The Source Code Documentation of Maxmat3

Stefan Kurtz[1]

April 3, 2003

This is the documentation of the Program Maxmat3. This program is based on a suffix tree library and a collection of general library functions. As a consequence, the source code is distributed over three directories. This is reflected in this documentation. The first section describes some general types and library functions as declared in the directory `libbasedir`. The include file `protodef.h` is of particular importance here, since it contains the prototypes for all non-static functions of `libbasedir`. Since `protodef.h` is generated from the C-source files, we do not provide any documentation for it. Note that many files contain a superset of the definitions and functions actually required for `maxmat3`. This is because the files have been taken verbatim from a larger collection of library functions. The second section describes the suffix tree library providing a convenient collection of types and functions to handle (i.e. construct and access) suffix trees. There are several functions of the library not required for `maxmat3`. They were implemented for other applications not discussed here. The third section of this document gives the implementation of `maxmat3` based on the functions described in the previous sections. In the appendix section A we define some necessary notation to understand the concept of suffix trees and their implementaion. Finally, in the appendix section B we provide an index of types and function names. The PDF-version of this document comes with hyperlinks, linking type names, function names, and macro names to their definition. If you encounter broken or missing links, please contact the author at `kurtz@zbh.uni-hamburg.de`. Other comments are also welcome.

---

[1] Zentrum für Bioinformatik, Universität Hamburg, Bundesstrasse 43, 20146 Hamburg, Germany, E-mail: kurtz@zbh.uni-hamburg.de

Contents

1 The Basic Library in libbasedir

1.1 Basic Header Files

1.1.1 The Header File `types.h`

This file contains some basic type definition.

```
typedef unsigned char  Uchar;
typedef unsigned short Ushort;
```

The following is the central case distinction to accomodate code for 32 bit integers and 64 bit integers.

```
#ifdef SIXTYFOURBITS


typedef unsigned long  Uint;
typedef signed   long  Sint;
#define LOGWORDSIZE    6              // base 2 logarithm of wordsize
#define UintConst(N)   (N##UL)        // unsigned integer constant


#else


typedef unsigned int  Uint;
typedef signed   int  Sint;
#define LOGWORDSIZE    5             // base 2 logarithm of wordsize
#define UintConst(N)   (N##U)        // unsigned integer constant


#endif
```

Type of unsigned integer in `printf`.

```
typedef unsigned long Showuint;
```

Type of signed integer in `printf`.

```
typedef signed long Showsint;
```

Type of integer in `scanf`.

```
typedef signed long Scaninteger;
```

Argument of a function from `ctype.h`.

```
typedef int Ctypeargumenttype;
```

Return type of `fgetc` and `getc`.

```
typedef int Fgetcreturntype;
```

Type of first argument of `fputc`.

```
typedef int Fputcfirstargtype;
```

Return type of `strcmp`.

```
typedef int Strcmpreturntype;
```

Type of a file descriptor.

```
typedef int Filedesctype;
```

Return type of `qsort` function.

```
typedef int Qsortcomparereturntype;
```

Return type of `sprintf` function.

```
typedef int Sprintfreturntype;
```

Type of fieldwidth in `printf` format string.

```
typedef int Fieldwidthtype;
```

Type of `argc`-parameter in main.

```
typedef int Argctype;
```

Return type of `getrlimit`

```
typedef int Getrlimitreturntype;
```

The following macros define some basic division, multiplication, and modulo operations on unsigned integers.

```
#define DIV2(N)      ((N) >> 1)
#define DIV4(N)      ((N) >> 2)
#define DIV8(N)      ((N) >> 3)
#define MULT2(N)     ((N) << 1)
#define MULT4(N)     ((N) << 2)
#define MULT8(N)     ((N) << 3)
#define MOD2(N)      ((N) & 1)
#define MOD4(N)      ((N) & 3)
#define MOD8(N)      ((N) & 7)
```

Here is a prototype for the main function.

```
#define MAINFUNCTION int main(Argctype argc,char *argv[])
```

A type for boolean values defined as a constant to allow checking if it has been defined previously.

```
#ifndef BOOL
#define BOOL unsigned char
#endif
```

```
#ifndef False
#define False ((BOOL) 0)
#endif
```

```
#ifndef True
#define True ((BOOL) 1)
#endif
```

Show a boolean value as a string or as a character 0 or 1.

```
#define SHOWBOOL(B) ((B) ? "True" : "False")
#define SHOWBIT(B)  ((B) ? '1' : '0')
```

Pairs, triples, and quadruples of unsigned integers.

```
typedef struct
{
  Uint uint0, uint1;
} PairUint;
```

```
typedef struct
{
  Uint uint0, uint1, uint2;
} ThreeUint;
```

```
typedef struct
{
  Uint uint0, uint1, uint2, uint3;
} FourUint;
```

The following type stores filenames and the length of the corresponding files.

```
typedef struct
{
  char *filenamebuf;     // pointer to a copy of a filename
  Uint filelength;       // the length of the corresponding file
} Fileinfo;
```

The following is the type of the comparison function to be provided to the function `qsort`.

```
typedef int (*Qsortcomparefunction)(const void *,const void *);
```

1.1.2 The Header File `chardef.h`

This file defines some character values used when storing multiple sequences.

```
#define SEPARATOR       UCHAR_MAX          // separator symbol in multiple seq
#define WILDCARD        (SEPARATOR-1)      // wildcard symbol in multiple seq
#define UNDEFCHAR       (SEPARATOR-2)      // undefined character in multiple seq
#define ISSPECIAL(C)    ((C) >= WILDCARD)  // either WILDCARD or SEPARATOR
#define ISNOTSPECIAL(C) ((C) < WILDCARD)   // neither WILDCARD nor SEPARATOR
```

1.1.3 The Header File `visible.h`

This header file defines some constants and macros to test for visibility of a character (ASCII code is between 33 and 126) and to show this character.

The smallest visible character is the blank with code 33.

```
#define LOWESTVISIBLE        33
```

The largest visible character is the tilde with code 126.

```
#define HIGHESTVISIBLE      126
```

Check if character is invisible according to the definition from above.

```
#define INVISIBLE(C)        ((C) < LOWESTVISIBLE || (C) > HIGHESTVISIBLE)
```

Rescale characters denoted by numbers starting at 0 to the visible ASCII characters.

```
#define VISIBLECHAR(I)      ((char)((I)+LOWESTVISIBLE))
```

Reverse the previous operation.

```
#define INVISIBLECHAR(C)    ((Sint)((C)-LOWESTVISIBLE))
```

The following macro prints a character to a file pointer. If the character is not visible, then it is shown as the corresponding ASCII-number with a prepended backslash.

```
#define SHOWCHARFP(FP,C)\
        if(INVISIBLE(C))\
        {\
          fprintf(FP,"\\%lu",(Showuint) (C));\
        } else\
        {\
          fputc(C,FP);\
        }
```

The following macro is a variation of the previous macro, always showing the output to standard out.

```
#define SHOWCHAR(C) SHOWCHARFP(stdout,C)
```

1.1.4 The Header File `minmax.h`

This file defines macros for maximum and minimum computation, if they are not already defined.

```
#ifndef MAX
#define MAX(X,Y) (((X) > (Y)) ? (X) : (Y))
#endif

#ifndef MIN
#define MIN(X,Y) (((X) < (Y)) ? (X) : (Y))
#endif
```

1.1.5 The Header File `megabytes.h`

The following macro transforms bytes into megabytes.

```
#define MEGABYTES(V)  ((double) (V)/((UintConst(1) << 20) - 1))
```

1.1.6 The Header File `fopen.h`

This file defines macros for opening and writing files via file pointers.

```
#define FILEOPEN(FP,FILENAME,MODE)\
        if ((FP = fopen(FILENAME,MODE)) == NULL)\
        {\
          fprintf(stderr,"(%s,%lu): Cannot open file \"%s\"\n",\
                  __FILE__,(Showuint) __LINE__,FILENAME);\
          exit(EXIT_FAILURE);\
        }

#define FPBINWRITE(FP,BUF,SIZE)\
        if(fwrite(BUF,SIZE,1,FP) != 1)\
        {\
          fprintf(stderr,"(%s,%lu): fwrite failed\n",__FILE__,\
                  (Showuint) __LINE__);\
          exit(EXIT_FAILURE);\
        }
```

1.1.7 The Header File `intbits.h`

This file contains some definitions manipulating bitvectors represented by a `Uint`. In the comment lines we use w for the word size and ^ for exponentiation of the previous character.

```
#define INTWORDSIZE\
        (UintConst(1) << LOGWORDSIZE)     // # of bits in Uint = w
#define FIRSTBIT\
        (UintConst(1) << (INTWORDSIZE-1)) // 10^w-1
#define ISBITSET(S,I)\
        (((S) << (I)) & FIRSTBIT)         // is ith bit set?
#define ITHBIT(I)\
        (FIRSTBIT >> (I))                 // 0^i10^w-i-1
#define SECONDBIT\
        (FIRSTBIT >> 1)                   // 010^w-2
#define THIRDBIT\
        (FIRSTBIT >> 2)                   // 0010^w-3
#define FIRSTTWOBITS\
        (UintConst(3) << (INTWORDSIZE-2)) // 11^w-2
#define EXCEPTFIRSTBIT\
        (~FIRSTBIT)                       // 01^w-1
#define EXCEPTFIRSTTWOBITS\
        (EXCEPTFIRSTBIT >> 1)             // 001^w-2
#define EXCEPTFIRSTTHREEBITS\
        (EXCEPTFIRSTBIT >> 2)             // 0001^w-3
#define DIVWORDSIZE(I)\
        ((I) >> LOGWORDSIZE)              // (I) div w
#define MODWORDSIZE(I)\
        ((I) & (INTWORDSIZE-1))           // (I) mod w
#define MULWORDSIZE(I)\
        ((I) << LOGWORDSIZE)              // (I) * w
```

The following macro allocates a bitarray of `N` bits. All bits are off.

```
#define INITBITTAB(TAB,N)\
        {\
          Uint *tabptr, tabsize = 1 + DIVWORDSIZE(N);\
          TAB = ALLOCSPACE(NULL,Uint,tabsize);\
          for(tabptr = TAB; tabptr < (TAB) + tabsize; tabptr++)\
          {\
            *tabptr = 0;\
          }\
        }
```

The following macro inititalizes a bitarray such tha all bits are off.

```
#define CLEARBITTAB(TAB,N)\
        {\
          Uint *tabptr, tabsize = 1 + DIVWORDSIZE(N);\
          for(tabptr = TAB; tabptr < TAB + tabsize; tabptr++)\
          {\
            *tabptr = 0;\
          }\
        }
```

`SETIBIT(TAB,I)` sets the `I`-th bit in bitarray `TAB` to 1.

```
#define SETIBIT(TAB,I)    (TAB)[DIVWORDSIZE(I)] |= ITHBIT(MODWORDSIZE(I))
```

`UNSSETIBIT(TAB,I)` sets the `I`-th bit in bitarray `TAB` to 0.

```
#define UNSETIBIT(TAB,I)  (TAB)[DIVWORDSIZE(I)] &= ~(ITHBIT(MODWORDSIZE(I)))
```

`ISIBITSET(TAB,I)` checks if the `I`-th bit in bitarray `TAB` is 1.

```
#define ISIBITSET(TAB,I)  ((TAB)[DIVWORDSIZE(I)] & ITHBIT(MODWORDSIZE(I)))
```

1.2 Debugging Messages

1.2.1 The Header File `debugdef.h`

This file defines macros for producing debugging messages. Except for the first two macros, all macros are only defined in case `DEBUG` is defined.

`STAMP` shows the file and the line number, where the macro is placed. Sometimes this should only be done if a condition is satisfied. So we define a macro `CONDSTAMP`.

```
#define STAMP\
        printf("STAMP(%lu,%s)\n",(Showuint) __LINE__,__FILE__);\
        fflush(stdout)
```

```
#define CONDSTAMPC(C) if(C){STAMP;}
```

The output of the debug macros is performed according to the value of environment variable `DEBUGLEVEL` and `DEBUGWHERE`. We allow a debug level from 0 to 6.

```
#define MAXDEBUGLEVEL 6
```

The general rule of using debug levels is as follows: the larger the debug level, the more messages will be printed. The debug level of the message to be printed should be set according to the following (very informal) rules:

- `DEBUGLEVEL` undefined means no output
- `DEBUGLEVEL=0` means no output
- `DEBUGLEVEL=1` checking, and if something wrong report it
- `DEBUGLEVEL=2` some important statistics
- `DEBUGLEVEL=3` output of the result
- `DEBUGLEVEL=4` output intermediate steps
- `DEBUGLEVEL=5` some less important steps
- `DEBUGLEVEL=6` some even less important steps

In addition `DEBUGLEVEL`, there are two environment variables that trigger the format of the debug message:

- `DEBUGWHERE` undefined means no file name and line number
- `DEBUGWHERE=off` means no file name and line number
- `DEBUGWHERE=on` means to show file and line number in C-Program where macro is used.

`GENDEBUG` is a generic macro to define the debug macros of different arity. `L` is the `debuglevel` according to the environment variable `DEBUGLEVEL`. We check if the variable `L` is in the range `[0,MAXDEBUGLEVEL]`. If `L` is smaller than the `debuglevel` and `debugwhere` is true, then the file and line number is printed. Otherwise, we only check the `debuglevel`. Only if `L` is smaller than the `debuglevel`, the function `fprintf` is called with the appropriate format string and the given arguments. We use `fprintf` to do the debug output, since it allows the compiler to check if the arguments are consistent with the format string.

```
#define GENDEBUG(L)\
        if((L) <= 0 || (L) > MAXDEBUGLEVEL)\
        {\
          fprintf(getdbgfp(),\
                  "file \"%s\", line %lu: level n [1..%lu] required\n",\
                  __FILE__,(Showuint) __LINE__,(Showuint) MAXDEBUGLEVEL);\
          exit(EXIT_FAILURE);\
        }\
        if((L) <= getdebuglevel() && getdebugwhere())\
        {\
          fprintf(getdbgfp(),"file \"%s\", line %lu: ",__FILE__,\
                  (Showuint) __LINE__);\
        }\
        if((L) <= getdebuglevel())
```

`GENDEBUG` is used as a prefix of the `fprintf`-commands followed by an `fflush`. The following macros should always be used for printing debugging info.

```
#define DEBUG0(L,F)\
        GENDEBUG(L){fprintf(getdbgfp(),F);fflush(stdout);}
#define DEBUG1(L,F,A1)\
        GENDEBUG(L){fprintf(getdbgfp(),F,A1);fflush(stdout);}
#define DEBUG2(L,F,A1,A2)\
        GENDEBUG(L){fprintf(getdbgfp(),F,A1,A2);fflush(stdout);}
#define DEBUG3(L,F,A1,A2,A3)\
        GENDEBUG(L){fprintf(getdbgfp(),F,A1,A2,A3);fflush(stdout);}
#define DEBUG4(L,F,A1,A2,A3,A4)\
        GENDEBUG(L){fprintf(getdbgfp(),F,A1,A2,A3,A4);fflush(stdout);}
#define DEBUG5(L,F,A1,A2,A3,A4,A5) \
        GENDEBUG(L){fprintf(getdbgfp(),F,A1,A2,A3,A4,A5);fflush(stdout);}
```

The following macros are abbreviations for calling functions that set the debug level. The macros must appear in the program text before the first debug command. The other macros open and close an extra file to write the debug messages to.

```
#define DEBUGLEVELSET            setdebuglevel()
#define DEBUGLEVELSETFILENAME(F) setdebuglevelfilename(F)
#define DEBUGCLOSEFILE           debugclosefile()
```

The following macros are used to declare local variables for debugging purposes and to call a function depending on the `DEBUGLEVEL`.

```
#define DEBUGDECL(S)   S
#define DEBUGCODE(L,S) if((L) <= getdebuglevel())\
                       {\
                         S;\
                       }
```

To follow the program flow the following macros can be used.

```
#define CASE(I)       fprintf(getdbgfp(),"Case(%ld)\n",(Showsint) (I));fflush(stdout)
#define CASELINE(I)   fprintf(getdbgfp(),"file %s, line %lu: Case(%ld)\n",\
                              __FILE__,(Showuint) __LINE__,(Showuint) (I))
```

Some forward declarations of the functions used for producing debugging messages.

```
Sint getdebuglevel(void);
BOOL getdebugwhere(void);
void showmemsize(void);
void setdebuglevel(void);
void setdebuglevelfilename(char *filename);
FILE *getdbgfp(void);
void debugclosefile(void);
```

1.2.2 The Code File `debug.c`

This module defines functions for handling debug levels and other information related to producing debugging messages. The debug mechanism is only available if the `DEBUG` compiler flag is used.

The following function returns the `DEBUGLEVEL`.

```
Sint getdebuglevel(void)
```

The following function returns the value of `DEBUGWHERE`.

```
BOOL getdebugwhere(void)
```

The following function sets the debug level by looking up the environment variable `DEBUGLEVEL`. Moreover, the environment variable `DEBUGWHERE` is read and `debugwhere` is set accordingly.

```
void setdebuglevel(void)
```

The following function opens the given filename for writing the debug messages to. It also sets the debug level. This function is called only very rarely. If only `setdebuglevel` is called, then the output goes to standard output.

```
void setdebuglevelfilename(char *filename)
```

The following function looks up the output pointer.

```
FILE *getdbgfp(void)
```

The following function closes the debug output pointer, if it is not standard out.

```
void debugclosefile(void)
```

1.3 Error Messages

1.3.1 The Header File `errordef.h`

This file contains some macros to write error messages into a buffer returned by the function `messagespace`.

```
          exit(EXIT_FAILURE);\
        }
```

The different error macros call `sprintf` with the corresponding number of arguments.

```
#define ERROR0(F)\
        GENERROR(sprintf(messagespace(),F))
```

```
#define ERROR1(F,A1)\
        GENERROR(sprintf(messagespace(),F,A1))
```

```
#define ERROR2(F,A1,A2)\
        GENERROR(sprintf(messagespace(),F,A1,A2))
```

```
#define ERROR3(F,A1,A2,A3)\
        GENERROR(sprintf(messagespace(),F,A1,A2,A3))
```

```
#define ERROR4(F,A1,A2,A3,A4)\
        GENERROR(sprintf(messagespace(),F,A1,A2,A3,A4))
```

```
#define ERROR5(F,A1,A2,A3,A4,A5)\
        GENERROR(sprintf(messagespace(),F,A1,A2,A3,A4,A5))
```

The following is a macro to show the usage line for all programs which have options and an indexname as the last argument.

```
#define USAGEOUT\
        ERROR2("Usage: %s options indexname\n"\
               "%s -help shows possible options",\
                argv[0],argv[0]);
```

The following is the standard message in the main function. It shows the program name and the error message as returned by the function `messagespace`.

```
#define STANDARDMESSAGE\
        fprintf(stderr,"%s: %s\n",argv[0],messagespace());\
        return EXIT_FAILURE
```

```
#define SIMPLESTANDARDMESSAGE\
        fprintf(stderr,"%s\n",messagespace());\
        return EXIT_FAILURE
```

1.3.2 The Code File `seterror.c`

This module implements a simple mechanism to write error messages into a global buffer, and to output this buffer when required. We also maintain a global error code.

The following function returns the size of the buffer, and thus the maximal length of an error message.

```
Sint maxerrormsg(void)
```

The following function returns a reference to the buffer for storing or retrieving the error message.

```
char *messagespace(void)
```

The following function sets the error code. It should be called in the innermost function in which the error occurs.

```
void seterror(Sint code)
```

The following function returns the error code.

```
Sint geterror(void)
```

The following function resets the error code to 0.

```
void reseterror(void)
```

1.4 Argument and Option Processing

1.4.1 The Header File `args.h`

This header file implements three macros for checking argument numbers and to parse integers and floats from strings.

The following macro checks if the number of argument is exactly `N`. Otherwise, an error message is thrown.

```
#define CHECKARGNUM(N,S)\
        if (argc != N)\
        {\
          fprintf(stderr,"Usage: %s %s\n",argv[0],S);\
          exit(EXIT_FAILURE);\
        }
```

The following scans an integer `readint` from a string.

```
#define PARSEINTARG(S)\
        if(sscanf(S,"%ld",&readint) != 1 || readint < 0)\
        {\
          fprintf(stderr,"invalid argument \"%s\": " \
                         "non-negative number expected\n",S);\
          exit(EXIT_FAILURE);\
        }
```

The following scans a floating point value `readfloat` from a string.

```
#define PARSEFLOATARG(S)\
        if(sscanf(S,"%f",&readfloat) != 1)\
        {\
          fprintf(stderr,"invalid argument \"%s\":"\
                         " floating point number expected\n",S);\
          exit(EXIT_FAILURE);\
        }
```

1.4.2 The Header File `optdesc.h`

This file defines some macros to manipulate and a type to store information about options.

The macro `OPTION` fills the component of a structure of type `OptionDesc`. This macro should not be used. It is mainly for compatibility of the code.

```
#define OPTION(S,N,O,D)\
        S[N].optname = O;\
        S[N].description = D;\
        S[N].optval = N;\
        S[N].isalreadyset = False;\
        S[N].declared = True
```

The macro `ADDOPTION` has a similar effect, but adding the option is done via the function `addoption`. This is the recommended way to add an option.

```
#define ADDOPTION(N,O,D)\
        if(addoption(&options[0],(Uint) NUMOFOPTIONS,(Uint) N,O,D) != 0)\
        {\
          return -1;\
        }
```

The following macro checks if an option is already set.

```
#define ISSET(N)  (options[N].isalreadyset)
```

The following macro checks if an option is used. If not, then an error is reported.

```
#define OPTIONMANDATORY(A)\
        if(!ISSET(A))\
        {\
          ERROR1("option %s is mandatory",options[A].optname);\
          return -1;\
        }
```

The following four macros check if all options `B` (`C`, `D`, `E`) are used, whenever option `A` is used.

```
#define OPTIONIMPLY(A,B)\
        if(ISSET(A) && !ISSET(B))\
        {\
          ERROR2("option %s requires option %s",\
                  options[A].optname,options[B].optname);\
          return -1;\
        }


#define OPTIONIMPLYEITHER2(A,B,C)\
        if(ISSET(A))\
        {\
          if(!ISSET(B) && !ISSET(C))\
          {\
            ERROR3("option %s requires either option %s or %s",\
                    options[A].optname,options[B].optname,\
                    options[C].optname);\
            return -1;\
          }\
        }


#define OPTIONIMPLYEITHER3(A,B,C,D)\
        if(ISSET(A))\
        {\
          if(!ISSET(B) && !ISSET(C) && !ISSET(D))\
          {\
            ERROR4("option %s requires one of the options %s, %s, %s",\
                    options[A].optname,\
                    options[B].optname,\
                    options[C].optname,\
                    options[D].optname);\
            return -1;\
          }\
        }


#define OPTIONIMPLYEITHER4(A,B,C,D,E)\
        if(ISSET(A))\
        {\
          if(!ISSET(B) && !ISSET(C) && !ISSET(D) && !ISSET(E))\
          {\
            ERROR5("option %s requires one of the options %s, %s, %s, %s",\
                    options[A].optname,\
                    options[B].optname,\
                    options[C].optname,\
                    options[D].optname,\
                    options[E].optname);\
            return -1;\
          }\
        }
```

The following macro checks if two options are used simultaneously. If so, then an error is reported.

```
#define OPTIONEXCLUDE(A,B)\
        if(ISSET(A) && ISSET(B))\
        {\
          ERROR2("option %s and option %s exclude each other",\
                  options[A].optname,options[B].optname);\
          return -1;\
        }
```

An option is described by the following type.

```
typedef struct
{
  char *optname,             // the option string, begins with -
       *description;         // help text describing purpose of option
  Uint optval;               // the unique number of an option
  BOOL isalreadyset,         // has the option already been set?
       declared;             // is the option declared by
                             // a line of the form ADDOPTION
} OptionDescription;
```

1.4.3 The Code File `procopt.c`

The following function initializes the table `options` storeing the information for all options. `numofoptions` is the number of options.

```
void initoptions(OptionDescription *options,Uint numofoptions)
```

The following function adds an option to table `options`. `numofoptions` is the number of options. `optnum` is the number of the option declared. `optname` is the option string to be used when calling the program. Finally, `optdesc` is the option description, i.e. a short explanation of the option. The function returns 0 if everything went fine. In case of an error, a negative error code is returned. The function should be called via the macro `ADDOPTION` as defined in `optdesc.h`.

```
Sint addoption(OptionDescription *options,Uint numofoptions,
               Uint optnum,char *optname,char *optdesc)
```

The following function looks up a string `optstring` in a table of option descriptions. If it has found an option description whose `optname` is identical to `optstring`, it is checked if this option has already been set. If so, then an error message is thrown and the function returns with an error code. If the option was not set before, then the index of the option is returned. If optstring is not found in the table, then this is an error and the program throws an error message and returns with an exit code.

```
Sint procoption(OptionDescription *opt,Uint numofopt,char *optstring)
```

The following function shows the options as specified by the option description `opt`. For each option, it checks if the component `optval` is identical to the index of the option. The option description is output to a file pointer `outfp` with the indentation level delivered by `getoptindent`. `program` is the program which calls `showoptions`.

```
void showoptions(FILE *outfp,char *program,OptionDescription *opt,
                 Uint numofopt)
```

The following function shows the options as specified by the option description `opt`, except for the options whose numbers are specified in list `exludetab`. The option description is output to file pointer `outfp`. `program` is the program which calls `showoptions`.

```
void showoptionswithoutexclude(FILE *outfp,char *program,
                               OptionDescription *opt,
                               Sint *excludetab,Uint numofopt)
```

The following function checks for all pairs of options if they have already been declared to be excluding each other.

```
Sint checkdoubleexclude(Uint numofopts,OptionDescription *opt,
                        Sint *excludetab,Uint len)
```

To specify pairs of options which exclude each other we use an exclude table. An exclude table is just an array of signed integers. Each block of this table consists of a maximal subsequence not beginning with `-1`, but ending with `-1`. Let $i_0, i_1, \ldots, l_r$ be a block. Each number is an option number. For $j \in [1, r]$, option number i_0 and option number i_j exclude each other.

The following function takes a table of option descriptions and an exclude table of length `len`. It checks if two options which exclude each other are set. If there are no such options, then the function returns -1 and throws an error message. The function must be called after parsing the options.

```
Sint checkexclude(OptionDescription *opt,Sint *excludetab,Uint len)
```

The following function takes a table of option descriptions and an exclude table of length `len`. It prints the exclude table in LaTeX format. We use this function to produce a table in the manual of a program showing which options exclude each other.

```
void showexclude(OptionDescription *opt,Sint *excludetab,Uint len)
```

1.5 Space Management

1.5.1 The Header File `spacedef.h`

This file defines macros to simplify the calls to the functions

- `allocandusespaceviaptr`,
- `freespaceviaptr`,

- `dynamicstrdup`,
- `creatememorymapforfiledesc`,
- `creatememorymap`,
- `deletememorymap`.

1. The first parameter to `ALLOCSPACE` is `NULL` or a pointer to a previously allocated space block.
2. The second argument of the macro is the type of the space block to be allocated.
3. The third argument is the number of elements of that type to be allocated space for.

```
#define ALLOCSPACE(S,T,N)\
        (T *) allocandusespaceviaptr(__FILE__,(Uint) __LINE__,S,sizeof(T),N)
```

The macro `FREESPACE` frees the space pointed to by `P`, if this is not `NULL`. It also sets the pointer to `NULL`.

```
#define FREESPACE(P)\
        if((P) != NULL)\
        {\
          freespaceviaptr(__FILE__,(Uint) __LINE__,P);\
          P = NULL;\
        }
```

The remaining macros call the corresponding function with the filename and the line number where the function call appears.

```
#define DYNAMICSTRDUP(S)\
        dynamicstrdup(__FILE__,(Uint) __LINE__,S)
```

```
#define CREATEMEMORYMAP(F,WM,NB)\
        creatememorymap(__FILE__,(Uint) __LINE__,F,WM,NB)
```

```
#define CREATEMEMORYMAPFORFILEDESC(FD,WM,NB)\
        creatememorymapforfiledesc(__FILE__,(Uint) __LINE__,FD,WM,NB)
```

```
#define DELETEMEMORYMAP(MF)\
        deletememorymap(__FILE__,(Uint) __LINE__,(void *) MF)
```

1.5.2 The Code File `space.c`

This file contains functions to store pointers to dynamically allocated spaceblocks, and to maintain the number of cells and their size in each block. The arguments `file` and `line` (if they occur) are always the filename and the linenumber the function is called from. To supply these arguments, we recommend to call the corresponding functions via some useful macros, as defined in the file `spacedef.h`.

1. The function `allocspaceviaptr` should be called via the macro `ALLOCSPACE`.
2. The function `freespaceviaptr` should be called via the macro `FREESPACE`.

3. The function `dynamicstrdup` should be called via the macro `DYNAMICSTRDUP`.

The following is a general macro to print fatal error messages.

```
typedef struct
{
  void *spaceptr;       // ptr to the spaceblock
  Uint sizeofcells,     // size of cells of the block
       numberofcells;   // number of cells in the block
  char *fileallocated;  // the filenames where the block was allocated
  Uint lineallocated;   // the linenumber where the
} Blockdescription;
```

The following function allocates `number` cells of `size` for a given pointer `ptr`. If this is `NULL`, then the next free block is used. Otherwise, we look for the block number corresponding to `ptr`. If there is none, then the program exits with exit code 1.

```
void *allocandusespaceviaptr(char *file,Uint line,void *ptr,
                             Uint size,Uint number)
```

The following function makes a copy of a 0-terminated string pointed to by `source`.

```
char *dynamicstrdup(char *file,Sint line,char *source)
```

The following function frees the space for the given pointer `ptr`. This cannot be `NULL`.

```
void freespaceviaptr(char *file,Sint line,void *ptr)
```

The following function prints a list of block numbers which have not been freed. For each block number the filename and line number in which the call appears allocating which allocated this block.

```
void activeblocks(void)
```

The following function checks if all blocks previously allocated, have explicitely been freed. If there is a block that was not freed, then an error is reported accordingly. We recommend to call this function before the program terminates. This easily allows to discover space leaks.

```
void checkspaceleak(void)
```

The following function shows the space peak in megabytes on `stderr`.

```
void showspace(void)
```

The following function returns the space peak in bytes.

```
Uint getspacepeak(void)
```

The following function delivers the space limit of the machine in megabytes. This only works if the variable `WITHSYSCONF` is defined. This is currently the case for Linux and Solaris.

```
void showmemsize(void)
```

1.5.3 The Code File `mapfile.c`

This file contains functions to map files to memory, to store pointers to the corresponding secondary memory, and to maintain the size of the mapped memory. The arguments `file` and `line` (if they occur) are always the filename and the linenumber the function is called from. To supply the arguments, we recommend to call the corresponding functions via some useful macros, as defined in the file `spacedef.h`.

1. The function `creatememorymap` should be called via the macro `CREATEMEMORYMAP`.
2. The function `deletememorymap` should be called via the macro `DELETEMEMORYMAP`.

The following function opens a file, and stores the size of the file in `numofbytes`. It returns the file descriptor of the file, or a negative error code if something went wrong.

```
Sint simplefileOpen(char *filename,Uint *numofbytes)
```

The following function creates a memory map for a given file descriptor `fd`. `writemap` is true iff the map should be writable, and `numofbytes` is the size of the file to be mapped.

```
void *creatememorymapforfiledesc(char *file,Sint line,Sint fd,
                                 BOOL writemap,Uint numofbytes)
```

The following function returns a memory map for a given filename, or `NULL` if something went wrong.

```
void *creatememorymap(char *file,Sint line,char *filename,
                      BOOL writemap,Uint *numofbytes)
```

The following function unmaps the memorymap referenced by `mappedfile`. It returns a negative value if the `mappedfile` is `NULL`, or the corresponding filedescriptor cannot be found, or the `munmap` operation fails.

```
Sint deletememorymap(char *file,Sint line,void *mappedfile)
```

The following function checks if all files previously mapped, have been unmapped. If there is a file that was not unmapped, then an error is reported accordingly. We recommend to call this function before the program terminates. This easily allows to discover space leaks.

```
void mmcheckspaceleak(void)
```

The following function frees the space for all memory maps which have not already been unmapped.

```
Sint mmwrapspace(void)
```

The following function shows the space peak in megabytes on `stderr`.

```
void mmshowspace(void)
```

The following function returns the space peak in bytes.

```
Uint mmgetspacepeak(void)
```

1.5.4 The Header File `arraydef.h`

This file defines macros to conveniently declare and manipulate dynamic arrays whose size grow on demand. Each dynamic array over some type `T` is implemented by a structure consisting of three components:

1. `space##T` is a pointer to the space block of type `T` allocated for the array.
2. `allocated##T` is an `Uint` storing the number of entries in the array currently allocated.
3. `nextfree##T` holds the smallest index of the array where no value is stored.

Here `##` is the concatenation operator of the C-preprocessor. The following macro expands to a corresponding type definition over some given `TYPE`.

```
#define DECLAREARRAYSTRUCT(TYPE)\
        typedef struct\
        {\
          TYPE *space##TYPE;\
          Uint allocated##TYPE, nextfree##TYPE;\
        } Array##TYPE
```

`INITARRAY` initializes an empty array.

```
#define INITARRAY(A,TYPE)\
        (A)->space##TYPE = NULL;\
        (A)->allocated##TYPE = (A)->nextfree##TYPE = 0
```

`CHECKARRAYSPACE` checks if the integer `nextfree##T` points to an index for which the space is not allocated yet. If this is the case, the number of cells allocated is incremented by `L`. The contents of the previously filled array elements is of course maintained.

```
#define CHECKARRAYSPACE(A,TYPE,L)\
        if((A)->nextfree##TYPE >= (A)->allocated##TYPE)\
        {\
          (A)->allocated##TYPE += L;\
          (A)->space##TYPE\
             = (TYPE *) allocandusespaceviaptr(__FILE__,(Uint) __LINE__,\
                                               (A)->space##TYPE,\
                                               sizeof(TYPE),\
                                               (A)->allocated##TYPE);\
        }
```

The next macro is a variation of `CHECKARRAYSPACE`, which checks if the next `L` cells have been allocated. If not, then this is done.

```
#define CHECKARRAYSPACEMULTI(A,TYPE,L)\
        if((A)->nextfree##TYPE + (L) >= (A)->allocated##TYPE)\
        {\
          (A)->allocated##TYPE += L;\
          (A)->space##TYPE\
             = (TYPE *) allocandusespaceviaptr(__FILE__,(Uint) __LINE__,\
                                               (A)->space##TYPE,\
                                               sizeof(TYPE),\
                                               (A)->allocated##TYPE);\
        }
```

This macro checks the space and delivers a pointer `P` to the next free element in the array.

```
#define GETNEXTFREEINARRAY(P,A,TYPE,L)\
        CHECKARRAYSPACE(A,TYPE,L)\
        P = (A)->space##TYPE + (A)->nextfree##TYPE++
```

This macro checks the space and stores `V` in the `nextfree`-component of the array. `nextfree` is incremented.

```
#define STOREINARRAY(A,TYPE,L,VAL)\
        CHECKARRAYSPACE(A,TYPE,L)\
        (A)->space##TYPE[(A)->nextfree##TYPE++] = VAL
```

This macro frees the space for an array if it is not `NULL`.

```
#define FREEARRAY(A,TYPE)\
        if((A)->space##TYPE != NULL)\
        {\
          FREESPACE((A)->space##TYPE);\
        }
```

Some declarations for the most common array types.

```
DECLAREARRAYSTRUCT(Uchar);
DECLAREARRAYSTRUCT(Ushort);
DECLAREARRAYSTRUCT(char);
DECLAREARRAYSTRUCT(Uint);
DECLAREARRAYSTRUCT(Sint);
DECLAREARRAYSTRUCT(PairUint);
DECLAREARRAYSTRUCT(ThreeUint);
```

And some type synonyms.

```
typedef ArrayUint  ArrayPosition;
typedef ArrayUchar ArrayCharacters;
```

The following array type has some extra components. However, it can be manipulated by the macros above since the record-components `spaceStrings`, `nextfreeStrings`, and `allocatedStrings` is declared appropriately.

```
typedef struct
{
  Stringtype *spaceStrings;
  Uchar *stringbuffer;
  Uint stringbufferlength, nextfreeStrings, allocatedStrings;
} ArrayStrings;
```

1.5.5 The Code File `safescpy.c`

The following function copies the 0-terminated string pointed to by `source` to the memory area pointed to by `dest`, provided `source` is shorter than `maxlen`. If this is not true, then the function returns a negative error code. Otherwise the return value is 0.

```
Sint safestringcopy(char *dest,char *source,Sint maxlen)
```

1.6 Multiple Sequences

1.6.1 The Header File `multidef.h`

This file defines the datatype `Multiseq` which stores information about k-sequences T_0, …, T_{k-1}:

1. For each $i \in [0, k-1]$, `startdesc[i]` stores the index in `descspace.spaceUchar`, where a textual description for sequence T_i starts. A description for sequence T_i ends with a newline character at index `startdesc[i+1]-1`. The description can e.g. be the text following the symbol `>` in a fasta formatted file.
2. For each $i \in [0, k-2]$, `markpos[i]` is the position of a *separator character* between sequence T_i and T_{i+1}.
3. Let $i \in [0, k-1]$. If $i = 0$, then T_i is stored in the component `sequence` from index 0 to index $|T_i| - 1$. If $i > 0$, then T_i is stored in the component `sequence` from index `markpos[i-1]+1` to index `markpos[i-1]` $+ 1 + |T_i|$.
4. `numofsequences` is the number k of sequences stored.
5. `totallength` is the total length of the stored sequences including the $k - 1$ separator characters.

The following defines the separator symbol for fasta files.

```
#define FASTASEPARATOR '>'
```

For a given multiseq and sequence number, the following macros deliver a pointer to the first character of the description, and the length of the description.

```
#define DESCRIPTIONSTARTDESC(MS,SN)\
        ((MS)->startdesc[SN])
```

```
#define DESCRIPTIONPTR(MS,SN)\
        ((MS)->descspace.spaceUchar + DESCRIPTIONSTARTDESC(MS,SN))
```

```
#define DESCRIPTIONLENGTH(MS,SN)\
        (DESCRIPTIONSTARTDESC(MS,(SN)+1) - DESCRIPTIONSTARTDESC(MS,SN))
```

The following macros specifies a default initialization of a structure of type `Showdescinfo`.

```
#define ASSIGNDEFAULTSHOWDESC(DESC)\
            (DESC)->defined = True;\
            (DESC)->skipprefix = 0;\
            (DESC)->maxlength = 0;\
            (DESC)->replaceblanks = False;\
            (DESC)->untilfirstblank = False
```

The following defines the undefined file separator position

```
#define UNDEFFILESEP 0
```

```
typedef struct
{
  ArrayPosition markpos;
  Uint *startdesc,                    // of length numofsequences + 1
       numofsequences,                // the number of sequences
       totallength;                   // the total length of all sequences
  ArrayCharacters descspace;          // the space for the descriptions
  Uchar *sequence,                    // the concatenated sequences
        *rcsequence,                  // NULL or points to
                                      // reverse complemented sequences
        *originalsequence;            // NULL or points to orig. sequence

} Multiseq;
```

The following type describes how to format a sequence description.

```
typedef struct
{
  BOOL defined,           // show a description
       replaceblanks,     // replaceblanks by underscore
       untilfirstblank;   // only show sequence until first blank
  Uint skipprefix,        // always skip this number of prefixes
       maxlength;         // maximal number of chars of description to be shown
} Showdescinfo;
```

The following type is used to store some basic information about a sequence stored in a `Multiseq`-record.

```
typedef struct
{
  Uint seqnum,        // the sequence number in multiseq
       seqstartpos,   // the position of the first character in multiseq.sequence
       seqlength,     // the length of the sequence
       relposition;   // the relative position of the sequence
} Seqinfo;
```

1.6.2 The Code File `multiseq.c`

This file implements the data type `Multiseq`.

`initmultiseq` initializes a `Multiseq` record such that it contains no sequences.

```
void initmultiseq(Multiseq *multiseq)
```

The following function frees the space allocated for a `multiseq`.

```
void freemultiseq(Multiseq *multiseq)
```

The following function applies a function `apply` to all sequences in a `multiseq`. `rcmode` is `True` iff the function is to be applied to the reverse complemented sequence. In each call, `apply` has the following arguments:

- the first argument is `applyinfo`
- the second argument is the number of the current sequence

- the third argument is a pointer to the start of the sequence
- the fourth argument is the length of the sequence

```
Sint overallsequences(BOOL rcmode,Multiseq *multiseq,void *applyinfo,
                      Sint(*apply)(void *,Uint,Uchar *,Uint))
```

Suppose we have a sequence of length `totalwidth` and a `position` in the range $[0, \texttt{totalwidth}-1]$. Given a sorted array of separators `recordspes` of length `numofrecords` such that each separator is a position in the range $[0, \texttt{totalwidth}-1]$. The record separator divides the sequence into records numbered from 0 to `numofrecords`. The following function `getrecordnum` delivers the number of the record in which position occurs. If the position is not in the correct range, then a negative error code is returned. The running time of `getrecordnum` is $O(\log_2 \texttt{numofrecords})$.

```
Sint getrecordnum(Uint *recordseps,Uint numofrecords,Uint totalwidth,
                  Uint position)
```

Given a `multiseq`, and a position in `multiseq->sequence`, the function `getseqnum` delivers the sequence number for `position`. If this cannot be found, then a negative error code is returned. The running time of `getseqnum` is $O(\log_2 k)$, where k is the number of sequences in `multiseq`.

```
Sint getseqnum(Multiseq *multiseq,Uint position)
```

The function `pos2pospair` computes a pair (i, j) stored in `pos->uint0` and `pos->uint1` such that i is the sequence number for `position` in `multiseq`, and j is the relative index for `position` in the ith sequence T_i. If this cannot be found, then a negative error code is returned. In case of success, the function returns 0. The running time of `pos2pair` is $O(\log_2 k)$, if k is the number of sequences in `multiseq`.

```
Sint pos2pospair(Multiseq *multiseq,PairUint *pos,Uint position)
```

1.7 MUM Candidates

1.7.1 The Header File `mumcand.h`

The following structure stores MUM candidates. That is, maximal matches which are unique in the subject-sequence but not necessarily in the query sequence.

```
typedef struct
{
  Uint mumlength,     // length of the mum
       dbstart,       // start position in the subject-sequence
       queryseq,      // number of the query sequence
       querystart;    // start position in the query sequence
} MUMcandidate;
```

We store MUM-candidates in a table and hence declare a corresponding array.

```
DECLAREARRAYSTRUCT(MUMcandidate);
```

1.7.2 The Code File `cleanMUMcand.c`

This module contains functions to extract from a table of MUM-candidates those which are also unique in the query sequence.

Output all MUM candidates that are unique in the query sequence. These are the MUMs. The MUM-candidates are stored in table `mumcand`. The MUM is processed further by the function `processmum` which takes the `processinfo` as an argument.

```
Sint mumuniqueinquery(void *processinfo,
                      Sint (*processmum)(void *,Uint,Uint,Uint,Uint),
                      ArrayMUMcandidate *mumcand)
```

1.8 Measuring Time

1.8.1 The Code File `clock.c`

The following function initializes the clock.

```
void initclock(void)
```

The following function delivers the time since the clock was initialized. The time is reported in seconds as a floating point value.

```
double getruntime(void)
```

The following function delivers the clock ticks betwenn `startclock` to `stopclock`.

```
Uint getclockticks(void)
```

2 The Suffix Tree Library in streesrc

2.1 The Header File `streetyp.h`

A `Reference` consists of an `address` pointing a leaf, or to a branching node. The boolean `toleaf` is `True` if and only if `address` points to a leaf.

```
typedef struct
{
  BOOL toleaf;
  Uint *address;
} Reference;
```

The following types are used for references to leaves and branching nodes, respectively. We will always identify a leaf and and branching node with their references.

```
typedef Uint * Bref;
typedef Uint * Lref;
```

For each branching node we store five values, as described in Section A.2. These values comprise the following structure.

```
typedef struct
{
  Uint headposition,          // the head position of the branching node
       depth;                 // the depth of the branching node
  Bref suffixlink;            // the suffix link is always to a branching node
  Reference firstchild,       // the reference to the first child
            branchbrother;    // the reference to the right brother;
                              // if this doesn't exist then it's NULL
} Branchinfo;
```

For each leaf, we store a reference to its right brother, which is `NULL`, if the right brother does not exist. This is expressed in the type synonym `Leafinfo`.

```
typedef Reference Leafinfo;
```

A suffix tree is implemented by the type `Suffixtree`. This structure contains several components which are mostly only used during the suffix tree construction. For applications, assume the following definition. Note that the input sequence is represented as an array of elements of type `SYMBOL`. The latter is a synonym for `Uchar` by default.

```
typedef struct
{
  SYMBOL *text;       // points to the input string
  Uint textlen;       // the length of the input string
  Uint *branchtab;    // stores the infos for the branching nodes
  Uint *leaftab;      // stores the brother-references of the leaves
} Suffixtree;
```

A location is implemented by the type `Location`.

```
typedef struct
{
  Stringtype locstring; // string represented by location
  Bref previousnode;    // reference to previous node (which is branching)
  SYMBOL *firstptr;     // pointer to first character of edge label
  Uint edgelen,         // length of edge
       remain;          // number of remaining characters on edge
  Reference nextnode;   // reference to node the edge points to
} Location;
```

If a location is a node $\overline{u}$, we set `remain` to 0, and store a reference to $\overline{u}$ in `nextnode`. Moreover, we store a position where u starts and its length in `locstring`. If the location is of the form $(\overline{u}, v, w, \overline{uvw})$, then the components of the location satisfies the following values:

1. `previousnode` is a reference to $\overline{u}$
2. `firstptr` points to the first symbol of the edge label vw.
3. `edgelen` $= |vw|$
4. `remain` $= |w|$
5. `nextnode` is a reference to $\overline{uvw}$.

Since w is not empty, a location is a node location if and only if `remain` is 0.

2.2 The Code File `streefiledoc.c`

2.3 Construction of Suffix Trees

Some parts of the suffix tree library are implemented as macros.

- `CONSTRUCTSTREE(ST,TEXT,TEXTLEN,ACTION)` calls the suffix tree construction function for a `TEXT` of length `TEXTLEN`. In case, the construction was successful, the suffix tree is stored in `ST`. In case, the construction was not successful, an error message is written to stderr, and `ACTION` (e.g. a return statement with an error code) is performed. The macro
- `ROOT(ST)` returns the address of the root of the suffix tree `ST`.
- `ROOTLOCATION(LOC)` returns `True` if and only if the location `LOC` refers to the root of the suffix tree `ST`.

The following function constructs the suffix tree for the input string `text` of length `textlen`. The result is stored in a structure pointed to by `stree`. The return value is `0` for success, and `-1` for failure. In the latter case an error message is returned by the function `messagespace()`.

```
Sint constructstree(Suffixtree *stree,SYMBOL *text,Uint textlen)
```

In some applications it is convenient to perform some extra computations during the suffix tree construction. For this purpose, there are two variations of `constructstree`:

The following function additionally marks all branching nodes which are maximal. A branching node $\overline{v}$ is *maximal* if and only if there is no suffix link to $\overline{v}$. The maximal nodes can be enumerated using the function `overmaximalstree`, see Section 2.5.

```
Sint constructmarkmaxstree(Suffixtree *stree,SYMBOL *text,
                           Uint textlen)
```

The following variation constructs the suffix tree and calls for each $j \in [0, n]$ the function `processhead` with the second argument being j, as soon as $head_j$ is computed. The first argument of `processhead` is the current suffix tree, and the third argument is the pointer `processheadinfo`.

```
Sint constructheadstree(Suffixtree *stree,SYMBOL *text,
                        Uint textlen,
                        void(*processhead)(Suffixtree *,Uint,
                                           void *),
                        void *processheadinfo)
```

The following variation constructs the suffix tree for `text`. Additionally, whenever $textlen > 1000$, the function `progress` is called after $i \cdot textlen/1000$ construction steps, for $i \in [1, 1000]$. Moreover, the function `finalprogress` is called at the end of the suffix tree construction. When they are called, both functions are supplied with the argument `info`.

```
Sint constructprogressstree(Suffixtree *stree,SYMBOL *text,
                            Uint textlen,
                            void (*progress)(Uint,void *),
                            void (*finalprogress)(void *),
                            void *info)
```

For example, one could define `progress` and `finalprogress` as follows, in order to write a line of 79 dots on the given file pointer, to show the progress of the suffix tree construction:

```
void progresswithdot(Uint nextstep,void *info)
```

```
void finalprogress(void *info)
```

The following function stores for each branching node $\overline{v}$ the *lefcount*, i.e. the number of leafs in the subtree below $\overline{v}$. The computation of the leafcounts is done in linear time. It can be retrieved using the function `getleafcountstree`, described below.

```
void addleafcountsstree(Suffixtree *stree)
```

The following function frees the space allocated for `stree`.

```
void freestree(Suffixtree *stree)
```

The following function returns the maximal length of an input string allowed for the suffix tree construction.

```
Uint getmaxtextlenstree(void)
```

2.4 Accessing the Stored Information

The following function stores the information for leaf `lref` in the structure `leafinfo`.

```
void getleafinfostree(Suffixtree *stree,Leafinfo *leafinfo,
                      Lref lptr)
```

The following function stores the information for branching node `bnode` in `branchinfo`. For efficiency reason, the function does not always compute all 5 components stored for a branching node. Instead, the components are delivered as specified by `whichinfo`. This has the following possible bits:

```
ACCESSDEPTH
ACCESSHEADPOS
ACCESSSUFFIXLINK
ACCESSFIRSTCHILD
ACCESSBRANCHBROTHER
```

```
void getbranchinfostree(Suffixtree *stree,Uint whichinfo,
                        Branchinfo *branchinfo,Bref btptr)
```

The following function shows the path for the node `bnode`. A symbol a is shown by applying the function `showchar` to it and the additional argument `info`.

```
void showpathstree(Suffixtree *stree,Bref bnode,
                   void (*showchar)(SYMBOL,void *),void *info)
```

The following function stores a start position and the length of $head_j$, for the current j in `str`. The start position is smaller than j.

```
void getheadstringstree(Suffixtree *stree,Stringtype *str)
```

The following function returns `True`, if and only if the branching node `bnode` has exactly two leaves, say with leaf numbers i and j, $i < j$. In this case, i is stored in `twoleaves->uint0` and j is stored in `twoleaves->uint1`.

```
BOOL exactlytwoleavesstree(Suffixtree *stree,
                           PairUint *twoleaves,Bref start)
```

The following function returns the leafcount for a node referenced by `nodeptr`. The function `addleafcountstree` must have been called exactly once before, in order to compute the leafcounts.

```
Uint getleafcountstree(Suffixtree *stree,Bref nodeptr)
```

The following function computes a table `depthtab`, such that `depthtab->spaceUint[i]` holds the number of branching nodes in the given suffix tree whose depth is exactly i. The maximal depth of any branching node is `depthtab->nextfreeUint-1`.

```
void makedepthtabstree(ArrayUint *depthtab,Suffixtree *stree)
```

2.5 Traversing the Suffix Tree

The following function implements the function *scanprefix*, restricted to the case that the starting location is a node location. `stree` is the suffix tree, `outloc` is the resulting location, `startnode` is the branching node, the scanning starts from, and `left` and `right` delimit the string, say s, to be scanned. The function returns `NULL`, if s is scanned completely. Otherwise, it points to a suffix of s. `rescanlength` is the length of a prefix of s that already occurs in the suffix tree. It is always safe to set `rescanlength` to 0.

```
SYMBOL *scanprefixfromnodestree(Suffixtree *stree,Location *loc,
                                Bref btptr,SYMBOL *left,
                                SYMBOL *right,Uint rescanlength)
```

The following function implements the function *scanprefix*. `stree` is the suffix tree, `outloc` is the resulting location, `inloc` is the location, the scanning starts from, and `left` and `right` delimit the string, say s, to be scanned. The function returns `NULL`, if s is scanned completely. Otherwise, it points to a suffix of s. `rescanlength` is the length of a prefix of s that already occurs in the suffix tree. It is always safe to set `rescanlength` to 0.

```
SYMBOL *scanprefixstree(Suffixtree *stree,Location *outloc,
                        Location *inloc,SYMBOL *left,
                        SYMBOL *right,Uint rescanlength)
```

The following function is similar to the function *scanprefixfromnodestree*. It additionally stores in the array `path` the sequence of branching nodes that has been visited during the scan to location `loc`. The array `path` may not be empty, in which case the list of visited branching nodes is appended to it.

```
SYMBOL *findprefixpathfromnodestree(Suffixtree *stree,
                                    ArrayPathinfo *path,
                                    Location *loc,Bref btptr,
                                    SYMBOL *left,SYMBOL *right,
                                    Uint rescanlength)
```

The following function is similar to the function *scanprefixstree*. It additionally stores in the array `path` the sequence of branching nodes that has been visited during the scan to location `inloc`. The array `path` may not be empty, in which case the list of visited branching nodes is appended to it.

```
SYMBOL *findprefixpathstree(Suffixtree *stree,
                            ArrayPathinfo *path,
                            Location *outloc,
                            Location *inloc,
                            SYMBOL *left,SYMBOL *right,
                            Uint rescanlength);
```

The following function implements the function *rescan*. `stree` is the suffix tree, `outloc` is the resulting location, `startnode` is the branching node, the scanning starts from, and `left` and `right` delimit the string, say s, to be rescanned.

```
void rescanstree(Suffixtree *stree,Location *loc,
                 Bref btptr,SYMBOL *left,SYMBOL *right)
```

The following function implements the function *linkloc*. `stree` is the suffix tree, `outloc` is the resulting location, and `inloc` is the input location.

```
void linklocstree(Suffixtree *stree,Location *outloc,
                  Location *inloc)
```

The following function applies the function `processnode` to all branching nodes of the suffix tree `stree`. The *root* is skipped, if and only if `skiproot` is `True`. The arguments of `processnode` are as follows: The suffix tree `stree`, the branching node, its depth, its head position, and the pointer `info`.

```
void overallstree(Suffixtree *stree,BOOL skiproot,
                  void(*processnode)(Suffixtree *,Bref,
                                     Uint,Uint,void *),
                  void *info)
```

The following function applies the function `processnode` to all maximal branching nodes of the suffix tree `stree`. The arguments of `processnode` are as follows: The suffix tree `stree`, the reference to the branching node, its depth, its head position, and the pointer `info`.

```
void overmaximalstree(Suffixtree *stree,
                      void(*processnode)(Suffixtree *,Bref,
                                         Uint,Uint,void *),
                      void *info)
```

The following function enumerates all immediate successors of the branching node `bnode`. Suppose the leaf with leaf number `j` is a successor of `bnode`. Then the function `processleaf` with arguments `stree`, `j`, and `info` is called. Suppose the branching node `bsucc` is a successor of `bnode`. Then the function `processbranch` is called with arguments `stree`, `bsucc`, and `info`.

```
void oversuccsstree(Suffixtree *stree,Bref bnode,
                    void(*processleaf)(Suffixtree *,Uint,void *),
                    void(*processbranch)(Suffixtree *,Bref,void *),
                    void *info)
```

The following function performs a (possibly) limited depth first traversal of the suffix tree rooted by `startnode`. `startnode` can be a reference to a leaf or a reference to a branching node. The nodes of the subtree are enumerated in depth first left to right order. The argument `processleaf` is not allowed to be `NULL`. `processbranch1` and `processbranch2` can either be both `NULL` or both different from `NULL`.

- Each time a leaf, say $\overline{v}$, with leaf number `j` is encountered, the function `processleaf` is called with arguments `j`, `lca`, and `info`. `lca` is the longest common ancestor of $\overline{v}$ and the previous leaf encountered during the traversal. `lca` is `NULL`, if $\overline{v}$ is the first leaf encountered in the traversal. If `processleaf` returns a value smaller than 0, then `depthfirststree` terminates with a return value `-1`.

- Each time a branching node `bsucc` is visited for the first time, the function `processbranch1` is called with arguments `bsucc` and `info`. If `processbranch1` returns `False`, then the entire subtree below `bsucc` is discarded. Otherwise the depth first traversal continues with the subtree rooted by `bsucc`.

- Each time a branching node `bsucc` is visited for the second time (i.e. the entire subtree below `bsucc` has been processed), the function `processbranch2` is called with arguments `bsucc` and `info`. If `processbranch2` returns a value smaller than 0, then `depthfirststree` returns with value `-1`.

Either `processleaf` or `processbranch1` and `processbranch2` can be `NULL`, in which case there is no function call. If `stoptraversal` is not `NULL`, then after each call the function `stoptraversal` is applied to `stopinfo`. If the return value of this call is `True`, then the depth first traversal stops.

In case everything goes right, `depthfirststree` returns 0.

```
Sint depthfirststree(Suffixtree *stree,Reference *startnode,
                     Sint (*processleaf)(Uint,Bref,void *),
                     BOOL (*processbranch1)(Bref,void *),
                     Sint (*processbranch2)(Bref,void *),
                     BOOL (*stoptraversal)(void *),
                     void *stopinfo,void *info)
```

3 The Implementation of Maxmat3 in MM3

3.1 The Header File `maxmatdef.h`

This file defines some constants and types for computing maximal matches using suffix trees.

The following two characters replace wildcard characters in a DNA sequences whenever the option `-n` (for match only nucleotides) is used. One is for the subject string and the other for the query string

```
#define MMREPLACEMENTCHARSUBJECT (WILDCARD-2)
#define MMREPLACEMENTCHARQUERY   (WILDCARD-3)
```

The maximal number of query files.

```
#define MAXNUMOFQUERYFILES       32
```

The following type contains all information derived from parsing the arguments of the program

```
typedef struct
{
  BOOL showstring,               // show the matching string
       reversecomplement,        // compute matches on reverse strand
       forward,                  // compute matches on forward strand
       showreversepositions,     // give reverse pos. rel. to orig. string
       showsequencelengths,      // show length of sequences on header line
       matchnucleotidesonly,     // match ONLY acgt's
       cmumcand,                 // compute all maximal matches
       cmum;                     // compute real matches unique in both sequences
  Uint minmatchlength,           // minimal length of a match to be reported
       numofqueryfiles;          // number of query files
  char program[PATH_MAX+1],      // the path of the program
       subjectfile[PATH_MAX+1],  // filename of the subject-sequence
       queryfilelist[MAXNUMOFQUERYFILES][PATH_MAX+1];
                                 // filenames of the query-sequences
} MMcallinfo;
```

Functions processing a maximal match are of the following type.

```
typedef Sint (*Processmatchfunction)
             (void *,Uint,Uint,Uint,Uint);
```

3.2 The Code File `maxmatopt.c`

This file contains functions to parse the possible options of `maxmat3` and to appropriately initialize the `mmcallinfo`-record according to the given options.

The following type declares symbolic constants for the options.

```
typedef enum
{
  OPTSHOWSTRING = 0,
  OPTCOMPUTEBOTHDIRECTIONS,
  OPTSHOWREVERSEPOSITIONS,
  OPTLEASTLENGTH,
  OPTSHOWSEQUENCELENGTHS,
  OPTMATCHNUCLEOTIDESONLY,
  OPTONLYREVERSECOMPLEMENT,
  OPTMUM,
  OPTMUMCAND,
  OPTHELP,
  NUMOFOPTIONS
} Optionnumber;
```

The following function stores the help-text for the option `-l`. This is necessary, since the text depends on the value of the symbolic constant `DEFAULTMINUNIQUEMATCHLEN`;

The following function declares the possible options in a record `options`. It then ananlyzes the `argv`-vector step by step. If everything is okay, 0 is returned and the `mmcallinfo` is correctly initialized. Otherwise, a negative value is returned.

```
Sint parsemaxmatoptions(MMcallinfo *mmcallinfo,Argctype argc, char **argv)
```

3.3 The Code File `maxmatinp.c`

This module contains all functions parsing the subject and query sequences.

For each sequence in the `Multiseq`-record, the dynamic array `startdesc` stores the positions in the dynamic array `descspace` where the sequence description starts. The following macro checks if enough memory has been allocated for `startdesc`. If not, then this is done by incrementing the size of the array by 128 entries. Finally, the appropriate entry in `startdesc` is assigned the correct value.

The following function scans a string containing the content of a multiple fasta formatted file. The parameter are as follows:

1. `multiseq` is the `Multiseq`-record to store the scanned information in.
2. `filename` is the information from which the file contents was read.
3. `replacewildcardchar` is the character used to replace a wildcard (then it should be different from the characters occuring in DNA sequences) or 0 if wildcards are not replaced.
4. `input` points to the inputstring to be scanned,
5. `inputlen` is the length of the input.

Each sequence description begins with the symbol `>`. If it does, then this symbol is skipped. The rest of the line up to the first white space character is stored in `descspace`. Otherwise, the rest of the line is discarded. The remaining lines (until the next symbol `>` or the end of the input string) are scanned for alphanumeric characters which make up the sequence. White spaces are ignored. Upper case characters are transformed to lower case. The input string must contain at least one sequence. In case of a error, an negative error code is returned. In case of success, the return code is 0.

```
Sint scanmultiplefastafile (Multiseq *multiseq,
                            char *filename,
                            Uchar replacewildcardchar,
                            Uchar *input,
                            Uint inputlen)
```

The following function reads the subject and queryfile and delivers the parsed multiple sequences in the corresponding `Multiseq`-records. The files are read via memory mapping. The subject file must contain exactly one sequence. Both files cannot be empty. The parameter `matchnucleotidesonly` is true iff if the programm was called with option `-n`, which means that only `ACGTs` are matched. If an error occurs, then the function delivers a negative error code. Otherwise the error code is 0.

```
Sint getmaxmatinput (Multiseq *subjectmultiseq,
                     BOOL matchnucleotidesonly,
                     char *subjectfile)
```

3.4 The Code File `findmaxmat.c`

This file contains functions to compute maximal matches of some minimum length between the subject-sequence and the query-sequence. This is done by a traversal of the suffix tree of the subject-sequence. For each suffix, say s, of the query, the location `ploc` of some prefix p of s is determined. Let `pmax` be the longest prefix of s that occurs as a substring of the subject-sequence. p is determined as follows.

- If the length of `pmax` is $\leq$ `minmatchlength`, then $p =$ `pmax`.
- If the length of `pmax` is $>$ `minmatchlength`, then p is the prefix of `pmax` of length `minmatchlength`.

Given `ploc`, the location `maxloc` of `pmax` is determined, thereby keeping track of the branching nodes visited during this matching step. Finally, the suffix tree below location `ploc` is traversed in a depth first strategy. This delivers the set of suffixes representing a match of length at least `minmatchlength` against a prefix of s. Using a stack one keeps track of the length of the longest common prefix of each encountered suffix and s. Finally, it is checked whether the match between the match is maximal by comparing the characters to the left and to the right of the two instances of the match.

For the depth first traversal we need a stack containing elements of the following type. Each stack element stores information for a branching node of the suffix tree. The top of the stack corresponds to the branching node, say b, currently visited. `querycommondepth` is the length of the longest common prefix of s and all suffixes represented by leaves in the subtree below b. `onmaxpath` is `true` iff the sequence corresponding to b is a prefix of the query.

```
typedef struct
{
  Uint querycommondepth;
  BOOL onmaxpath;
} Nodeinfo;
```

The stack is represented by a dynamic array of elements of type `Nodeinfo`.

The following type contains all information required during the computation of the maximal matches.

```
typedef struct
{
  Suffixtree *stree;                // reference to suffix tree of subject-seq
  ArrayNodeinfo commondepthstack; // stack to store depth values
  ArrayPathinfo matchpath;        // path of br. nodes from ploc to maxloc
  Location maxloc;                // location of pmax
  Uchar *query,                   // the query string
        *querysuffix;             // current suffix of query
  Uint querylen,                  // length of the current query
       queryseqnum,               // number of query sequence
       minmatchlength,            // min length of a match to be reported
       depthofpreviousmaxloc;     // the depth of the previous maxloc
  Processmatchfunction processmatch; // this function processes found match
  void *processinfo;            // first arg. when calling previous function
} Maxmatchinfo;
```

The following function finds all maximal matches between the subject sequence and the query sequence of length at least `minmatchlength`. To each match the function `processmatch` is applied, with `processinfo` as its first argument. `query` is the reference to the query, `querylen` is the length of the query and `queryseqnum` is the number of the query sequence. Initially, the function appropriately intializes the `maxmatchinfo`-record. It then scans `query` to find `ploc` for the longest suffix of `query`. The depth of `ploc` is stored in `depthofpreviousmaxloc`. In the `for`-loop each instance of `ploc` is determined and processed further by `enumeratemaxmatches` whenever its depth is longer than the minimum match length.

```
Sint findmaxmatches(Suffixtree *stree,
                    Uint minmatchlength,
                    Processmatchfunction processmatch,
                    void *processinfo,
                    Uchar *query,
                    Uint querylen,
                    Uint queryseqnum)
```

3.5 The Code File `findmumcand.c`

This module contains functions to compute MUM-candidates using a linear time suffix tree traversal.

The following function traverses the suffix tree guided by some query string. The parameters are as follows:

1. `stree` is the suffix tree constructed from the subject sequence.
2. `minmatchlength` is the minimal length of the MUMs as specified
3. `processmumcandidate` is the function to further process a MUM-candidate.
4. `processinfo` points to some values additionally required by the function `processmumcandidate`.
5. `query` points to the query which is of length `querylen`
6. `seqnum` is the number of the current query sequence in the `Multiseq`-record.

For each suffix, say s, of the query sequence the following function computes the location of the longest prefix of s that is a substring of the subject sequence. This is done by iteratively calling the function `scanprefixfromnodestree`. In each step, the scan starts at a location which represents a prefix of the maximaly matching substring. The locations are computed using the function `linklocstree`. In case an error occurs, a negative number is returned. Otherwise, 0 is returned.

```
Sint findmumcandidates(Suffixtree *stree,
                       Uint minmatchlength,
                       Processmatchfunction processmumcandidate,
                       void *processinfo,
                       Uchar *query,
                       Uint querylen,
                       Uint seqnum)
```

3.6 The Code File `procmaxmat.c`

This file contains functions to appropriately call the function `findmumcandidates` and `findmaxmatches` and to process their result according to the options given by the user.

The following structure contains all information required while computing and processing the matches.

```
typedef struct
{
  Suffixtree stree;             // the suffix tree of the subject-sequence
  Multiseq *subjectmultiseq,    // reference to multiseq of subject
           querymultiseq;       // the Multiseq record of the queries
  ArrayMUMcandidate mumcandtab;// a table containing MUM-candidates
                                // when option -mum is on
  Uint minmatchlength,          // minimum length of a match
       maxdesclength,           // maximal length of a description
       currentquerylen;         // length of the current query sequence
  BOOL showstring,              // is option -s on?
       showsequencelengths,     // is option -L on?
       showreversepositions,    // is option -c on?
       forward,                 // compute forward matches
       reversecomplement,       // compute reverse complement matches
       cmumcand,                // compute MUM candidates
       cmum,                    // compute MUMs
       currentisrcmatch;        // true iff currently rc-matches are computed
} Matchprocessinfo;
```

The following code fragement is used when computing MUMs. It calls the function `mumuniqueinquery`. The MUM-candidates are stored in the dynamic array `mumcandtab`. After the real MUMs are output, the table of MUM-candidates are declared to be empty.

The following function constructs the suffix tree, initializes the `Matchprocessinfo`-record appropriately, initializes the dynamic array `mumcandtab` (if necessary), and then iterates the function `findmaxmatchesonbothstrands` over all sequences in `querymultiseq`. Finally, the space allocated for the suffix tree and the space for `mumcandtab` is freed.

```
Sint procmaxmatches(MMcallinfo *mmcallinfo,Multiseq *subjectmultiseq)
```

3.7 The Code File `maxmat3.c`

This module contains the main function of maxmatch3. It calls the following three functions in an appropriate order and with proper arguments.

The following function is imported form `maxmatopt.c`.

```
Sint parsemaxmatoptions (MMcallinfo *maxmatcallinfo,
                         Argctype argc,
                         char **argv);
```

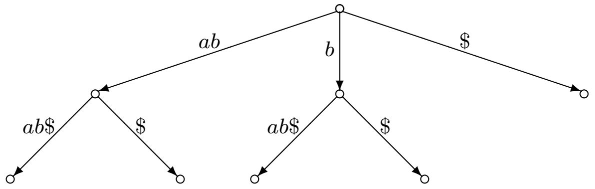
The following function is imported form `maxmatinp.c`.

```
Sint getmaxmatinput (Multiseq *subjectmultiseq,
                     BOOL matchnucleotidesonly,
                     char *subjectfile);
```

The following function is imported form `procmaxmat.c`.

```
Sint procmaxmatches(MMcallinfo *mmcallinfo,
                    Multiseq *subjectmultiseq);
```

Figure 1: The suffix tree for $x = abab$



A Preliminaries

A.1 Suffix Trees

Let Σ be a finite ordered set, the *alphabet*. The size of Σ is k. Σ^* denotes the set of all strings over Σ and ε is the *empty string*. We use Σ^+ to denote the set $\Sigma^* \setminus \{\varepsilon\}$ of non-empty strings. Let $x \in \Sigma^*$ and $x = uvw$ for some possibly empty strings u, v, w. Then u is a *prefix* of x, v is a *substring* of x, w is a *suffix* of x. $|x|$ is the number of characters in x. x_i is the ith character in x. If $|x| = n$, then $x = x_1 x_2 \ldots x_n$.

A Σ^+-*tree* T is a finite rooted tree with edge labels from Σ^+. For each $a \in \Sigma$, every node u in T has at most one a-edge $u \xrightarrow{av} w$ for some string v and some node w.

Let T be a Σ^+-tree. A node in T is *branching* if it has at least two outgoing edges. A *leaf* in T is a node in T with no outgoing edges. An *internal node* in T is either the *root* or a node with at least one outgoing edge. An edge leading to an internal node is an *internal edge*. An edge leading to a leaf is a *leaf edge*.

path(v) denotes the concatenation of the edge labels on the path from the *root* of T to the node v. Due to the requirement of unique a-edges at each node of T, paths are also unique. Therefore, we denote v by $\overline{w}$ if and only if *path*$(v) = w$. The node $\overline{\varepsilon}$ is the *root*. For any node $\overline{w}$ in T, $|w|$ is the *depth* of $\overline{w}$.

A string w *occurs* in T if T contains a node $\overline{wu}$, for some string u. *words*(T) denotes the set of strings occurring in T.

A Σ^+-tree is *compact* if every node is either the *root*, a leaf, or a branching node. A compact Σ^+-tree T is uniquely determined by *words*(T).

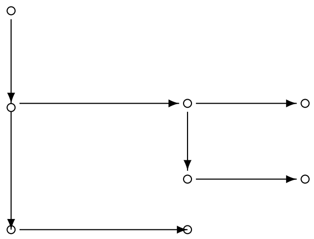
From now on we assume that $x \in \Sigma^+$ is a string of length $n \geq 1$ and that $\$ \in \Sigma$ is a character not occurring in x, the *sentinel*. The *suffix tree* for x, denoted by ST, is the compact Σ^+-tree T s.t. *words*$(T) = \{w \in \Sigma^* \mid w \text{ is a substring of } x\$\}$. Figure 1 shows the suffix tree for $x = abab$.

For any $i \in [1, n+1]$, let $S_i = x_i \ldots x_n\$$ denote the ith non-empty suffix of $x\$$. Note that due to the sentinel no S_i is a proper prefix of any S_j. Thus there is a one-to-one correspondence between the non-empty suffixes of $x\$$ and the leaves of ST. This implies that ST has exactly $n+1$ leaves. Moreover, since $n \geq 1$ and $x_1 \neq \$$, the *root* of ST is branching. Hence each internal node in ST is branching. This means that there are at most n internal nodes in ST.

Let T be a compact Σ^+-tree. The *suffix link* for a node $\overline{aw}$ in T is an unlabeled directed edge in T from $\overline{aw}$ to the node $\overline{w}$, if the latter exists in T. We consider suffix links to be a part of the suffix tree data structure. They are required for most of the linear time suffix tree constructions (see [Wei73, McC76, Ukk95]), and for some applications of suffix trees (see [Gus97]).

Let $head_1 = \varepsilon$ and for $i \in [2, n+1]$ let $head_i$ be the longest prefix of S_i which is also a prefix of S_j

Figure 2: The references of the suffix tree for $x = abab$ (see Figure 1). Vertical arcs stand for *firstchild* references, and horizontal arcs for *branchbrother* and T_{leaf} references.



for some $j \in [1, i-1]$.

For each branching node $\overline{w}$ in ST, let *headposition*$(\overline{w})$ denote the smallest integer $i \in [1, n+1]$ s.t. $w = head_i$. As shown in [Kur99], *headposition*$(\overline{w})$ is well defined. If *headposition*$(\overline{w}) = i$, then we say that the *head position* of $\overline{w}$ is i.

A.2 A Simple Representation

The following representation of suffix trees is suggested in [Kur99]: ST is represented by two tables T_{leaf} and T_{branch} which store the following values: For each *leaf number* $j \in [1, n+1]$, $T_{leaf}[j]$ stores a reference to the right brother of leaf $\overline{S_j}$. If there is no such brother, then $T_{leaf}[j]$ is a nil reference. For each branching node $\overline{w}$, $T_{branch}[\overline{w}]$ stores a *branch record* consisting of five components *firstchild*, *branchbrother*, *depth*, *headposition*, and *suffixlink* specified as follows:

1. *firstchild* refers to the first child of $\overline{w}$
2. *branchbrother* refers to the right brother of $\overline{w}$. If there is no such brother, then the component *branchbrother* is a nil reference.
3. *depth* is the depth of $\overline{w}$
4. *headposition* is the head position of $\overline{w}$
5. *suffixlink* refers to the branching node $\overline{v}$, if w is of the form av for some $a \in \Sigma$ and some $v \in \Sigma^*$

The successors of a branching node are therefore found in a list whose elements are linked via the *firstchild*, *branchbrother*, and T_{leaf} references. To speed up the access to the successors, each such list is ordered according to the first character of the edge labels. Figure 2 shows the child and brother references of the nodes of the suffix tree of Figure 1. We use the following notation to denote a record component: For any component c and any branching node $\overline{w}$, $\overline{w}.c$ denotes the component c stored in the branch record $T_{branch}[\overline{w}]$. Note that the head position j of some branching node $\overline{wu}$ tells us that the leaf $\overline{S_j}$ occurs in the subtree below node $\overline{wu}$. Hence wu is the prefix of S_j of length $\overline{wu}.depth$, i.e. the equality $wu = x_j \dots x_{j+\overline{wu}.depth-1}$ holds. As a consequence, the label of the incoming edge to node $\overline{wu}$ can be obtained by dropping the first $\overline{w}.depth$ characters of wu, where $\overline{w}$ is the predecessor of $\overline{wu}$:

Observation 1 If $\overline{w} \xrightarrow{u} \overline{wu}$ is an edge in ST and $\overline{wu}$ is a branching node, then we have $u = x_i \dots x_{i+l-1}$ where $i = \overline{wu}.headposition + \overline{w}.depth$ and $l = \overline{wu}.depth - \overline{w}.depth$.

Similarly, the label of the incoming edge to a leaf is determined from the leaf number and the depth of the predecessor:

Observation 2 If $\overline{w} \xrightarrow{u} \overline{wu}$ is an edge in ST and $\overline{wu} = \overline{S_j}$ for some $j \in [1, n+1]$, then $u = x_i \dots x_n\$$ where $i = j + \overline{w}.depth$.

A.3 Locations

Every string occurring in an compact Σ^+-tree T can uniquely be described in terms of the nodes and edges of T. This is formalized in the notion of *locations*:

Definition 1 Let T be a compact Σ^+-tree and $s \in words(T)$. The *location* of s in T, denoted by $loc_T(s)$ is defined as follows:

- If $\overline{s}$ is an internal node, then $loc_T(s) = \overline{s}$.
- If $\overline{s}$ is a leaf in T, then there is a leaf edge $\overline{u} \xrightarrow{v} \overline{s}$ in T and $loc_T(s) = (\overline{u}, v, \varepsilon, \overline{s})$.
- If there is no node $\overline{s}$ in T, then there is an edge $\overline{u} \xrightarrow{vw} \overline{uvw}$ in T such that $s = uv$, $v \neq \varepsilon$, $w \neq \varepsilon$ and $loc_T(s) = (\overline{u}, v, w, \overline{uvw})$.

$locations(T) = \{loc_T(s) \mid s \in words(T)\}$ is the *set of locations* in T. If a location is a node, we call it *node location*, otherwise *edge location*. Sometimes we identify a node location with the corresponding node.

By convention the *root* is the location of ε in the empty Σ^+-tree. As locations allow a unified representation of the strings occurring in a compact Σ^+-tree, they are a central (but sometimes implicit) data structure in many algorithms on suffix trees.

The location of a string corresponding to a leaf is not the leaf itself. Instead, it is defined in terms of the edge leading to the leaf. Note that an edge-location $(\overline{u}, v, w, \overline{uvw})$ corresponds to a leaf if and only if $w = \varepsilon$. In an edge-location $(\overline{u}, v, w, \overline{uvw})$ the string w and the node $\overline{uvw}$ are redundant. Both can uniquely be determined from $\overline{u}$ and v provided the compact Σ^+-tree is given.

For convenience we introduce some additional notions related to locations.

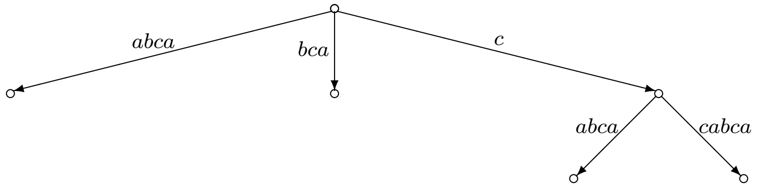
Definition 2 Let T be a compact Σ^+-tree. Suppose $s \in words(T)$.

1. $|loc_T(s)| = |s|$ is the *depth* of the location $loc_T(s)$.
2. Let v be the shortest string s.t. $\overline{sv} \in nodes(T)$. Then $\overline{sv}$ is denoted by $ceiling(loc_T(s))$.
3. For all $a \in \Sigma$ we define: $occurs(loc_T(s), a) \iff sa$ occurs in T.
4. $rescan(loc_T(s), w)$ denotes $loc_T(sw)$ for all $sw \in words(T)$.

Example 1 Let T be the compact Σ^+-tree shown in Figure 3. Then, for instance,

$$\begin{array}{lcllcl}
loc_T(\varepsilon) & = & root & ceiling(loc_T(\varepsilon))) & = & root \\
loc_T(a) & = & (root, a, bca, \overline{abca}) & ceiling(loc_T(a)) & = & \overline{abca} \\
loc_T(abca) & = & (root, abca, \varepsilon, \overline{abca}) & ceiling(loc_T(abca)) & = & \overline{abca} \\
loc_T(c) & = & \overline{c} & ceiling(loc_T(c)) & = & \overline{c} \\
loc_T(cab) & = & (\overline{c}, ab, ca, \overline{cabca}) & ceiling(loc_T(cab)) & = & \overline{cabca}
\end{array}$$

Figure 3: A compact Σ^+-tree



Following a path is the most important operation on suffix trees. In some applications we have to perform this operation starting from an arbitrary location, going down as far as possible. We describe this by a function *scanprefix*.

Definition 3 Let T be a compact Σ^+-tree. For each $s \in words(T)$ and each string w the function $scanprefix : locations(T) \times \Sigma^* \to locations(T) \times \Sigma^*$ is specified as follows:

$$scanprefix(loc_T(s), w) = (loc_T(su), v),$$

where $uv = w$ and u is the longest prefix of w such that $su \in words(T)$.

Example 2 Let T be the compact Σ^+-tree as shown in Figure 3. Then, for instance,

$$\begin{aligned} scanprefix(loc_T(cab), cd) &= (loc_T(cabc), d) \text{ and} \\ scanprefix(loc_T(bca), d) &= (loc_T(bca), d) \end{aligned}$$

For some constructions and many applications of compact Σ^+-trees it is very important to have an efficient access from $loc_T(cy)$ to $loc_T(y)$. This access is provided by a function *linkloc*, that uses the suffix links of the inner nodes as a "shortcut".

Definition 4 Let T be a compact Σ^+-tree. We define the function $linkloc : locations(T) \setminus \{root\} \to locations(T)$ as follows:

$$linkloc(\overline{s}) = \overline{z} \quad \text{where } \overline{s} \to \overline{z} \text{ is the suffix link for } \overline{s}$$

$$linkloc(\overline{u}, av, w, \overline{uavw}) = \begin{cases} loc_T(v) & \text{if } \overline{u} = root \\ rescan(\overline{z}, av) & \text{otherwise} \end{cases}$$
$$\text{where } \overline{u} \to \overline{z} \text{ is the suffix link for } \overline{u}.$$

Note that *linkloc* never uses the suffix link of a leaf.

Lemma 1 Let T be a suffix tree. Suppose that cy and y occur in T. Then

$$linkloc(loc_T(cy)) = loc_T(y)$$

B Index of Types and Functions

We use emphasize mode for type names, sans serif font for function names, and typewriter font for macro names

References

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, New York, 1997.

[Kur99] S. Kurtz. Reducing the Space Requirement of Suffix Trees. *Software—Practice and Experience*, 29(13):1149–1171, 1999.

[McC76] E.M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, **23**(2):262–272, 1976.

[Ukk95] E. Ukkonen. On-line Construction of Suffix-Trees. *Algorithmica*, **14**(3), 1995.

[Wei73] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, The University of Iowa, 1973.