
}

void freeLempelZivFact(LempelZivFact *lzf){
  free(lzf->lz);
  free(globalLpf);
  if(globalLcp)
```

```
    free(globalLcp);
  free(lzf);
}

LempelZivFact *computeLempelZivFact(Esa *esa){
  long i, n, *lpf;
  LempelZivFact *lzf;

  globalSa = esa->sa;
  globalLcp = esa->lcp;

  n = esa->n;

  lpf = computeLpf(esa);
  globalLpf = lpf;
  lzf = (LempelZivFact *)emalloc(sizeof(LempelZivFact));
  lzf->lz = (long *)emalloc(n*sizeof(long));
  lzf->lz[0] = 0;
  i = 0;
  while(lzf->lz[i] < n){
    lzf->lz[i+1] = lzf->lz[i] + maximum(1,lpf[lzf->lz[i]]);
    i++;
  }
  lzf->n = i;

  return lzf;
}

void printLzDecomp(LempelZivFact *lzf, Args *args){
  int i, j;

  globalLcp[0] = -1; /* follow convention used in my lectures */

  if(args->t){
    if(args->l){
      printf("\\begin{center}\n\\begin{tabular}{rcrrrrl}\\hline\\hline\n");
      printf("$i$␣&␣$\\mathtt{w}[i]$␣&␣$\\mathtt{sa}[i]$␣&␣$\\mathtt{lcp}[i
          ]$␣&␣$\\mathtt{isa}[i]$␣&␣$\\mathtt{lpf}[i]$␣&␣$\\mathtt{suf}[i]$
          \\\\\\hline\n");
    }else
      printf("i\tw[i]\tsa[i]\tlcp[i]\tisa[i]\tlpf[i]\tsuf[i]\n");
    for(i=0;i<lzf->strLen-1;i++){
      if(args->l){
        if(args->o)
          printf("%d␣&␣$\\mathtt{%c}$␣&␣%ld␣&␣%ld␣&␣%ld␣&␣%ld␣&␣$\\mathtt{"
              ,i+1,lzf->str[i],globalSa[i]+1,globalLcp[i],globalIsa[i]+1,
              globalLpf[i]);
        else
          printf("%d␣&␣$\\mathtt{%c}$␣&␣%ld␣&␣%ld␣&␣%ld␣&␣%ld␣&␣$\\mathtt{"
              ,i,lzf->str[i],globalSa[i],globalLcp[i],globalIsa[i],globalLpf
              [i]);
      }else{
        if(args->o)
          printf("%d\t%c\t%ld\t%ld\t%ld\t%ld\t",i+1,lzf->str[i],globalSa[i
       ]+1,globalLcp[i],globalIsa[i]+1,globalLpf[i]);
    else
     printf("%d\t%c\t%ld\t%ld\t%ld\t%ld\t",i,lzf->str[i],globalSa[i],
       globalLcp[i],globalIsa[i],globalLpf[i]);
   }
   for(j=globalSa[i];j<lzf->strLen-1;j++)
    printf("%c",lzf->str[j]);
   if(args->l)
    printf("}$\\\\");
   printf("\n");
  }
  if(args->l)
   printf("\\hline\\hline\\end{tabular}\n\\end{center}\n");
 }
 j = 0;
 if(args->l)
  printf("\\[\n");
 for(i=0;i<lzf->n-1;i++){
  if(args->l)
   printf("\\mathtt{");
  for(j=lzf->lz[i];j<lzf->lz[i+1];j++)
   printf("%c",lzf->str[j]);
  if(args->l)
   printf("}");
  if(args->l)
   printf("\\cdot");
  else
   printf(".");
 }
 if(args->l)
  printf("\\mathtt{");
 for(j=lzf->lz[i];j<lzf->strLen-1;j++)
  printf("%c",lzf->str[j]);
 if(args->l)
  printf("}\n\\]\n");
 else
  printf("\n");
}
```

4 Change Log

- Version 0.1 (November 25, 2015)
  - First version that works.
- Version 0.2 (April 13, 2017)
  - Output factors per site.
- Version 0.3 (February 13, 2018)
  - Allow one-based counting (`-o`).
- Version 0.4 (November 6, 2018)
  - Fixed bug in `interface.c`.
- Version 0.5 (November 17, 2018)

  - First entry in lcp-array is now -1 rather than 0.

- Version 0.6 (July 4, 2019)
  - When printing the match length decomposition, the variable `gloablLcp` was not initialized, which lead to a core dump. Fixed.

References

M. Crochemore, L. Ilie, and W.F. Smyth. A simple algorithm for computing the lempel-ziv factorization. In *Data Compression Conference, 2008. DCC 2008*, pages 482–488, 2008. doi: 10.1109/DCC.2008.36.

J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEE Transactions on Information Theory*, IT-23:337–343, 1977.