kr version 2.0.2: Efficient Estimation of Pairwise Distances between Genomes

Mirjana Domazet-Lošo[1,2] and Bernhard Haubold[1]

[1]Max-Planck-Institute for Evolutionary Biology Department of Evolutionary Genetics, 24306 Plön, Germany

[2]University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia

October 5, 2009

1 Introduction

kr is a program for efficiently computing the statistic K_r, an estimator of the pairwise number of substitutions between long DNA sequences. For details on this statistic see [3], which also accompanies version 1 of kr. The algorithmic improvements implemented in version 2 are described in [1].

2 Input and Output Formats

The input consists of DNA sequences in FASTA format. The alphabet of these sequences is restricted to the four canonical bases A, C, G, and T. The output is a distance matrix in PHYLIP format.

3 Getting Started

kr is written in C and is intended to run on any UNIX system with a C compiler and the GNU Scientific Library installed [2]. However, please contact MDL at mdomazet@evolbio.mpg.de if you have problems with the program.

- Unpack the software

  ```
  tar -xvzf kr_XXX.tgz
  ```

  where XXX indicates the version.

- Change into the newly created directory

  ```
  cd Kr_XXX/Src/Kr
  ```

  and list its contents

  ```
  ls
  ```

- Users of Mac OSX need to edit the file Makefile such that the assignments to the variables CFLAGS and CFLAGS64 are changed to

  ```
  CFLAGS= -O3 -Wall -Wshadow -pedantic -D_GNU_SOURCE                    \
  -D_FILE_OFFSET_BITS=64 -std=c99 -DVER32 -DUNIX -I/opt/local/include/  \
  -L/opt/local/lib
  ```

  and

```
CFLAGS64= -O3 -Wall -Wshadow -pedantic -D_GNU_SOURCE          \
-D_FILE_OFFSET_BITS=64 -std=c99 -DUNIX -I/opt/local/include/  \
-L/opt/local/lib
```

- Generate `kr`

  ```
  make
  ```

- List its options

  ```
  ./kr -h
  ```

- Test the program and record its run time

  ```
  time ./kr ../../Data/hiv42.fasta
  ```

  This carries out the standard distance computation. In the case of HIV sequences, however, it is safe to assume that the GC content does not vary greatly between sequences. By telling the program to use a global GC content rather than the GC content of each pair of sequences, the computation of K_{r} can be sped up:

  ```
  time ./kr -g ../../Data/hiv42.fasta
  ```

  The difference in run time becomes particularly pronounced for large data sets.

4 Listings

The following listings document central parts of the code for `kr`.

4.1 The Driver Program: `mainKr.c`

```
/***** mainKr.c
    ************************************************************
 * Description: Main program for kr2 (distance matrix computation based on
    Kr)
 * Author: Mirjana Domazet-Loso
 * File created on September 5th, 2008
 *
 ************************************************************************
    */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#if WIN
        #include <io.h>
        #include <windows.h> /* needed for file listing */
        #include "fileListWin.h"
#elif UNIX
        #include <unistd.h>
        #include <fcntl.h>
        #include "fileListUnix.h"
#endif

#include "commonSC.h"
```

```
#include "eprintf.h"
#include "interface.h"
#include "sequenceData.h"
#include "sequenceUnion.h"
#include "subjectUnion.h"
#include "mainKr.h"

/* #define MAXSIZE_32 536870912 */
#define MAXSIZE_32 500000000 /* upper limit for input data size for 32-bit
    program version, approx. 0.5 Gb, approx. 1 GB */

#if defined(_DEBUG) && defined(WIN)
        #include "leakWatcher.h"

        #define new DEBUG_NEW
  #undef THIS_FILE
  static char THIS_FILE[] = __FILE__;
#endif

int main(int argc, char *argv[]){

        FILE *fpout = NULL;
        FILE *fn = NULL;
        int i; // j, k;
        Int64 subjectNumSeq; // numStrain, lBorder;
        char *version;
  Args *args;
        char **seqNames = NULL;

        Sequence **subject;
        Sequence **sArrayAll; /* **sArray  */
        SequenceUnion *seqUnion;

        clock_t endAll, startAll;

        #if defined(_DEBUG) && defined(WIN)
                _CrtSetDbgFlag ( _CRTDBG_ALLOC_MEM_DF |
                    _CRTDBG_LEAK_CHECK_DF );
        #endif

        #if VER32
                setprogname2("kr");
        #else // VER64
                setprogname2("kr64");
        #endif
        #if DEBUG
                fprintf(stdout, "%s\n", progname());
                fprintf(stdout, "sizeof(Int64)=%llu\n", (unsigned long long
                    )sizeof(Int64));
        #endif
        startAll = clock();

        version = "2.0.1";
        // get arguments (including query and subject files)
```

```
args = getArgs(argc, argv);

        if (args->h == 1) {
                printUsage(version);
                return 0;
        }
        else if (args->p){
                printSplash(version);
                exit(0);
}
        else if (args->e){
                printUsage(version);
                exit(EXIT_FAILURE);
}

/*
        1) open input files: 1 or more subject file (S1.. Sn)
        2) read from input files and generate sequences
        --> every sequence has flag that is set to 1, .., N=numOfSubjects(
            for subjects)
        --> concatenate all the seq-s in one big sequenceUnion
        3) form suffix array (sa) from all sequences (from sequenceUnion)
        4) form longest common prefix array (lcp) from sa
        5) simulate sufix tree traversal using algorithm by Abouelhoda et
            al.
        --> every node in the tree keeps a list of its subjects and a list
            of its leaves;
    6) calculate shulens sl[i][j][k] from a subject Si,k-th position
        referring to subject Sj
          --> The shustring length at that position is the length of the path
              from the root to the deepest inner node common Si,k and Sj
          7) calculate Ir and Kr for every pair Si, Sj and form a matrix N x
             N
          */

        /* 1) open input files: 1 or more subject file (S1.. Sn) */
        /* check whether subject file(s) exist(s); if it doesn't, and the
            subject subdirectory is not specified, then consider the stdin
            as the subject file */
        if (args->subjectFileNumber == 0 && !args->d) { /* this means that
            the subject comes from the stdin */
        args->j = 0;
                args->subjectFileNumber = 1; /* one input file, but in fact
                     it is stdin */
        }
        else if (args->d) { /* subject files are in the subdirectory (of
            the current directory) specified with -d option */
                #if WIN
                        args->j = listFilesWin(args->d, &i); //(char **)
                            erealloc(args->j, numberOfFiles * sizeof (char
                            *));
                #elif UNIX
                        args->j = listFilesUnix(args->d, &i); //(char **)
                            erealloc(args->j, numberOfFiles * sizeof (char
```

```
                        *));
                #endif
                args->subjectFileNumber = i;
        }

        /* read data from subject file(s); array of subjects consists of
           Sequence objects where each object represents one file */
        subject = (Sequence **)/*e*/malloc(sizeof(Sequence *) * args->
           subjectFileNumber);
        readSubjects(subject, args, &subjectNumSeq);

        /* form a sequence union:
         * (i) concatenate all subject seq-s
         * (ii) define borders in seqBorders and bordersWithinSeq
        */
        prepareSeqUnion(&seqUnion, args, subject, subjectNumSeq, &sArrayAll
           );

        // open output file - results in phylip/neighbor format
        if (args->o) {
    fn = efopen(args->o, "w");
        }
        else {
    fn = stdout;
        }
        if (args->n) { /* detailed output */
    fpout = efopen(args->n, "w");
        }

        /* determine seq. names */
        seqNames = /*e*/malloc(sizeof(char *) * subjectNumSeq);
        for (i = 0; i < subjectNumSeq; i++) {
                /* skip > and start from the 1st character */
                seqNames[i] = /*e*/malloc((strlen(sArrayAll[i]->headers[0])
                     + 1 - 1) * sizeof(char));
                strncpy(seqNames[i], sArrayAll[i]->headers[0] + 1, strlen(
                   sArrayAll[i]->headers[0]) - 1);
                seqNames[i][strlen(sArrayAll[i]->headers[0]) - 1] = '\0';
        }

        /* print to the output stream */
        if (args->n) {
                fprintf(fpout,"ALL␣SUBJECT(S):\n");
                for (i = 0; i < subjectNumSeq; i++) {
                        //fprintf(fpout,"%-40.40s\t\t(%d)\n", sArrayAll[i
                           ]->headers[0], i + 1); // print all headers
                        fprintf(fpout,"%-40.40s\t\t(%d)\n", seqNames[i], i
                           + 1); // print all headers
                        #if DEBUG
                                fprintf(fpout,"%s\n", sArrayAll[i]->seq);
                                printf("%s\n", sArrayAll[i]->seq);
                        #endif
                }
                #if DEBUG
```

```
                        fprintf(fpout,"\n%s\n", seqUnion->seqUnion->seq);
                        printf("\n%s\n", seqUnion->seqUnion->seq);
#endif
        }

        /* release some of the memory */
        memoryDealloc1(subjectNumSeq, args->subjectFileNumber, subject,
            sArrayAll);

        scanSubjectUnion(fpout, seqNames, args, seqUnion, fn);

        #if DEBUG
                fprintf(stdout, "\nFinished!\n\n");
        #endif

        endAll = clock();
        if (args->t && fpout) { /* print run-time */
                fprintf(fpout, "\nWhole program: %.2f seconds.\n", (double)
                    (endAll - startAll) / CLOCKS_PER_SEC);
        }

        if (args->o) {
                fclose(fn);
        }
        if (args->n) {
                fclose(fpout);
        }

        /* memory deallocation */
        memoryDealloc2(args, seqUnion, seqNames);

#if defined(_DEBUG) && defined(WIN)
                _CrtDumpMemoryLeaks();
        #endif
        return 0;
}

/*
    ****************************************************************************
    */

/* memory deallocation for all the objects called in main */
#if UNIX
void memoryDealloc1(Int64 subjectNumSeq, int subjectFileNumber, Sequence **
    subject, Sequence **sArrayAll) {
#elif WIN
static void memoryDealloc1(Int64 subjectNumSeq, int subjectFileNumber,
    Sequence **subject, Sequence **sArrayAll) {
#endif

        /* when there is only 1 subject, sArrayAll[0] will be deallocated
            with freeSequenceUnion, because it is seqUnion->seqUnion */
        if (subjectNumSeq > 1) {
                freeSequenceArray(subjectNumSeq, sArrayAll);
```

```
206         }
            else { // only 1 subject
                    free(sArrayAll);
            }
            freeSequenceArray(subjectFileNumber, subject);
211 }


    #if UNIX
    void memoryDealloc2(Args *args, SequenceUnion *seqUnion, char **seqNames) {
216 #elif WIN
    static void memoryDealloc2(Args *args, SequenceUnion *seqUnion, char **
        seqNames) {
    #endif
            int i;
            for (i = 0; i < seqUnion->numOfSubjects; i++) {
221                 free(seqNames[i]);
            }
            free(seqNames);
            freeArgs(args);

226         freeSequenceUnion(seqUnion, seqUnion->numOfSubjects);
            free(progname());
    }


231 /* read data from subject file(s); array of subjects consists of Sequence
        objects where each object represents one file */
    void readSubjects(Sequence **subject, Args *args, Int64 *subjectNumSeq) {
            int subjectDscr = -1;
            Int64 i, j;
            Int64 totalLen = 0L;
236
            *subjectNumSeq = 0;

            for (i = 0; i < args->subjectFileNumber; i++) {
                    if (args->j == 0) { /* this means that the subject comes
                        from the stdin */
241                         subjectDscr = 0;
                    }
                    else if (subjectDscr == -1) {
                            subjectDscr = open((const char *)(args->j)[i], 0);
                    }
246                 /* if a file cannot be open, deallocate memory and
                        terminate the program */
                    if (subjectDscr < 0) {
                            for (j = 0; j < i; j++) {
                                    freeSequence(subject[j]);
                            }
251                         eprintf("ERROR [kr 2]: could not open subject file
                                %s\n", (args->j)[i]);
                    }
```

```
                /* if it's an empty file or an error file, then skip the
                   rest of the loop; NOW - this is allowed because stdin
                   can also be input!! */
                if (0) {//lseek(subjectDscr, 0L, SEEK_END) <= 0L ) { /* -1
                   for error, 0 for the file beginning */
256                     fprintf(stderr, "[WARNING] File: %s is empty!", (
                           const char *)(args->j)[i]);
                        subject[i] = NULL;
                }
                else { /*       if a file is ok, then read its sequence(s)
                   */
                        //lseek(subjectDscr, 0L, SEEK_SET); /* return to
                           the file beginning */
261                     subject[i] = readFasta(subjectDscr);
                        close(subjectDscr);
                        *subjectNumSeq += subject[i]->numSeq;
                        totalLen += subject[i]->len;
                        /* total input size exceeded the upper bound for
                           32-bit version --> error */
266                     if (totalLen >= MAXSIZE_32 && sizeof(Int64) == 4) {
                                fprintf(stderr, "ERROR[kr 2]: Total input
                                   size exceeded the upper bound for the
                                   input size of the 32-bit version of the
                                   program.\n Please use the 64-bit version
                                   .\n");
                                exit(EXIT_FAILURE);
                        }
                }
271             subjectDscr = -1;
        } /* end for */
}

/* form a sequence union as a concatenation of all subjects' sequences */
276 #if UNIX
void prepareSeqUnion(SequenceUnion **seqUnion, Args *args, Sequence **
   subject, Int64 subjectNumSeq, Sequence ***sArrayAll) {
#elif WIN
static void prepareSeqUnion(SequenceUnion **seqUnion, Args *args, Sequence
   **subject, Int64 subjectNumSeq, Sequence ***sArrayAll) {
#endif
281
        Sequence **sArray, *cat;
        Int64 i, j, k;

        *seqUnion = getSequenceUnion(subjectNumSeq, subjectNumSeq); /*
           subjectNumSeq = actual number of subjects in subject files */
286
        /* if subject[i] is consisted of multiple strains, then divide
           those strains in separate Sequence objects - each strain in one
           Sequence object */
        sArray = NULL; /* array of strains from one subject */
        *sArrayAll = (Sequence **)/*e*/malloc((size_t)subjectNumSeq *
           sizeof(Sequence *)); /* array of strains from all subjects */
```

```
        for (i = 0, j = 0; i < args->subjectFileNumber; i++) {          /*
            j: position in the sArrayAll which contains strains from all
            subjects */
291             if (subject[i]) { /* not an empty file */
                        sArray = getSubjectArray(subject[i]); /* sArray is
                            the list of sequences (Sequence objects) from i-
                            th subject */
                        for (k = 0; k < subject[i]->numSeq; k++) {
                                (*sArrayAll)[j++] = sArray[k];
                        }
296                     free(sArray);
                }
        }

        /* set borders of the sequence union */
301     cat = getSeqUnionBorders(sArrayAll, subjectNumSeq, seqUnion);

        (*seqUnion)->seqUnion = cat;
        (*seqUnion)->numOfSubjects = subjectNumSeq;
        (*seqUnion)->len = (*seqUnion)->seqBorders[(*seqUnion)->
            numOfSubjects - 1] + 1;
306     (*seqUnion)->seqUnion->queryStart = -1; /* there is no query */
        (*seqUnion)->seqUnion->queryEnd = -1;
}

/* form seq. union borders */
311 Sequence *getSeqUnionBorders(Sequence ***sArrayAll, Int64 subjectNumSeq,
    SequenceUnion **seqUnion) {

        Int64 j, k, numStrain, lBorder, numChar;
        Sequence *cat = NULL, *catOld;

316     lBorder = 0; /* lower border of a subject strain */
        /* form a seqUnion borders */
        for (j = 0; j < subjectNumSeq; j++) {
                prepareSeq((*sArrayAll)[j]);
    /* concatenate the strain[j] to the concatenated strains of all the
        subjects so far */
321             if (j == 0) {
                        cat = (*sArrayAll)[0];
                }
                else {
                        catOld = cat;
326                     cat = catSeq(catOld, (*sArrayAll)[j], 'S'); /*
                            catSeq allocates new object! */
                        if (j > 1) { /* when j == 1, catOld points to
                            sArrayAll[0] and this should not be deallocated
                            */
                                freeSequence(catOld);
                        }
                }
331             /* set borders of the sequence union */
                (*seqUnion)->seqBorders[j] = lBorder + (*sArrayAll)[j]->len
                    - 1; // len icludes border
```

```
                numStrain = (*sArrayAll)[j]->numSeq * 2; // include reverse
                    strain
                (*seqUnion)->bordersWithinSeq[j] = (Int64 *)/*e*/malloc(
                    sizeof(Int64) * (size_t)numStrain); /* fwd + rev strain
                    */
                for (k = 0; k < numStrain; k ++) {
                        (*seqUnion)->bordersWithinSeq[j][k] = lBorder + (*
                            sArrayAll)[j]->borders[k]; // len icludes border
                }
                lBorder = (*seqUnion)->seqBorders[j] + 1;
                (*seqUnion)->gc[j] = (double)(*sArrayAll)[j]->freqTab[0]['G
                    '] + (*sArrayAll)[j]->freqTab[0]['C'];
                numChar = (Int64)(*seqUnion)->gc[j] + (*sArrayAll)[j]->
                    freqTab[0]['A'] + (*sArrayAll)[j]->freqTab[0]['T'];
                (*seqUnion)->gc[j] /= numChar;
        }
        return cat;
}
```

4.2 Traverse Suffix Tree: `lcpSubjectTree.c`

```
/***** lcpSubjectTree.c
    *************************************************************
 * Description: Functions for lcp-interval tree processing.
 * Reference: Abouelhoda, M. I., Kurtz, S., and Ohlebusch, E. (2002).
 *   The enhanced suffix array and its applications to genome analysis.
 *   Proceedings of the Second Workshop on Algorithms in Bioinformatics,
 *   Springer-Verlag, Lecture Notes in Computer Science.
 * Author: Mirjana Domazet-Loso
 * File created on September 5th, 2008.
 ************************************************************************
    */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "commonSC.h"
#include "eprintf.h"
#include "sequenceData.h"
#include "sequenceUnion.h"
#include "intervalKr.h"
#include "intervalStack.h"
#include "shulen.h"

#include "interface.h"
#include "expectedShulen.h"
#include "lcpSubjectTree.h"

#if defined(UNIX)
#include <unistd.h>
#include <time.h>
#elif defined(WIN)
#include <time.h>
#endif
```

```
#if defined(_DEBUG) && defined(WIN)
#include "leakWatcher.h"
#endif

#if defined(_DEBUG) && defined(WIN)
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

#define MAXNUMLEAVES 6 /* size of the alphabet: A,C,T,G,Z,$ and possibly N */

/* global var - only for testing */
Int64 cntGetShulensForLeaves = 0;
Int64 cntGetShulensForChildrenNodes1 = 0;
Int64 cntGetShulensForChildrenNodes2a = 0;
Int64 cntGetShulensForChildrenNodes2b = 0;
Int64 cntGetShulensForChildrenNodes3 = 0;
Int64 cntGetShulensForChildrenNodes4 = 0;

Int64 cntSubjectLeavesInInterval = 0;
Int64 cntCheckIntervalBorder = 0;

/* allocate and calculate for every subject max shulen that can occur by
   chance alone */
Int64 *getMaxShulen(Args *args, Int64 numOfSubjects, Int64 *seqBorders,
    double *gc) {
  Int64 *ml, i, numSbjctNuc, lBorder;

  ml = (Int64 *)/*e*/malloc(numOfSubjects * sizeof(Int64));
  lBorder = 0;
  for (i = 0; i < numOfSubjects; i ++) {
    numSbjctNuc = seqBorders[i] - lBorder + 1 - 2; /* -2 to exclude borders
        */
#if DEBUG
    ml[i] = 0;
#else
    ml[i] = maxShulen(args->P, numSbjctNuc, gc[i]);
#endif

    lBorder = seqBorders[i] + 1;
  }
  return ml;
}

/* allocate and calculate for every subject max shulen
 * that can occur by chance alone using new formula that
 * includes gcS and gcQ
*/
#if defined(WIN)
static
#endif
```

```
Int64 **getMaxShulenNew(Args *args, Int64 numOfSubjects, Int64 *seqBorders,
    double *gc) {
  Int64 **mlNew, i, j, numSbjctNuc, lBorder;

  /* initialize mlNew */
  mlNew = (Int64 **)/*e*/malloc(numOfSubjects * sizeof(Int64 *));
  for (i = 0; i < numOfSubjects; i++) {
    mlNew[i] = (Int64 *)/*e*/malloc(numOfSubjects * sizeof(Int64));
    for (j = 0; j < numOfSubjects; j++) {
      mlNew[i][j] = 0;
    }
  }

  /* calculate new max shulens */
  lBorder = 0;
  for (i = 0; i < numOfSubjects; i ++) {
    numSbjctNuc = seqBorders[i] - lBorder + 1 - 2; /* -2 to exclude borders
        */
    //for (j = i + 1; j < numOfSubjects; j++) {
    for (j = 0; j < numOfSubjects; j++) {
      if (i != j) {
        // maxShulenNew(argsP, lS, gcQ, gcS); -- Si is subject, Sj is query
        mlNew[j][i] = maxShulenNew(args->P, numSbjctNuc, gc[j], gc[i]);
      }
    }
    lBorder = seqBorders[i] + 1;
  }
  return mlNew;
}


/* getLcpTreeShulens: compute shulens from lcp tree traversal.
 * If calculate Ae = 1, then calculate Ae, otherwise Ao
 * This is the only entry point to the functions in this file.
 */
long long **getLcpTreeShulens(Args *a, SequenceUnion *seqUnion, Int64 **
    effNuc, Int64 **nucAe, FILE *fpout) {

  Int64 *sa = NULL, *lcpTab = NULL;
  long long **sl = NULL;
  Int64 maxDepth = a->D;
#if DEBUG
  Int64 i;
#endif

  //time_t end, start, end2, end3;
  clock_t end, start;
  double elapsed_time1, elapsed_time2, elapsed_time3;

  start = clock();
  sa = getSuffixArray(seqUnion->seqUnion);
  end = clock();
  elapsed_time1 = (double)(end - start) / CLOCKS_PER_SEC;
```

```
    start = clock();
    lcpTab = getLcp(seqUnion->seqUnion, sa);
    end = clock();
    elapsed_time2 = (double)(end - start) / CLOCKS_PER_SEC;
    lcpTab[1] = -1;

#if DEBUG
    printf("\n---------- SA ----------\n");
    for (i = 1; i <= seqUnion->len; i++) {
      printf("sa[%lld]=%lld\n", (long long)(i - 1), (long long)sa[i]);
    }

    printf("\n---------- LCP ----------\n");
    for (i = 1; i <= seqUnion->len; i ++) {
      printf("lcp[%lld]=%lld\n", (long long)(i - 1), (long long)lcpTab[i]);
    }
#endif
    start = clock();
    sl = traverseLcpTree(lcpTab, sa, seqUnion->seqUnion, seqUnion->
        numOfSubjects, \
                         seqUnion->seqBorders, maxDepth, a, seqUnion->gc,
                             effNuc, nucAe);
    end = clock();
    elapsed_time3 = (double)(end - start) / CLOCKS_PER_SEC;

    if (a->t && fpout) { /* print run-time*/
      fprintf(fpout, "\nSA calculation: %.2f seconds.\n", elapsed_time1);
      fprintf(fpout, "\nLCP calculation: %.2f seconds.\n", elapsed_time2);
      fprintf(fpout, "\nLCP-tree traversal calculation: %.2f seconds.\n",
          elapsed_time3);
    }
    /*
      printf("\ncntGetShulensForLeaves = %lld\n", (long long)
          cntGetShulensForLeaves);
      printf("cntGetShulensForChildrenNodes1 = %lld\n", (long long)
          cntGetShulensForChildrenNodes1);
      printf("cntGetShulensForChildrenNodes2a = %lld\n", (long long)
          cntGetShulensForChildrenNodes2a);
      printf("cntGetShulensForChildrenNodes2b = %lld\n", (long long)
          cntGetShulensForChildrenNodes2b);
      printf("cntGetShulensForChildrenNodes3 = %lld\n", (long long)
          cntGetShulensForChildrenNodes3);
      printf("cntGetShulensForChildrenNodes4 = %lld\n", (long long)
          cntGetShulensForChildrenNodes4);
      printf("\ncntSubjectLeavesInInterval = %lld\n", (long long)
          cntSubjectLeavesInInterval);
      printf("cntCheckIntervalBorder = %lld\n", (long long)
          cntCheckIntervalBorder);
    */
    free(sa);
    free(lcpTab);
    return sl;

}
```

```
/* allocate space for aggregate sum of shulens of forward & reverse strain,
    dimensions: N x N */
long long **initializeShulens(Int64 numOfSubjects) {
  long long **shulens;
  Int64 i, j;
 /* Next programming iteration: change the type to int!! */
  shulens = (long long **)/*e*/malloc(numOfSubjects * sizeof(long long *));
  for (i = 0; i < numOfSubjects; i++) {
    shulens[i] = (long long *)/*e*/malloc(numOfSubjects * sizeof(long long)
        );
    for (j = 0; j < numOfSubjects; j++) { /* 2nd dimension: N-1 (to save
        the space)*/
      shulens[i][j] = 0;
    }
  }
  return shulens;
}

/* calculate list of left borders of all strains */
void calculateLBorders(Int64 *seqBorders, Int64 *lBorders, Int64
    numOfSubjects) {

  Int64 i;
  for (i = 0; i < numOfSubjects; i ++) {
    /* position of the left border of the Si sequence */
    lBorders[i] = (i > 0) ? (seqBorders[i - 1] + 1) : 0;
  }
}

/* traverseLcpTree: bottom-up traversal lcp-interval tree */
long long **traverseLcpTree(Int64 *lcpTab, Int64 *suffixArray, Sequence *
    seq, Int64 numOfSubjects, \
                            Int64 *seqBorders, Int64 maxDepth, Args *args,
                                \
                            double *gc, Int64 **effNuc, Int64 **nucAe){

  Interval *lastInterval, *interval;
  Int64 i, j;
  Int64 lb, rightEnd;
  Stack *treeStack;
  Stack *reserveStack;
  Int64 *lBorders;
  long long **shulens = NULL; /* 3D array: number of subjects x number of
      characters in the query seq */
  Int64 **leaves = NULL;
  int numLeaves = 0, maxChildren = MAXNUMLEAVES;
  int maxNumLeaves = MAXNUMLEAVES;
  Int64 **complementSI, **SI;
  Int64 *ml = NULL;
  Int64 **mlNew = NULL;
  Int64 lastIsNull;
  short *subjects = NULL;
```

```
    /* adjust SA, since data in sa and lcpTab start with position 1, and not
       0 */
    suffixArray++;
    lcpTab++;
    lcpTab[0] = 0; /* or -1 */

    /* allocate space for leaves of an interval; this array would be used for
        all intervals */
    leaves = initLeaves(maxNumLeaves);
    /* allocate space for every interval's child a complement list of subject
        indices (alphabet-size x (N+1)) */
    complementSI = initComplementSI(maxChildren, numOfSubjects);
    /* allocate space for every interval's child a list of subject indices (
       alphabet-size x N) */
    SI = initSI(maxChildren, numOfSubjects);

    subjects = getSubjects(seqBorders, numOfSubjects, 1);

    /* allocate and calculate for every subject max shulen that can occur by
       chance alone */
    ml = getMaxShulen(args, numOfSubjects, seqBorders, gc);
    mlNew = getMaxShulenNew(args, numOfSubjects, seqBorders, gc);

    /* allocate space for shulens of forward & reverse strain, dimensions: N
       x (N - 1) x length of i-th subject */
    shulens = initializeShulens(numOfSubjects);

    lBorders = /*e*/malloc(numOfSubjects * sizeof(Int64));
    calculateLBorders(seqBorders, lBorders, numOfSubjects);

    /* initialization of lcp-tree-traversal variables */
    rightEnd = seq->len - 1;
    treeStack = createStack();     /* true stack */
    interval = NULL;

    reserveStack = createStack(); /* helping stack; used for saving used
       allocated locations; since stack operations (pop/push) are faster than
        malloc */

    lastInterval = NULL;
    lastIsNull = 1;
    push(treeStack, getIntervalKr(0, 0, rightEnd, NULL, numOfSubjects, NULL,
       maxDepth, maxChildren)); /* push root node to the stack */

    for(i = 1; i < seq->len; i++){
      lb = i - 1;
      while(lcpTab[i] < ((Interval *)(treeStack->top))->lcp) { /* end of the
          previous interval, so the interval should be popped from the stack*/
        ((Interval *)(treeStack->top))->rb = i - 1;
        lastInterval = (Interval *)pop(treeStack);

        /* process: find list of subjects and list of immediate leaves that
           belong to the interval  */
```

```
265         //process(lastInterval, leaves, &numLeaves, shulens, seqBorders,
                numOfSubjects, args, suffixArray, lBorders, complementSI, SI, ml,
                effNuc, nucAe);
            process(lastInterval, leaves, &numLeaves, shulens, seqBorders,
                numOfSubjects, args, suffixArray, lBorders, complementSI, SI, ml,
                effNuc, nucAe, mlNew, subjects);

            /* save child intervals of popped interval */
            for(j = 0; j < lastInterval->numChildren; j ++) {
270           lastInterval->children[j]->numChildren = 0;
              lastInterval->children[j]->parent = NULL;
              push(reserveStack, lastInterval->children[j]);
            }

275         lb = lastInterval->lb;
            lastIsNull = 0;

            //(1) If lcptab[i] <= top 1.lcp, then top is the child interval of
                 top 1.
            //(2) If top 1.lcp < lcptab[i] < top.lcp, then top is the child
                interval of the lcptab[i]-interval that contains i
280         if (lcpTab[i] <= ((Interval *)treeStack->top)->lcp) { /* the new top
                is the parent of the ex top*/
              lastInterval->parent = (Interval *)treeStack->top;
              addChildKr(treeStack->top, lastInterval, maxChildren);
              lastIsNull = 1;
            }
285       } /* end while */

          /* when without reserveStack - begin */
          //if(lcpTab[i] > ((Interval *)treeStack->top)->lcp) { /* add interval
              to the stack */
          //  //interval = getIntervalKr(lcpTab[i], lb, rightEnd, NULL,
              numOfSubjects, treeStack->top, maxDepth, maxChildren);// treeStack->
              top or null
290       //  interval = getIntervalKr(lcpTab[i], lb, rightEnd, NULL,
              numOfSubjects, NULL, maxDepth, maxChildren);// treeStack->top or
              null
          //  if(!lastIsNull) {
          //          freeIntervalChildren(lastInterval);
          //          lastInterval->children = NULL;
          //          lastInterval->parent = interval;
295       //          addChildKr(interval, lastInterval, maxChildren);
          //          lastIsNull = 1;
          //    }
          //    push(treeStack,interval);
          //  }
300       /* when without reserveStack - end */


          if (lcpTab[i] > ((Interval *)treeStack->top)->lcp) { /* add interval to
               the stack */
            if (isEmpty(reserveStack)) {
```

```
305           interval = getIntervalKr(lcpTab[i], lb, rightEnd, NULL,
                  numOfSubjects, NULL, maxDepth, maxChildren);// treeStack->top or
                   null
          }
          else { /* use locations from the reserveStack */
            interval = pop(reserveStack); // pop the last child of the
                lastInterval from the reservestack
            interval->lcp = lcpTab[i];
310         interval->lb = lb;
            interval->rb = rightEnd;
            interval->numChildren = 0;
            interval->parent = NULL;
            interval->numSubjects = 0;
315       }
          if (!lastIsNull){
            lastInterval->parent = interval;
            addChildKr(interval, lastInterval, maxChildren);
            lastIsNull = 1;
320       }
          push(treeStack, interval);
        }
      }

325   #if DEBUG
      printf("Empting stack...\n");
      printf("Number of intervals allocated:%lld\n\n", (long long)numIntervalKr
          );
      #endif

330   while(!isEmpty(treeStack)){
        interval = pop(treeStack);
        //process(interval, leaves, &numLeaves, shulens, seqBorders,
            numOfSubjects, args, suffixArray, lBorders, complementSI, SI, ml,
            effNuc, nucAe);
        process(interval, leaves, &numLeaves, shulens, seqBorders,
            numOfSubjects, args, suffixArray, lBorders, complementSI, SI, ml,
            effNuc, nucAe, mlNew, subjects);

335     /* when the stack is not empty and the current interval has no parent,
            then his parent is on the top of the stack*/
        if (!interval->parent && !isEmpty(treeStack)) {
          interval->parent = (Interval *)(treeStack->top);
          addChildKr((Interval *)(treeStack->top), interval, maxChildren);
        }
340
        for (i = 0; i < interval->numChildren; i++) {
          freeIntervalKr((void *)interval->children[i]);
        }

345     //freeIntervalKr(interval); /* this cannot work - this interval will be
            allocated as the child of the next on the top of the stack */
      }
      freeIntervalKr((void *)interval); /* free the root interval */
      freeStack(treeStack, numOfSubjects, freeIntervalKr);
```

```
  /* free memory allocated for the reserveStack */
  j = 0; /* number of intervals on reserveStack */
  while (!isEmpty(reserveStack)) {
    interval = pop(reserveStack);
    ++ j;
    for (i = 0; i < interval->numChildren; i++) {
      freeIntervalKr((void *)interval->children[i]);
    }
    freeIntervalKr((void *)interval);
  }
  freeStack(reserveStack, numOfSubjects, freeIntervalKr);

#if DEBUG
  printf("reserveStack:%lld\n\n", (long long)j);
  printf("Finished.. Number of intervals allocated:%lld\n\n", (long long)
      numIntervalKr);
#endif
  free(lBorders);
  free(ml);
  // free mlNew
  for (i = 0; i < numOfSubjects; i++) {
    free(mlNew[i]);
  }
  free(mlNew);
  freeLeaves(leaves, maxNumLeaves);
  freeComplementSI(complementSI, maxChildren);
  freeSI(SI, maxChildren);
  free(subjects);
  return shulens;
}


/* calculate shulen for j-th subject and the given position pos;
 *  the position pos and right border rBorder are given in relative values,
      that is, as left border were 0!
 */
Int64 getShulen (Int64 intervalLcp, Int64 pos, Int64 rBorder) {

  Int64 len, slen;
  Int64 rBorderFwd = rBorder / 2;

  len = intervalLcp + 1;
  if (pos <= rBorderFwd) { // adjust for the end of the strain
    slen = ((pos + len - 1) <= rBorderFwd) ? len : rBorderFwd - pos + 1;
  }
  else { // adjust for the end of the sequence
    slen = ((pos + len - 1) <= rBorder) ? len : rBorder - pos + 1;
  }
  return slen;
}

/* determine which positions (from whole seq.union) belong to this interval
    and put them in the matrix interval->unresPosition
```

```
 * (a row i of the matrix represent positions of the subject Si that belong
     to this interval)
 * O(n) (|A| x n)
 * Function returns 1 if there are unres. positions for children, and 0
     otherwise (all positions resolved)
 */
int subjectLeavesInInterval(Interval *interval, Int64 *seqBorders, Int64
    numOfSubjects, Int64 *suffixArray, Int64 *lBorders, Int64 **leaves, int
    *numLeaves, short *subjects) {

  Int64 i, j, k;
  int numPos_j; /* number of positions left to be resolved for the j-th
      subject (>= 0) */
  int retValue = 0;
  *numLeaves = 0;

  ++cntSubjectLeavesInInterval;
  /* (1) when the interval is a leaf of the lcp tree */
  if (interval->numChildren == 0) { // O(|A|n)
    checkIntervalBorder(interval->lb, interval->rb, interval, seqBorders,
        numOfSubjects, suffixArray, lBorders, leaves, numLeaves, subjects);
    retValue = 0; /* leaf of the lcp tree has no unres. positions, since it
        doesn't have children */
  }
  /* (2) when the interval is an internal node of the lcp tree */
  else {
    for (k = 0; k < interval->numChildren; k++) {
      //if (interval->children[k]->numChildren < numOfSubjects) {
      if (interval->children[k]->numSubjects < numOfSubjects) {
        retValue = 1; /* unres. positions*/
        break;
      }
    }
    /* when there are unres. positions */
    if (retValue == 1) { //(k < interval->numChildren) {
      for (j = 0; j < numOfSubjects; j++) { // O(|A|n)
        /* when at least one of the children belongs to the subject Sj,
            then the interval itself belongs to the Sj subject */
        for (i = 0; i < interval->numChildren; i++) {
          numPos_j = interval->children[i]->subjectIndex[j];
          if (numPos_j >= 0) { /* when unresPos_j == -1, it means that Sj
              is not present at child's subject list */
            /* add the number of positions that haven't been resolved from
                a child for the j-th subject */
            if (interval->subjectIndex[j] == -1) { /* first occurence of
                the j-th subject */
              interval->subjectIndex[j] = numPos_j;
              ++ interval->numSubjects;
            }
            else {
              interval->subjectIndex[j] += numPos_j;
            }
          }
        } // end for i..
```

```
        }
      }
      else { /* there are no unres. positions, so the current interval covers
          all subjects */
        for (j = 0; j < numOfSubjects; j++) {
          interval->subjectIndex[j] = 0;
        }
        interval->numSubjects = (int)numOfSubjects;
      }
      /* add immediate leaves list to the currently processed internal node,
          if there are any */
      checkLeaves(interval, seqBorders, numOfSubjects, suffixArray, lBorders,
          leaves, numLeaves, subjects); //O(|A|n)
    } /* end else */
    return retValue;
}

/* for every immediate leaf (Si,pi) of the interval
 *     add shulens to aggregate sum shulens(Si, Sj) where Sj are all
     current's interval subjects
 */
//void getShulensForLeaves(Interval *interval, Int64 **leaves, int
    numLeaves, long long **shulens, Int64 *seqBorders, Int64 intervalLcp,
    Int64 numOfSubjects,
//
                                Int64 *lBorders, Int64 *ml, Int64 **effNuc,
    Int64 **nucAe) {
void getShulensForLeaves(Interval *interval, Int64 **leaves, int numLeaves,
     long long **shulens, Int64 *seqBorders, Int64 intervalLcp, Int64
    numOfSubjects,
                         Int64 *lBorders, Int64 *ml, Int64 **effNuc, Int64
                             **nucAe, Int64 **mlNew) {

  Int64 i, j, k, shulen;
  Int64 Si, pi;

  ++ cntGetShulensForLeaves;
  /*  for every leaf L(Si, pi) e L(z)
   *     for every subject Sj e SubjectList(z) \ Si
   *        sl[Si][Sj] := sl[Si][Sj] + lcp-value(z) + 1 //CORRECTION FOR
      BORDERS!!
   */
  for (i = 0; i < numLeaves; i++) { // leaves[i][0] --> subject, leaves[i
      ][1] --> position
    Si = leaves[i][0];
    pi = leaves[i][1];
    for (j = 0, k = 0; j < numOfSubjects && k < interval->numSubjects; j++)
        {
      // when Si <> Sj and Sj belongs to interval's subject list
      if (interval->subjectIndex[j] >= 0) {
        if (Si != j) {
          if (!nucAe) { /* when calculating Ao, matrix nucAe is null */
            shulen = getShulen(intervalLcp, pi, seqBorders[Si] - lBorders[
                Si]);
```

```
            //if (shulen > ml[j]) {
            if (shulen > mlNew[Si][j]) {
              shulens[Si][j] += shulen; // aggregate sum of shulens
              ++ effNuc[Si][j];
            }
            //if (interval->lcp + 1 > ml[j]) { // only those values that
                are longer then random ones add to the sum, but take care of
                 the borders!
            //  shulens[Si][j] += getShulen(intervalLcp, pi, seqBorders[Si]
                - lBorders[Si]);// aggregate sum of shulens
            //  ++ effNuc[Si][j];
            //}
          }
          else if (nucAe[Si][j] < effNuc[Si][j]) { /* when calculating Ae,
              check whether enough shulens have already been added */
            shulens[Si][j] += getShulen(intervalLcp, pi, seqBorders[Si] -
              lBorders[Si]);// aggregate sum of shulens
            ++ nucAe[Si][j];
          }
        }
        ++ k;
      } // end if (interval->subjectIndex[j] >= 0) {
    }
  }
}

/* calculate SI (subjectIndex list) and complementSI for every child
    interval */
void getSiComplementSi(Interval *interval, Int64 numOfSubjects,  Int64 **
    complementSI, Int64 **SI) {

  Int64 i, j, k, l, m;
  Interval *child;

  // construction of complement of subjectIndex for every child interval (
      each interval has at most |A| children)
  for (i = 0; i < interval->numChildren; i++) {
    child = interval->children[i];
    l = 0; /* cnt of complement subjects for the i-th child */
    m = 0; /* cnt of subjects for the i-th child */
    for (j = 0, k = 0; k < interval->numSubjects && j < numOfSubjects; j++)
        { // k - cnt of subjects for a parent interval
      if (interval->subjectIndex[j] >= 0) {
        ++ k;
        if (child->subjectIndex[j] == -1) { // complement: Sj is not in the
            child's list, but it is in the parent's list
          complementSI[i][l ++] = j;
        }
        else { // Sj is both in the child's and parent's list
          SI[i][m ++] = j;
        }
      }
    }
    complementSI[i][l] = -1; // end of the complement list
```

```
  }
}

/* Function returns sl (shulen) as it is if the prefix of the current
   interval doesn't include border or,
 * if it does, then returns the correction of the sl value
 * currently: O(n), to do --> change to O(log n)
 */
Int64 prefixIncludesBorder(Interval *child, Interval *interval, Int64 *
   suffixArray, Int64 *seqBorders, Int64 *lBorders, Int64 numOfSubjects) {

  Int64 sl = interval->lcp + 1, i, Sj = -1;
  /* takes first (but it could be any other position) of the interval to
     check whether the string S[pos.. pos+lcp]
   * contains border character('Z'); if it does, correct the lcp value of
     the interval */

  Int64 beginPrefix = suffixArray[child->lb];
  Int64 endPrefix = beginPrefix + sl - 1;
  Int64 endFwdStrain;

  // determine Sj for beginPrefix
  for (i = 0; i < numOfSubjects; i++) {
    if (child->subjectIndex[i] >= 0 && (beginPrefix >= lBorders[i] &&
       beginPrefix <= seqBorders[i])) {
      Sj = i;
      break;
    }
  }
  if (Sj == -1) {
    fprintf(stderr, "Error[kr 2]: prefixIncludesBorder - cannot determine
       the subject sequence for the child interval!\n");
  }

  endFwdStrain = lBorders[Sj] + (seqBorders[Sj] - lBorders[Sj]) / 2;
  if (beginPrefix <= endFwdStrain && endPrefix > endFwdStrain) { // border
     of the strain --> correct lcp!
    sl = endFwdStrain - beginPrefix + 1;
  }
  else if (beginPrefix > endFwdStrain && endPrefix > seqBorders[Sj]) { //
     border of the sequence --> correct lcp!
    sl = seqBorders[Sj] - beginPrefix + 1;
  }
  return sl;
}

/* for every child interval Ci that has unresolved positions, find the Ci's
     complement subject list (in current interval)
 *  and solve those positions.
 * This function (O (|A|n^2)) is only called in, at most |A|n cases, when
     the interval has less than maxNumSubjects
 */
//void getShulensForChildrenNodes(Interval *interval, Int64 numOfSubjects,
   Int64 **complementSI, Int64 **SI, long long **shulens,
```

```
//
                                                    Int64* ml, Int64
    **effNuc, Int64 **nucAe, Int64 *suffixArray, Int64 *seqBorders, Int64 *
    lBorders) {
void getShulensForChildrenNodes(Interval *interval, Int64 numOfSubjects,
    Int64 **complementSI, Int64 **SI, long long **shulens,
                                Int64* ml, Int64 **effNuc, Int64 **nucAe,
                                   Int64 *suffixArray, Int64 *seqBorders,
                                   Int64 *lBorders, Int64 **mlNew) {

  Interval *child;
  Int64 i, j, k, p, sl;
  Int64 Sj, Sk;

  int temp= 0;
  ++ cntGetShulensForChildrenNodes1;

  // construction of complement of subjectIndex for every child interval (
      each interval has at most |A| children)
  getSiComplementSi(interval, numOfSubjects,  complementSI, SI);

  /*  for every child-interval Ci of the node z
   *     for every (Sj, pj) e SubjectList(Ci)
   *        for every (Sk, pk) e SubjectList(z) \ SubjectList(Ci)
   *            sl[Sj][Sk] := sl[Sj][Sk] + pj * (lcp-value(z) + 1)
   */
  for (i = 0; i < interval->numChildren; i++) {
    child = interval->children[i];
    ++ cntGetShulensForChildrenNodes2a;
    /* shulen correction for the border (if the prefix includes border) */
    sl = prefixIncludesBorder(child, interval, suffixArray, seqBorders,
        lBorders, numOfSubjects);

    for (j = 0; j < child->numSubjects; j++) { // for every (Sj, pj) e
        SubjectList(Ci), pj-number of unres. positions for Sj
      Sj = SI[i][j];
      ++ cntGetShulensForChildrenNodes2b;

      temp = 0;
      for (k = 0; complementSI[i][k] != -1; k++) { // for every (Sk, pk) e
          SubjectList(z) \ SubjectList(Ci)

        ++ cntGetShulensForChildrenNodes3;
        if (temp == 0) {
          ++ cntGetShulensForChildrenNodes4; /* how many times function
              ends in this loop per function call */
          temp = 1;
        }

        Sk = complementSI[i][k];
        /* when calculating Ao */
        if (!nucAe) { /* when calculating Ao, matrix Ae is null */
          //if (sl > ml[Sk]) { // only those values that are longer then
              random ones
```

```
            if (sl > mlNew[Sj][Sk]) { // only those values that are longer
                then random ones
              shulens[Sj][Sk] += child->subjectIndex[Sj] * sl; // aggregate
                  sum of shulens
              effNuc[Sj][Sk] += child->subjectIndex[Sj];
            }
          }
          /* when calculating Ae */
          else if (nucAe[Sj][Sk] + child->subjectIndex[Sj] < effNuc[Sj][Sk])
              { /* when calculating Ae, check whether enough shulens have
              already been added */
            /* correction for borders -- shulen should not go across strain
                border !!!! precalculated */
            shulens[Sj][Sk] += child->subjectIndex[Sj] * sl; // aggregate sum
                of shulens
            nucAe[Sj][Sk] += child->subjectIndex[Sj];
          }
          else if (nucAe[Sj][Sk] < effNuc[Sj][Sk]) { /* when calculating Ae,
              do not add all the positions*/
            p = effNuc[Sj][Sk] - nucAe[Sj][Sk]; // number of positions to be
                added
            /* correction for borders -- shulen should not go across strain
                border !!!! precalculated */
            shulens[Sj][Sk] += p * sl; // aggregate sum of shulens
            nucAe[Sj][Sk] += p;
          }
        }
      }
    }
  }

  /* add a subject and its position (calculated from the S.A.) to an interval
      */
  void addSubjectLeaf(Int64 subject_j, Int64 pos, Int64 **leaves, int *
      numLeaves) {

    /* add a new position in the list and increase the number of leaves */
    if (*numLeaves >= MAXNUMLEAVES) {
      eprintf("[ERROR: kr 2] Exceeded the number of immediate leaves for an
          interval - increase the maximum !\n");
    }
    leaves[*numLeaves][0] = subject_j;
    leaves[*numLeaves][1] = pos;
    ++ (*numLeaves);
  }

  /* check borders of an interval */
  void checkIntervalBorder(Int64 left, Int64 right, Interval *interval, Int64
      *seqBorders, Int64 numOfSubjects, Int64 *suffixArray, Int64 *lBorders,
      Int64 **leaves, int *numLeaves, short *subjects) {

    Int64 i, j;

    for (i = left; i <= right; i++) {
```

```
    ++cntCheckIntervalBorder;

    //  for (j = 0; j < numOfSubjects; j++) {
    //          if (suffixArray[i] >= lBorders[j] && suffixArray[i] <=
        seqBorders[j]) {
    j = subjects[suffixArray[i]];
    if (interval->subjectIndex[j] == -1) { /* first occurence of the j-th
        subject for the interval */
      interval->subjectIndex[j] = 0; /* leaves' positions that are
          unresolved are added later */
      ++ interval->numSubjects;
    }
    /* add a subject and its position in the sequence (calculated from the
        S.A.) to a node */
    addSubjectLeaf(j, suffixArray[i] - lBorders[j], leaves, numLeaves);
  }
}

/* note the suffix tree leaves attached to interval */
void checkLeaves(Interval *interval, Int64 *seqBorders, Int64 numOfSubjects
    , Int64 *suffixArray, Int64 *lBorders, Int64 **leaves, int *numLeaves,
     short *subjects) {

  Int64 i;

  /* check children[0] - this is special case because children[0] doesn't
      have a predecessor (like other children) */
  /* if the left border of the parent interval is smaller than the border
      of the left most child */
  if (interval->lb < interval->children[0]->lb) {
    checkIntervalBorder(interval->lb, interval->children[0]->lb - 1,
        interval, seqBorders, numOfSubjects, suffixArray, lBorders, leaves,
        numLeaves, subjects);
  }

  /* intervals covered by interval children (from children[1] to last child
      )*/
  for (i = 1; i < interval->numChildren; i++) {
    if (interval->children[i - 1]->rb + 1 < interval->children[i]->lb){
      /*if there is a gap between intervals covered by children[i-1] and
          children[i] */
      checkIntervalBorder(interval->children[i-1]->rb + 1, interval->
          children[i]->lb - 1, interval, seqBorders, numOfSubjects,
          suffixArray, lBorders, leaves, numLeaves, subjects);
    }
  }

  /* right border of the parent interval */
  if(interval->children[interval->numChildren - 1]->rb < interval->rb){
    checkIntervalBorder(interval->children[interval->numChildren - 1]->rb +
         1, interval->rb, interval, seqBorders, numOfSubjects, suffixArray,
        lBorders, leaves, numLeaves, subjects);
  }
}
```

```
690
   /* process a current node n(z) / interval */
   //void process(Interval *interval, Node *node, Int64 ***shulens, Int64 *
       seqBorders, Int64 numOfSubjects, Args *args, Int64 *suffixArray, Int64 *
       lBorders) {
   void process(Interval *interval, Int64 **leaves, int *numLeaves, long long
       **shulens, Int64 *seqBorders, Int64 numOfSubjects, Args *args
                , Int64 *suffixArray, Int64 *lBorders, Int64 **complementSI,
                    Int64 **SI
695              , Int64 *ml, Int64 **effNuc, Int64 **nucAe, Int64 **mlNew,
                    short *subjects) {

     Int64 i, Si;
     int unresolvedPos = 0; // if 0, then there are still some unres.
         positions from chldrens' intervals

700  #if DEBUG
     printf("\nproces..");
   #endif
     /* initially: no subjects belonging to this interval */
     for (i = 0; i < numOfSubjects; i++) {
705    interval->subjectIndex[i] = -1;
     }
     interval->numSubjects = 0;

     /* initially: no leaves belong to this node */
710  *numLeaves = 0;

     /* determine:
      * (1) which subjects belong directly or indirectly to this interval and
          put them in list interval->subjectIndex
      * (2) leaves of the interval
715   * (3) number of unresolved positions coming from interval's children
      * Function returns 1 if there are unres. positions for children, and 0
          otherwise (all positions resolved)
      */
     unresolvedPos = subjectLeavesInInterval(interval, seqBorders,
         numOfSubjects, suffixArray, lBorders, leaves, numLeaves, subjects);

720  /*  for every child Ci of the interval z
      *     for every (Sj, pj) e SubjectList(Ci)
      *        for every (Sk, pk) e SubjectList(z) \ SubjectList(Ci)
      *        sl[j][k] := sl[k][j] := lcp-value(z) + 1
      */
725  /* correction of the shulen/lcp value only for the leaves' positions and
         not children's positions */
     if (unresolvedPos) {
       //getShulensForChildrenNodes(interval, numOfSubjects, complementSI, SI,
           shulens, ml, effNuc, nucAe, suffixArray, seqBorders, lBorders);
       getShulensForChildrenNodes(interval, numOfSubjects, complementSI, SI,
           shulens, ml, effNuc, nucAe, suffixArray, seqBorders, lBorders, mlNew
           );
     }
```

```
730
    /*  for every leaf L(Si, pi) e L(z)
     *      for every subject Sp e SubjectList(z) \ Si
     *          sl[i][p] := lcp-value(z) + 1 //CORRECTION FOR BORDERS!!
     */
    if (*numLeaves) {
      //getShulensForLeaves(interval, leaves, *numLeaves, shulens, seqBorders
          , interval->lcp, numOfSubjects, lBorders, ml, effNuc, nucAe);
      getShulensForLeaves(interval, leaves, *numLeaves, shulens, seqBorders,
          interval->lcp, numOfSubjects, lBorders, ml, effNuc, nucAe, mlNew);
    }

    if (unresolvedPos > 0 || interval->numChildren == 0) {
      /* when the first time all subjects are encountered in the subtree of
          interval */
      if (interval->numSubjects == numOfSubjects) {
        for (i = 0; i < numOfSubjects; i++) {
          interval->subjectIndex[i] = 0;
        }
      }
      else {
        /* add leaves to the sum of unresolved positions for evey subject
            index */
        for (i = 0; i < *numLeaves; i++) {
          Si = leaves[i][0];
          if (interval->subjectIndex[Si] == -1) {
            interval->subjectIndex[Si] = 1;
          }
          else {
            ++ interval->subjectIndex[Si];
          }
        }
      }
    }

  #if DEBUG
    printf("%lld-[%lld..%lld]\n",(long long)interval->lcp,(long long)interval
        ->lb,(long long)interval->rb);
    if (interval->parent) {
      printf("\tparent:%lld-[%lld..%lld]\n",(long long)interval->parent->lcp
          ,(long long)interval->parent->lb,(long long)interval->parent->rb);
    }
    printf("\tsubjects: ");
    if (!interval->subjectIndex) { /* all subjects present --> all positions
        resolved*/
      for (i = 0; i < numOfSubjects; i++) printf("\t%d", 0);
    }
    else {
      for (i = 0; i < numOfSubjects; i++) printf("\t%d",interval->
          subjectIndex[i]);
    }
    printf("\n");
  #endif
  }
```

```
/* allocate and initialize leaves - the same array would be used for all
   nodes */
Int64 **initLeaves(int maxNumLeaves) {

  Int64 **leaves; /* leaves - list of subjects and their positions, e.g.
     leaves[0][0]=0, leaves[0][1]=2 (subject = 0, position = 2) */
  int i;
  leaves = (Int64 **)emalloc((size_t)maxNumLeaves * sizeof(Int64 *));
  for (i = 0; i < maxNumLeaves; i++) {
    leaves[i] = (Int64 *)emalloc(2 * sizeof(Int64));
  }
  return leaves;
}

/* deallocate leaves */
void freeLeaves(Int64 **leaves, int maxNumLeaves) {
  int i;
  for (i = 0; i < maxNumLeaves; i++) {
    free(leaves[i]);
  }
  free(leaves);
}


/* allocate and initialize complementSI */
Int64 **initComplementSI(int maxChildren, Int64 numOfSubjects) {

  Int64 **complementSI;
  int i;

  complementSI = (Int64 **)emalloc((size_t)maxChildren * sizeof(Int64 *));
  for (i = 0; i < maxChildren; i++) {
    complementSI[i] = (Int64 *)emalloc((numOfSubjects + 1) * sizeof(Int64))
        ; // + 1 for the end of the list, when the list has exactly
        numOfSubjects subjects
  }
  return complementSI;
}

/* deallocate list of complement subject indices */
void freeComplementSI(Int64 **complementSI, int maxChildren) {
  int i;
  for (i = 0; i < maxChildren; i++) {
    free(complementSI[i]);
  }
  free(complementSI);
}

/* allocate and initialize SI list for every child*/
Int64 **initSI(int maxChildren, Int64 numOfSubjects) {

  Int64 **SI;
  int i;
```

```
  SI = (Int64 **)emalloc((size_t)maxChildren * sizeof(Int64 *));
  for (i = 0; i < maxChildren; i++) {
    SI[i] = (Int64 *)emalloc(numOfSubjects * sizeof(Int64)); // + 1 for the
        end of the list, when the list has exactly numOfSubjects subjects
  }
  return SI;
}

/* deallocate list of subject indices for every interval */
void freeSI(Int64 **SI, int maxChildren) {
  int i;
  for (i = 0; i < maxChildren; i++) {
    free(SI[i]);
  }
  free(SI);
}


/* When the processing of the interval is over, set its children to NULL (
    since they're already processed);
   There is no actual memory deallocation, since the memory wasn't
       allocated to children.
   Children are just pointers to intervals and not intervals themselves! */
/* used without reserveStack !! */
//void freeIntervalChildren(Interval *interval) {
//
//      int i, numOfChildren;
//      numOfChildren = interval->numChildren;
//
//      if (numOfChildren < interval->maxNumChildren) { /* children
    intervals that are allocated, but not used, should also be deallocated
    */
//              numOfChildren = interval->maxNumChildren;
//      }
//  for(i = 0; i < numOfChildren; i++) {
//
//              if (interval->children[i] && interval->children[i]->
    numChildren > 0) {
//                      eprintf("freeInterval: the interval %ulld has %d
    allocated children", (unsigned long long)interval->children[i], interval
    ->children[i]->numChildren);
//              }
//              freeInterval(interval->children[i]);
//              //freeInterval(interval->children[i]);
//              interval->children[i] = NULL;
//  }
//      interval->numChildren = 0;
//      interval->maxNumChildren = 0;
//}

/* find a subject for each position in the suffix array */
short *getSubjects(Int64 *seqBorders, Int64 numOfSubjects, int step) {
```

```
    Int64 i, j, lb;
    short *subjects = NULL;

    subjects = (short *)/*e*/malloc(sizeof(short) * (seqBorders[numOfSubjects
        - 1] + 1));
    //subjects = (short *)/*e*/malloc(sizeof(short) * (seqBorders[
        numOfSubjects - 1] / step + 1));

    // subject - positive values
    lb = 0;
    for (i = 0; i < numOfSubjects; i ++) {
      //for (; j <= seqBorders[i] / step; j ++) {
      for (j = lb; j <= seqBorders[i]; j ++) {
        subjects[j] = (short)i;
      }
      lb = seqBorders[i] + 1;
    }
#if DEBUG
    printf("Subjects:\n");
    for (j = 0; j <= seqBorders[numOfSubjects - 1]; j ++) {
      printf("[%lld] = %hd\n", j, subjects[j]);
    }
#endif
    return subjects;
}
```

4.3 Estimate Divergence: `divergence.c`

```
/***** divergence.c
    ************************************************************
 * Description: divergence calculation
 * The divergence computation is based on the mathematical model described
    in:
 * Haubold, B., Pfaffelhuber, P., Domazet-Loso, M., and Wiehe, T. 2009.
 * Estimating mutation distances from unaligned genomes, J. Comput. Biol.
 *
 * Author: Bernhard Haubold, haubold@evolbio.mpg.de
 * Modified by: Mirjana Domazet-Loso, 02/12/2008
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301,
    USA.
 ************************************************************************
    */
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>

#include "commonSC.h"
#include "eprintf.h"
#include "interface.h"
#include "divergence.h"

#if WIN
#include "gsl/gsl_sf_gamma.h"
#include "gsl/gsl_nan.h"
#else
#include <gsl/gsl_sf_gamma.h>
#include "gsl/gsl_nan.h"
#endif

double **pmaxMat=NULL;
int *maxX;
double globalP;

/* non-static function - calculate shulens */
double divergence (Args *args, double shulen, Int64 seqLen, double gc,
    double **lnChoose, double gcS, int n_choose, int si) {
  double p, q;
  double du, dl, dm, t, d, pS;
  double s1[N_S + 1] = { 0 };
  double errd = args->E; /* relative error for shulen length */

  p = gc / 2.0;
  q = (1.0 - gc) / 2.0;
  pS = gcS / 2.0;
  du = 0;
  dl = 1.0 - (2 * p * p + 2 * q * q); // dl < 0.75
  t = THRESHOLD;

  while ((dl - du) / 2.0 > t) {
    dm = (du + dl) / 2.0;
    if (shulen < expShulen(args, dm, p, seqLen, lnChoose, pS, n_choose, s1,
        si)) {
      du = dm;
    }
    else {
      dl = dm;
    }
    /* test the relative error between du and dl; if it is smaller than
       some threshold, then break !!*/
    if (fabs(dl - du) / dl <= errd) {
      break;
    }
  }
  d = (du + dl) / 2.0;
  return d;
```

```
}

#if defined(WIN)
static
#endif
double expShulen (Args *args, double d, double p, Int64 l, double **
    lnChoose, double pS, int n_choose, double *s1, int si) {
  Int64 i;
  int thresholdReached = 0;

  double e = 0.0; /* expectation */
  double prob_i, delta;
  double factor;
  double probOld = 0;

  double t = 1.0 - d; // d = d' / l
  double p_t = t; /* pow(t, 1)*/

  //absolute error: double t1 = 1e-5; /* 1e-4 ok */
  //double t1 = 1e-5;
  double t1 = args->T;

  for (i = 1; i < l; i++) { //since for i = 0, the whole expression is 0
    factor = 1.0 - p_t; //factor = 1.0 - pow(1.0 - d, i);
    if (!thresholdReached) {
      prob_i = factor * pmax(args, i, p, l, &thresholdReached, lnChoose, pS
          , n_choose, s1, si);
    }
    else {
      prob_i = factor; /* prob_i = factor * s, where s = 1 */
    }
    delta = (prob_i - probOld) * i; /* delta should always be positive */
    e += delta; /* expectation of avg shulen l(Q, S)*/
                /* relative error - a little bit faster than the
                   calculation with the absolute error */
    if (e >= 1 && delta / e <= t1) {
      break;
    }
    p_t *= t;
    probOld = prob_i;
  }
  return e;
}

#if defined(WIN)
static
#endif
double pmax(Args *args, Int64 x, double p, Int64 l, int *thresholdReached,
    double **lnChoose, double pS, int n_choose, double *s1, int si) {

  Int64 k, i;
  double s = 0, x_choose_k;
  double t, t1, m, delta;
  /* m_t value should be explored by simulation */
```

```
    const double m_t = args->M; //pow(10, DBL_MIN_10_EXP); /* 10^(-307) */
126
    if (x > N_S) {
      eprintf("Error: x = %lld. The maximum number of elements in the array
          s1 should be increased!\n", (long long) x);
    }
    if (s1[x] != 0) {
131   return s1[x];
    }

    if(si != -1){
      if(x>maxX[si]){
136     pmaxMat[si]=(double *)erealloc(pmaxMat[si],(x+1)*sizeof(double));
        for(i=maxX[si]+1;i<=x;i++)
          pmaxMat[si][i] = -1;
        maxX[si] = x;
      }
141   if(pmaxMat[si][x] != -1){
        if(pmaxMat[si][x] >= 1.0){
          *thresholdReached = 1;
          return pmaxMat[si][x];
        }
146   }
      p = globalP;
    }

    s = 0;
151 for (k = 0; k <= x; k++) {
      if (x < n_choose) {
        if (lnChoose[x][k] == 0) {
          lnChoose[x][k] = gsl_sf_lnchoose(x, k);
        }
156     x_choose_k = lnChoose[x][k];
      }
      else {
        x_choose_k = gsl_sf_lnchoose(x, k);
      }
161   m = (pow(2.0, (double)x) * pow(p, (double)k) * pow(0.5 - p, (double)x -
          k)
           * pow(1.0 - pow(pS, (double)k) * pow(0.5 - pS, (double)x - k), (
               double)1));
      if (m == 0) {
        delta = 0;
      }
166   else if (m >= m_t) {
        t = log(m);
        if (t == GSL_NEGINF) {
          delta = 0;
        }
171     else {
          delta = exp (t + x_choose_k);
        }
      }
      else {
```

```
176         t1 = log(1 + m); // for small values of m - to avoid overflow (-INF)
            delta = exp (t1 + x_choose_k) - exp(x_choose_k);
          }
          s += delta;
          if (s >= 1.0) {
181         s = 1;
            *thresholdReached = 1;
            break;
          }
        } /* end for */
186     if(si != -1)
          pmaxMat[si][x] = s;
        s1[x] = s;
        return s;
      }
191
      void initDivergence(int numSeq, double gc){
        int i;

        globalP = gc/2.0;
196     pmaxMat = (double **)emalloc(numSeq*sizeof(double *));
        maxX = (int *)emalloc(numSeq*sizeof(int));
        for(i=0;i<numSeq;i++){
          maxX[i] = -1;
          pmaxMat[i] = (double *)emalloc((maxX[i]+1)*sizeof(double));
201     }
      }
```

5 Change Log

- Version 2.0.0 (June 27, 2009)
  - First running version
- Version 2.0.1 (July 28, 2009)
  - Implemented shortcut for K_r computation (`-g`)
- Version 2.0.2 (October 5, 2009)
  - Added more detailed error messages

References

[1] M. Domazet-Lošo and B. Haubold. Efficient estimation of pairwise distances between genomes. *Bioinformatics*, submitted, 2009.

[2] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. Gnu scientific library reference manual. Network Theory Ltd 1.6, for gsl version 1.6, 17 March 2005 edition, 2005.

[3] B. Haubold, P. Pfaffelhuber, M. Domazet-Lošo, and T. Wiehe. Estimating mutation distances from unaligned genomes. *Journal of Computational Biology*, in press, 2009.