GRAPHEN UND NETZWERKE
UNIVERSITÄT OLDENBURG
DEPARTMENT FÜR INFORMATIK
Fakultät 2 – Informatik, Wirtschafts- und Rechtswissenschaften

Günther Stiege

GHS Graph Handling System
User Manual GHScore
(Version 5.0)

Oldenburg · December 2009

Author's address:

Prof. Dr. Günther Stiege
Universität Oldenburg
Department für Informatik
D-26111 Oldenburg, Germany
`www-bvs.informatik.uni-oldenburg.de`

Bibliographische Angaben:
Stiege, Günther:
GHS Graph Handling System User Manual Version 5.0 / Stiege, Günther – Oldenburg: Universität Oldenburg, 2009

Preface

During the past years, the Graph Handling System GHS *has evolved from a program for finding the biblock decomposition of undirected graphs to a rather large and complex system for handling general graphs. Its kernel is a library of C routines for a variety of graph functions termed* GHScore. *In addition to GHScore, a graphical user interface* GHSgui *which uses a library of java functions has been implemented by Sergej Alekseev and some external contributions have been added to GHS.*

This manual describes GHScore version 4.0 . In the manual the C routines are grouped together in chapters according to the kind of function they perform. The chapters are subdivided in a uniform manner into four sections. The first section gives some graph theoretic background of the functions. It is followed by a short section on formats and data structures. The third section contains a detailed description of the individual functions. A last section with examples completes each chapter.

I have programmed most of the GHScore functions myself. Functions `generatefromed` *and* `cpchtp` *were designed and implemented by Dirk Ahlers and Tobe Toben. Sergej Alekseev is responsible for the implementation of function* `gcomponents`.

I gratefully acknowledge the patience Ingo Stierand and Sergej Alekseev have shown in long and intensive discussions and the many creative ideas they have contributed.

Oldenburg, October 2004 *G. Stiege*

Preface to version 5.0

During the past years GHS has not progressed as much as originally planned. This version 5.0 includes extensions and revisions of red-black trees. Paths, Menger structures and higher decompositions are still lacking. The work on the graphical user interface GHSgui has been suspended. GHSgui is not part of this version.

Again, I have to thank Ingo Stierand and Sergej Alekseev for very valuable help.

Oldenburg, December 2009 *G. Stiege*

Contents

Chapter 1

Introduction

1.1 General Remarks and Overview

The *Graph Handling System GHS* consists of a core component *GHScore* and complementary programs. This manual is concerned with GHScore, some complementary programs are mentioned.
Essentially, GHScore is a collection of C programs offering functions for a variety of graph operations and graph algorithms. The collection is far from being complete and there is no intention to achieve completeness in the future. Beside some functions for elementary graph manipulation, the collection contains mainly functions to decompose graphs into simpler components. With respect to this problem, however, there are some new ideas and algorithms not available in other packages. For an example see chapter 6 “The Biblock Decomposition”.

GHScore is being implemented in C. Users are assumed to have a working knowledge of this programming language and familiarity with an Unix operating system. A basic knowledge of graph theory and graph algorithms is of course assumed, too. The system has been tested and is as stable (or unstable) as a continualy evolving system which is used as research tool is expected to be.

The function groups supported by GHScore are reflected in the chapters following this introduction.

Chapter 2 “Basic Graph Functions"
Reading a graph from a file. Saving a graph to a file. Graph printing programs. Adding and deleting single vertices and single lines. Adding a depth-first forest structure to the graph.

Chapter 3 “Sets”
Functions to handle sets of vertices, sets of lines, and general sets (i. e. sets of character strings).

Chapter 4 “Graph Generating Functions"
Programs for generating subgraphs from sets of vertices and sets of lines. Functions to copy a graph and adapt it to a new type. Functions to generate subgraphs where all vertices up to degree n have been deleted.

Chapter 5 “Weak and Strong Components”
Programs for finding weakly and strongly connected components in general graphs.

Chapter 6	"The Biblock Decomposition" A function which decomposes the cyclic weak connected components of a general graph (of any type) into a stopfree kernel, the peripheral trees, the subcomponents, the internal trees, and the biblocks. In a general graph, the direction of arcs is ignored and the a-cyclic weak components are decomposed. Print functions. A Functions for generating the biblock graph.
Chapter 9	"Paths" *(not yet released)*
Chapter 10	"Menger Structures" *(not yet released)*
Chapter 11	"Higher Decompositions" *(not yet released)*
Chapter 13	"Red-Black Trees" Inserting a record into a red-black tree. Searching a record in a red-black tree. Printing red-black trees. Providing the name of an object given its record. Though not graph functions in the strict sense, a complete set of funtions on binary trees has been added.
Chapter 14	"Stacks and Queues" Push, pop, top, enqueue, dequeue. Variants: a. Multiple simultaneous membership allowed and membership attributes not allowed. b. Multiple simultaneous membership not allowed and membership attributes allowed.

Care has been taken to complement each chapter with a section containing a set of examples. Additional examples are contained in the GHS directory `GHStests` (see section 1.2). The examples are intended to illustrate simple 'normal' cases and not so much special applications. Together with the introductory chapter 1 they may serve as a primer for GHScore.

Errors and system errors: GHScore programs detect a variety of errors. In the case of an error, a explanatory message is written to standard output, indicating the kind of error and the detecting routine. Most errors are "user errors" (inconsistent data and the like). The routine returns. If the routine yields a return value, this will be an error value.

Some errors should never occur assuming the system programs correct. If such a "system error" is detected the program exits with an appropriate error message.

Dynamic memory: GHScore programs frequently allocate and deallocate dynamic memory. An attempt is made to free dynamic memory as soon and as complete as possible. The actual amount of dynamic memory allocated by GHScore programs is recorded in the global extern variable `ghsmemsize` of type `long int`. See examples 2.3, page 16 and 3.1, page 38.[1]

Remark 1.1 Sometimes GHS functions detect an user error only after processing has started and some memory has been assigend. For instance, a graph structure is being constructed from an external description and an inicidence vertex' name is missing in the vertex list. In these cases GHS returns an error code to the calling program, *but does not free the memory used up this point.*

[1] Note that the amount of memory allocated may depend on the computer architecture and the software systems used. The examples of this manual correspond to an PC-based Linux system and a GNU gcc compiler.
The amount of memory allocated depends also on the actual version of GHS since the data structures in GHS may change. Therefore, in an actual program run the numerical values of the examples cited above may differ from those given in this manual.

1.2 How to Use the System

The complete GHS system can freely be used.

The distribution policy of GHS is stated in table 1.1, page 3.

```
# * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * #
#                  Graph Handling System (GHS)                            #
#                   Copyright (c) 2000 - 2003                             #
#               Guenther Stiege and Sergej Alekseev                       #
#                Universitaet Oldenburg, Germany                          #
#                    All rights reserved.                                 #
#                                                                         #
# Redistribution and use in source and binary forms,  with or without     #
# modification, are permitted provided that:                              #
# (1) source code distributions retain the above copyright notice and     #
# this paragraph in its entirety.                                         #
# (2) distributions including binary code include the above copyright     #
# notice and this paragraph in its entirety  in  the documentation or     #
# other materials provided with the distribution.                         #
# (3) all  advertising  materials  mentioning features or use of this     #
# software display the following acknowledgement:                         #
# ‘‘This product includes software developed by Guenther Stiege et al.    #
# and Sergej Alekseev, Universitaet Oldenburg, Germany.’’ Neither the     #
# name of the University nor the names of its authors may be used  to     #
# endorse or promote products  derived  from  this  software  without     #
# specific prior written permission.                                      #
# (4) changes are  permissible  only if the changed files are given a     #
# new name,  different from the names of  existing  files of GHS, and     #
# only if the changed files are clearly  identified as not being part     #
# of GHS.                                                                 #
#                                                                         #
# THE AUTHORS HAVE TRIED  THEIR  BEST TO PRODUCE A CORRECT AND USEFUL     #
# PROGRAM, IN ORDER TO HELP  PROMOTE  COMPUTER  SCIENCE RESEARCH, BUT     #
# THIS  SOFTWARE  IS  PROVIDED ‘‘AS IS’’ AND  WITHOUT ANY  EXPRESS OR     #
# IMPLIED  WARRANTIES,  INCLUDING,  WITHOUT  LIMITATION, THE  IMPLIED     #
# WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.     #
# * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * #
```

Table 1.1: *Distribution policy of GHS*

GHS comes as a package consisting of text files (program sources and others) and documentation. From this package all what is needed can be installed to run GHS under a Unix system or a Windows system[2]. In a Unix system the package is decompressed using `gzip` and installed in a directory using `tar`. GHScore uses the following directories:

- `GHScore` (required).
  Contains the source code of the GHScore C functions. In addition, it contains a `Makefile` and the header file `GHSstructure.h`. When the command `make` is executed using `Makefile` a library is created and stored in the directory `GHScorelib`. `GHScorelib` is a directory with the same father directory as `GHScore`. The name of the library is `libGHS.so`.

- `GHSgraphs` (optional).
  In this directory a number of graphs, including all graphs which are used as examples in this manual, are collected. The most important is the graph `GHSwords`, which is the GHS version of Knuth's `words.dat` [Knut1993].

- `GHStests` (optional).
  Finally, the directory `GHStests` is used to store all example programs of this manual. To execute the programs, each comes with a small command file having the same name and suffix `cmd`.

After installing the directories corresponding to GHScore we have a directory structure as shown in figure 1.1.

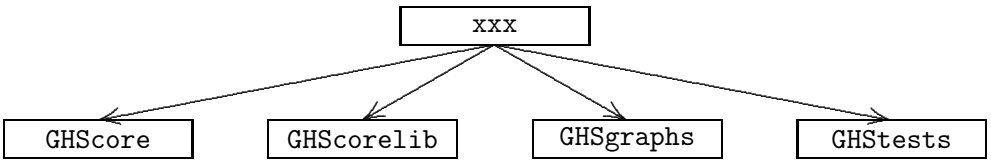


Figure 1.1: *GHScore Directory Structure*

Before GHScore can be used some Unix variables have to be set:

```
export ghscore=/home/..../xxx/GHScore
export ghsgraphs=/home/..../xxx/GHSgraphs
export ghstests=/home/..../xxxx/GHStests
export ghscorelib=/home/..../xxx/GHScorelib
export LD_LIBRARY_PATH=/home/..../xxx/GHScorelib
```

There are two ways to use GHS.
The first way is the "programmer's way". It normally consists of the following four steps:

1. Prepare the input file(s).

2. Program a main program in C using the appropriate functions from the GHS library.

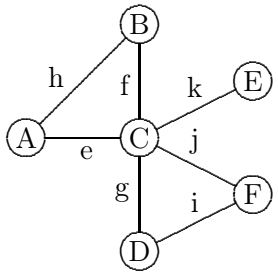
3. Compile the program using the library in `GHScorelib`.

[2]Using GHS under Windows is not described in this manual.

4. Run the program and evaluate the results.

The second way is starting the Graphical User Interface for GHS (GHSgui). GHSgui is not described in this manual.

The programmer's way is best explained by an example. Assume that we want to read the undirected graph *Graph0* of figure 1.2, print it as a raw graph, and than print its vertices in




```
$GRAPH Graph0
$TYPE  UGSLF
$VERTICES B  A  C  E  D  F
$EDGES
_h   A  B
_k   C  E
_f   B  C
_e   A  C
_g   C  D
_j   C  F
_i   D  F
$END
```

Figure 1.2: *Graph0*

breadth-first sequence starting at vertex D.

Step 1: The graph must be available in a computer readable format in a file. GHS assumes files in ASCII code with a simple structure. At the right side of figure 1.2 a file representation of the Graph *Graph0* is shown. It consists of key words which start with a $ sign, the name and the type of the graph, the list of vertices, and the list of lines, divided in list of edges and list of arcs. For each line the line name (starting with _) and both its end points are listed. A detailed description of the GHS graph file format is given in section 2.2, page 11. In the case of our example the graph file can easily be written manually. In the case of larger and more complex graphs the files may come from GHS programs, other graph handling systems or other external sources.

Step 2: Table 1.2 shows a program in C, called `int01.c` (see also directory `GHStests`), which solves the problem of our example. In it the variable `graph` of type `GRAPH` is declared. Then the function `readgraphlist` is called. It reads the file representation of the graph from standard input, builds the internal data structure of the graph and returns a pointer to the new graph record. If the pointer is `NULL`, an error has occurred.

The function `savegraphlist` is called next. It writes the graph to standard output in the external graph format (file format) so that it can be read by `readgraphlist`.

After printing an intermediate heading line, the function `printbfs` is called which prints the vertices of the graph in breadth-first sequence starting at vertex D. It also prints those edges which are used in the search and it prints the distance to the starting vertex. The results of the program run are shown in table 1.3.

A detailed description of the functions `readgraphlist`, `savegraphlist`, and `printbfs` is given in chapter 2 "Basic Graph Functions".

Step 3: The program `int01.c` is compiled with the command

```
/*************************************************************/
/*     Program  main.                                        */
/*                                                           */
/*     Read a graph, print it and print it in bfs sequence.  */
/*                                                           */
/*************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

int  main()
{
  GRAPH      *graph;

  graph = readgraphlist(NULL);
  if (graph ==  NULL)
     { printf("Incorrect graph input\n");
       exit(0);
     };

  savegraphlist(graph, NULL);

  printf("\nBreadth-first search starting at 'D\':\n");
  printbfs(graph, "D", "forward", NULL);
  return 0;
}
```

Table 1.2: *Program* `int01.c`

gcc -I $ghscore -L$ghscorelib int01.c -lGHS -lm -o ghs

After compiling the program we get the executable code in file `ghs`.

Step 4: Finally the program is run, for instance with the Unix command

`cat $ghsgraphs/graph0 | ghs`

reading the graph description data from file `graph0` in the directory `GHSgraphs` and writing the results (table 1.3) to standard output.

See also the command `int01.cmd` in the directory `GHStests` and start this command to see steps 3 and 4 being executed.

```
$GRAPH
Graph0
$TYPE
UGSLF
$VERTICES
A
B
C
D
E
F
$EDGES
_e
  A
  C
_f
  B
  C
_g
  C
  D
_h
  A
  B
_i
  D
  F
_j
  C
  F
_k
  C
  E
$END

Breadth-first search starting at ‘D’:
    0   D
           _i
    1   F
           _g
    1   C
           _e
    2   A
           _k
    2   E
           _f
    2   B
```

Part I

GHS Functions

Chapter 2

Basic Graph Functions

2.1 Problem Description

In this chapter the functions which are basic for all graph algorithms are described. First of all, there is the function `readgraphlist` which constructs the internal incidence structures of a sequence of graphs from their external representations read in from a file or standard input. There is also the function `savegraphlist` which transforms the internal representations of a list of graphs into their external representations and outputs these to a file or standard output. Finally, the function `releasegraphlist` releases a list of graphs.

A second group of functions – `printgrlist, printdfs, printbfs` – serve to print different aspects of a graph: Some statistics, the incidence lists of vertices, list of vertices in depth-first or breadth-first sequence.

The functions `addvertex` and `addline` serve to enlarge an existing graph. Functions `deletevertex` and `deleteline` can be used for diminishing an existing graph. ***They have to be used with special care*** since adding (or deleting) a vertex or a line may change drastically graph properties and hence make inavlid secondary structures built upon those.

Finally, there are functions `dfsstructure` and `releasedfs` which add a depth-first forest structure to the graph, respectively release it.

2.2 Formats and Data Structures

External file format: Files which represent graphs in external GHS format are organized as shown in table 2.1. A graph file is a character file where relevant tokens are defined according to the C function `scanf`. A formal syntax description in Backus-Naur-Form is given in the appendix, page 153.

`$GRAPH` <*graphname*>	Each graph must have a name, which is a string not starting with '$' or '_'.
`$TYPE` <*graphtype*>	Each graph must have a type which is one of the values: `GG, DG, DGS, DGSLF, UG, UGS, UGSLF`. `GG` *(general graph)*: Mixtures of edges and arcs are allowed, multiple edges and/or arcs are allowed, loops (including multiple loops) are allowed. `UG` *(undirected graph)*: Only (undirected) edges are allowed. Multiple edges (including multiple loops) may occur. `UGS` *(undirected graph, single edges)*. Only simple edges are allowed. Simple loops may occur. `UGSLF` *(undirected graph, single edges, loop-free)*: Only simple edges are allowed. Loops are forbidden. `DG, DGS, DGSLF`: Directed versions of `UG, UGS, UGSLF`. Only arcs (i.e. directed edges) are allowed.
`$VERTICES` <*list of vertices*>	Vertices are specified by their names. Vertex names are strings whose first character must not be '$' or '_'. Vertex names must be unique within a graph. A graph must have at least one vertex.
`$EDGES` <*list of edges*>	Edges are specified by their names. Edge names are strings whose first character must be '_'. Each edge name must be followed by two vertex names, the names of its end points. Edge names must be unique within a graph. They must also be different from all arc names of the graph. The number of edges of a graph may be 0.
`$ARCS` <*list of arcs*>	Arcs are specified by their names. Arc names are strings whose first character must be '_'. Each arc name must be followed by two vertex names, the first of which specifies the start point of the arc (head) whereas the second specifies the end point (tail). Arc names must be unique within a graph. They must also be different from all edge names of the graph. The number of arcs of a graph may be 0.
`$END`	

Table 2.1: *External File Format of GHS Graphs*

Internal data structures: The internal data structures used by GHS functions are specified in the file `GHSstructure.h` which must be included in each GHS program. Chapter C, page 155, in the appendix contains a complete listing.

`GHSstructure.h` is divided into subsections. In the first subsection some constants and all common types are defined. The second subsection contains C function definitions and the definition of global variables. The remaining subsections describe data structures specific to a group of GHS functions. In general, these data structures are defined as records using the C statement `struct`. For easier programming, some enumeration types have been introduced, too.

Most of the record types (also called data types) are suitable to be handled in a uniform manner as elements of red-black trees (see chapter 13, page 117). All fields of a record type have a fixed mode and a fixed semantic meaning. Occasionally, a different mode or a different meaning is used with a field. If this is the case, it is mentioned explicitly in the corresponding function description (under remarks). Most record types have an auxiliary field of mode `(void *)` which is explicitly reserved for variable applications and future enhancements. The use of these fields in a function is also described under remarks.

In subsection C.5, page 166, the data structures pertaining to the basic graph functions of this chapter are defined. To represent and manipulate graphs, vertices and edges/arcs the record types `GRAPH, VERTEX`, `EDGE`, and `INC` (for incidence records) have been introduced yielding together a complete incidence list structure. To represent vertices and edges in different additional roles, the types `RVERTEX` and `REDGE` are used. The enumeration type `GRAPHTYPE` is also used with the basic functions. *The basic data types are required for all other function classes, too.*

2.3 Functions

2.3.1 readgraphlist

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `GRAPH *readgraphlist(char *filename)`

Description: Reads a sequence of external graph descriptions from file `filename`. If `filename` equals NULL, the graph descriptions are read from standard input. For each description an incidence structure is built: GRAPH record, VERTEX records, EDGE records, and INC records. The records of each class are linked by a red-black tree structure, ordered by vertex name (VERTEX) or edge name (EDGE, INC). The individual graph structures are linked in a list (red-black tree) sorted by graph name. Returns a pointer to list.

Error Exits: Return value is NULL if an error has occurred.

Remarks: None.

Examples: Section 1.2, page 3. Example 2.1, page 16.

2.3.2 savegraphlist

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void savegraphlist(GRAPH *graphlist, char *filename)`

Description: Writes a list of graphs organized as a red-black tree to file `filename`. If `filename` equals NULL the output is written to standard output. For each graph of the list its incidence structure is output in GHS external graph format so that it can be read again by `readgraphlist`.

Error Exits: List pointer NULL. Graph has no vertices.

Remarks: None.

Examples: Section 1.2, especially table 1.3, page 7.

2.3.3 releasegraphlist

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void releasegraphlist(GRAPH *graphlist)`

Description: `graphlist` points to the root of the tree. Releases a list of graphs (organized as red-black tree) with all its vertices and edges freeing completely the memory. Additional decomposition structures of the graph, for instance the biblock decomposition (see chapter 6, page 63) are also released without any warning.

Error Exits: Graph pointer NULL (Warning).

Remarks: Releasing graphs must be done with care. During processing a series of dependent structures, like paths or Menger structures may have been created. These point to the graph they have been derived from, but there is no pointer from the graph to them. So, they cannot be deleted automatically together with the graph but must be released explicitly by the user program using the pertinent functions.

Examples: Example 2.3.

2.3.4 printgrlist

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void printgrlist(GRAPH *graph, char *option, char *filename)`
Option must be one of the values `"short"`, `"normal"` or `"detailed"`.

Description: Writes a list of graphs (organized as red-black tree) to file `filename`. If `filename` equals NULL the output is written to standard output. For each graph the name, the type, and the numbers of vertices, edges and arcs are written. With options `"normal"` or `"detailed"` a degree summary is also written. Option `"detailed"` causes the additional output of a list of vertices with their incident edges/arcs.

Error Exits: None.

Remarks: The field `edauxiliary` in record type `EDGE` is used for detection of loops and multiple edges/arcs.

Examples: Example 2.1, page 16.

2.3.5 printdfs

Program Author: Günther Stiege, Universität Oldenburg

Syntax:
```
void    printdfs(GRAPH *graph, char *vname, char *option, char
        *filename)
```
Option must be one of the values `"forward"`, `"backward"` or `"any"`.

Description: Writes to file `filename` in depth-first sequence the vertices of graph `graph` which are reachable from vertex `vname`. If `filename` equals NULL the output is written to standard output. Edges, if present, are followed in any direction. Arcs, if present, are followed in the direction specified by `option`. The edges/arcs followed are also printed as well as the depth level reached.

Error Exits: Error message if vertex `vname` is not vertex of graph `graph`.

Remarks: The field `vtauxiliary` in record type `VERTEX` is used for the depth level reached.

Examples: Example 2.2, page 16.

2.3.6 printbfs

Program Author: Günther Stiege, Universität Oldenburg

Syntax:
```
void    printbfs(GRAPH *graph, char *vname, char *option, char
        *filename)
```
Option must be one of the values `"forward"`, `"backward"` or `"any"`.

Description: Writes to file `filename` in breadth-first sequence the vertices of graph `graph` which are reachable from vertex `vname`. If `filename` equals NULL the output is written to standard output.
Edges, if present, are followed in any direction. Arcs, if present, are followed in the direction specified by `option`. The edges/arcs followed are also printed as well as the distance from the actual vertex to the start vertex.

Error Exits: Error message if vertex `vname` is not vertex of `graph`.

Remarks: The field `vtauxiliary` in record type `VERTEX` is used as a pointer to a `RVERTEX` record, which in turn is used for queuing.

Examples: Example 2.2, page 16.

2.3.7 addvertex (not yet released)

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void addvertex(GRAPH *graph, char *vtname)`

Description: A new vertex with name `vtname` is added to graph `graph`.
The graph must be of type `GG` (general graph).

Error exits:

Examples:

2.3.8 addline (not yet implemented)

2.3.9 deletevertex (not yet implemented)

2.3.10 deleteline (not yet implemented)

2.4 Examples

Example 2.1 In this example graphs Graph0 (figure 1.2) and Graph1 (figure 2.1) are considered. Graph1 is a general graph showing (undirected) edges, arcs, multiple edges/arcs, and directed as well as undirected loops. Its external file description is shown in table 2.2. In addition, two modifications of Graph1 are also considered. Graph1a results from Graph1 by making all arcs undirected edges whereas Graph1b results from transforming all edges into arcs.

Program `bas01.c` (table 2.3) shows a main program which reads the 4 graphs Graph0, Graph1, Graph1a, and Graph1b building the internal graph structures and linking the graph structures in a list. This is done with the function `readgraphlist`. The graphs are then printed calling function `printgrlist` with option `"short"`. Afterwards it is called again, this time with option `"detailed"`. The results are shown in tables 2.4, 2.5, 2.6, and 2.7. □

Example 2.2 In this example Graph1 (figure 2.1 and table 2.2) is constructed using `readgraphlist` and then all vertices reachable from vertex `K00` are written in depth-first and breadth-first sequence using the options “forward”, “any”, and “backward” to specify the direction in which arcs are allowed to be followed. Table 2.8 shows the main program used. Table 2.9 contains the results. □

Example 2.3 Program `bas03.c` in `GHStests` – see also tables 2.10, 2.11, and 2.12 – is a program which reads five graphs and builds a list of them. 3 of the graphs are copied (without type changes) and then the list and the 3 individual graphs are released. After each graph creation and after each graph releasing the actual amount of dynamic memory used is printed using variable `ghsmemsize`. The results are shown in table 2.13.

Note: Due to changes in the data structures of GHScore, the memory values given by `bas03.cmd` may differ somewhat from those in table 2.13.

□

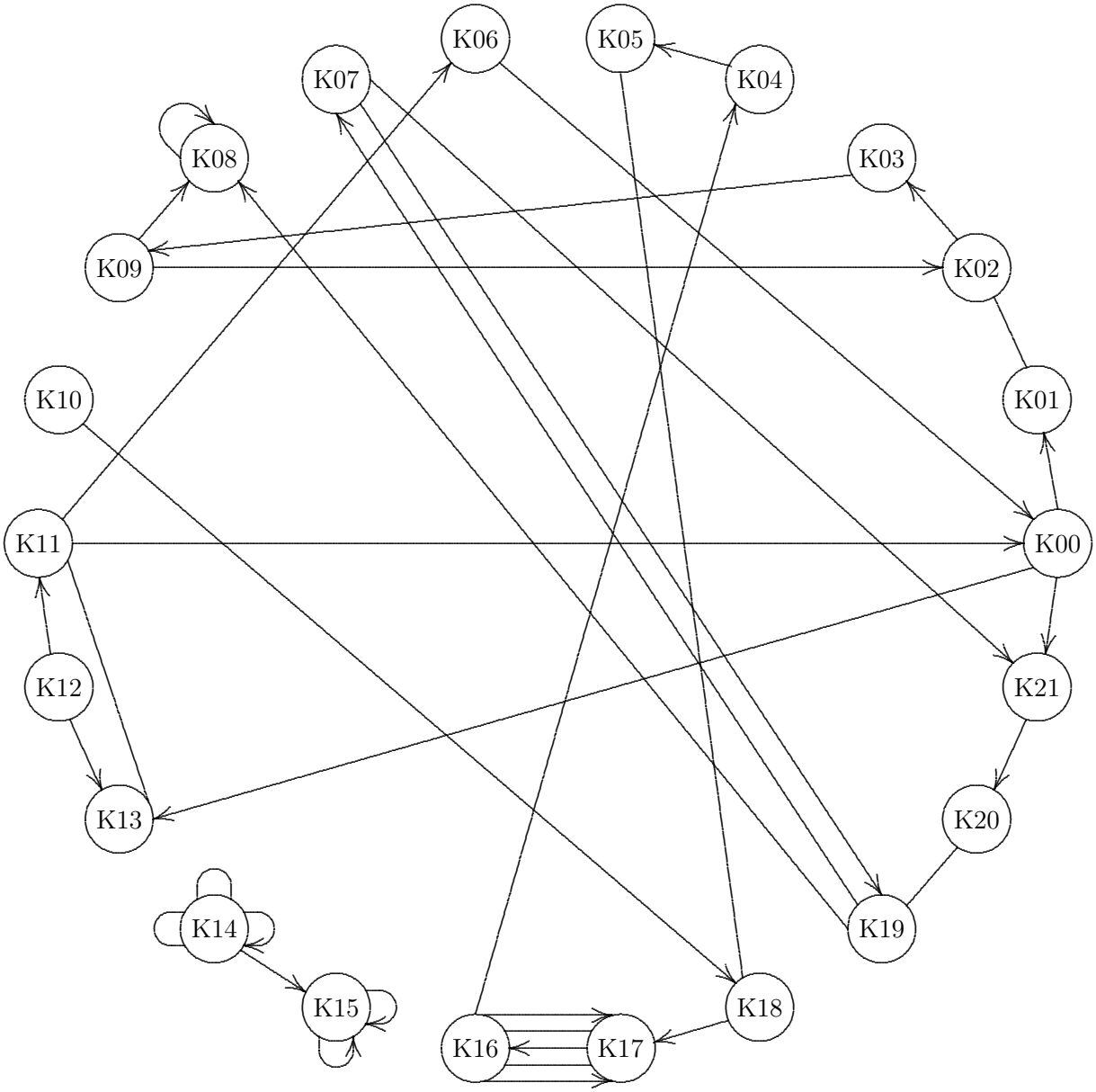


Figure 2.1: *Graph1*

```
$GRAPH Graph1
$TYPE  GG
$VERTICES K00  K01  K02  K03  K04  K05  K06  K07  K08  K09  K10
          K11  K12  K13  K14  K15  K16  K17  K18  K19  K20  K21

$EDGES
_u*K01*K02   K01  K02
_u*K05*K18   K05  K18
_u*K11*K13   K11  K13
_u*K14*K14a  K14  K14     _u*K14*K14b  K14  K14
_u*K16*K17a  K16  K17     _u*K16*K17b  K16  K17
_u*K19*K20   K19  K20

$ARCS
_d*K00*K01   K00  K01     _d*K00*K13   K00  K13   _d*K00*K21   K00  K21
_d*K02*K03   K02  K03
_d*K03*K09   K03  K09
_d*K04*K05   K04  K05
_d*K06*K00   K06  K00
_d*K07*K19   K07  K19     _d*K07*K21   K07  K21
_d*K08*K08   K08  K08
_d*K09*K02   K09  K02     _d*K09*K08   K09  K08
_d*K10*K18   K10  K18
_d*K11*K00   K11  K00     _d*K11*K06   K11  K06
_d*K12*K11   K12  K11     _d*K12*K13   K12  K13
_d*K14*K14   K14  K14     _d*K14*K15   K14  K15
_d*K15*K15a  K15  K15     _d*K15*K15b  K15  K15
_d*K16*K04   K16  K04     _d*K16*K17a  K16  K17     _d*K16*K17b  K16  K17
_d*K17*K16   K17  K16
_d*K18*K17   K18  K17
_d*K19*K07   K19  K07     _d*K19*K08   K19  K08
_d*K21*K20   K21  K20
$END
```

Table 2.2: *External File Description of Graph1*

```
/*************************************************************/
/*     Program  main.                                        */
/*                                                           */
/*     Reads 4 graphs in sequence building a graph list.     */
/*     Prints the list in formats "short" and "detailed".    */
/*                                                           */
/*************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

int main()
{
  GRAPH      *graphlist;

  graphlist = NULL;

  graphlist = readgraph(NULL);
  if (graphlist ==  NULL)
    { printf("Incorrect graph input\n");
      exit(0);
    }
  printgrlist(graphlist, "short", NULL);
  printgrlist(graphlist, "detailed", NULL);
  return 0;
}
```

Table 2.3: *Program* `bas01.c`

Results of calling `printgrlist` with option `"short"`

```
  Graph0        Type = UGSLF  #Vertices =     6  #Edges =     7  #Arcs =     0
  Graph1        Type = GG     #Vertices =    22  #Edges =     8  #Arcs =    29
  Graph1a       Type = UG     #Vertices =    22  #Edges =    37  #Arcs =     0
  Graph1b       Type = DG     #Vertices =    22  #Edges =     0  #Arcs =    37
```

Results of calling `printgrlist` with option `"detailed"`
(only Graph1)

```
Statistics of graph Graph1    Type = GG
  #Vertices =    22
  #Edges    =     8  #Mult.edges =   4  #Loops(u) =   2  #Mult.loops(u) =   2
  #Arcs     =    29  #Mult.arcs  =   4  #Loops(d) =   4  #Mult.loops(d) =   2

Degree summary (undirected loops are counted twice!):
      Mean total degree                   3.36
      Standard deviation of total degree 1.55
      1 Vertices of total degree     1
      7 Vertices of total degree     2
      6 Vertices of total degree     3
      3 Vertices of total degree     4
      2 Vertices of total degree     5
      2 Vertices of total degree     6
      1 Vertices of total degree     7

Vertex list:
    K10                  degree =   0  indegree =   0  outdegree =   1
      Outgoing arcs
          _d*K10*K18
    K01                  degree =   1  indegree =   1  outdegree =   0
      Undirected edges
          _u*K01*K02
      Incoming arcs
          _d*K00*K01
    K03                  degree =   0  indegree =   1  outdegree =   1
      Outgoing arcs
          _d*K03*K09
      Incoming arcs
          _d*K02*K03
    K04                  degree =   0  indegree =   1  outdegree =   1
      Outgoing arcs
          _d*K04*K05
      Incoming arcs
          _d*K16*K04
```

Table 2.4: *Results of Program* `bas01.c` *(a)*

```
K05                 degree =   1  indegree =   1  outdegree =   0
  Undirected edges
      _u*K05*K18
  Incoming arcs
      _d*K04*K05
K06                 degree =   0  indegree =   1  outdegree =   1
  Outgoing arcs
      _d*K06*K00
  Incoming arcs
      _d*K11*K06
K12                 degree =   0  indegree =   0  outdegree =   2
  Outgoing arcs
      _d*K12*K11
      _d*K12*K13
K20                 degree =   1  indegree =   1  outdegree =   0
  Undirected edges
      _u*K19*K20
  Incoming arcs
      _d*K21*K20
K02                 degree =   1  indegree =   1  outdegree =   1
  Undirected edges
      _u*K01*K02
  Outgoing arcs
      _d*K02*K03
  Incoming arcs
      _d*K09*K02
K07                 degree =   0  indegree =   1  outdegree =   2
  Outgoing arcs
      _d*K07*K19
      _d*K07*K21
  Incoming arcs
      _d*K19*K07
K09                 degree =   0  indegree =   1  outdegree =   2
  Outgoing arcs
      _d*K09*K02
      _d*K09*K08
  Incoming arcs
      _d*K03*K09
```

Table 2.5: *Results of Program* `bas01.c` *(b)*

```
K13                 degree =   1  indegree =   2  outdegree =   0
  Undirected edges
      _u*K11*K13
  Incoming arcs
      _d*K00*K13
      _d*K12*K13
K18                 degree =   1  indegree =   1  outdegree =   1
  Undirected edges
      _u*K05*K18
  Outgoing arcs
      _d*K18*K17
  Incoming arcs
      _d*K10*K18
K21                 degree =   0  indegree =   2  outdegree =   1
  Outgoing arcs
      _d*K21*K20
  Incoming arcs
      _d*K00*K21
      _d*K07*K21
K08                 degree =   0  indegree =   3  outdegree =   1
  Outgoing arcs
      _d*K08*K08 (loop)
  Incoming arcs
      _d*K08*K08 (loop)
      _d*K09*K08
      _d*K19*K08
K11                 degree =   1  indegree =   1  outdegree =   2
  Undirected edges
      _u*K11*K13
  Outgoing arcs
      _d*K11*K00
      _d*K11*K06
  Incoming arcs
      _d*K12*K11
K19                 degree =   1  indegree =   1  outdegree =   2
  Undirected edges
      _u*K19*K20
  Outgoing arcs
      _d*K19*K07
      _d*K19*K08
  Incoming arcs
      _d*K07*K19
```

Table 2.6: *Results of Program* `bas01.c` *(c)*

```
K00                 degree =   0  indegree =   2  outdegree =   3
  Outgoing arcs
      _d*K00*K01
      _d*K00*K13
      _d*K00*K21
  Incoming arcs
      _d*K06*K00
      _d*K11*K00
K15                 degree =   0  indegree =   3  outdegree =   2
  Outgoing arcs
      _d*K15*K15a (multiple loop)
      _d*K15*K15b (multiple loop)
  Incoming arcs
      _d*K14*K15
      _d*K15*K15a (multiple loop)
      _d*K15*K15b (multiple loop)
K16                 degree =   2  indegree =   1  outdegree =   3
  Undirected edges
      _u*K16*K17a (multiple)
      _u*K16*K17b (multiple)
  Outgoing arcs
      _d*K16*K04
      _d*K16*K17a (multiple)
      _d*K16*K17b (multiple)
  Incoming arcs
      _d*K17*K16
K17                 degree =   2  indegree =   3  outdegree =   1
  Undirected edges
      _u*K16*K17a (multiple)
      _u*K16*K17b (multiple)
  Outgoing arcs
      _d*K17*K16
  Incoming arcs
      _d*K16*K17a (multiple)
      _d*K16*K17b (multiple)
      _d*K18*K17
K14                 degree =   4  indegree =   1  outdegree =   2
  Undirected edges
      _u*K14*K14a (multiple loop)
      _u*K14*K14b (multiple loop)
  Outgoing arcs
      _d*K14*K14 (loop)
      _d*K14*K15
  Incoming arcs
      _d*K14*K14 (loop)
```

Table 2.7: *Results of Program* `bas01.c` *(d)*

```
/*************************************************************/
/*     Program  main.                                        */
/*                                                           */
/*     Reads a graph.                                        */
/*                                                           */
/*     Prints all vertices reachable from K00                */
/*        a. in depth-first sequence, forward direction,     */
/*        b. in depth-first sequence, any direction,         */
/*        c. in breadth-first sequence, backward direction.  */
/*************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

int main()
{
  GRAPH      *graph;

  graph = readgraphlist(NULL);
  if (graph ==  NULL)
    { printf("Incorrect graph input\n");
      exit(0);
    };

  printdfs(graph, "K00", "forward", NULL);
  printdfs(graph, "K00", "any", NULL);
  printbfs(graph, "K00", "backward", NULL);
  return 0;
}
```

Table 2.8: *Program* `bas02.c`

```
0   K00
      _d*K00*K01
1   K01
      _u*K01*K02
2   K02
      _d*K02*K03
3   K03
      _d*K03*K09
4   K09
      _d*K09*K08
5   K08
      _d*K00*K21
1   K21
      _d*K21*K20
2   K20
      _u*K19*K20
3   K19
      _d*K19*K07
4   K07
      _d*K00*K13
1   K13
      _u*K11*K13
2   K11
      _d*K11*K06
3   K06
```

Part A.
`printdfs(graph, "K00", "forward", NULL)`

```
0   K00
      _d*K00*K01
1   K01
      _u*K01*K02
2   K02
      _d*K02*K03
3   K03
      _d*K03*K09
4   K09
      _d*K09*K08
5   K08
      _d*K19*K08
6   K19
      _u*K19*K20
7   K20
      _d*K21*K20
8   K21
      _d*K07*K21
9   K07
      _d*K00*K13
1   K13
      _u*K11*K13
2   K11
      _d*K11*K06
3   K06
      _d*K12*K11
3   K12
```

Part B.
`printdfs(graph, "K00", "any", NULL)`

```
0   K00
      _d*K11*K00
1   K11
      _d*K06*K00
1   K06
      _u*K11*K13
2   K13
      _d*K12*K11
2   K12
```

Part C.
`printbfs(graph, "K00", "backward", NULL)`

Table 2.9: *Results of Program* `bas02.c`

```
/*************************************************************/
/*     Program  main.                                        */
/*                                                           */
/*     Reads 5 graphs in sequence building a graph list.     */
/*     Then copies 3 of them without changing the graph      */
/*     type. Finally, the list and the 3 copies are          */
/*     released. The amount of memory used is printed after  */
/*     each graph creation and each graph releasing.         */
/*                                                           */
/*************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

int   main()
{ char       *buffer;
  GRAPH      *graph, *graphlist, *graphwords, *graph1b, *graph0;
  int         i, bl;
  graphlist = NULL;

  buffer = getenv("ghsgraphs");
  bl = strlen(buffer);
  strcpy(buffer+bl, "/Graph0");
  printf("\n\n%8ld  ", (int)ghsmemsize);
  printf("Memory before reading the graphs\n");
  graph = readgraphlist(buffer);
  if (graph ==  NULL)
    { printf("Incorrect graph input\n");
      exit(0);
    };
  rbtreeinsert((RB **)&graphlist, (RB *)graph, GRNM);
  printf("%8ld  ", (int)ghsmemsize);
  printf("Memory after reading graph %s\n",graph->grname);

  strcpy(buffer+bl, "/Graph1");
  graph = readgraphlist(buffer);
  if (graph ==  NULL)
    { printf("Incorrect graph input\n");
      exit(0);
    };
  rbtreeinsert((RB **)&graphlist, (RB *)graph, GRNM);
  printf("%8ld  ", (int)ghsmemsize);
  printf("Memory after reading graph %s\n",graph->grname);
```

Table 2.10: *Program* `bas03.c` *(Part I)*

```
 strcpy(buffer+bl, "/Graph1a");
 graph = readgraphlist(buffer);
 if (graph ==  NULL)
   { printf("Incorrect graph input\n");
     exit(0);
   };
 rbtreeinsert((RB **)&graphlist, (RB *)graph, GRNM);
 printf("%8ld  ", (int)ghsmemsize);
 printf("Memory after reading graph %s\n",graph->grname);

 strcpy(buffer+bl, "/GHSwords");
 graph = readgraphlist(buffer);
 if (graph ==  NULL)
   { printf("Incorrect graph input\n");
     exit(0);
   };
 rbtreeinsert((RB **)&graphlist, (RB *)graph, GRNM);
 printf("%8ld  ", (int)ghsmemsize);
 printf("Memory after reading graph %s\n",graph->grname);

 strcpy(buffer+bl, "/Graph1b");
 graph = readgraphlist(buffer);
 if (graph ==  NULL)
   { printf("Incorrect graph input\n");
     exit(0);
   };
 rbtreeinsert((RB **)&graphlist, (RB *)graph, GRNM);
 printf("%8ld  ", (int)ghsmemsize);
 printf("Memory after reading graph %s\n",graph->grname);

graph = (GRAPH *)rbtreefind((RB *)graphlist, (RB *) "Graph0", GRNC);
graph0 = cpchtp(graph, "UGSLF", "Newgraph0");
printf("%8ld  ", (int)ghsmemsize);
printf("Memory after copying graph Graph0 to Newgraph0\n");
graph = (GRAPH *)rbtreefind((RB *)graphlist, (RB *) "Graph1b", GRNC);
graph1b = cpchtp(graph, "DG", "Newgraph1b");
printf("%8ld  ", (int)ghsmemsize);
printf("Memory after copying graph Graph1b to Newgraph1b\n");
graph = (GRAPH *)rbtreefind((RB *)graphlist, (RB *) "GHSwords", GRNC);
graphwords = cpchtp(graph, "UGSLF", "Newgraphwords");
printf("%8ld  ", (int)ghsmemsize);
printf("Memory after copying graph GHSwords to Newgraphwords\n");
```

Table 2.11: *Program* `bas03.c` *(Part II)*

```
 releasegraphlist(graph0);
 printf("%8ld  ", (int)ghsmemsize);
 printf("Memory after releasing Newgraph0\n");
 releasegraphlist(graphlist);
 printf("%8ld  ", (int)ghsmemsize);
 printf("Memory after releasing list of graphs\n");
 releasegraphlist(graphwords);
 printf("%8ld  ", (int)ghsmemsize);
 printf("Memory after releasing Newgraphwords\n");
 releasegraphlist(graph1b);
 printf("%8ld  ", (int)ghsmemsize);
 printf("Memory after releasing Newgraph1b\n");
 return 0;
}
```

Table 2.12: *Program* `bas03.c` *(Part III)*

```
       0  Memory before reading the graphs
    1052  Memory after reading graph Graph0
    6014  Memory after reading graph Graph1
   10897  Memory after reading graph Graph1a
 1762027  Memory after reading graph GHSwords
 1767030  Memory after reading graph Graph1b
 1768085  Memory after copying graph Graph0 to Newgraph0
 1773091  Memory after copying graph Graph1b to Newgraph1b
 3524226  Memory after copying graph GHSwords to Newgraphwords
 3523171  Memory after releasing Newgraph0
 1756141  Memory after releasing list of graphs
    5006  Memory after releasing Newgraphwords
       0  Memory after releasing Newgraph1b
```

Table 2.13: *Results of Program* `bas03.c`

Chapter 3

Sets

3.1 Problem Description

GHS offers a number of set handling functions. These are necessary mainly to generate subgraphs from sets of vertices or sets of lines. In addition to sets of vertices and set of lines, a third data type, namely general sets, whose elements a mere character strings, are provided. All three kinds of sets can be read in from a file, saved to a file, and released. It is also possible to add a single element to them. A set of vertices (lines) can be merged into another set of vertices (lines). Pure set operations (union, intersection, difference) are allowed with general sets only and yield a new general set as result. Finally, there are functions, which transform a general set into a set of vertices (lines).

3.2 Formats and Data Structures

For a general description and basic data types see section 2.2, page 11.
The data structures used with the set handling functions are described in subsection C.6, page 171. The data types `VTSET`, `EDSET`, and `GSET` respectively respresent sets of vertices, sets of lines, and general sets. Elements of general sets are represented by records of type `SETELEM`. In contrast, elements of vertex sets (line sets) are represented by records of type `RVERTEX (REDGE)`, see chapter 2.

3.3 Functions for Vertex Sets

3.3.1 readvtset

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `VTSET *readvtset(GRAPH *graph, char *filename)`

Description: Reads a sequence of names from file `filename`. If `filename` equals NULL the names are read from standard input. The end is indicated by EOF or $END. Builds the corresponding set of vertices of the graph. Returns a pointer to the set. NULL if the set is empty or if an error has occurred.

Error Exits: Graph pointer NULL, vertex set empty, vertex name not from a vertex of the graph.

Remarks: None.

Examples: Example 4.1, page 48.

3.3.2 gset2vtset

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `VTSET *gset2vtset(GSET *gset, GRAPH *graph)`

Description: Constructs a vertex set from a general set. Returns a pointer to the set. NULL if an error has occurred.

Error Exits: Graph pointer NULL, pointer to general set NULL, when calling `add2vtset`: vertex with given name not in graph.

Remarks: None.

Examples: Example 3.1. See also program `gen03.c` in GHS directory `GHStests`.

3.3.3 savevtset

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void savevtset(VTSET *vtset, char *filename)`

Description: Writes the vertex names of a vertex set to file `filename`. If `filename` equals NULL the output is written to standard output. The output format is such that it can be read by `readvtset`.

Error Exits: Pointer NULL, vertex set empty, number of elements incorrect.

Remarks: None.

Examples: Example 4.1, page 48.

3.3.4 releasevtsetlist

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void releasevtsetlist(VTSET *vtset)`

Description: Deletes a list of vertex sets and frees completely the memory.

Error Exits: None.

Remarks: None.

Examples: Example 3.1.

3.3.5 add2vtset

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void add2vtset(GRAPH *graph, VTSET **pvtset, char *name)`

Description: Adds a new vertex given by its name to a vertex set. Creates new vertex set for first vertex.

Error Exits: Vertex with given name not in graph, duplicte vertex name to be added.

Remarks: None.

Examples: Example 4.1, page 48.

3.3.6 addvtset2vtset

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void addvtset2vtset(GRAPH *graph, VTSET **pvtset, RVERTEX *rvt)`

Description: Joins two vertex sets of the same graph.

Error Exits: When calling `add2vtset`: Vertex with given name not in graph.

Remarks: None.

Examples: Example 4.1, page 48.

3.4 Functions for Edge Sets

3.4.1 readedset

Program Authors: Dirk Ahlers and Tobe Toben, Universität Oldenburg

Syntax: `EDSET *readedset(GRAPH *graph, char *filname)`

Description: Reads a sequence of names from file `filename`. If `filename` equals NULL the input is read from standard input. The end is indicated by EOF or $END. Builds the corresponding set of lines of the graph. Returns a pointer to the set. NULL if the set is empty or if an error has occurred.

Error Exits: Graph pointer NULL, vertex set empty, vertex name not from a vertex of the graph.

Remarks: None.

Examples: See example `gen02.c` in GHS directory `GHStests`. The example is analogous to example 4.1.

3.4.2 saveedset

Program Authors: Dirk Ahlers and Tobe Toben, Universität Oldenburg

Syntax: `void saveedset(EDSET *edset, char *filename)`

Description: Writes the line names of a line set to file `filename`. If `filename` equals NULL the output is written to standard output. The output format is such that it can be read by `readedset`.

Error Exits: Pointer NULL, vertex set empty, number of elements incorrect.

Remarks: None.

Examples: See example `gen02.c` in GHS directory `GHStests`. The example is analogous to example 4.1.

3.4.3 gset2edset

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `EDSET *gset2edset(GSET *gset, GRAPH *graph)`

Description: Constructs an line set from a general set. Returns a pointer to the set. NULL if an error has occurred.

Error Exits: Graph pointer NULL, pointer to general set NULL, when calling `add2edset`: line with given name not in graph.

Remarks: None.

Examples: Example 3.1.

3.4.4 releaseedsetlist

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void releaseedsetlist(VTSET *vtset)`

Description: Deletes a list of line sets and frees completely the memory.

Error Exits: None.

Remarks: None.

Examples: Example 3.1.

3.4.5 add2edset

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void add2edset(GRAPH *graph, EDSET **pedset, char *name)`

Description: Adds a new line given by its name to an line set. Creates new line set for first line.

Error Exits: Edge/arc with given name not in graph, duplicate name to be included.

Remarks: None.

Examples: See example `gen02.c` in GHS directory `GHStests`. The example is analogous to example 4.1.

3.4.6 addedset2edset

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void addedset2edset(GRAPH *graph, EDSET **pedset, REDGE *red)`

Description: Joins two line sets of the same graph.

Error Exits: When calling `add2edset`: Edge/arc with given name not in graph.

Remarks: None.

Examples: See example `gen02.c` in GHS directory `GHStests`. The example is analogous to example 4.1.

3.5 Functions for General Sets

3.5.1 readgset

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `GSET *readgset(char *filename)`

Description: Reads a sequence of names from file `filename` to build a general set. If `filename` equals NULL the names are read from standard input. The end is indicated by EOF or $END. Returns a pointer to the set. NULL if the set is empty.

Error Exits: Set empty.

Remarks: None.

Examples: Example 3.2, page 38.

3.5.2 savegset

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void savegset(GSET *vtset, char *filename)`

Description: Writes the elements of a general set to file `filename`. If `filename` equals NULL the output is written to standard output. The output format is such that it can be read by `readgset`.

Error Exits: Pointer NULL, vertex set empty, number of elements incorrect.

Remarks: None.

Examples: Example 3.2, page 38.

3.5.3 releasegsetlist

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void releasegsetlist(VTSET *vtset)`

Description: Deletes a list of general sets and frees completely the memory.

Error Exits: None.

Remarks: None.

Examples: Example 3.1.

3.5.4 add2gset

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `BOOLEAN add2gset(GSET **pvtset, char *name)`

Description: Returns `FALSE` if the character string `name` is already an element of the general set. Otherwise, the string is added as a new element to the general set and `TRUE` is returned. Creates new general set for first set element. The string must be different from `$END`.

Error Exits: String pointer NULL. String length 0. String equals `$END`.

Remarks: None.

Examples: See example `set03.c` in GHS directory `GHStests`.

3.5.5 gsetunion

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `GSET *gsetunion(RB *lista, int keya, RB *listb, int keyb)`

Description: Creates a new general set as the set union of the two red-black trees (see chapter 13) given by `lista` and `listb`. The elements of the general set are names of the records of the red-black trees. The key classes `keya` and `keyb` must be such that function `getname` is allowed (see subsection C.4.2, page 165, in the appendix).

Error Exits: None.

Remarks: None.

Examples: Example 3.1

3.5.6 gsetintersect

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `GSET *gsetintersect(RB *lista, int keya, RB *listb, int keyb)`

Description: Creates a new general set as the set intersection of the two red-black trees (see chapter 13) given by `lista` and `listb`. The elements of the general set are names of the records of the red-black trees. Key class `keya` must be such that function `getname` is possible with records of `lista`. Key class `keyb` must be such that function `rbtreefind` must be possible in `listb` with a character string as compare value (see subsection C.4.2, page 165, in the appendix).

Error Exits: None.

Remarks: None.

Examples: See example `gen03.c` in GHS directory `GHStests`.

3.5.7 gsetdiff

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `GSET *gsetdiff(RB *lista, int keya, RB *listb, int keyb)`

Description: Creates a new general set as the set difference between the red-black tree (see chapter 13) given by `lista` and the red-black tree given by `listb`. The elements of the general set are names of the records of the red-black trees. Key class `keya` must be such that function `getname` is possible with records of `lista`. Key class `keyb` must be such that function `rbtreefind` must be possible in `listb` with a character string as compare value (see subsection C.4.2, page 165, in the appendix).

Error Exits: None.

Remarks: None.

Examples: See example `set03.c` in GHS directory `GHStests`.

3.6 Examples

Example 3.1 Tables 3.1 and 3.2 show a program which builds a graph from an external description (`readgraphlist`). It then builds a general set joining the graph's vertex list with an empty set (`gsetunion`). From the general set the corrsponding vertex set is derived (`gset2vtset`) and then the general set is deleted (`releasegsetlist`). In the next step, using function `gsetunion` again a new general set is created. It contains all edges and all arcs of the graph. From it the correponding line set is derived (`gset2edset`). Finally all sets are released (`releaseedsetlist, releasegsetlist, releasevtsetlist`). The amount of memory allocated is printed at different points of the program run. □

Example 3.2 Table 3.4 shows in the upper part the command procedure `set02.cmd`. This procedure compiles and starts program `set02a.c` which is shown in the lower part of table 3.4. The program reads a graph and writes the list of vertex names as a general set to standard output (`savegset`) which by the command procedure is stored in the temporary file `XYYX`. The command procedure then compiles and starts program `set02b.c`. Table 3.5 shows the program in the upper part. It reads the graph again and then reads the general set from file `XYYX` (`readsgset`). From the general set the corresponding set of vertices (`gset2vtset`) is constructed which in turn is used to generate the graph again (`generatefromvt`). The names and the number of elements are printed for both, the old and the new graph, which are identical. □

```
/*************************************************************/
/*     Program  main.                                        */
/*                                                           */
/*     Reads a graph, constructs a sequence of sets (general  */
/*     set, vertex set, edge set), and releases the sets.     */
/*     Traces the amount of allocated memory.                */
/*************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

main()
{ GRAPH      *graph, *vtgra;
  VTSET      *vtset;
  EDSET      *edset;
  GSET       *gset, *gset1 = NULL;
  char        mgrname[100];

  graph = readgraph();
    if (graph ==  NULL)
      { printf("Incorrect graph input\n");
        exit(0);
      };
  strcpy(mgraphname, graph->grname);
  printf("Memory after reading graph %s:           %6d Bytes\n",
           mgraphname, ghsmemsize);

  gset = gsetunion((RB *)(graph->grvtlist), SND, (RB *)gset1, SND);
  printf("Memory after reading first general set:      %6d Bytes\n",
           ghsmemsize);
  vtset = gset2vtset(gset, graph);
  // savevtset(vtset);
  printf("Memory after building vertex set:            %6d Bytes\n",
           ghsmemsize);
  releasegsetlist(gset);
  printf("Memory after releasing first general set:    %6d Bytes\n",
           ghsmemsize);
  gset = gsetunion((RB *)(graph->gredlist), SED,
                (RB *)(graph->grdedlist), SED);
  edset = gset2edset(gset, graph);
  // saveedset(edset);
  printf("Memory after building edge set:               %d Bytes\n",
           ghsmemsize);
}
```

Table 3.1: *Program* `set01.c` *(Part I)*

```
  releaseedsetlist(edset);
  releasegsetlist(gset);
  releasevtsetlist(vtset);
  printf("Memory after releasing edge set,general sets, \n");
  printf("and vertex set:                                  %d Bytes\n",
                      ghsmemsize);
  strcpy(mgrname, graph->grname);
  releasegraph(graph);
  printf("Memory after releasing graph %s:         %6d Bytes\n",
           mgraphname, ghsmemsize);
  return 0;
}
```

Table 3.2: *Program* `set01.c` *(Part II)*

```
Memory after reading graph Graph1:             4478 Bytes
Memory after reading first general set:        5030 Bytes
Memory after building vertex set:              5494 Bytes
Memory after releasing first general set:      4942 Bytes
Memory after building edge set:                7033 Bytes
Memory after releasing edge set,general sets,
and vertex set:                                4478 Bytes
Memory after releasing graph Graph1:              0 Bytes
```

Table 3.3: *Results of Program* `set01.c`

```
cd ../GHSsources
cp ../GHStests/set02a.c    ./maingraph.c
make
 cat ../GHSgraphs/Graph1 | ghs > XYYX
#
rm GHSobjects/maingraph.o
rm ghs
cp ../GHStests/set02b.c    ./maingraph.c
make
 (cat ../GHSgraphs/Graph1; cat XYYX) | ghs
rm XYYX


/*************************************************************/
/*     Program  main.                                        */
/*                                                           */
/*     Reads a graph, builds a general set from its vertex   */
/*     names, and writes the general set to standard output.  */
/*************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

int   main()
{ GRAPH      *graph;
  GSET       *gset, *gset1 = NULL;

  graph = readgraph();
    if (graph ==  NULL)
      { printf("Incorrect graph input\n");
        exit(0);
      };
  gset = gsetunion((RB *)(graph->grvtlist), SND, (RB *)gset1, SND);
  savegset(gset);
  return 0;
}
```

Table 3.4: *Command Procedure* `set02.cmd` *and Program* `set02a.c`

```
/*************************************************************/
/*     Program  main.                                        */
/*                                                           */
/*     Reads a graph and then the elements of a general set  */
/*     from standard input, constructs the corresponding     */
/*     vertex set, and generates the a graph from the vertex */
/*     The names and the number of elements of both graphs   */
/*     are printed.                                          */
/*************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

int   main()
{ GRAPH      *graph, *graph1;
  VTSET      *vtset;
  GSET       *gset;

  graph = readgraph();
    if (graph ==  NULL)
      { printf("Incorrect graph input\n");
        exit(0);
      };
  printgrlist(graph, "short");
  gset = readgset();
  vtset = gset2vtset(gset, graph);
  graph1 = generatefromvt(graph, "NEWGRAPH", vtset->vtsetlist);
  printgrlist(graph1, "short");
  return 0;
}

  Graph1        Type = GG     #Vertices =    22  #Edges =     8  #Arcs =    29
  NEWGRAPH      Type = GG     #Vertices =    22  #Edges =     8  #Arcs =    29
```

Table 3.5: *Program* `set02b.c`

Chapter 4

Graph Generating Functions

4.1 Problem Description

4.1.1 General Remarks

This chapter describes important GHS functions which generate new graphs from existing ones.

General graph generating functions. The most important functions of this type are `generatefromvt` and `generatefromed` which construct the subgraph of a given graph generated by a set of vertices, respectively edges/arcs. Another function is `cpchtp` which copies a graph while simultaneously changing its type. Finally, function `degreedelete` deletes recursively all vertices of degree n or lower from a graph and generates subgraphs from the remaining and from the deleted edges. It also constructs a list of the vertices common to both subgraphs.

Functions for generating graphs from elementary graph decompositions. Elementary general graph decompositions are the decomposition into weak and strong components (chapter 5) at the one hand and the biblock decomposition (chapter 6) at the other hand. Function `generatefromcomp` builds the subgraph corresponding to a specific decomposition element given by its name. For details on names and addresses see subsection 4.1.2 below.

Other graph generating functions. Other graph generating functions are scattered throughout the remaining chapters.

- Condensed graph and component graph of connected components.
  See subsections 5.3.4, page 58, and 5.3.5, page 58.
- Biblock graph of a biblock decomposition.
  See subsection 6.3.4, page 69.
- For subgraphs generated from elements of higher decompositions see chapter 11.

4.1.2 Names and Adresses

Weak and strong components together with the external dag lead to a hierarchial decomposition of a general graph. The biblock decomposition of a general graph introduced in chapter 6 gives rise to the stopfree kernel, peripheral trees, subcomponents, internal trees, and biblocks, another

hierarchial decomposition. Together, we call these elements *decomposition elements* or *building blocks*. Those which can occur multiply are described by structure records, for instance `BLB` for biblocks or `WCOMP` for weak components. Those which occur only once per weak component, namely external dag and stoprfree kernel, have their data stored in the corresponding `WCOMP` record. The address of a decomposition element is the address of its describing record. It is run dependent and, in general, will change from one construction of the graph to the next. In additon, hierarchical names based on the number of a decomposition element are provided, too. Table 4.1 shows how these name are built. The number assigned to a decomposition element

<*graphname*>.WCOMP<*no*>	weak component
<*graphname*>.WCOMP<*no*>.SCOMP<*no*>	strong component
<*graphname*>.WCOMP<*no*>.EXD	external dag
<*graphname*>.WCOMP<*no*>.STP	stopfree kernel
<*graphname*>.WCOMP<*no*>.PT<*no*>	peripheral tree
<*graphname*>.WCOMP<*no*>.IT<*no*>	internal tree
<*graphname*>.WCOMP<*no*>.SUB<*no*>	subcomponent
<*graphname*>.WCOMP<*no*>.SUB<*no*>.BLB<*no*>	biblock
<*no*> is the identifying number of the decomposition element.	

Table 4.1: *Decomposition hierarchies and names*

depends also of the construction run of the graph and may vary from one run to the next. However, it is useful and desirable to identify the decomposition elements by names which are run independent. Therefore, connected components and biblock decomposition elements, get a second name, a *charname*. A such the smallest line name whitin the lines of a decomposition element is chosen.

Remark 4.1 1. Weak and strong components could have been characterized by vertex names since they are vertex disjoint. Elements of the biblock decomposition are in general not vertex disjoint. However, they have no common lines. For consistency reasons line names have also been chosen for connected components.

In general, higher components as introduced in chapter 11, have lines in common. Therefore, no charnames but only hierarchical names based on component numbers are used. □

4.2 Formats and Data Structures

For a general description and basic data types see section 2.2, page 11.
The data structures used with the graph generating functions are the same as the ones used with the set handling functions of chapter 3. For function `generatefromcomp` the data structures for weak and strong components (chapter 5) and for the biblock decomposition (chapter 6) are of concern, too.

4.3 Functions

4.3.1 generatefromvt

Program Author: Günther Stiege, Universität Oldenburg

Syntax:
```
GRAPH    *generatefromvt(GRAPH *oldgraph, char *newname,
RVERTEX *rvt)
```

Description: Constructs the subgraph of a given graph generated by a set of vertices.

Error Exits: Graph pointer NULL, old and new graph have the same name, vertex not in graph.

Remarks: None.

Examples: Example 4.1, page 48. See also example `gen03.c` in GHS directory `GHStests`.

4.3.2 generatefromed

Program Authors: Dirk Ahlers and Tobe Toben, Universität Oldenburg

Syntax:
```
GRAPH    *generatefromed(GRAPH *oldgraph, char *newname,
REDGE *redge)
```

Description: Constructs the subgraph of a given graph generated by a set of edges/arcs.

Error Exits: Graph pointer NULL, old and new graph have the same name, edge/arc not in graph.

Remarks: None.

Examples: See example `gen02.c` in GHS directory `GHStests`. The example is analogous to example 4.1.

4.3.3 cpchtp

Program Authors: Dirk Ahlers and Tobe Toben, Universität Oldenburg

Syntax:
```
GRAPH    *cpchtp(GRAPH *graph, char *newtype, char *newname)
```

Description: Constructs a new graph by copying vertices and edges/arcs from the old graph and simultaneously changing its type by modifying the copied edges/arcs.

Error Exits: Graph pointer NULL, old and new graph have the same name, incorrect graph type. See also the warning under remarks below.

Remarks: The sequences of specialization of the graph types are

GG → UG → UGS → UGSLF and GG → DG → DGS → DGSLF

for undirected graphs and for digraphs, respectively (see table 2.1, page 12). Changing a graph to a less specialized type will do nothing but change the graph type in the new graph. Changing a graph to a more specialized type goes as follows:

1. GG → UG:
   All edges are kept as they are. All arcs are made edges and keep their names.

2. GG → DG:
   All arcs are kept as they are. All edges except loops are transformed into two arcs, one in each direction. One of the arcs keeps the edge's name, the name of the other is built from the edge name adding the suffix `_e2a`. Undirected loops are transformed into directed loops with the same name.

3. UG → UGS:
   For all pairs of vertices all multiple edges but one joining the vertices are deleted. It is not specified which edge is kept.

4. DG → DGS: For any pair of a starting vertex and an end vertex all multiple arcs but one are deleted. It is not specified which arc is kept.

5. UGS → UGSLF and DGS → DGSLF:
   All loops are deleted.

When type conversion encompasses several steps of a specializing sequence, it is performed as such. For instance, conversion GG → UGS is decomposed into the steps

a. GG → UG and b. UG → UGS

When type conversion goes from an U-type to a D-type or vice versa the source type is considered GG and the above rules apply.
Warning: *Do not use edge/arc names ending in* `_e2a`. *With type conversions, such names may cause a system error* 'Duplicate key cannot be inserted'.

Examples: See example `gen05.c` in GHS directory `GHStests`. It uses `Graph1` of figure 2.1, page 18, and performs the following three conversions:

a. GG → UG b. UG → DG c. DG → UGSLF .

The results are printed with option `"normal"`.

4.3.4 degreedelete

Program Author: Günther Stiege, Universität Oldenburg

Syntax:
```
void degreedelete(GRAPH *oldgraph, GRAPH **delgraph, char
*delname, GRAPH **remgraph, char *remname, VTSET **att,
int dg)
```

Description: Recursively collects all vertices of total degree `dg` or less together with the edges/arcs they are incident with. The subgraphs generated by the 'deleted' edges/arcs and by the 'remaining' edges are constructed and assigned to delgraph (with name delname) and remgraph (with name remname), respectively. The original graph is not modified. The list of vertices common to both subgraphs is constructed and assigned as a name list to att. For an empty subgraph or an empty vertex list a NULL pointer is returned.

Error Exits: graph, delgraph, remgraph or att equal NULL, graphs have not pairwise distinct names, zero or negative degree specified.

Remarks: Isolated vertices of the original graph are neither part of the 'remaining' nor of the 'deleted' subgraph.
With a fixed value of `dg` the set of edges/arcs of the 'remaining' subgraph and the 'deleted' subgraph are a partition of the set of edges/arcs of the original graph. The subgraphs may have vertices in common, a list of these is returned in att. The total degree of a vertex of the 'remaining' subgraph is at least `dg` + 1. The total degree of a vertex of the 'deleted' subgraph is bounded only by the maximal total degree of the original subgraph.

Examples: See example `gen04.c` in GHS directory `GHStests`. In this example the vertices of degree 1 are deleted from graph `GHSwords`. The degree statistics of the original graph, the 'deleted' subgraph, and the 'remaining' subgraph are printed as well as the list of common vertices of the subgraphs. If the lists are printed with option `"detailed"` instead of `"normal"` one can check that vertex `chiff` has degree 3 in the 'deleted' subgraph and degree 2 in the 'remaining' subgraph. If graphs `GHSwords-biblock` or `tree-u` are used instead of `GHSwords` the 'deleted', respectively the 'remaining' subgraph is empty.

4.3.5 generatefromcomp

Program Author: Günther Stiege, Universität Oldenburg

Syntax:
```
void      *generatefromcomp(GRAPH *graph, char *gstdname,
          char *newname)
```

Description: `gstdname` specifies the hierachical name of a decomposition element. For hierarchical names see subsection 4.1.2 and table 4.1. If such an element exists, the subgraph it generates is returned as a new graph with name `newname`.
The type of the new graph is the same as that of the original graph with the following exception for the subgraph generated from the external dag of a weak component (`EXD`):

`GG`	$\mapsto$	`DG`
`DGS`	$\mapsto$	`DGSLF`

Error Exits: NULL returned if syntax of specified name wrong or decomposition element does not exist.

Remarks: The hierarchical names of the elements of a decomposition are displayed by function `gprstd`, page 56, and by function `aprstd`, page 68. The name of a decomposition element given its address can also be obtained directly with function `getname`, page 125.

Examples: See example 5.2, page 58.

4.3.6 generatefromchnm

Program Author: Günther Stiege, Universität Oldenburg

Syntax:
```
void      *generatefromchnm(GRAPH *graph, char *linename,
          char *comptype, char *newname)
```

Description: `linename` specifies the name of a decomposition element as `charname` (see subsection 4.1.2). To uniquely identify the element `comptype` must be specified as one of the following strings:

`"WCOMP"`	weak component
`"SCOMP"`	strong component
`"EXD"`	external dag
`"STP"`	stopfree kernel
`"PT"`	peripheral tree
`"IT"`	internal tree
`"SUB"`	subcomponent
`"BLB"`	biblock

If such an element exists, the subgraph it generates is returned as a new graph with name `newname`. The type of the new graph is the same as that of the original graph with the following exception for the subgraph generated from the external dag of a weak component (`EXD`):

```
GG   ↦  DG
DGS  ↦  DGSLF
```

Error Exits: NULL returned if wrong parameter or decomposition element does not exist.

Remarks: The charnames of the elements of a decomposition are displayed by function `gprstd`, page 56, and by function `aprstd`, page 68. The charname of a decomposition element given its address can also be obtained directly with function `getcharname`, page 125.

Examples: See example 5.2, page 58.

4.4 Examples

Example 4.1 In this example we use Graph1 (figure 2.1) again. Table 4.2 shows the program in the lower part. Graph1 is constructed using `readgraphlist`. After that the vertex sets `vtset1` and `vtset2` are filled using `readvtset` with the vertices shown in the upper part of table 4.2 For each vertex set its number of elements is printed and then the set itself using function `savevtset`. Then `vtset2` is joined to `vtse1` (function `addvtset2vtset`) and the single vertex `K00` is added to `vertex1` (function `add2vtset`). Finally, the subgraph of Graph1 generated by `vtset1` is constructed (function `generatefromvt`) and written to standard output (function `printgrlist`). The results of the program run are shown in tables 4.3 and 4.4. □

```
K11 K06   K13    $END
K15   K14

/*************************************************************/
/*     Program  main.                                        */
/*                                                           */
/*     Reads a graph, reads 2 vertex sets, joins the vertex  */
/*     sets, adds an additonal vertex to the set,            */
/*     and uses the resulting vertex set to generate         */
/*     the correponding subgraph.                            */
/*                                                           */
/*************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

main()
{
  GRAPH      *graph, *newgraph;
  VTSET      *vtset1, *vtset2;

  graph = readgraphlist("xxx");
  vtset1 = readvtset(graph, NULL);
  printf("No. of elements in vertex set = %d\n", vtset1->vtsetcard);
  savevtset(vtset1, NULL);

  vtset2 = readvtset(graph, NULL);
  printf("No. of elements in vertex set = %d\n", vtset2->vtsetcard);
  savevtset(vtset2, NULL);
  addvtset2vtset(graph, &vtset1, vtset2->vtsetlist);
  add2vtset(graph, &vtset1, "K00");

  newgraph = generatefromvt(graph, "NEWGRAPH", vtset1->vtsetlist);
  printgrlist(newgraph, "detailed");
}
```

Table 4.2: *Program* `gen01.c`

```
No. of elements in vertex set = 3
K06
K11
K13
No. of elements in vertex set = 2
K14
K15

Statistics of graph NEWGRAPH    Type = GG
  #Vertices =     6
  #Edges    =     3  #Mult.edges =   2  #Loops(u) =   2  #Mult.loops(u) =   2
  #Arcs     =     8  #Mult.arcs  =   2  #Loops(d) =   3  #Mult.loops(d) =   2

Degree summary (undirected loops are counted twice!):
      Mean total degree                  3.67
      Standard deviation of total degree 1.80
      2 Vertices of total degree     2
      2 Vertices of total degree     3
      1 Vertices of total degree     5
      1 Vertices of total degree     7
```

Table 4.3: *Results of Program* `gen01.c` *(a)*

```
Vertex list:
    K06                 degree =   0  indegree =   1  outdegree =   1
      Outgoing arcs
          _d*K06*K00
      Incoming arcs
          _d*K11*K06
    K13                 degree =   1  indegree =   1  outdegree =   0
      Undirected edges
          _u*K11*K13
      Incoming arcs
          _d*K00*K13
    K00                 degree =   0  indegree =   2  outdegree =   1
      Outgoing arcs
          _d*K00*K13
      Incoming arcs
          _d*K06*K00
          _d*K11*K00
    K11                 degree =   1  indegree =   0  outdegree =   2
      Undirected edges
          _u*K11*K13
      Outgoing arcs
          _d*K11*K00
          _d*K11*K06
    K15                 degree =   0  indegree =   3  outdegree =   2
      Outgoing arcs
          _d*K15*K15a (multiple loop)
          _d*K15*K15b (multiple loop)
      Incoming arcs
          _d*K14*K15
          _d*K15*K15a (multiple loop)
          _d*K15*K15b (multiple loop)
    K14                 degree =   4  indegree =   1  outdegree =   2
      Undirected edges
          _u*K14*K14a (multiple loop)
          _u*K14*K14b (multiple loop)
      Outgoing arcs
          _d*K14*K14 (loop)
          _d*K14*K15
      Incoming arcs
          _d*K14*K14 (loop)
```

Table 4.4: *Results of Program* `gen01.c` *(b)*

Chapter 5

Weak and Strong Components

5.1 Problem Description

Connected components are the most obvious and the most important decomposition of undirected graphs. For digraphs, i. e. directed graphs, the situation is more complex (see [Hara1969], [CharL1996] or [Volk1996]). It gets still more complex if we allow for general graphs, i. e. graphs of type `GG`. In these graphs (undirected) edges and (directed) arcs may be present in any mixture. In addition, loops and multiple edges/arcs are allowed.
In the following the definitions and properties of weakly connected components and strongly connected components are summarized. For more details see Stiege [Stie2007a] and [Stie2006]. The algorithms used by GHS can be found there, too.

Paths: We consider three kinds of paths: f-paths, b-paths and a-paths. f stands for `forward`, b for `backward`, and a for `any`. An f-path is a sequence

$$v_0, l_1, v_1, l_1, v_2, \ldots, v_{n-1}, l_n, v_n$$

with $n \geq 1$. l_i is a an edge or an arc. If l_i is an edge v_{i-1} is one of its end points and v_i is the other end point. If l_i is an arc, v_{i-1} is its head and v_i is its tail. b-paths and a-paths are defined accordingly.

A path is closed if $v_0 = v_n$, otherwise it is open. A path is simple if all v_i are pairwise distinct with the exception of the first and the last vertex of a closed path. A path is edge-simple if all l_i are pairwise distinct. A closed simple and edge-simple path is a circuit. We distinguish f-circuits, b-circuits, and a-circuits. A (sub)graph with no f-circuits is called f-acyclic, otherwise it is called f-cyclic. b-acyclic/b-cyclic and a-acyclic/a-cyclic graphs are defined accordingly. Vertex v is f-reachable (b-reachable, a-reachable) from vertex u if there is an f-path (b-path, a-path) from u to v.

Note: In this chapter paths are mere graph-theoretic entities used to define weakly and strongly connected components and to establish their properties. However, paths exist in GHS also as objects in their own right. For details see chapter 9.

Definitions and Properties of Connected Components: A weakly connected component (weak component for short) of a general graph is a subgraph generated by a maximal set of mutually a-reachable vertices. We call isolates vertices improper weakly connected components. In the following, by weak components we always mean proper weakly connected components, i. e those which have at least one edge or arc. A strongly connected component (strong component

for short) of a general graph is a subgraph of a proper weak component generated by a maximal set of mutually f-reachable vertices. We get the same strong components if we use mutual b-reachability instead. An edge and its end points always belong to a strong component. In a weak component, the subgraph generated by the arcs not belonging to any strong component is called the external dag[1] of the weak component. Vertices common to the external dag and to one of the strong components are called weak attachment points.
By

$$z' \text{ is f-reachable from } z \text{ but } z \text{ is not f-reachable from } z'$$

a strict partial ordering $z \prec z'$ is defined on the set of vertices of a weak component. From the partial ordering a unique level number for each z is derived. Level number 0 is assigned to the starting points of this partial ordering. All vertices of a strong component have the same level number. A vertex which is not element of a strong component is called vertex of no return. If in a weak component there is only one element of level 0 (vertex of no return or strong component) the weak component is called rooted.

Weak connected components can easily be found with a-depth-first search, i. e. a depth-first search where arcs may be passed through in any direction. The same algorithm tests whether the weak component is a-acyclic or not. Within a weak component the strong components are found by a combination of f-depth-first search and b-depth-first search. The same algorithms find the level numbers of the strong components as well as those of the vertices of no return. Finally, the algorithms also test whether a strong component found is f-acyclic or not.

We thus get the following classification of weak and strong components. A weak component is of one of the following types:

1. ***a-acyclic weak component with no strong components.***
It is a dag. It is an a-tree, i. e. there is exactly one simple a-path joining any pair of distinct vertices. If and only if there is only one vertex with level number 0, the weak component is an f-tree (directed tree), too, and the vertex is its root.

2. ***a-acyclic weak component with strong components.*** The strong components consist of edges only. As subgraphs they are free trees. The weak component is an a-tree. It is also an f-tree if and only if there is only one element with level number 0. If this element is a vertex of no return, this vertex is the root. If the element with level number 0 is a strong component then any of its vertices can be taken as root of the f-tree.

3. ***a-cyclic weak component with no strong components.*** This is again a dag, but not an a-tree. If and only if there is only one vertex with level number 0, from this vertex all others are f-reachable. However, at least one other vertex is f-reachable by two different line-simple paths. Such weak components are called rooted.

4. ***a-cyclic weak component with strong components.*** This is the general case. In the external dag a-circuits may be present. The strong components of a weak component may be classified in the following manner:

- *The strong component is f-acyclic.*
  It then consists of edges only and is a free tree. If all strong components have this property then there must be an a-circuit containing arcs of the external dag.

- *The strong component is f-cyclic.*
  Edges and arcs may occur in any combination. The following holds:

[1] Directed acyclic graph.

- If an arc is part of a strong component, there is an f-cycle through that arc and the strong component is f-cyclic.
- If a strong component contains an a-circuit, it is f-cyclic and every edge on that a-circuit lies on a an f-circuit.

If and only if there is exactly one element with level number 0, there are vertices with root property, i. e. vertices from which all other vertices are f-reachable. Again such weakly connected components are called rooted. If the element is a vertex of no return, then there is exactly one such vertex. If it is a strong component all its vertices have root property. In any case, in general there is more than one simple f-path from a vertex with root property to another vertex.

Condensed Graph and Component Graph:

STILL TO DO

Periods of Components: The a-period p of a vertex v of a weak component is defined as the greatest common devisor of the lengths of all closed a-paths which start and end in v. The f-period q of a vertex u of a strong component is defined as the greatest common devisor of the lengths of all closed f-paths which start and end in u. It is not difficult to prove that all vertices of a weak component have the same a-period and that all vertices of a strong component have the same f-period. Therefore, the a-period of a weak component and the f-period of a strong component can be defined based on the period of their vertices. The a-period of a weak component can only be 1 or 2. If it is 2 the weak component is called bipartite. If a strong component contains an edge, the f-period of the strong component can only be 1 or 2. If it is a pure digraph, any f-period is possible. If the a-period (f-period) of a weak (strong) component is p then the set of vertices of the weak (strong) component is partitioned into p classes with a cyclic ordering such that every step of an a-path (f-path) always passes from one class to the next in the cyclic ordering. For more details on periods in graphs see Stiege [Stie2006] and [Stie2007a].

5.1.1 Survey of functions

The main function of this chapter is `gcomponents`. It finds the weakly connected and strongly connected components of a general graph. Function `gprstd` is used to print the structure found by `gcomponents`. A list of vertices is printed with `gprstdvt`. To generate a subgraph from a decomposition element (weak component, strong component, external dag) function `generatefromcomp`, page 47, of chapter 4 is used.

5.2 Formats and Data Structures

For a general description and basic data types see section 2.2, page 11.
The data structures used with the decomposition into weak and strong components are described in subsection C.7, page 172. The main record to describe the decomposition into weak and strong components is of type `GSTDD`. It is pointed to from the `grgstruct` field in the `GRAPH` record (subsection C.5). The decomposition elements of a general graph are represented by the data types `WCOMP` (weak component) and `SCOMP` (strong component), With data type `GVTSTD` general vertex properties corresponding to weak and strong components (e. g. weak attachment point) are recorded. In the same way, records of type `GEDSTD` are used for edges/arcs.

Remark 5.1 Weak components of a general graph are numbered[2]. These numbers are used as identifiers. Within a weak component the strong components are numbered[2]. These numbers are used as internal identifiers. The complete identification of a weak (strong) component is its name as specified in table 4.1, page 44. □

5.3 Functions

5.3.1 gcomponents

Program Author: Sergeij Alekseev, Universität Oldenburg

Syntax: `void gcomponents(GRAPH *graph)`

Description: Finds the decomposition of a general graph (of any type): Weak components, strong components, external dag, weak attachment points, level numbers, periods. Creates a `GSTDD` record and complementary records. After successful termination, the field `grgstruct` of the graph's `GRAPH` record points to the `GSTDD` record.

Error Exits: Graph pointer NULL. General decomposition already exists (pointer `grgstruct` is not NULL).

Remarks: The field `vtptgstd` in the `VERTEX` records is used to point to the corresponding `GVTSTD` records. The field `edptgstd` in the `EDGE` records is used to point to the corresponding `GEDSTD` records.

Examples: Example 5.1, page 58.

5.3.2 gprstd

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void gprstd(GRAPH *graph, int stroption, char *filname)`

Description: Writes the general decomposition (weak components, strong components, and external dags) of a general graph to file `filename`. If `filename` equals NULL, the output is written to standard oputput. Has the following print options:

`STR`	Prints complete decomposition structure.
`STRA`	Prints weak attachment points but no other vertices and no edges.
`STRR`	Prints reduced structure. No vertices and no lines are printed.
`STRCS`	Prints condensed statistics only. If available, this statstistics also includes information about the biblock decomposition of the graph (see chapter 6).

Error Exits: Illegal print option. Graph pointer Null. Decomposition into weak and strong components does not exist.

[2]The numbering depends on the specific run of the decomposition algorithms.

Remarks: A line like

```
$STRONG_COMPONENT wcompgraph.WCOMP6.SCOMP1 <_uE13E16> (2V, 1E, 0A) (2WAP) lv=1 fper = 2
```

is to be read: Strong component No.1 of weak component No.6 of graph 'wcompgraph' has `charname _uE13E16`. It consists of 2 vertices, 1 edge, and 0 arcs. It has 2 weak attachment points, level No. 1, and f-period 2.

Examples: Example 5.1, page 58. For option `STRCS` see example 6.1, page 70.

5.3.3 gprstdvt

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void gprstdvt(GRAPH *graph, int poption, char *filename)`

Description: Writes a list of vertices ordered by total degree and vertex name to file `filename`. If `filename` equals NULL the output is written to standard output. For each vertex and the edges/arcs incident with it graph decomposition information is written. Has the following print options:

`VPV`	All vertices.
`VPA`	All weak attachment points.

Error Exits: Graph pointer Null. Decomposition into weak and strong components does not exist. Wrong print option.

Remarks: A list entry like

```
E12              $DG 1    $ODG 2    $IDG 3    $LV 1    wcompgraph.WCOMP12
  wcompgraph.WCOMP12.SCOMP3             weak attachment point
  $No_OUTGOING_EXTERNAL_TREE_ARCS  1
                _dE12E13
  $No_INCOMING_EXTERNAL_TREE_ARCS  0
  $No_EDGES 1
                _uE09E12
  $No_OUTGOING_STRONG_COMPONENT_ARCS  1
                _dE12E11
  $No_INCOMING_STRONG_COMPONENT_ARCS  3
                _dE08E12
                _dE10E12
                _dE15E12
```

is to be read as follows. Vertex `E12` belongs to weak component No. 12 of graph `wcompgraph`. It has total degree 6 (1 edge, 2 outgoing arcs, 3 incoming arcs). The level number is 1. It is a vertex of return in strong component No. 3. It is a weak attachment point. A list of arcs and edges `E12` is incident with follows, stating for each arc whether it is an external dag arc or not. For instance, `_dE12E13` is an outgoing arc of vertex `E12` and belongs to the external dag. `_dE15E12` is an incoming arc und belongs to the strong component `SCOMP3`.

Examples: Example 5.2, page 58.

5.3.4 condgraph

Still to do

5.3.5 compgraph

Still to do

5.4 Examples

Example 5.1 This example corresponds to `gcomp01.cmd` and `gcomp01.c` in GHStests. Function `gcomponents` is applied to the general graph *wcompgraph* shown in figure 5.1. In a second step the resulting structure is printed calling function `gprstd`. Tables 5.1 and 5.2 show the output. Examples `gcomp02.cmd` and `gcomp02.c` in GHStests show a condensed structure printing (parameter `STRCS`) and a complete structure printing (parameter `STR`) of the same graph. The output is not included in the manual.

Example 5.2 Program `gcomp03.c` generates a subgraph from weak component `WCOMP2` of graph *wcompgraph*, applies `gcomponents` to this subgraph, and prints a list of vertices using function `gprstdvt` with options `VPV` and `VPA`. The results correponding to option `VPA` are shown in table 5.3.

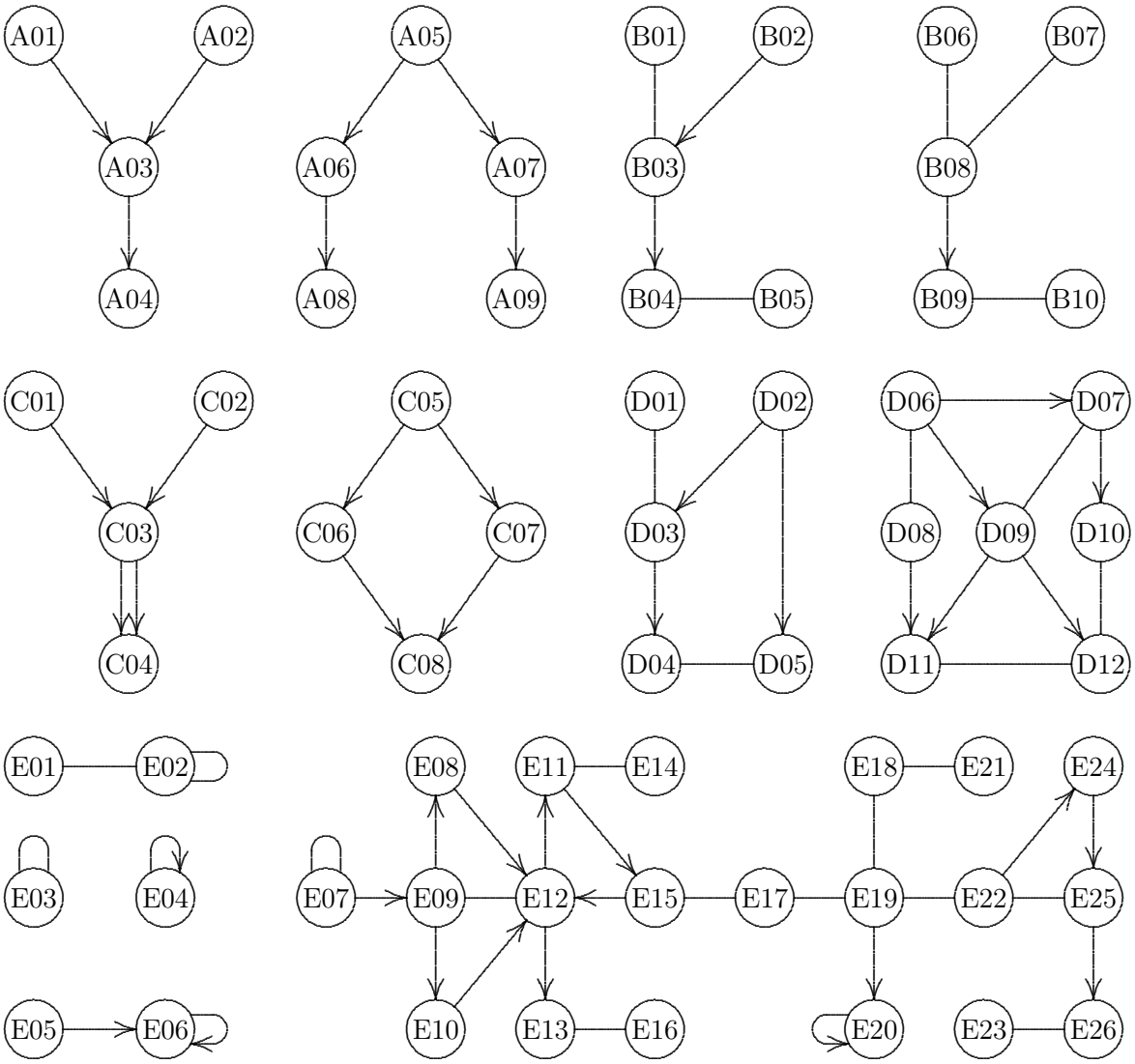


Figure 5.1: *wcompgraph*

```
BEGIN WEAK AND STRONG COMPONENTS - REDUCED STRUCTURE
$GRAPH wcompgraph
$TYPE  GG
$No_VERTICES                65
$No_EDGES                   25
$No_ARCS                    44
$No_ISOLATED_VERTICES        0
$No_WEAK_COMPONENTS                                          13
$No_A-ACYCLIC_WEAK_COMPONENTS_WITHOUT_STRONG_COMPONENTS      2     (9V, 0E, 7A)
   $WEAK_COMPONENT wcompgraph.WCOMP0 <_dA01A03> (4V, 0E, 3A)  aper = 2  MAXLV = 2
   $WEAK_COMPONENT wcompgraph.WCOMP1 <_dA05A06> (5V, 0E, 4A)  aper = 2  MAXLV = 2
      f-TREE   root: A05 (vertex)
$No_A-ACYCLIC_WEAK_COMPONENTS_WITH_STRONG_COMPONENTS         2     (10V, 5E, 3A)
   $WEAK_COMPONENT wcompgraph.WCOMP2 <_dB02B03> (5V, 2E, 2A) aper = 2  MAXLV = 2
      f-TREE   root: B02 (vertex)
   $No_STRONG_COMPONENTS                                     2
   $No_f-ACYCLIC_STRONG_COMPONENTS                           2     (4V, 2E, 0A)
      $STRONG_COMPONENT wcompgraph.WCOMP2.SCOMP0 <_uB01B03> (2V, 1E, 0A)  (1WAP) lv = 1  fper = 2
      $STRONG_COMPONENT wcompgraph.WCOMP2.SCOMP1 <_uB04B05> (2V, 1E, 0A)  (1WAP) lv = 2  fper = 2
      $EXTERNAL_DAG wcompgraph.WCOMP2.SCOMP1 <_dB02B03>  (3V, 0E, 2A)  (2WAP)
   $WEAK_COMPONENT wcompgraph.WCOMP3 <_dB08B09> (5V, 3E, 1A) aper = 2  MAXLV = 1
      f-TREE   root: wcompgraph.WCOMP3.SCOMP0 <_uB06B08> (strong component)
   $No_STRONG_COMPONENTS                                     2
   $No_f-ACYCLIC_STRONG_COMPONENTS                           2     (5V, 3E, 0A)
      $STRONG_COMPONENT wcompgraph.WCOMP3.SCOMP0 <_uB06B08> (3V, 2E, 0A)  (1WAP) lv = 0  fper = 2
      $STRONG_COMPONENT wcompgraph.WCOMP3.SCOMP1 <_uB09B10> (2V, 1E, 0A)  (1WAP) lv = 1  fper = 2
      $EXTERNAL_DAG wcompgraph.WCOMP3.SCOMP1 <_dB08B09>  (2V, 0E, 1A)  (2WAP)
$No_A-CYCLIC_WEAK_COMPONENTS_WITHOUT_STRONG_COMPONENTS       2     (8V, 0E, 8A)
   $WEAK_COMPONENT wcompgraph.WCOMP4 <_dC01C03> (4V, 0E, 4A) aper = 2  MAXLV = 2
   $WEAK_COMPONENT wcompgraph.WCOMP5 <_dC05C06> (4V, 0E, 4A) aper = 2  MAXLV = 2
      rooted   root: C05 (vertex)
$No_A-CYCLIC_WEAK_COMPONENTS_WITH_STRONG_COMPONENTS          2     (12V, 6E, 9A)
    $(ALL_STRONG_COMPONENTS_F-ACYCLIC)
   $WEAK_COMPONENT wcompgraph.WCOMP6 <_dD02D03> (5V, 2E, 3A) aper = 2  MAXLV = 2
      rooted   root: D02 (vertex)
   $No_STRONG_COMPONENTS                                     2
   $No_f-ACYCLIC_STRONG_COMPONENTS                           2     (4V, 2E, 0A)
      $STRONG_COMPONENT wcompgraph.WCOMP6.SCOMP0 <_uD01D03> (2V, 1E, 0A)  (1WAP) lv = 1  fper = 2
      $STRONG_COMPONENT wcompgraph.WCOMP6.SCOMP1 <_uD04D05> (2V, 1E, 0A)  (2WAP) lv = 2  fper = 2
      $EXTERNAL_DAG wcompgraph.WCOMP6.EXD <_dD02D03> (4V, 0E, 3A)  (3WAP)
   $WEAK_COMPONENT wcompgraph.WCOMP7 <_dD06D07> (7V, 4E, 6A) aper = 1  MAXLV = 2
      rooted   root: wcompgraph.WCOMP7.SCOMP0 <_uD06D08> (strong component)
   $No_STRONG_COMPONENTS                                     3
   $No_f-ACYCLIC_STRONG_COMPONENTS                           3     (7V, 4E, 0A)
      $STRONG_COMPONENT wcompgraph.WCOMP7.SCOMP0 <_uD06D08> (2V, 1E, 0A)  (2WAP) lv = 0  fper = 2
      $STRONG_COMPONENT wcompgraph.WCOMP7.SCOMP1 <_uD07D09> (2V, 1E, 0A)  (2WAP) lv = 1  fper = 2
      $STRONG_COMPONENT wcompgraph.WCOMP7.SCOMP2 <_uD10D12> (3V, 2E, 0A)  (3WAP) lv = 2  fper = 2
      $EXTERNAL_DAG wcompgraph.WCOMP7.EXD <_dD06D07> (7V, 0E, 6A)  (7WAP)
END REDUCED STRUCTURE
```

Table 5.1: *Weak and strong components of graph wcompgraph* (Part A)

```
$No_A-CYCLIC_WEAK_COMPONENTS_WITH_STRONG_COMPONENTS          5     (26V, 14E, 17A)
    $(F-CYCLIC_STRONG_COMPONENTS_EXIST)
   $WEAK_COMPONENT wcompgraph.WCOMP8 <_uE01E02> (2V, 2E, 0A)  aper = 1  MAXLV = 0
      rooted   root: wcompgraph.WCOMP8.SCOMP0 <_uE01E02>(strong component)
   $No_STRONG_COMPONENTS                                     1
   $No_f-ACYCLIC_STRONG_COMPONENTS                           0
   $No_f-CYCLIC_STRONG_COMPONENTS                            1
      $STRONG_COMPONENT wcompgraph.WCOMP8.SCOMP0 <_uE01E02> (2V, 2E, 0A)  (0WAP) lv = 0  fper = 1
      $EXTERNAL_DAG wcompgraph.WCOMP8.EXD  <> (0V, 0E, 0A)  (0WAP)
   $WEAK_COMPONENT wcompgraph.WCOMP9 <_dE04E04> (1V, 0E, 1A)  aper = 1  MAXLV = 0
      rooted   root: wcompgraph.WCOMP9.SCOMP0 <_dE04E04>(strong component)
   $No_STRONG_COMPONENTS                                     1
   $No_f-ACYCLIC_STRONG_COMPONENTS                           0
   $No_f-CYCLIC_STRONG_COMPONENTS                            1
      $STRONG_COMPONENT wcompgraph.WCOMP9.SCOMP0 <_dE04E04> (1V, 0E, 1A)  (0WAP) lv = 0  fper = 1
      $EXTERNAL_DAG wcompgraph.WCOMP9.EXD  <> (0V, 0E, 0A)  (0WAP)
   $WEAK_COMPONENT wcompgraph.WCOMP10 <_uE03E03> (1V, 1E, 0A)  aper = 1  MAXLV = 0
      rooted   root: wcompgraph.WCOMP10.SCOMP0 <_uE03E03>(strong component)
   $No_STRONG_COMPONENTS                                     1
   $No_f-ACYCLIC_STRONG_COMPONENTS                           0
   $No_f-CYCLIC_STRONG_COMPONENTS                            1
      $STRONG_COMPONENT wcompgraph.WCOMP10.SCOMP0 <_uE03E03> (1V, 1E, 0A)  (0WAP) lv = 0  fper = 1
      $EXTERNAL_DAG wcompgraph.WCOMP10.EXD  <> (0V, 0E, 0A)  (0WAP)
   $WEAK_COMPONENT wcompgraph.WCOMP11 <_dE05E06> (2V, 0E, 2A)  aper = 1  MAXLV = 1
      rooted   root: E05 (vertex)
   $No_STRONG_COMPONENTS                                     1
   $No_f-ACYCLIC_STRONG_COMPONENTS                           0
   $No_f-CYCLIC_STRONG_COMPONENTS                            1
      $STRONG_COMPONENT wcompgraph.WCOMP11.SCOMP0 <_dE06E06> (1V, 0E, 1A)  (1WAP) lv = 1  fper = 1
      $EXTERNAL_DAG wcompgraph.WCOMP11.EXD  <_dE05E06> (2V, 0E, 1A)  (1WAP)
   $WEAK_COMPONENT wcompgraph.WCOMP12 <_E07E07> (20V, 11E, 14A)  aper = 1  MAXLV = 2
      rooted   root: wcompgraph.WCOMP12.SCOMP2 <_E07E07>(strong component)
   $No_STRONG_COMPONENTS                                     5
   $No_f-ACYCLIC_STRONG_COMPONENTS                           2
      $STRONG_COMPONENT wcompgraph.WCOMP12.SCOMP0 <_uE23E26> (2V, 1E, 0A)  (1WAP) lv = 2  fper = 2
      $STRONG_COMPONENT wcompgraph.WCOMP12.SCOMP1 <_uE13E16> (2V, 1E, 0A)  (1WAP) lv = 2  fper = 2
   $No_f-CYCLIC_STRONG_COMPONENTS                            3
      $STRONG_COMPONENT wcompgraph.WCOMP12.SCOMP2 <_E07E07> (1V, 1E, 0A)  (1WAP) lv = 0  fper = 1
      $STRONG_COMPONENT wcompgraph.WCOMP12.SCOMP3 <_dE08E12> (14V, 8E, 9A)  (4WAP) lv = 1  fper = 1
      $STRONG_COMPONENT wcompgraph.WCOMP12.SCOMP4 <_dE20E20> (1V, 0E, 1A)  (1WAP) lv = 2  fper = 1
      $EXTERNAL_DAG wcompgraph.WCOMP12.EXD  <_dE07E09> (8V, 0E, 4A)  (8WAP)
END REDUCED STRUCTURE
```

Table 5.2: *Weak and strong components of graph wcompgraph* (Part B)

```
WEAK AND STRONG COMPONENTS
PRINT OPTION VPA (Undirected loops are counted twice!)
$GRAPH wcomp
$TYPE GG
$No_VERTICES 5
$No_EDGES 2
$No_ARCS 2
$VERTICES

B04              $DG 1    $ODG 0    $IDG 1    $LV 2    wcomp.WCOMP0
  wcomp.WCOMP0.SCOMP1                   weak attachment point
  $No_OUTGOING_EXTERNAL_TREE_ARCS  0
  $No_INCOMING_EXTERNAL_TREE_ARCS  1
                _dB03B04
  $No_EDGES 1
                _uB04B05
  $No_OUTGOING_STRONG_COMPONENT_ARCS  0
  $No_INCOMING_STRONG_COMPONENT_ARCS  0

B03              $DG 1    $ODG 1    $IDG 1    $LV 1    wcomp.WCOMP0
  wcomp.WCOMP0.SCOMP0                   weak attachment point
  $No_OUTGOING_EXTERNAL_TREE_ARCS  1
                _dB03B04
  $No_INCOMING_EXTERNAL_TREE_ARCS  1
                _dB02B03
  $No_EDGES 1
                _uB01B03
  $No_OUTGOING_STRONG_COMPONENT_ARCS  0
  $No_INCOMING_STRONG_COMPONENT_ARCS  0
```

Table 5.3: *List of attachment points of the third weak component of graph wcompgraph*

Chapter 6

The Biblock Decomposition

6.1 Problem Description

Mutual a-reachablity is used to define weak components in a general graph. In the same way mutual f-reachability is used to define strong components. Higher components are defined by mutual k-a-reachability, respectively mutual k-f-reachability. Vertex v is k-a-reachable (k-f-reachable) from vertex v if there are (at least) k internally disjoint a-paths (f-paths) from u to v, i.e. a-paths which have only the endpoints in common. Line-disjoint paths, i.e. the paths have no common lines, lead to higher linecomponents.

For $k = 2$ and a-paths the decomposition is called biblock decomposition. It is easy to obtain and very useful. Its remarkable properties are due to the fact that two vertices which are mutually 2-a-reachable lie on an a-circuit. They lie on a line-simple closed a-path if they are mutually 2-a-linereachable. This makes it possible, to define the biblock decomposition using partitions of the set of lines of a general graph. For details see [Stie2006] or [Stie2007a]. There the algorithms used in GHS can be found.[1]

2-a-lineconnected subgraphs (and therefore also 2-a-connected subgraphs) must contain a-circuits. Hence, the biblock decomposition starts with the a-cyclic weak components of a general graph. In these a hierarchy of closed a-path is considered:

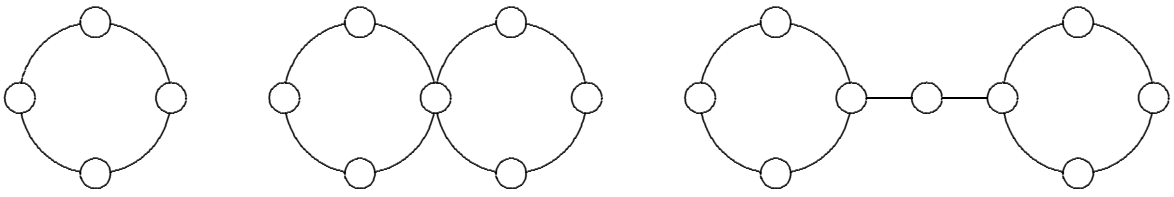


Figure 6.1: *Examples of closed paths*

$$circuits \subseteq closed\ line-simple\ paths \subseteq stopfree\ paths$$

[1] Traditionally, connected components of simple graphs are decomposed further by identifying the blocks – i. e. maximal non separable subgraphs – of each component and representing the blocks a well as the cutpoints which separate them by a block-cutpoint graph (see for instance Harary [Hara1969]). It is possible to construct the block-cutpoint graph with a depth-first search algorithm proposed by Hopcroft and Tarjan [HopcT1973], [Tarj1972]).

A stopfree path is a closed a-path where no line is immediately followed by itself and the first line is different from the last. Schematic pictures of the three kinds of paths are shown in figure 6.1. From this hierarchy of closed paths equivalence relations on lines are defined whose equivalence classes generate the decomposition subgraphs of the biblock decomposition. The result is shown in figure 6.2. The biblock decomposition proper starts with the a-cyclic weak components. Each

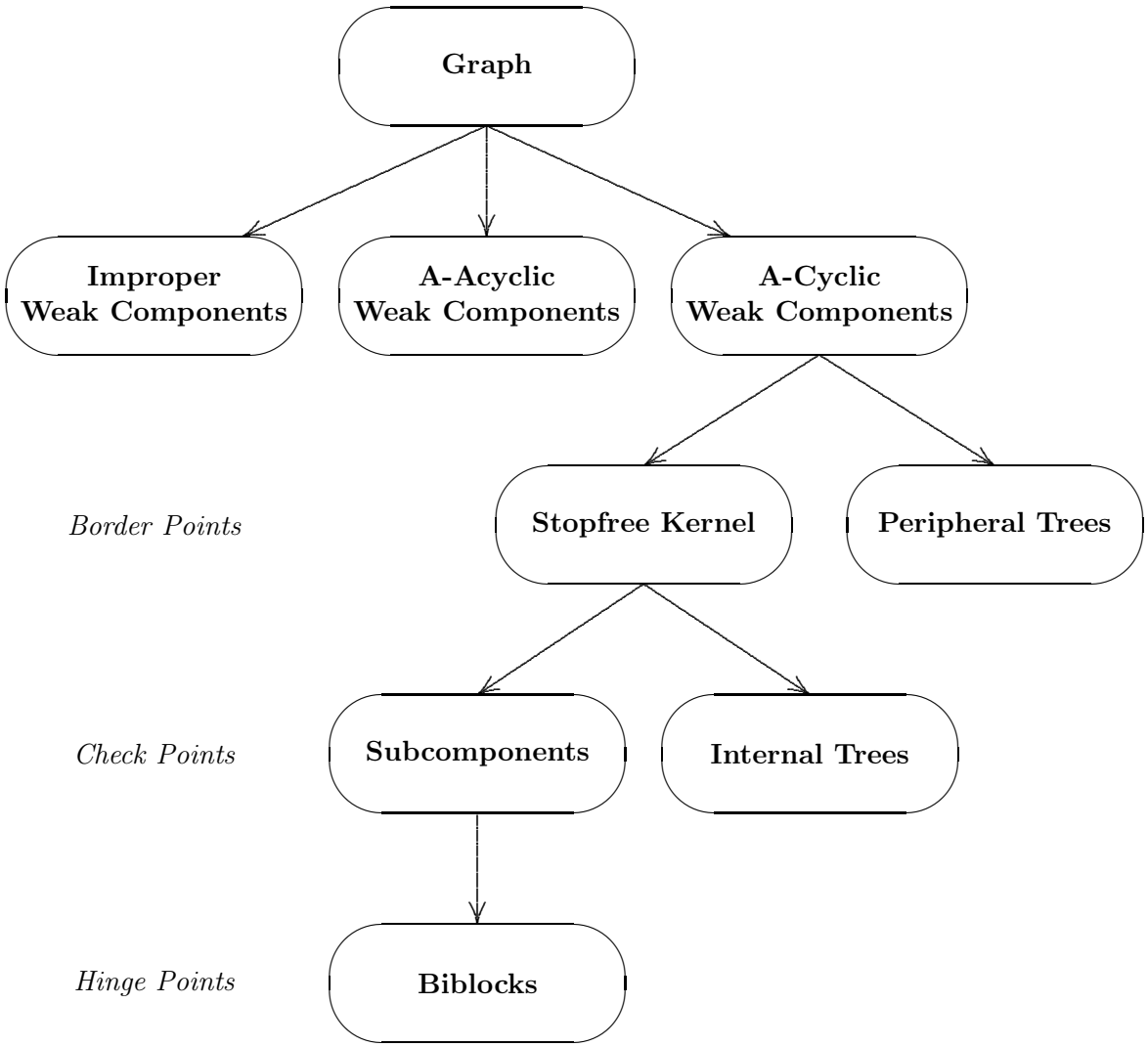


Figure 6.2: *Biblock decomposition of a general graph*

contains exactly one equivalence class of lines generated from stopfree paths. These lines generate the stopfree kernel. The lines not contained in the stopfree kernel generate an a-forest, whose weak components are the peripheral trees. In each stopfree kernel we find at least one equivalence class of lines generated by closed line-simple a-paths. These classes generate the subcomponents. In the stopfree kernel, there may be lines not belonging to any subcomponent. These again generate an a-forest. The corresponding weak components are the internal trees. Finally, each subomponent contains at least one equivalence class of lines generated by circuits. These classes generate the biblocks (biconnected blocks).

Structural units of the same level are line-disjoint but in general not vertex disjoint. The points

where these units are attached to each other are border points, check points, and hinge points. These are called attachment points, they are cut points. The lines of the peripheral and internal trees are bridges. All other lines are non-bridges.

The biblock decomposition of a general graph leads to an important derived bipartite undirected graph, the *biblock graph*. The elementary connectedness elements, namely improper connected components, acyclic components, peripheral trees, internal trees, and biblocks at the one hand and the attachment points at the other hand are the vertices. An attachment point is joined by an edge to every connectedness element it is part of. There are no other edges. Isolated vertices and a-acyclic weak components of the original graph become isolated vertices in the biblock graph. It is easy to see that in the biblock graph of a general graph there are no a-circuits and that the biblock graph is a-connected if and only if the original graph is a-connected. Therefore, the biblock graph of an a-connected general graph is an a-tree, its *biblock tree*.
The tree structure of the biblock graph of an a-connected general graph has important consequences. It allows a rather simple general and uniform treatment of most graph theoretic and many practical questions: 1. Solve the problem for the biblock tree and 2. Solve the problem for each biblock. In general, step 1 is easy and step 2 is the hard core of the problem.

Example: Figure 6.3 shows a simple graph which consists of 3 weak components (specified by

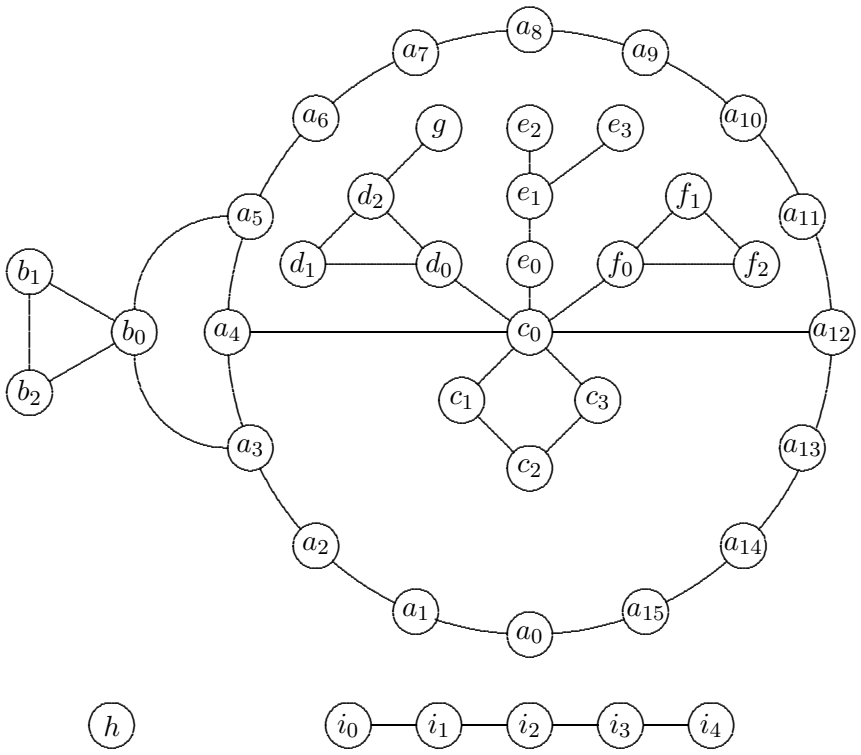


Figure 6.3: *Ugraph1*

their vertices)

1. $\{h\}$ Improper connected component.

2. $\{i_0, i_1, i_2, i_3, i_4\}$
   Proper a-acyclic weak component. No stopfree kernel.

3. $\{a_0, \ldots, a_{15}, b_0, b_1, b_2, c_0, c_1, c_2, c_3, d_0, d_1, d_2, e_0, e_1, e_2, e_3, f_0, f_1, f_2\}$
   Proper a-cyclic weak component. The stopfree kernel consists of the biblocks $\{a_0, \ldots, a_{15}, b_0, c_0\}, \{b_0, b_1, b_2\}, \{c_0, c_1, c_2, c_3\}, \{d_0, d_1, d_2\}, \{f_0, f_1, f_2\}$ and the internal tree $\{c_0, d_0, f_0\}$. In addition, there are the peripheral trees $\{d_2, g\}$ and $\{c_0, e_0, e_1, e_2, e_3\}$.

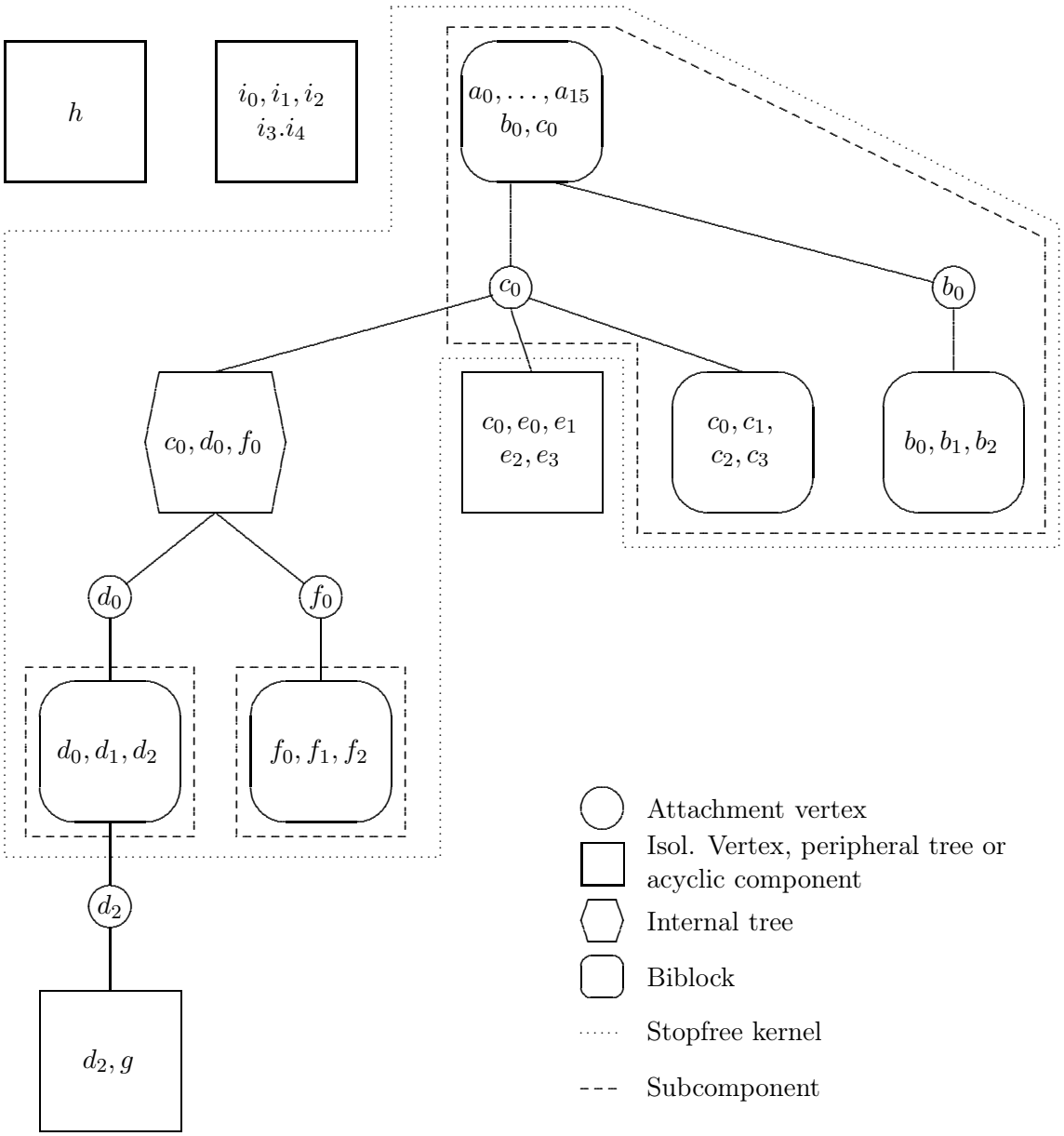


Figure 6.4: *The Biblock Graph Corresponding to Ugraph1*

Figure 6.4 shows the corresponding biblock graph.

Survey of Functions:

The main function of this chapter is `astd`. It constructs the biblock decomposition of a general graph. Function `aprstd` is used to write the biblock decomposition of a general graph. Function `aprstdvt` writes a list of vertices. The biblock graph corresponding to a general graph is constructed by function `generateblbgraph`. To generate a subgraph from an a-decomposition element (peripheral tree, stopfree kernel, subcoponent, internal tree, biblock) use the function `generatefromcomp` of chapter 4, page 47.

6.2 Formats and Data Structures

For a general description and basic data types see section 2.2, page 11.
The data structures used with the biblock decomposition functions are described in subsection C.8, page 176. The main record for each stopfree kernel is of type `WCOMP`, i. e. a record describing a weak component. This record also describes the stopfree kernel and has pointers to lists of `PT` records (peripheral trees), `SUB` records (subcomponents), `IT` records (internal trees), and `BLB` records (biblocks). With data type `AVTSTD` biblock decomposition properties of a vertex (e. g. kind of attachment point) are recorded. In the same way, records of type `AEDSTD` are used for edges/arcs. Edges/arcs a vertex is incident with may belong to more than one biblock (hinge point). Records of type `INCSQR` indicate the biblocks a vertex is element of. Several enumeration types are used by the biblock decomposition functions, mostly for internal purposes.

Remark 6.1 Peripheral trees within a weak component are numbered[2]. These numbers are used as identifiers. The same applies to subcomponents and internal trees. Within a subcomponent the biblocks are numbered[2]. These numbers are used as internal identifiers. The complete identification of an element of the biblock composition is its name as specified in table 4.1, page 44. □

6.3 Functions

6.3.1 astd

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void astd(GRAPH *graph)`

Description: Finds the biblock decomposition of a general graph (of any type). Sets the values for the decomposition elements (stopfree kernel, peripheral tree, internal trees, subcomponents, biblocks) in the corresponding WCOMP records and creates complementary records. After successful termination, the field `grastd` of the graph's `GRAPH` record is set TRUE.

Error Exits: Graph pointer NULL. Biblock decomposition already exists.

Remarks: To find the biblock decomposition of a general graph the weak components must be known. If they are not, function `gcomponents`, page 56, is called implicitly before the a-decomposition analysis is done.

[2]The numbering depends on the specific run of the decomposition algorithms.

Examples: Example 6.1, page 70.

6.3.2 aprstd

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void aprstd(GRAPH *graph, int stroption, char *filename)`

Description: Writes the the biblock decomposition of a general graph (of any type) to file `filename`. If `filename` equals NULL the output is written to standard output. The function has the following print options:

- `STR` Prints complete decomposition structure.
- `STRA` Prints attachment points but no other vertices and no lines.
- `STRR` Prints reduced structure. No vertices and no lines.

Note: Option `STRCS` for condensed output is not provided with function `aprstd`. See function `gprst`, page 56, instead.

Error Exits: Illegal print option. Graph pointer Null. Biblock decomposition does not exist.

Remarks: A line like

```
$A_CYCLIC_WEAK_COMPONENT Ugraph1.WCPOMP1    (34V, 40E, 0A, 5AP, 2BP, 3CP, 2HP)
```

is to be read: The a-cyclic weak component with name `Ugraph1.WCOMP1` consists of 34 vertices, 40 edges, and 0 arcs. 5 of the vertices are attachment points of which 2 are border points, 3 are check points, and 2 are hinge points. For name conventions see subsection 4.3.5, page 47. See also remark 5.1, page 56, and remark 6.1, page 67.

Examples: Example 6.1, page 70.

6.3.3 aprstdvt

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void aprstdvt(GRAPH *graph, int poption, char *filename)`

Description: Writes to file `filename` a list of vertices ordered by degree and vertex name with biblock decomposition information about the vertex and the edges/arcs incident with it. If `filename` equals NULL the output is written to standard output. Has the following print options:

VPV	All vertices.
VPA	All attachment points.
VPB	All border points.
VPC	All check points.
VPH	All hinge points.
VPBC	All vertices which are border points and check points, but not hinge points.
VPBH	All vertices which are border points and hinge points, but not check-points.
VPCH	All vertices which are check points and hinge points, but not border points.
VPBCH	All vertices which are border points, check points and hinge points.

Error Exits: Graph pointer Null. Biblock decomposition does not exist. Wrong print option.

Remarks: A list entry like

```
c0               $TDG 7    $DG 7    $ODG 0    $IDG 0    Ugraph1.WCOMP1
   Border Point  Check Point  Hinge Point
   $NO_PERIPHERAL_TREE_EDGES              1  (Ugraph1.WCOMP1.PT0)
                _c0*e0
   $NO_BIBLOCK_EDGES                      2  (Ugraph1.WCOMP1.SUB0.BLB1)
                _c0*c1
                _c0*c3
   $NO_BIBLOCK_EDGES                      2  (Ugraph1.WCOMP1.SUB0.BLB2)
                _c0*a12
                _c0*a4
   $NO_INTERNAL_TREE_EDGES                2  (Ugraph1.WCOMP1.IT0)
                _c0*d0
                _c0*f0
```

is to be read as follows. Vertex `c0` has total degree 7: 7edges, 0 outgoing arcs, 0 incoming arcs. The vertex belongs to weak component `Ugraph1.WCOMP1`. It is incident with peripheral tree edge `_c0*e0`. It is also incident with 4 edges which belong to biblocks. Edges `_co*c1` and `_co*c3` are from biblock `Ugraph1.WCOMP1.SUB0.BLB1` and edges `_c0*a12` and `_c0*a4` are from biblock `Ugraph1.WCOMP1.SUB0.BLB2`. Finally, `c0` is incident with 2 internal tree edges, namely `_c0*d0` and `_co*f0`.

Error Exits: Graph pointer Null. Biblock decomposition does not exist. There is no cyclic weak componenent with name `wkcomp`. File `filename` cannot be opened.

Examples: Example 6.1, page 70.

6.3.4 blbgraph

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `GRAPH *blbgraph(GRAPH *graph, char* filename)`

Description: The biblock graph corresponding to `graph` is constructed in external GHS format and output to file `filename`. If `filename` is NULL, it is output to standard output.

6.4 Examples

Example 6.1 In this example we consider the undirected graph *Ugraph1* of figure 6.3. It consists of the improper weak component[3] $\{h\}$, the a-acyclic weak component $\{i_0, \ldots, i_5\}$ and the a-cyclic weak component $\{a_0 \ldots, a_{15}, b_0, \ldots, g\}$. We process the graph with the program shown in table 6.1. The program reads the external description of the graph and constructs it using function

```
/*************************************************************/
/*     Program  main.                                        */
/*                                                           */
/*     Reads a graph and constructs its biblock decomposition.*/
/*     A condensed decomposition information (weak and       */
/*     strong components as well as biblock decomposition)   */
/*     is printed.                                           */
/*     The biblock decomposition is printed.                 */
/*     A list of vertices, ordered by degree and vertex name, */
/*     is printed, too. It shows all relevant information of  */
/*     of each vertex and all edges incident with it.        */
/*                                                           */
/*************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

int main()
{
  GRAPH      *graph;

  graph = readgraphlist(NULL);
  astd(graph);
  gprstd(graph, STRCS, NULL);
  aprstd(graph, STRR, NULL);
//aprstd(graph, STRA, NULL);
//aprstd(graph, STR, NULL);
  aprstdvt(graph, VPA, NULL);
  return 0;
}
```

Table 6.1: *Program* `acomp01.c`

`readgraphlist`. Then function `astd` is called to find and construct the biblock decomposition of the graph. Next, function `gprstd` is called with print option `STRCS` for condensed statistics. The

[3]Here we specify subgraphs by their vertices.

reduced listing of the biblock decomposition is then printed with function `aprstd`, print option `STRR`. Finally, with function `aprstdvt` a list of attachment points (option `VPA`) is printed.
The results of these steps are shown in the following tables. Table 6.2 shows the condensed results of the decomposition into weak and strong components (which for undirected graphs are identical) and the biblock decomposition. Table 6.3 contains a listing of the biblock decomposition of *Ugraph1* printed with option `STRR`. Table 6.4 shows a list of those vertices of *Ugraph1* which are attachment points.

□

```
BEGIN CONDENSED STRUCTURE OF GENERAL DECOMPOSITION
$GRAPH Ugraph1
$TYPE  UGSLF
$No_VERTICES                          40
$No_EDGES                             44
$No_ARCS                               0
$No_ISOLATED_VERTICES                  1
$No_WEAK_COMPONENTS_TYPE_1             0  (0 f-trees)
$No_WEAK_COMPONENTS_TYPE_2             1  (1 f-trees)
$No_WEAK_COMPONENTS_TYPE_3             0  (0 rooted)
$No_WEAK_COMPONENTS_TYPE_4             0  (0 rooted)
$No_WEAK_COMPONENTS_TYPE_5             1  (1 rooted)
$No_WEAK_ATTACHMENT_POINTS             0
$No_STRONG_COMPONENTS_(f-ACYCLIC)      1
$No_STRONG_COMPONENTS_(f-CYCLIC)       1
$No_STOPFREE_KERNELS                   1
$No_PERIPHERAL_TREES                   2
$No_SUBCOMPONENTS                      3
$No_BIBLOCKS                           5
$No_INTERNAL_TREES                     1
$No_HINGE_POINTS                       2
$No_CHECK_POINTS                       3
END CONDENSED STRUCTURE OF GENERAL DECOMPOSITION
```

Table 6.2: *Results of Program* `acomp01.c` *(Part I)*

```
BEGIN BIBLOCK DECOMPOSITION
REDUCED STRUCTURE
$GRAPH Ugraph1
$TYPE  UGSLF
$No_VERTICES                40
$No_EDGES                   44
$No_ARCS                     0
$No_ISOLATED_VERTICES        1
$No_A-ACYCLIC_WEAK_COMPONENTS       1     (5V, 4E, 0A)
$No_A-CYCLIC_WEAK_COMPONENTS        1     (34V, 40E, 0A)
   $A-CYCLIC_WEAK_COMPONENT Ugraph1.WCOMP1   (34V, 40E, 0A, 5AP, 2BP, 3CP, 2HP)
   $No_PERIPHERAL_TREES 2
        $PERIPHERAL_TREE Ugraph1.WCOMP1.PT0  (5V, 4E, 0A)
        $PERIPHERAL_TREE Ugraph1.WCOMP1.PT1  (2V, 1E, 0A)
        $STOPFREE_KERNEL Ugraph1.WCOMP1.STP  (29V, 35E, 0A, 5AP, 2BP, 3CP, 2HP)
        $No_SUBCOMPONENTS 3
            $SUBCOMPONENT Ugraph1.WCOMP1.SUB0     (23V, 27E, 0A, 2AP, 1BP, 1CP, 2HP)
            $No_BIBLOCKS 3
                $BIBLOCK Ugraph1.WCOMP1.SUB0.BLB0          (3V, 3E, 0A, 1AP, 0BP, 0CP, 1HP)
                $BIBLOCK Ugraph1.WCOMP1.SUB0.BLB1          (4V, 4E, 0A, 1AP, 1BP, 1CP, 1HP)
                $BIBLOCK Ugraph1.WCOMP1.SUB0.BLB2          (18V, 20E, 0A, 2AP, 1BP, 1CP, 2HP)
            $SUBCOMPONENT Ugraph1.WCOMP1.SUB1     (3V, 3E, 0A, 2AP, 1BP, 1CP, 0HP)
            $No_BIBLOCKS 1
                $BIBLOCK Ugraph1.WCOMP1.SUB1.BLB0          (3V, 3E, 0A, 2AP, 1BP, 1CP, 0HP)
            $SUBCOMPONENT Ugraph1.WCOMP1.SUB2     (3V, 3E, 0A, 1AP, 0BP, 1CP, 0HP)
            $No_BIBLOCKS 1
                $BIBLOCK Ugraph1.WCOMP1.SUB2.BLB0          (3V, 3E, 0A, 1AP, 0BP, 1CP, 0HP)
        $No_INTERNAL_TREES 1
            $INTERNAL_TREE Ugraph1.WCOMP1.IT0    (3V, 2E, 0A, 3AP, 1BP, 3CP, 1HP)
END REDUCED STRUCTURE
```

Table 6.3: *Results of Program* `acomp01.c` *(Part IIa)*

```
BIBLOCK DECOMPOSITION
PRINT OPTION VPA  (Undirected loops are counted twice!)
$GRAPH Ugraph1
$TYPE UGSLF
$No_VERTICES 40
$No_EDGES 44
$No_ARCS 0
VERTICES

d0               $TDG 3    $DG 3    $ODG 0    $IDG 0    Ugraph1.WCOMP1
   Check Point
   $NO_BIBLOCK_EDGES                     2  (Ugraph1.WCOMP1.SUB1.BLB0)
                _d0*d1
                _d0*d2
   $NO_INTERNAL_TREE_EDGES               1  (Ugraph1.WCOMP1.IT0)
                _c0*d0

d2               $TDG 3    $DG 3    $ODG 0    $IDG 0    Ugraph1.WCOMP1
   Border Point
   $NO_PERIPHERAL_TREE_EDGES             1  (Ugraph1.WCOMP1.PT1)
                _d2*g
   $NO_BIBLOCK_EDGES                     2  (Ugraph1.WCOMP1.SUB1.BLB0)
                _d0*d2
                _d1*d2

f0               $TDG 3    $DG 3    $ODG 0    $IDG 0    Ugraph1.WCOMP1
   Check Point
   $NO_BIBLOCK_EDGES                     2  (Ugraph1.WCOMP1.SUB2.BLB0)
                _f0*f1
                _f0*f2
   $NO_INTERNAL_TREE_EDGES               1  (Ugraph1.WCOMP1.IT0)
                _c0*f0

b0               $TDG 4    $DG 4    $ODG 0    $IDG 0    Ugraph1.WCOMP1
   Hinge Point
   $NO_BIBLOCK_EDGES                     2  (Ugraph1.WCOMP1.SUB0.BLB0)
                _b0*b1
                _b0*b2
   $NO_BIBLOCK_EDGES                     2  (Ugraph1.WCOMP1.SUB0.BLB2)
                _b0*a3
                _b0*a5

c0               $TDG 7    $DG 7    $ODG 0    $IDG 0    Ugraph1.WCOMP1
   Border Point  Check Point  Hinge Point
   $NO_PERIPHERAL_TREE_EDGES             1  (Ugraph1.WCOMP1.PT0)
                _c0*e0
   $NO_BIBLOCK_EDGES                     2  (Ugraph1.WCOMP1.SUB0.BLB1)
                _c0*c1
                _c0*c3
   $NO_BIBLOCK_EDGES                     2  (Ugraph1.WCOMP1.SUB0.BLB2)
                _c0*a12
                _c0*a4
   $NO_INTERNAL_TREE_EDGES               2  (Ugraph1.WCOMP1.IT0)
                _c0*d0
                _c0*f0
```

Table 6.4: *Results of Program* `acom01.c` *(Part III)*

Example 6.2 Sometimes it is useful to aply the biblock decomposition to a subgraph found in a previous decomposition, for instance to a strong component or to an external dag. Program `acom02.c` (see table 6.5) shows how this is done. We consider graph *wcompgraph*, figure 5.1, page 59. This graph is read using `readgraphlist` and then its weak and strong components are found (`gcomponents`) and printed (`gprstd`). This decomposition is not shown in this manual but we find from it that the last (large) weak component is `WCOMP12`. Using function `generatefromcomp`, page 47, a new graph, termed *wcompgraph12*, is generated from this weak component. Its decomposition into weak and strong components and its biblock decomposition is calculated and printed, tables 6.6 and 6.7. Of course, these decompositions of *wcompgrap12* are equivalent to the decompositions of `WCOMP12` in the original graph. Within *wcompgraph12* the largest strong component is `SCOMP3`. From it again a new graph, termed *wcompgraph12.3*, is generated. Its biblock decomposition is found and printed, table 6.8.

The biblock decomposition of a strongly connected general graph is found considering exclusively a-paths. However, it reveals also some f-path properties. For instance, an edge lies on a f-circuit if and only if it is element of some biblock. □

```
/*************************************************************/
/*     Program  main.                                        */
/*                                                           */
/*     Generates the subgraph correponding to a weak         */
/*     component and the one corresponding to one of its     */
/*     strong components.                                    */
/*     The biblock decomposition is applied to that strong   */
/*     component.                                            */
/*************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

int main()
{
  GRAPH      *ograph, *graph, *stronggraph;

  ograph = readgraphlist(NULL);
  gcomponents(ograph);
  gprstd(ograph, STRR, NULL);
  graph = generatefromcomp(ograph, "wcompgraph.WCOMP12", "wcompgraph12");
  if (graph == NULL)
    { printf("Connot generate subgraph\n");
      exit(0);
    }
  astd(graph);
  gprstd(graph, STRR, NULL);
  aprstd(graph, STRR, NULL);
  stronggraph = generatefromcomp(graph, "wcompgraph12.WCOMP0.SCOMP3",
                 "wcompgraph12.3");
  astd(stronggraph);
  aprstd(stronggraph, STRR, NULL);
  return 0;
}
```

Table 6.5: *Program* `acomp02.c`

```
$GRAPH wcompgraph12
$TYPE  GG
$No_VERTICES                20
$No_EDGES                   11
$No_ARCS                    14
$No_ISOLATED_VERTICES        0
$No_WEAK_COMPONENTS                                          1
$No_A-ACYCLIC_WEAK_COMPONENTS_WITHOUT_STRONG_COMPONENTS      0
$No_A-ACYCLIC_WEAK_COMPONENTS_WITH_STRONG_COMPONENTS         0
$No_A-CYCLIC_WEAK_COMPONENTS_WITHOUT_STRONG_COMPONENTS       0
$No_A-CYCLIC_WEAK_COMPONENTS_WITH_STRONG_COMPONENTS          0
    $(ALL_STRONG_COMPONENTS_F-ACYCLIC)
$No_A-CYCLIC_WEAK_COMPONENTS_WITH_STRONG_COMPONENTS          1     (20V, 11E, 14A)
    $(F-CYCLIC_STRONG_COMPONENTS_EXIST)
   $WEAK_COMPONENT wcompgraph12.WCOMP0  (20V, 11E, 14A)  aper = 1
      rooted   root: wcompgraph12.WCOMP0.SCOMP2 (strong component)
   $No_STRONG_COMPONENTS                                     5
   $No_f-ACYCLIC_STRONG_COMPONENTS                           2
      $STRONG_COMPONENT wcompgraph12.WCOMP0.SCOMP0  (2V, 1E, 0A)  (1WAP) lv = 2  fper = 2
      $STRONG_COMPONENT wcompgraph12.WCOMP0.SCOMP1  (2V, 1E, 0A)  (1WAP) lv = 2  fper = 2
   $No_f-CYCLIC_STRONG_COMPONENTS                            3
      $STRONG_COMPONENT wcompgraph12.WCOMP0.SCOMP2  (1V, 1E, 0A)  (1WAP) lv = 0  fper = 1
      $STRONG_COMPONENT wcompgraph12.WCOMP0.SCOMP3  (14V, 8E, 9A)  (4WAP) lv = 1  fper = 1
      $STRONG_COMPONENT wcompgraph12.WCOMP0.SCOMP4  (1V, 0E, 1A)  (1WAP) lv = 2  fper = 1
      $EXTERNAL_DAG wcompgraph12.WCOMP0.EXD  (8V, 0E, 4A)  (8WAP)
END REDUCED STRUCTURE
```

Table 6.6: *Results of Program* `acomp02.c` *(Part I)*

```
BEGIN BIBLOCK DECOMPOSITION
REDUCED STRUCTURE
$GRAPH wcompgraph12
$TYPE  GG
$No_VERTICES                20
$No_EDGES                   11
$No_ARCS                    14
$No_ISOLATED_VERTICES        0
$No_A-ACYCLIC_WEAK_COMPONENTS       0     (0V, 0E, 0A)
$No_A-CYCLIC_WEAK_COMPONENTS        1     (20V, 11E, 14A)
   $A-CYCLIC_WEAK_COMPONENT wcompgraph12.WCOMP0  (20V, 11E, 14A, 9AP, 4BP, 5CP, 1HP)
   $No_PERIPHERAL_TREES 4
        $PERIPHERAL_TREE wcompgraph12.WCOMP0.PT0  (3V, 1E, 1A)
        $PERIPHERAL_TREE wcompgraph12.WCOMP0.PT1  (3V, 2E, 0A)
        $PERIPHERAL_TREE wcompgraph12.WCOMP0.PT2  (3V, 1E, 1A)
        $PERIPHERAL_TREE wcompgraph12.WCOMP0.PT3  (2V, 1E, 0A)
        $STOPFREE_KERNEL wcompgraph12.WCOMP0.STP  (13V, 6E, 12A, 9AP, 4BP, 5CP, 1HP)
        $No_SUBCOMPONENTS 4
            $SUBCOMPONENT wcompgraph12.WCOMP0.SUB0     (3V, 1E, 2A, 2AP, 1BP, 1CP, 0HP)
            $No_BIBLOCKS 1
                $BIBLOCK wcompgraph12.WCOMP0.SUB0.BLB0          (3V, 1E, 2A, 2AP, 1BP, 1CP, 0HP)
            $SUBCOMPONENT wcompgraph12.WCOMP0.SUB1     (1V, 0E, 1A, 1AP, 0BP, 1CP, 0HP)
            $No_BIBLOCKS 1
                $BIBLOCK wcompgraph12.WCOMP0.SUB1.BLB0          (1V, 0E, 1A, 1AP, 0BP, 1CP, 0HP)
            $SUBCOMPONENT wcompgraph12.WCOMP0.SUB2     (6V, 1E, 7A, 4AP, 2BP, 2CP, 1HP)
            $No_BIBLOCKS 2
                $BIBLOCK wcompgraph12.WCOMP0.SUB2.BLB0          (3V, 0E, 3A, 3AP, 2BP, 1CP, 1HP)
                $BIBLOCK wcompgraph12.WCOMP0.SUB2.BLB1          (4V, 1E, 4A, 2AP, 1BP, 1CP, 1HP)
            $SUBCOMPONENT wcompgraph12.WCOMP0.SUB3     (1V, 1E, 0A, 1AP, 0BP, 1CP, 0HP)
            $No_BIBLOCKS 1
                $BIBLOCK wcompgraph12.WCOMP0.SUB3.BLB0          (1V, 1E, 0A, 1AP, 0BP, 1CP, 0HP)
        $No_INTERNAL_TREES 2
            $INTERNAL_TREE wcompgraph12.WCOMP0.IT0    (5V, 3E, 1A, 4AP, 1BP, 3CP, 0HP)
            $INTERNAL_TREE wcompgraph12.WCOMP0.IT1    (2V, 0E, 1A, 2AP, 0BP, 2CP, 0HP)
END REDUCED STRUCTURE
```

Table 6.7: *Results of Program* `acomp02.c` *(Part II)*

```
BEGIN BIBLOCK DECOMPOSITION
REDUCED STRUCTURE
$GRAPH wcompgraph12.3
$TYPE  GG
$No_VERTICES                14
$No_EDGES                    8
$No_ARCS                     9
$No_ISOLATED_VERTICES        0
$No_A-ACYCLIC_WEAK_COMPONENTS       0     (0V, 0E, 0A)
$No_A-CYCLIC_WEAK_COMPONENTS        1     (14V, 8E, 9A)
   $A-CYCLIC_WEAK_COMPONENT wcompgraph12.3.WCOMP0  (14V, 8E, 9A, 5AP, 2BP, 2CP, 1HP)
   $No_PERIPHERAL_TREES 2
        $PERIPHERAL_TREE wcompgraph12.3.WCOMP0.PT0  (3V, 2E, 0A)
        $PERIPHERAL_TREE wcompgraph12.3.WCOMP0.PT1  (2V, 1E, 0A)
        $STOPFREE_KERNEL wcompgraph12.3.WCOMP0.STP  (11V, 5E, 9A, 5AP, 2BP, 2CP, 1HP)
        $No_SUBCOMPONENTS 2
            $SUBCOMPONENT wcompgraph12.3.WCOMP0.SUB0     (3V, 1E, 2A, 1AP, 0BP, 1CP, 0HP)
            $No_BIBLOCKS 1
                $BIBLOCK wcompgraph12.3.WCOMP0.SUB0.BLB0          (3V, 1E, 2A, 1AP, 0BP, 1CP, 0HP)
            $SUBCOMPONENT wcompgraph12.3.WCOMP0.SUB1     (6V, 1E, 7A, 3AP, 1BP, 1CP, 1HP)
            $No_BIBLOCKS 2
                $BIBLOCK wcompgraph12.3.WCOMP0.SUB1.BLB0          (3V, 0E, 3A, 3AP, 1BP, 1CP, 1HP)
                $BIBLOCK wcompgraph12.3.WCOMP0.SUB1.BLB1          (4V, 1E, 4A, 1AP, 0BP, 0CP, 1HP)
        $No_INTERNAL_TREES 1
            $INTERNAL_TREE wcompgraph12.3.WCOMP0.IT0    (4V, 3E, 0A, 3AP, 1BP, 2CP, 0HP)
END REDUCED STRUCTURE
```

Table 6.8: *Results of Program* `acomp02.c` *(Part III)*

Chapter 7

Distances (*not yet released*)

Not yet released

7.1 Problem Description

In this chapter functions for distances are presented. We start considering a-paths (f-paths, b-paths) from vertex u to vertex v. The length of such a path is defined as the a-distance (f-distance, b-distance) from u to v *along that path*. If a path is not mentioned, distance means the length of a shortest path where shortest is always understood as minimal number of lines. It is easy to determine all shortes paths from a vertex u to a different vertex v. An a-path (f-path, b-path) from u to v is called *direct*, if it is either a shortest path or none of its inner vertices lies on a shorter direct path.

to be updated
In this section the distance structure of the giant component of `words.dat` , especially the distance structure of the giant biblock are analyzed. As usual, the distance between two vertices a and b of the same component is the minimum length of the paths from a to b. The length of a path is its number of edges, length 0 is allowed.

In general, there is more than one shortest path between two vertices. The vertices and edges of the shortest paths form a subgraph, the *distance graph with respect to a and b*. Figure 7.1 shows the distance graphs (`point`, `block`), (`point`, `check`), (`point`, `trees`) and (`point`, `hinge`) combined in a single drawing. In principle, the distance graph with respect to a and b is independent of the order of these vertices. It is convenient, however, to impose a direction on the distance graph by fixing one of the two vertices as starting point of all shortest paths. For reasons which shortly will become clear, we call it the *center* of the distance graph, whereas the other vertex is called *goal*[1].

The direction of the distance graph assigns a unique *level number* to each of its vertices. The center has level 0, the goal has level d, where d is the distance from a vertex of level i $(0 < i < d-1)$ to a vertex of level $i + 1$.

7.2 Formats and Data Structures

For a general description and basic data types see section 2.2, page 11.

[1] Knuth uses the names *start* and *goal*.

7.3 Functions

- shortpath
- distances
- fdistances

7.4 Examples

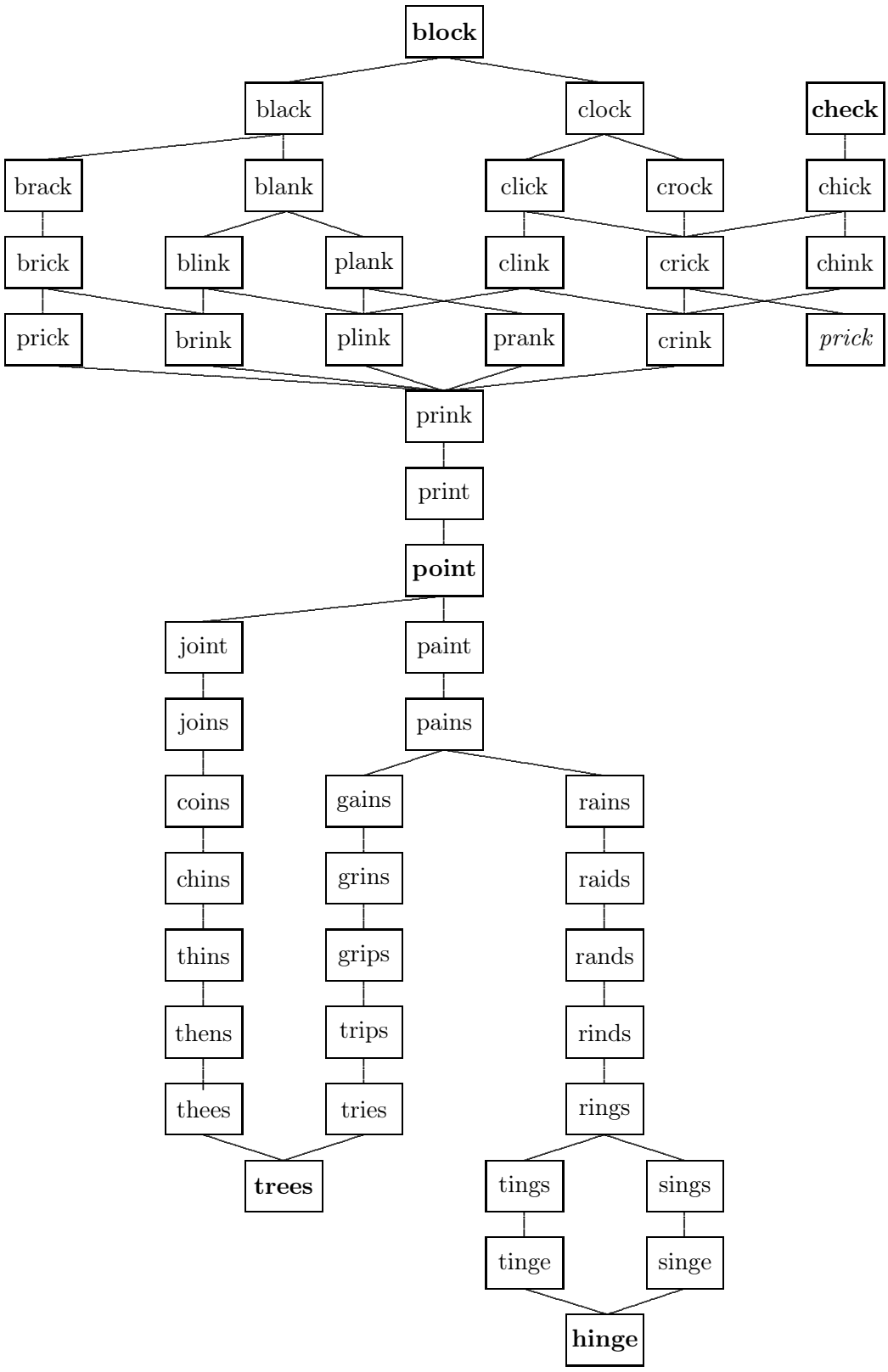


Figure 7.1: *Combined distance graphs (`point`, `block`), (`point`, `check`), (`point`, `trees`), (`point`,*

Chapter 8

Edge and Vertex Partitions *(not yet released)*

In this chapter only simple graphs, i. e. graphs of type `UGSLF` *are considered.*

8.1 Problem Description

Strictly speaking, a family of edge sets of a graph is an edge partition if the sets are pairwise disjoint and their union is the complete set of edges. However, it is more expedient to allow general sets of edges. `GHS` knows two types of edge partitions:

1. *Strict partitions:* The edge sets are pairwise disjoint.

2. *General partitions:* The edge sets may have elements in common.

In no case the union of the edge sets is required to contain all edges of the graph. We shall deal with non-empty edge sets only.

The edge sets will be called *classes*. Vertices (edges) common to two or more classes are *attachment points (attachment edges)*. The basic example is the partition generated by a set W of *limiting vertices* and by a set F of *limiting edges*, W and F not both empty. Two edges belong to the same class if the following holds: *There is a path through both edges with not internal vertex in W and no edge in F. The path must start in a vertex in W or in an end point of an edge in F.*[1] This partition is strict. In a non-connected graph the union of its classes may be a proper subset of the set of all edges, however. For more details see [Stie1998]. Function `edpartgen` constructs such a partition. The case where F is the set of edges of a family of paths and W the set of end points of the path edges is important when determining Menger structures (see chapter ?

A second example is the edge partition which results from another edge partition, when we delete from the first partition all 1-class edges where both vertices also belong to the same multi-edge class and join the edge to all multi-edge classes with this property. The resulting partition may have non-disjoint classes, i.e attachment edges. It is a general partition. Function `cppartition`, among other options, yields partitions of that type. They are used, for instance, in determining the k-components of a graph.

[1] Non-simple paths are allowed and there are cases where two edges of a class are joinable by non-simple paths only.

Sometimes a strict edge partition is given by the application problem. In such cases, all edges of a class are 'painted' with the same color. If there is an external C function yielding the color of an edge, GHS is able to construct the corresponding edge partition with function `paint2part`. Function `paint2cpart` constructs the partition of the 'color-connected' classes of the original painting. In addition to the 'normal' colors of a painting, the external function may yield the optional color 'invisible'. If this is the case, the corresponding edge is not included in any class. For more details see example 8.1, page 85.

Function `add2edpart` which adds a new class to a general partition, function `add2class` which adds a new edge to a class of a general partition, function `releaseedpartlist` which deletes a list of partitions, function `generatefromclass` which generates a new graph form an edge class, and printing functions `prpartstr` and `prpartvted` complement the set of functions.

8.2 Formats and Data Structures

For a general description see section 2.2, page 11.
The data structures used with general edge partition functions are listed in subsection C.10, page 183. Records of type `EDPART` describe partitions, records of type `EDCLASS` describe edge classes. The partition specific properties of a vertex are recorded in an instance of type `PARTVT`, for edges type `PARTED` is used. To represent records of these data types in different additional roles the data types `REDCLASS`, `RPARTV`, and `RPARED` are used.

8.3 Functions

8.3.1 edpartgen

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `EDPART *edpartgen(GRAPH *graph, RVERTEX *vtlist, REDGE *edlist)`

Description: Builds the strict edge partition generated from a list of limiting vertices `vtlist` and a list of limiting edges `edlist`. Two edges belong to the same class if both lie on a path with no internal vertex from `vtlist` and no edge from `edlist`. The path must start in a vertex from `vtlist` or in an endpoint of an edge from `edlist`. Creates an `EDPART` record and complementary records.

Error Exits: Graph pointer NULL. Graph not of type UGSLF. Vertex list and edge list, both empty.

Remarks: None

Examples:

8.3.2 cppartition

8.3.3 paint2part

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `EDPART *paint2part(GRAPH *graph, int pclrno)`

Description: Assumes a function `int paintval(EDGE *ed)` yielding for every edge of the graph a valid color $0 \leq$ `pcrl` $<$ `pclrno` or the color -1 meaning 'invisible'.
Builds the strict edge partition generated from a list of limiting vertices `vtlist` and a list of limiting edges `edlist`. Two edges belong to the same class if both lie on a path with no internal vertex from `vtlist` and no edge from `edlist`. The path must start in a vertex from `vtlist` or in an ednpoint of an edge from `edlist`. Creates an `EDPART` record and complementary records.

Error Exits: Graph pointer NULL. Graph not of type UGSLF. Vertex list and edge list, both empty.

Remarks: None

Examples:

8.3.4 paint2cpart

8.3.5 add2edpart

8.3.6 add2class

8.3.7 releaseedpartlist

8.3.8 generatefromclass

8.3.9 prpartstr

8.3.10 prpartvted

8.4 Examples

Example 8.1 *To be completed*

Chapter 9

Paths *(not yet released)*

9.1 Problem Description

Paths have been introduced in chapter 5, page 53. In the present chapter, paths are GHS objects in their own right. They are represented by specific data structures and may be manipulated by specific GHS functions.

Survey of Functions:

A path header for a new path is created by `newphdr`. A line is added to a path with `insertline2path`. It is removed by `removelinefrompath`. Function `insertpath2path` inserts a path into another path. A path is read from a file or standard input with `readpath`, it is written to a file or standard output by `savepath`. If several paths are to be read or written functions `readpathlist`, respectively `savepathlist` are to be used. The lists are organized as red-black trees. Function `releasepathlist` is used to release a list of paths. Function `printpath` or `printpathlist` is used to print paths

9.2 Formats and Data Structures

For a general description and basic data types see section 2.2, page 11.

The special data structures corresponding to paths are described in subsection C.11, page 183. A path header (type `PHDR`) describes a path, a member record (type `PTHA`) stands for each occurrence of a line in the path. As shown in figure 9.1, the member records are organized as a

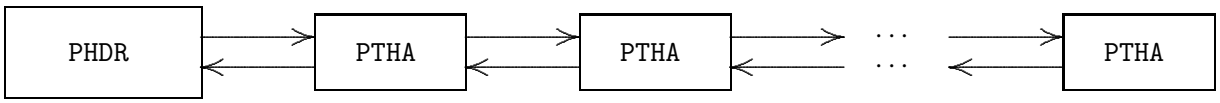


Figure 9.1: *Paths as linked lists*

doubly linked list. The list indicates the path direction. The field `pddir` in the header indicates the allowed direction of lines in the paths, i.e. whether we have a-paths, f-paths or b-paths.

External file format: Files which represent paths in external GHS format are organized as shown in table 9.1.

`$PATH` <*pathname*>	Each path must have a name, which is a string not starting with '$' or '_'.
`$GRAPH` <*graphname*>	Each path must belong to a graph. The name of the graph is recorded here. The name of the graph is a string not starting with '$' or '_'.
`$DIRECTION` <*dirvalue*>	`A` for "any". `F` for "forward". `B` for "backward".
`$POSITION` <*posvalue*>	`S` for "start". `E` for "end".
`$PATHLINES` <*plinelist*>	List of path line entries. May be empty. A path line entry consists of the name of the line, followed by the name of the first incidence vertex in path direction. The name of a line is a string which must start with '_'. The name of a vertex is a string which must not start with '$' or '_'.
`$END`	

Table 9.1: *External File Format of GHS Graphs*

9.3 Functions

9.3.1 newphdr

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `PHDR *newphdr(GRAPH *graph, char *direction, char* name)`

Description: Creates a new path header for a path in `graph`.
Parameter `direction` specifies whether the path is an a-paths, f-paths or b-path.

"A" or "a":	a-path; arcs may be traversed in any direction
"F" or "f":	f-path; arcs must be traversed in forward direction only
"B" or "b"	b-path; arcs must be traversed in backward direction only

In all cases edges may be traversed in any direction. When lines are inserted into a path by GHS, consistency with parameter `direction` *is checked.*
`name` is a user-specific name for the path. If no name is provided GHS uses "dummy".

Error Exits: The function returns NULL if an error occurs. Errors are: Graph pointer NULL, incorrect parameter value.

Remarks: Additonal path properties like line-simple, open, circuit and so on, are neither recorded nor checked.

Examples:

9.3.2 insertline2path

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `PTHA *insertline2path(PHDR *phdr, EDGE *line, VERTEX *first, char *position)`

Description: Inserts the line given by `line` into the the path specified by `phdr`. Vertex `first` is the first vertex of `line` in path direction. The function checks whether the conditions of a correct path are fulfilled. Parameter `position` controls where `line` is inserted:

"start" (or "S" or "s"): inserted as new first line of the path
"end" (or "E" or "e"): inserted as last line of the path

The `PTHA` record of the inserted line is returned. NULL if an error occurred.

Error Exits:

Remarks: Take care to not confound options "start" and "end" with path modes "any", "forward", and "backward".

Examples:

9.3.3 removelinefrompath

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `PTHA *insertline2path(PHDR *phdr, EDGE *line, VERTEX *first, char *position)`

9.3.4 insertpath2path

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `BOOLEAN insertpath2path(PHDR *phdr1, PHDR phdr2)`

Description: Inserts the lines of the path given by `phdr2` into the path given by `phdr1`. Path `phdr1` is extended, path `phrd2` remains unchanged.
1. If the last vertex of `phdr1` and the first vertex of `phdr2` are identical, then the lines of `phdr2` follow `phdr1` in the resulting path.
2. If the first vertex of `phdr1` and the last vertex of `phdr2` are identical then the lines of `phdr2` precede `phdr1` in the resulting path
3. If neither 1. nor 2. hold, then `phdr2` must be a closed path and its starting point must occur in `phdr1`. The lines of `phdr2` are inserted into `phdr1` following the first occurrence of that vertex.

The function returns `TRUE` if the insertion was correct. It returns FALSE if an error occurred.

Error Exits:

Remarks: 1. It may be that both conditions 1. and 2. above hold. In that case the result is as described in condition 1.
2. `phdr1` and `phdr2` must be valid header records and they must refer to the same graph. However, it is not necessary that these paths have positive lengths. The following cases are possible:

length `phdr1`	*length* `phdr2`	*result*
0	0	no effect
positive	0	no effect
0	positive	second path is copied

Examples:

9.3.5 readpath

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `PHDR *readpath(GRAPH *graph, char *filename)`

Description: Reads an external path description (Table 9.1) from file `filename`. If `filename` equals NULL, the path description is read from standart input. Using vertices and lines from graph `graph`, an internal path description (path header and path menbers) is created. A pointer to the path header is returnend by the function. Returns NULL if an error ocurred.

Lines are added to the path being contructed at the start if $POSITION indicates `S`. They are added at the end if $POSITION indicates `E`.

Error exits:

Remarks: 1. The path maybe of length 0, i.e. plinelist (see table 9.1, page 88) is empty.

9.3.6 readpathlist

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `PHDR *readpathlist(GRAPH *graph, char *filename)`

Description: Reads a sequence of external path descriptions (Table 9.1) from file `filename`. If `filename` equals NULL, the path descriptions are read from standart input. Using vertices and lines from graph `graph`, for each path in external format an internal path description (path header and path members) is created. The path header is inserted into a list. The list is organized as red-black tree sorted by path name. A pointer to the list is returnend by the function. Returns NULL if an error ocurred.

Lines are added to the actual path being contructed at the start if $POSITION indicates `S`. They are added at the end if $POSITION indicates `E`.

Error exits:

Remarks: 1. All external paths descriptions must refer to the same graph.
2. A path may be of length 0, i.e. plinelist (see table 9.1, page 88) is empty.

9.3.7 savepathlist

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void printpathlist(PHDR *phdr, char *option, char *filename)`

9.3.8 printpathlist

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void printpathlist(PHDR *phdr, char *option, char *filename)`
Option must be one of the values `"short"` or `"detailed"`.

Description: Writes a list of paths (organized as red-black tree) to file `filename`. If `filename` equals NULL the output is written to standard output. For each path the name, the mode, the directon, and the length are written. With options `"detailed"` a complete list of all lines and vertices of the paths is written, too.

Error Exits: None.

Examples:

9.3.9 releasepathlist

Program Author: Günther Stiege, Universität Oldenburg

9.3.10 simplifypath

Program Author: Günther Stiege, Universität Oldenburg

9.3.11 generategraphfrompath

Program Author: Günther Stiege, Universität Oldenburg

9.4 Examples

Example 9.1 In this example we again consider Graph1 from page 18. Program `path01.c` of table 9.2 shows how different GHS functions can be used to construct from Graph1 an a-path through vertices K00, K13, K11, K12, K13, K11, K06, K00. As a first step subpath K00, K13, K11, K06, K00 is built from its external representation using function `readgraph`. The external representation is shown in the upper part of table 9.3, followed by the resulting subpath as output by the program.

In a second step the closed subpath K11, K12, K13, K11 is built. To this end a new path header is explicitly created and the three lines individually added using function `insertline2path`. The resulting subpath is output and again shown in table 9.3.

The final step consists in again creating a new path header and inserting into this path the first and the second subpath using function `insertpath2path`. The finally resulting path is shown in table 9.3, too.

```
/*************************************************************/
/*     Program  main.                                        */
/*     Example for path functions                            */
/*************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

int  main(int argc, char **argv)
{
  GRAPH      *graph;
  PHDR       *phdr, *phdr1, *phdr2;
  EDGE       *line;

    graph = readgraphlist(argv[1]);
    if (graph ==  NULL)
     { printf("Incorrect graph input\n");
       exit(0);
     };
    phdr1 = readpath(graph, NULL);
    if (graph == NULL)
      { printf("Error when reading paths");
        exit(0);
      }
    printpathlist(phdr1, "detailed", NULL);
//
    phdr2 = newphdr(graph, "A", "subpath2");
    line = (EDGE *)rbtreefind((RB *)(graph->grdedlist), (RB *)"_d*K12*K11", SEN);
    insertline2path(phdr2, line, line->edsec, "E");
    line = (EDGE *)rbtreefind((RB *)(graph->grdedlist), (RB *)"_d*K12*K13", SEN);
    insertline2path(phdr2, line, line->edfirst, "E");
    line = (EDGE *)rbtreefind((RB *)(graph->gredlist), (RB *)"_u*K11*K13", SEN);
    insertline2path(phdr2, line, line->edsec, "E");
    printpathlist(phdr2, "detailed", NULL);
//
    phdr = newphdr(graph, "A", "mainpath");
    insertpath2path(phdr, phdr1);
    insertpath2path(phdr, phdr2);
    printpathlist(phdr, "detailed", NULL);
    return 0;
}
```

Table 9.2: *Program* `path01.c`

```
file path01.dat:
$PATH      subpath1
$GRAPH     Graph1
$DIRECTION A
$POSITION  E
$PATHLINES _d*K00*K13     K00
           _u*K11*K13     K13
           _d*K11*K06     K11
           _d*K06*K00     K06
$END

results of program path01:
BEGIN PATH LIST
PATH subpath1
PATH DIRECTION     A (any)
PATH LENGTH        4
       _d*K00*K13   K00   K13
       _u*K11*K13   K13   K11
       _d*K11*K06   K11   K06
       _d*K06*K00   K06   K00
END PATH LIST (detailed)

BEGIN PATH LIST
PATH subpath2
PATH DIRECTION     A (any)
PATH LENGTH        3
       _d*K12*K11   K11   K12
       _d*K12*K13   K12   K13
       _u*K11*K13   K13   K11
END PATH LIST (detailed)

BEGIN PATH LIST
PATH mainpath
PATH DIRECTION     A (any)
PATH LENGTH        4
       _d*K00*K13   K00   K13
       _u*K11*K13   K13   K11
       _d*K11*K06   K11   K06
       _d*K06*K00   K06   K00
END PATH LIST (detailed)
```

Table 9.3: *Input and output of program path01*

Chapter 10

Menger Structures *(not yet released)*

10.1 Problem Description

The name of Menger theorems is given to several similar graph theoretical results: Let u and v be distinct and non-adjacent vertices of a general graph. Then

1. The maximum number of internally vertex-disjoint a-paths[1] from u to v equals the minimum number of vertices a-separating u and v.

2. The maximum number of internally vertex-disjoint f-paths from u to v equals the minimum number of vertices f-separating u and v.[2]

3. The maximum number of line-disjoint a-paths from u to v equals the minimum number of lines a-separating u and v.

4. The maximum number of line-disjoint f-paths from u to v equals the minimum number of lines f-separating u and v.[2]

In the case of line-disjoint paths vertices u and v are allowed to be adjacent.
Neither the paths nor the separating vertex (line) sets are uniquely determined. We call such paths a *system of Menger paths* and the vertex (line) sets *Menger separating sets*. It can always be assumed that the paths are simple.

For a given type (a-path/f-path, internally vertex-disjoint/line-disjoint) a simple path from u to v is called a *Menger path* if it can be complemented by other simple paths such that a system of Menger paths of the given type results. A single vertex (line) is called a *Menger vertex (Menger line)* for u and v if it can be complemented by other vertices (lines) such that a Menger a-separating set (Menger f-separating set) for u and v results.

Menger vertices and Menger lines have nice ordering properties. For two Menger vertices x and y we define $\boldsymbol{x \preceq_{uv} y}$ ("x lies before y in direction from u to v") if $x = y$ or if there is a Menger path on which x comes before y in direction from u to v. This definition holds without changes for Menger lines. In most cases u and v are clear from the context and the subscript $_{uv}$ can be omitted. It turns out that $\preceq$ is a partial ordering on the set of Menger vertices (Menger lines). For given vertices u and v, a given type, and a given Menger vertex (line) x there exists a unique Menger set $(x, x_1^\circ, x_2^\circ, \ldots, x_{k-1}\circ)$ such that for every Menger set $(x, x_1, x_2, \ldots, x_{k-1})$ it is true

[1] The paths have no inner vertices in common.

[2] Note that a set of vertices (set of lines) f-separating u and v need not be a set f-separating v and u.

that $x_j^\circ \preceq x_j$ for $j = 1, 2, \ldots k-1$[3]. We call $(x, x_1^\circ, x_2^\circ, \ldots, x_{k-1}^\circ)$ the *minimal Menger separating set corresponding to* x

Sets of Vertices as Sources and Sinks:

The Menger theorems still hold if instead of vertices u and v nonempty vertex sets U and W are considered. This remains true even if elements in U are W are adjacent or if the sets have common elements. In these cases mixed separating sets containing both, vertices and lines have to be admitted. Sometimes it is of interest to find all a-paths (f-paths) from U to W which start in U, end in W and have no vertices at all in common, i.e. *disjoint UW-paths*. This can be achieved by adding two additional vertices to the graph, adding edges joining all vertices in U to the first new vertex and all vertices in W to the second new vertex, and then applying the corresponding Menger algorithm to the new vertices.

For a detailed treatment of Menger structures, including algorithms, see Stiege ([Stie2006] and [Stie2007a]).

Survey of Functions:

The central function for Menger structures is `mengerstr`. It finds a maximal set of simple paths of a specified mode and a specified direction joining a vertex in `VTSOURCE` to a vertex in `VTSINK`. It also finds all Menger vertices (Menger lines) and the corresponding minimal Menger separating sets. Function `mgprstd` is used to output a Menger structure.

10.2 Formats and Data Structures

For a general description and basic data types see section 2.2, page 11.

The data structures specific for Menger structures are described in section C.12, page 184. A Menger descriptor (type `MGDESCR`) describes the sets `VTSOURCE` and `VTSINK` and the mode and the direction of the Menger structure found. It points to a path descriptor (type `PDESCR`) where the Menger paths are recorded. It also points to a separating descriptor (type `SEPDESCR`) where the details of the separating structure are recorded.

10.3 Functions

10.3.1 mengerstr

Program Author: Günther Stiege, Universität Oldenburg

Syntax:

```
MGDESCR *mengerstr(GRAPH *graph, VTSET *vtsource, VTSET *vtsink,
                   char *mode, char direction, char *name, int bound, char
                   *sep)
```

[3]To identify x_j by the index j we chose an arbitrary Menger paths system $P_0, P_1, P_2, \ldots, P_{k-1}$ with x on P_0 and x_j on P_j.

Description: Finds a set of maximal cardinality of simple paths joining a vertex in vertex set `vtsource` to a vertex in vertex set `vtsink`. Returns the corresponding Menger descriptor. The kind of paths found depends on parameter `mode`:

`mode` equals "vt" (or "VT"):	the paths are internally disjoint.
`mode` equals "li" (or "LI"):	the paths are line-disjoint
`mode` equals "evt" (or "EVT"):	the graph is temporarily enlarged (see page 96) and "vt" is applied to the larger graph.
`mode` equals "eli" (or "ELI"):	the graph is temporarily enlarged (see page 96) and "li" is applied to the larger graph.

If `direction` equals 'A' (or 'a') a-paths are found whereas `direction` 'F' (or 'f') indicates f-paths. `name` is a user-specific name for the Menger structure.

Parameter `bound` delimits the number of paths:

`bound` < 0 :	*all* paths are found.
`bound` > 0 :	the function returns at most the specified number of paths. *In this case no Menger separating elements are determined*, even if they existed.

Parameter `sep`: Values "first" or "complete". See the following description of paths and separating elements.

Paths and separating elements: Function `mengertstr` yields a complete description of the paths joining `vtsource` to `vtsink` as well as complete description of the elements separating these sets. All paths found are simple. They are also shortest in the sense that only the first vertex is element of `vtsource` and only the last vertex is element of `vtsink`. The paths are described by a path descriptor `PDESC`.

a. `mode` **"vt":**

1. All vertices common to `vtsource` and `vtsink` are separating elements.
2. All lines joining directly a vertex of `vtsource` (not in `vtsink`) to a vertex of `vtsink` (not in `vtsource`) are Menger paths (see the problem description in section 10.1, page 95) and at the same time separating elements.
3. All remaining Menger paths have length at least 2 and pass through one or more Menger vertices. If parameter `sep` equals "complete" all Menger vertices of all these paths are determined together with its minimal Menger separating sets (see section 10.1). If parameter `sep` equals "first" only the first system of Menger sets (in direction from `vtsource` to `vtsink`) is determined.

b. `mode` **"li":**

1. All vertices common to `vtsource` and `vtsink` are separating elements.
2. All Menger paths join a vertex in `vtsource` (not in `vtsink`) to a vertex in `vtsink` (not in `vtsource`) and pass through at least one Menger line. If parameter `sep` equals "complete", for each path all its Menger lines together with their minimal Menger separating sets are determined (see section 10.1). If parameter `sep` equals "first" only the first system of Menger sets (in direction from `vtsource` to `vtsink`) is determined.

c. `mode` **"evt":**

1. All vertices common to `vtsource` and `vtsink` are separating elements.
2. The Menger paths found are vertex-disjoint.
3. All Menger paths join a vertex in `vtsource` (not in `vtsink`) to a vertex in `vtsink` (not in `vtsource`) and pass through at least one Menger vertex. *The starting point and the end points of Menger paths are admitted as Menger vertices, too.* If parameter `sep` equals "complete" all Menger vertices of all paths are determined together with its minimal Menger separating sets (see section 10.1). If parameter `sep` equals "first" only the first system of Menger sets (in direction from `vtsource` to `vtsink`) is determined.

c. `mode` **"eli":**

1. All vertices common to `vtsource` and `vtsink` are separating elements.
2. The Menger paths found are line-disjoint. Their starting points in `vtsource` and their end points in `vtsink` are pairwise distinct. Vertices from `vtsource` and/or vertices from `vtsink` may be internal vertices of the path found.
3. All Menger paths join a vertex in `vtsource` (not in `vtsink`) to a vertex in `vtsink` (not in `vtsource`) and pass through at least one Menger line. If parameter `sep` equals "complete" all Menger lines of all paths are determined together with its minimal Menger separating sets (see section 10.1). If parameter `sep` equals "first" only the first system of Menger sets (in direction from `vtsource` to `vtsink`) is determined.

Error Exits: Graph pointer NULL. Graph not of type UGSLF. Vertex list and edge list, both empty.

Remarks: None

Examples:

10.4 Examples

Example 10.1 *To be completed*

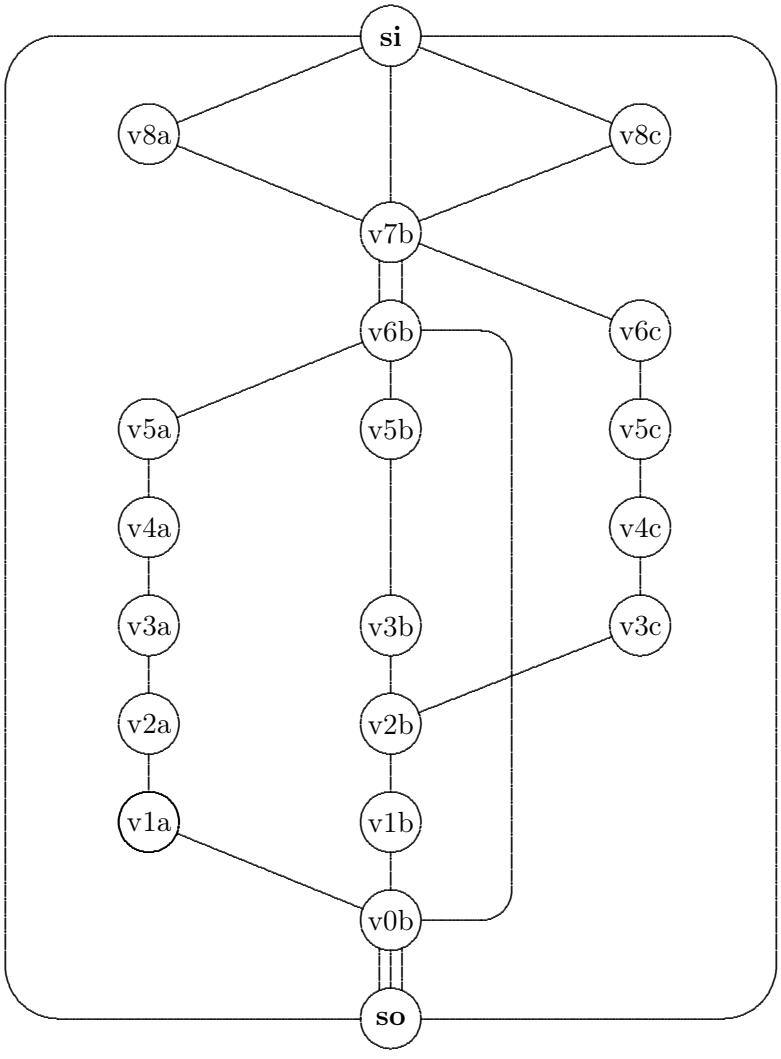


Figure 10.1: *Menger line theorem (graph Menglgraph1)*

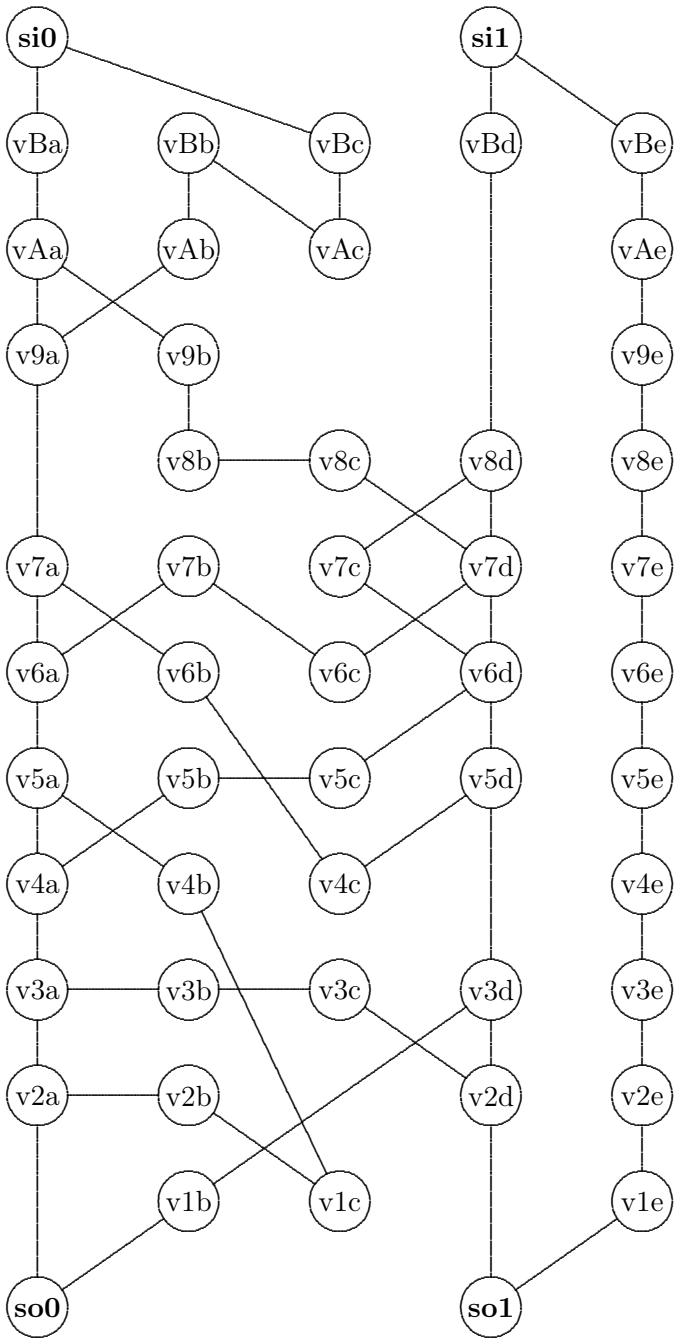


Figure 10.2: *Menger line theorem (graph Menglgraph2)*

Chapter 11

Higher Decompositions *(not yet released)*

11.1 Problem Description

Loosely speaking, a decomposition of a general graph is a family of subgraph whose union is the given graph and which have as few as possible vertices and lines in common. We call a decomposition hierarchical if some of the decomposing subgraphs are decomposed themselves and this procedure is reiterated. Graph decompositions are in general based on reachability properties. Extending the definition of page 53 we say that vertex v is k-a-reachable from vertex u if there are (at least) k internally disjoint a-paths from u to v. We say that v is k-a-linereachable from u if there are (at least) k line-disjoint a paths from u to v. In an analogous manner k-f-reachability is defined. A subgraph is k-a-connected (k-f-connected) if every pair of different vertices is mutually k-a-reachable (k-f-reachable) *in the subgraph.* Maximal k-a-connected (k-f-connected) subgraphs are uniquely determined though not necessarily disjoint. They are called k-a-components (k-f-components) of the graph. In the same way lineconnectedness is defined. Maximal k-a-lineconnected (k-f-lineconnected) subgraphs are uniquely determined and pairwise disjoint. They are called k-a-linecomponents (k-f-linecomponents).

Instead of subgraphs we also may consider sets of vertices. A vertex set of a general graph is called k-a-connected if every pair of different vertices is mutually k-a-reachable by internally disjoint a-paths where the paths are allowed to traverse vertices not in the set. k-f-connected vertex sets as well as lineconnected vertex sets are defined in an analogous manner. Again such sets are uniquely determined if they are maximal.

For more details see Stiege [Stie2006] or[Stie2007a]. The algorithms used by GHS can be found there, too.

Hierarchical Decomposition: Hierarchical decomposition of a general graph is possible in four dimensions due to the following implications:

$$k\text{-connectedness} \Rightarrow (k-1)\text{-connectedness} \tag{11.1}$$

$$\text{(vertex) connectedness} \Rightarrow \text{lineconnectedness} \tag{11.2}$$

$$\text{f-connectedness} \Rightarrow \text{a-connectedness} \tag{11.3}$$

$$\text{vertex set} \Rightarrow \text{subgraph} \tag{11.4}$$

The first three implications mean that every vertex set, respectively every subgraph for which the left side holds also satisfies the right side. The last implication says that the vertex set of a

subgraph has (at least) the connectedness properties of the subgraph.

Decomposition for small k: Every general graph is a 0-a-linecomponent, 0-a-component, 0-f-linecomponent, and a 0-f-component. The 1-a-linecomponents and the 1-a-components of a general graph coincide. They are called the weak components of the graph. The 1-f-linecomponents and the 1-f-components also coincide. They are called the strong components of the graph. GHS functions for determining weak and strong components of a general graph are described in chapter 5.

2-a-linecomponents are different from 2-a-components. The former are called subcomponents and the latter biblocks. See chapter 6 for details. In general graphs, 2-f-linecomponents and 2-f-components also differ.

Survey of Functions:

The central function for higher decompositions of a general graph is `highcomponents`. For a given decomposition type (internally disjoint paths/line-disjoint paths, a-paths/f-paths, subgraphs/vertex sets) it finds the complete hierarchical decomposition according to equation 11.1. It uses heavily Menger paths (see chapter 10) and applies a heuristics called RGB (see the literature cited above). Function `highprst` is used to output de decompostions found.

11.2 Formats and Data Structures

For a general description and basic data types see section 2.2, page 11.
The data structures used with the biblock decomposition functions are described in subsection C.8. The main record to describe a biblock decomposition is of type `STDGD`. It is pointed to from the `grstruct` field in the `GRAPH` record (subsection C.5). The building blocks of a biblock decomposition are represented by the data types `CFR` (acyclic component), `CLC` (cyclic component and stopfree kernel), `SUB` (subcomponent), `BLB` (biblock), `IT` (internal tree), and `PT` (peripheral tree). As a generic element, data type `RELEM` is used. With data type `RVSTD` biblock decomposition properties of a vertex (e. g. kind of attachment point) are recorded. In the same way, records of type `EDSTD` are used for edges. Edges a vertex is incident with may belong to more than one biblock (hinge point). Records of type `INCSQR` indicate the biblocks a vertex is element of. Several enumeration types (`RELEMCLASS, RVSTDCLASS, RVSTDSTAT, EDCLASS, EDSTAT, POPTION, STROPTION`) are used by the biblock decomposition functions, mostly for internal purposes.

Remark 11.1 The proper building blocks of the biblock decomposition (acyclic component, cyclic component, peripheral tree, stopfree kernel, internal tree, subcomponent, biblock) are given unique and permanent names. The name of a building block is not recored in a name field but is the name of one of its edges **(falsch !**) identified by a specific procedure. □

11.3 Functions

11.3.1 highcomponents

Program Author: Günther Stiege, Universität Oldenburg

Examples: Example 6.1, page 70.

11.3.2 prbiblocktrees

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void prbiblocktrees(GRAPH *graph)`

Description: Prints the biblock graph of a simple graph. Only cyclic components are considered. For each cyclic component the corresponding biblock tree is printed.

Error Exits: Graph pointer Null. Graph not of type UGSLF. Biblock decomposition does not exist.

Remarks: The biblock tree of a cyclic component is printed in breadth-first sequence starting with a biblock. The level of each vertex in the biblock tree is indicated as well as the father vertex.

Examples: Example 6.1, page 70.

11.4 Examples

As an example let H be a 5-a-linecomponent of general graph G. From implication 11.1 follows that any 6-a-linecomponent of G is either disjoint to H or a subgraph of H. So, H has no 6-a-lineconnected subgraph or decomposes into 1 or more 6-a-linecomponents and a set of edges/arcs not belonging to any 6-a-linecomponent. If this set is not empty, the subgraph it generates is called the 6-a- of H. From implication 11.2 follows than any 5-a-component of G is either disjoint to H or a subgraph of H. Every 5-a-component in turn decomposes into 0 ore more 6-a-components and possibly a 6-a-cocomponent (implication 11.1). Finally, from implication 11.3 follows that any 5-f-linecomponent of G is either disjoint to H or a subgraph of H. Again, the 5-f-linecomponents may be decomposed further. The decomposition follows the same arguments as for a-decomposition.

Example 11.1 In this example program 5.1 (table 11.1) is used. It is applied to graph Ugraph1 (figure 6.3). The program reads the external description of the graph and constructs it using function `readgraphlist`. Then function `stddecomp` is called to find and construct the biblock decomposition of the graph. Afterwards function `checkstr` performs consistency checks. Next, function `prstd` is called with print option `STRCS` for condensed statistics. The results of these steps are shown in table 11.2.

As the next step function `prstd` is called again, this time with print option `STR` for a complete listing of the biblock decomposition. The listing is shown in tables 11.4, 11.5, 11.6, and 11.7.

With function `prstdvt` a complete list of vertices (option `VPV`) is printed. Only the output of the last two vertices of the list (`b0` and `c0`) is shown in table 11.8.

Finally, function `biblocktrees` is called to find the biblock graph of the biblock decomposition and to print it. The result is shown in table 11.9. □

```
/*************************************************************/
/*     Program  main.                                        */
/*                                                           */
/*     Reads a graph and constructs its biblock              */
/*     decomposition.                                        */
/*     The biblock decomposition is printed.                 */
/*     A list of vertices, ordered by degree and vertex name, */
/*     is printed, too. It shows all relevant information of  */
/*     of each vertex and all edges incident with it.        */
/*     Finally, the biblock graph of the biblock             */
/*     decomposition is computed and printed.                */
/*                                                           */
/*************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

int main()
{
  GRAPH      *graph;

  graph = readgraphlist();

  stddecomp(graph);

  prstd(graph, STRCS);
  prstd(graph, STR);

  prstdvt(graph, VPV);

  prbiblocktrees(graph);
  return 0;
}
```

Table 11.1: *Program 5.1*

```
$Checking biblock decomposition of graph Graph2
$TYPE  UGSLF
              $No_VERTICES 40
              $No_EDGES 44
OK   1: No. isolated vertices correct.
OK   2: Incidences from vertices to edges correct
OK   3: Numbers of vertices and edges in proper components correct.
OK   4: Attachment points correct.
OK   5: Edges correct.
OK   6: SUB/BLB correct.
End of checking biblock decomposition of graph Graph2

BEGIN CONDENSED STRUCTURE
$GRAPH Graph2
$TYPE  UGSLF
$No_VERTICES                40
$No_EDGES                   44
$No_ISOLATED_VERTICES        1
$No_ACYCLIC_COMPONENTS       1     (5V, 4E)
$No_CYCLIC_COMPONENTS        1     (34V, 40E)
$No_PERIPHERAL_TREES         2
$No_STOPFREE_KERNELS         1
$No_SUBCOMPONENTS            3
$No_BIBLOCKS                 5
$No_INTERNAL_TREES           1
$No_ATTACHMENT_POINTS        5
$No_BORDER_POINTS            2
$No_CHECK_POINTS             3
$No_HINGE_POINTS             2
END CONDENSED STRUCTURE
```

Table 11.2: *Results of Program 5.1 (Part I)*

```
BEGIN COMPLETE STRUCTURE
$GRAPH Graph2
$TYPE  UGSLF
$No_VERTICES                40
$No_EDGES                   44
$No_ISOLATED_VERTICES        1
h
$No_ACYCLIC_COMPONENTS       1     (5V, 4E)
   $ACYCLIC_COMPONENT _i0*i1          (5V, 4E)
   $VERTICES
   i0
   i1
   i2
   i3
   i4
   $EDGES
   _i0*i1
   _i1*i1
   _i2*i3
   _i3*i4
$No_CYCLIC_COMPONENTS        1     (34V, 40E)
   $CYCLIC_COMPONENT _a0*a1           (34V, 40E, 5AP, 2BP, 3CP, 2HP)
   $No_PERIPHERAL_TREES 2
        $PERIPHERAL_TREE _d2*g            (2V, 1E)
        $VERTICES
        d2
        g
        $EDGES
        _d2*g
        $BORDER_POINT
        d2
        $PERIPHERAL_TREE _c0*e0           (5V, 4E)
        $VERTICES
        c0
        e0
        e1
        e2
        e3
        $EDGES
        _c0*e0
        _e0*e1
        _e1*e2
        _e1*e3
        $BORDER_POINT
        c0
```

Table 11.3: *Results of Program 5.1 (Part IIa)*

```
$STOPFREE_KERNEL _a0*a1            (29V, 35E)
$No_SUBCOMPONENTS 3
    $SUBCOMPONENT _f0*f1               (3V, 3E, 1AP, 0BP, 1CP, 0HP)
    $No_BIBLOCKS 1
        $BIBLOCK _f0*f1                    (3V, 3E, 1AP, 0BP, 1CP, 0HP)
        $VERTICES
        f0
        f1
        f2
        $EDGES
        _f0*f1
        _f1*f2
        _f2*f0
        $ATTACHMENT_POINTS
        f0
        $CHECK_POINTS
        f0
    $SUBCOMPONENT _d0*d1               (3V, 3E, 2AP, 1BP, 1CP, 0HP)
    $No_BIBLOCKS 1
        $BIBLOCK _d0*d1                    (3V, 3E, 2AP, 1BP, 1CP, 0HP)
        $VERTICES
        d0
        d1
        d2
        $EDGES
        _d0*d1
        _d1*d2
        _d2*d0
        $ATTACHMENT_POINTS
        d0
        d2
        $BORDER_POINTS
        d2
        $CHECK_POINTS
        d0
```

Table 11.4: *Results of Program 5.1 (Part IIb)*

```
$SUBCOMPONENT _c0*c1                (23V, 27E, 2AP, 1BP, 1CP, 2HP)
$No_BIBLOCKS 3
    $BIBLOCK _b0*b1                     (3V, 3E, 1AP, 0BP, 0CP, 1HP)
    $VERTICES
    b0
    b1
    b2
    $EDGES
    _b0*b1
    _b1*b2
    _b2*b0
    $ATTACHMENT_POINTS
    b0
    $HINGE_POINTS
    b0
    $BIBLOCK _c0*c1                     (4V, 4E, 1AP, 1BP, 1CP, 1HP)
    $VERTICES
    c0
    c1
    c2
    c3
    $EDGES
    _c0*c1
    _c1*c2
    _c2*c3
    _c3*c0
    $ATTACHMENT_POINTS
    c0
    $BORDER_POINTS
    c0
    $CHECK_POINTS
    c0
    $HINGE_POINTS
    c0
```

Table 11.5: *Results of Program 5.1 (Part IIc)*

```
$BIBLOCK _a0*a1                    (18V, 20E, 2AP, 1BP, 1CP, 2HP)
$VERTICES
a0
a1
a10
a11
a12
a13
a14
a15
a2
a3
a4
a5
a6
a7
a8
a9
b0
c0
$EDGES
_a0*a1
_a1*a2
_a10*a11
_a11*a12
_a12*a13
_a13*a14
_a14*a15
_a15*a0
_a2*a3
_a3*a4
_a4*a5
_a5*a6
_a6*a7
_a7*a8
_a8*a9
_a9*a10
_b0*a3
_b0*a5
_c0*a12
_c0*a4
```

Table 11.6: *Results of Program 5.1 (Part IId)*

```
            $ATTACHMENT_POINTS
            b0
            c0
            $BORDER_POINTS
            c0
            $CHECK_POINTS
            c0
            $HINGE_POINTS
            b0
            c0
        $No_INTERNAL_TREES 1
            $INTERNAL_TREE _c0*d0              (3V, 2E, 3AP, 1BP, 3CP, 1HP)
            $VERTICES
            c0
            d0
            f0
            $EDGES
            _c0*d0
            _c0*f0
            $ATTACHMENT_POINTS
            c0
            d0
            f0
            $BORDER_POINTS
            c0
            $CHECK_POINTS
            c0
            d0
            f0
            $HINGE_POINTS
            c0
END COMPLETE STRUCTURE
```

Table 11.7: *Results of Program 5.1 (Part IIe)*

```
PRINT OPTION VPV
$GRAPH
Graph2
$TYPE
UGSLF
$No_VERTICES 40
$No_EDGES 44
VERTICES

.............................................................


b0                              $DEGREE     4      $GRAPH Graph2
                                $CLASS RVDAP
                                $HINGEPOINT
  $NO_PERIPHERAL_TREE_EDGES     0
  $NO_INTERNAL_TREE_EDGES       0
  $NO_BIBLOCKS                  2

  $NO_BIBLOCK_EDGES             2  _b0*b1
   _b0*b1                       $CLASS EDBB
   _b2*b0                       $CLASS EDBB
  $NO_BIBLOCK_EDGES             2  _a0*a1
   _b0*a3                       $CLASS EDBB
   _b0*a5                       $CLASS EDBB

c0                              $DEGREE     7      $GRAPH Graph2
                                $CLASS RVDAP
                                $HINGEPOINT
                                $CHECKPOINT
                                $BORDERPOINT
  $NO_PERIPHERAL_TREE_EDGES     1  _c0*e0
   _c0*e0                       $CLASS EDPT
  $NO_INTERNAL_TREE_EDGES       2  _c0*d0
   _c0*d0                       $CLASS EDIT
   _c0*f0                       $CLASS EDIT
  $NO_BIBLOCKS                  2

  $NO_BIBLOCK_EDGES             2  _c0*c1
   _c0*c1                       $CLASS EDBB
   _c3*c0                       $CLASS EDBB
  $NO_BIBLOCK_EDGES             2  _a0*a1
   _c0*a12                      $CLASS EDBB
   _c0*a4                       $CLASS EDBB
```

Table 11.8: *Results of Program 5.1 (Part III)*

```
Biblock trees for graph Graph2
(Cyclic components only)

Cyclic component _a0*a1           (34V, 40E)
    Level =   0  Biblock          _a0*a1  (18V,20E)              ('root')
    Level =   1  Attachment point b0                        (_a0*a1)
    Level =   1  Attachment point c0                        (_a0*a1)
    Level =   2  Biblock          _b0*b1  (3V,3E)                (b0)
    Level =   2  Peripheral tree  _c0*e0  (5V,4E)                (c0)
    Level =   2  Internal Tree    _c0*d0  (3V,2E)                (c0)
    Level =   2  Biblock          _c0*c1  (4V,4E)                (c0)
    Level =   3  Attachment point d0                        (_c0*d0)
    Level =   3  Attachment point f0                        (_c0*d0)
    Level =   4  Biblock          _d0*d1  (3V,3E)                (d0)
    Level =   4  Biblock          _f0*f1  (3V,3E)                (f0)
    Level =   5  Attachment point d2                        (_d0*d1)
    Level =   6  Peripheral tree  _d2*g  (2V,1E)                 (d2)

 End Biblock trees
```

Table 11.9: *Results of Program 5.1 (Part IV)*

Chapter 12

The Triblock Decomposition *(not yet released)*

12.1 Problem Description

12.2 Formats and Data Structures

For a general description and basic data types see section 2.2, page 11.

12.3 Functions

Not yet implemented

- hoptar

12.4 Examples

Chapter 13

Red-Black Trees

13.1 Problem Description

On the one hand, the basic incidence structure of GHS graph representations as well as the multitude of derived structures (e. g. weak and strong components, biblock decomposition) conceptually require numerous linked lists (lists of vertices, lists of edges, incidence lists, etc) which frequently are processed in sequence. On the other hand, there is the need to rapidly find an element in these lists by its key value. To cope with both requirements, the decision has been made to consistently use red-black trees, and the experience with these trees has been completely satisfactory.

In former versions of GHS only a reduced set of functions for red-black trees had been implented. The actual version comprises a complete set of operations for red-black trees. It also includes 'naive inserting' and 'naive deleting', so that non-balanced search trees can be buildt and manipulated. The actual implementation of the algorithms follows closely [Stie2009], which in turn is based largely on Cormen/Leiserson/Rivest [CormLR1990]. For the differences between older versions and the actual version see section 13.4 'Note on Compatibility'.

The following set of operations have been implemented:

`ntreeinsert`
Naive insertion of a new record to a general binary search tree. If there is already a record with the same key value in the tree, the new record is not inserted and the calling program is notified.

`rbtreeinsert`
A new record is inserted into a red-black tree. If there is already a record with the same key value in the tree, the new record is not inserted and the calling program is notified.

`ntreedelete`
Naive deletion of a record given by its key from a general binary seach tree.

`rbtreedelete`
A record given by its key is removed from the red-black tree. A correct red-black tree remains.

`rbtreefind`
A record with a given key value is searched for in the tree. If the record exists, its address is the return value. Otherwise, NULL is returned.

`rbtreepfind`
A record with a given key value is searched for in the tree. If the record exists, its address is the return value. If such a record does not exist and the provided key value is not greater then the largest existing key value and not smaller than the smallest existing key value, the record with the next lower key value is returned. Otherwise, NULL is returned.

`rbtreesize`
Returns the number of records in the tree.

`rbtreenext`
For a given record, finds the record with the next higher ordering value. NULL if such a record does not exist.

`rbtreeprevious`
For a given record, finds the record with the next lower ordering value. NULL if such a record does not exist.

`rbtreemin`
Finds the record of the tree with minimum ordering value, i.e. the left-most record.

`rbtreemax`
Finds the record of the tree with maximum ordering value, i.e. the right-most record.

In addition to the functions mentioned above, there exist in GHS the following service functions related to trees.

`printorder`
Prints the names of the records of a red-black tree / a general binary sarch tree in ascending order.

`printrbtree`
Prints the actual structure of a red-black tree / a general binary sarch tree.

`testtree`
Tests wether the tree, given by its root, is a correct binary search tree. Optionally, it is also tested, wether it is a correct red-black tree.

`getname`
This function returns the name of an object when the address of the corresponding record is given. The function does not use any tree structure and is included in the present chapter only because it uses the same definitions of key values and key classes (see next section).

`getcharname`
This function returns the charname of a decomposition element (see subsection 4.1.2, page 43, on namesd and addresses). This function uses neither tree structures nor key lasses. It is included in the present chapter for its relationship to `getname`,

13.2 Formats and Data Structures

To handle red-black trees in a uniform manner, the generic date type `RB` has been introduced.

```
struct  rbtree  /* RB  */
{       RB    *left;
        RB    *right;
        RB    *par;
        COLOR  color;
        void  *auxiliary;
};
```

The definitions of almost all records in the GHS data structures start with these five fields and thus these records can be inserted and searched for in red-black trees.

To handle the large number of different search criteria, the enumeration type `SORTKEY` has been included into the file `GHSstructure.h` (see section C.14, page 189). The values of this data type indicate a *key class*.

Key classes are used for different purposes. So care has to be taken when using them with the functions of this chapter. This is best explained by an example. In the definition of the enumeration type `SORTKEY` (page 189) we find, among others, the following

```
    SND,      /* rbtcomp        VERTEX compared to                    */
              /* rbtreefind     VERTEX by vtname                      */
              /* rbtreeinsert                                         */
              /* getname                                              */
.............................................................
    SNN,      /* rbtcomp        VERTEX->vtname compared to char*      */
              /* rbtreefind                                           */
.............................................................
    SED1,     /* getname        Uses EDGE record. Provides edge name and */
              /*                names of end vertices in a 3 lines    */
              /*                print format                          */
```

Key classes `SND` and `SNN` may be used to search (`rbtreefind`) within red-black trees, since they are supported by the central comparison routine `rbtcomp` (which is of internal GHS use only). `SNN` uses as search value a mere character string whereas with `SND` the search value is the name of a vertex whose record is given. Therefore, insertion (`rbtreeinsert`) is allowed with `SND` but is not with `SNN`. For the same reason, `getname` is allowed with `SND` but not with `SNN`.

On the other hand, if a record is to be searched for in a red-black tree of vertex records and the search criterion is the bare vertex name (in contrast to a given vertex record), key class `SNN` has to be used.

Key class `SED1` is not supported by `rbtcomp` and must therefore not be used with red-black trees. It may be used with `getname` when an EDGE record is given to obtain the name of the edge and the names of its end points (see example 13.10, page 127).

Note: Key values for insertion (like `SND`*)* are valid also for `ntreeinsert`. *Key values (like* `SND` *or* `SNN`*) usable for comparisons are valid also for* `rbtreepfind, rbtreenext, rbtreeprevious`.

13.3 Functions

13.3.1 ntreeinsert

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `BOOLEAN ntreeinsert(RB **root, RB *elem, int key)`

Description: `root` points to a general binary search tree. `elem` is the record to be inserted. TRUE is returned if a new record has been added to the tree. If the key value already exists, FALSE will be returned. The type of the records and the search key used depend on the key class `key` (see section 13.2). The insertion is done in a naive way, i.e. the record is added at the place which its key value indicates. No reordering of the tree is done.

Error Exits: System error if NULL record is to be inserted. System error if key class is not allowed.

Remarks: None

Examples: See example 13.1, page 126.

13.3.2 rbtreeinsert

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `BOOLEAN rbtreeinsert(RB **root, RB *elem, int key)`

Description: `root` points to a red-black tree. `elem` is the record to be inserted. TRUE is returned if a new record has been added to the tree. If the key value already exists, FALSE will be returned. The type of the records and the search key used depend on the key class `key` (see section 13.2).

Error Exits: System error if NULL record is to be inserted. System error if key class is not allowed.

Remarks: Uses `ntreeinsert` internally.

Examples: Example 13.2, page 126.

13.3.3 ntreedelete

Syntax: `BOOLEAN ntreedelete(RB **root, RB *elem, int key)`

Description: `root` points to a binary search tree. `elem` indicates the record to be deleted. It may be the record itself, given by its address, or the name of the record. Whatever is the case, is determined by `key` (see section 13.2). TRUE is returnend if the record has been deleted correctly. If the record does not exist, FALSE will be returned.

Error Exits: System error if NULL record is to be deleted. System error if key class is not allowed.

Remarks: Deletion is done in the naive way. See [CormLR1990].

Examples: See example 13.3, page 126.

13.3.4 rbtreedelete

Syntax: `BOOLEAN rbbreedelete(RB **root, RB *elem, int key)`

Description: `root` points to a red-black tree. `elem` indicates the record to be deleted. It may be the record itself, given by its address, or the name of the record. Whatever is the case, is determined by `key` (see section 13.2). TRUE is returnend if the record has been deleted correctly. If the record does not exist, FALSE will be returned.

Error Exits: System error if NULL record is to be deleted. System error if key class is not allowed.

Remarks: Deletion is done in such a way that afterwards a correct red-black tree remains.

Examples: See examples 13.4, page 126, and 13.5, page 126.

13.3.5 rbtreefind

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `RB *rbtreefind(RB *tree, RB *elem, int key)`

Description: Searches a record in the red-black tree given by `tree`. The search value is given by `elem`. The type of the record and how the search key is used depend on the key class `key` (see section 13.2). The return value is the address of the found record. It is NULL if no record is found.

Error Exits: System error if search value is NULL. System error if key class is not allowed.

Remarks: None.

Examples: See example 13.6, page 126.

13.3.6 rbtreepfind

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `RB *rbtreepfind(RB *tree, RB *elem, int key)`

Description: Searches a record in the binary search tree given by `tree`. The search value is given by `elem`. The type of the record and how the search key is used depend on the key class `key` (see section 13.2). The return value is the address of the found record, if it exists. If a record with the given key does not exist and the provided key value is not greater than the largest existing key and not smaller than the smallest existing key, the record with the next lower key value is returned. Otherwise NULL is returned.

Error Exits: System error if search value is NULL. System error if key class is not allowed.

Remarks: None.

Examples: See example 13.6, page 126.

13.3.7 rbtreesize

Program Author: Günther Stiege, Universität Oldenburg

Syntax:
```
int       *rbtreesize(RB *tree)
```

Description: Returns the number of records of the tree given by `tree`.

Error Exits: None.

Remarks: None.

Examples: None.

13.3.8 rbtreenext

Program Author: Günther Stiege, Universität Oldenburg

Syntax:
```
RB        *rbtreenext(RB *tree, RB *elem, BOOLEAN *b, int key)
```

Description: If `tree` equals `NULL` `b` is set to `TRUE` und `NULL` is returned. Otherwise, the record given by `elem` and `key` is located in the search tree. `b` is set to `FALSE` and `NULL` is returned if the record does not exist. If it does exist `b` is set to `TRUE` and then the record with the next higher key value is searched for and its address is returned. If no reord with higher key value exists `NULL` is returned.

Error Exits: Indirect system errors if `elem` equals `NULL` or an illegal `key` has been specified.

Remarks: None.

Examples: See example 13.7 (page 126).

13.3.9 rbtreeprevious

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `int *rbtreeprevious(RB *tree, RB *elem, BOOLEAN *b, int key)`

Description: If `tree` equals `NULL` `b` is set to `TRUE` and `NULL` is returned. Otherwise, the record given by `elem` and `key` is located in the search tree. `b` is set to `FALSE` and `NULL` is returned if the record does not exist. If it does exist `b` is set to `TRUE` and then the rocrd with the next lower key value is searched for and its address is returned. If no record with lower key value exists `NULL` is returned.

Error Exits: Indirect system errors if `elem` equals `NULL` or an illegal `key` has been specified.

Remarks: None.

Examples: See example 13.7 (page 126).

13.3.10 rbtreemax

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `RB *rbtreemax(RB *tree)`

Description: Returns the record of the tree which has maximum ordering value, i.e. the right-most record. Returns NULL if `tree` is empty.

Error Exits: None

Remarks: None.

Examples: See example 13.8 (page 127).

13.3.11 rbtreemin

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `RB *rbtreemin(RB *tree)`

Description: Returns the record of the tree which has minimum ordering value, i.e. the left-most record. Returns NULL if `tree` is empty.

Error Exits: None

Remarks: None.

Examples: See example 13.8 (page 127).

13.3.12 printorder

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void printorder(FILE *fd, RB *ndx, int key, char *ftext)`

Description: Writes to the file given by file descriptor `fd` the names of the records of the red-black tree pointed to by `ndx` in ascending order.
The type of the records depends on the key class `key` (see section 13.2). A constant prefix printed with each record is specified by `ftext`. Nothing is printed if `ndx` is NULL.

Error Exits: File descriptor `fd` is NULL (nothing is printed).
System error if key class is not allowed.

Examples: See examples 13.1 (page 126), 13.2 (page 126), and 13.9 (page 127).

Remarks: The file descriptor `fd` must have been set by `fopen` or alternately to `stdout`.

13.3.13 printrbtree

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void printrbtree(FILE *fd, RB *ndx, int level, int key)`

Description: Writes to the file given by file descriptor `fd` the actual structure of a red-black tree in in-order. The type of the records depends on the key class `key` (see section 13.2). The root of the tree is given by `ndx`. The 'root' may be any node in the tree. With `level` the level of this node is specified.
Remark: `level` is for printing purposes only. Any number can be used. It is recommended to use 0, if the real level of the 'root' is unknown.

Error Exits: File descriptor `fd` is NULL (nothing is printed).
System error if key class is not allowed.

Examples: Examples 13.1 (page 126) and 13.2 (page 126).

Remarks: The file descriptor `fd` must have been set by `fopen` or alternately to `stdout`.

13.3.14 testtree

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `BOOLEAN testtree(FILE *fd, RB *root, int key, char option)`

Description: Tests a binary tree, given by its `root`. If `option` equals `S` the tree is tested for correct binary search tree. If `option` equals `R`, the tree is tested in addition for correct red-black tree. TRUE is returned if no erros have been detected. FALSE is returned otherwise. If errors are detected they are output to `fd`. Comparisons for greater/less use `key`.

Error Exits: File descriptor `fd` is NULL (nothing is printed).

Examples: Examples 13.1 (page 126), 13.2,

Remarks: The file descriptor `fd` must have been set by `fopen` or alternately to `stdout`.

13.3.15 getname

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void getname(RB *ndx, int key)`

Description: The name of the record pointed to by `ndx` is provided. The name is not the return value of the function, but is available as a character string in the global array `namebuffer`.

Error Exits: System error if `ndx` is NULL. System error if key class is not allowed.

Remarks: None.

Examples: See example 13.10, page 127, and example 13.11, page 127.

13.3.16 getcharname

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void getcharname(RB *ndx, char *comptype)`

Description: The charname of the decomposition element pointed to by `ndx` and specified by `comptype` is provided as a pointer to the corresponding line name (`edname`). See also subsection 4.1.2, page 43, on names and addresses. `comptype` must be specified as one of the following strings:

`"WCOMP"`	weak component
`"SCOMP"`	strong component
`"EXD"`	external dag
`"STP"`	stopfree kernel
`"PT"`	peripheral tree
`"IT"`	internal tree
`"SUB"`	subcomponent
`"BLB"`	biblock

Error Exits: System error if `ndx` is NULL. System error if `comptype` is NULL or has illegal value.

Remarks: None.

Examples: See example 13.11, page 127.

13.4 Note on Compatibility

13.5 Examples

Example 13.1 For this example see `rbt01.c` and `rbt01.cmd` in directory `GHStests`. Using `ntreeinsert`, a binary search tree is constructed from the list of keys in file `bookrnd.klist` in directory `GHSgraphs`. After that the program tries to insert value `HAT` again and a warning occurs. Then `testtree` is called to check wether the tree is a correct binary search tree, which it is. Afterwards `testtree` is called again to check wether we obtained a correct red-black tree, which we did not. The tree is printed using `printorder` and `printrbtree`. The program code is shown in table 13.1, the results are shown in tables 13.2 and 13.3. □

Example 13.2 For this example see `rbt02.c` and `rbc02.cmd` in directory `GHStests`. Program `rbt02.c` is almost the same as `rbt01.c`. It uses `rbtreeinsert` instead of `ntreeinsert`. The results are shown in tables 13.4 and 13.5. Table 13.5 shows clearly the better balancing of the tree as compared to table 13.3. □

Example 13.3 See `rbt07.c` and `rbt07.cmd` in directory `GHStests`. Program `rbt07.c` is very similar to program `rbt01.c` and is partially shown in table 13.6. After constructing a red-black tree from a file of key values this tree is completely removed by deleting all its records one by one. The file used is `GHSwords.klist` in directory `GHSgraphs`. It consists of 5757 English 5-letter words and was extracted from Knuth's `words.dat` (see [Knut1993]). Table 13.7 shows the reults. □

Example 13.4 See `rbt09.c` and `rbt09.cmd` in directory `GHStests`. Program `rbt09.c` is the almost the same as program `rbt07.c`. Insertion and deletion is done for a red-bleck tree. □

Example 13.5 See `rbt08.c` and `rbt08.cmd` in directory `GHStests`. Program `rbt08.c` builds the red-black tree shown in table 13.5. In the subsequent interactive phase records are deleted one by one. Table 13.8, page 135 shows a run of the program where key `ROT, ANTON, KNABE, ZANK, HUT` are deleted in that order. □

Example 13.6 See program `rbt06.c` and command `rbt06.cmd` in directory `GHStests`. File `GHSwords` in directory `GHSgraphs` is used to construct the graph `wordsgraph`. The list of vertices `wordsgraph->grvtlist` is organized as red-black tree. The key values `aa, aargh, magma, magmi, flora, flori, zul` are processed with `rbtreefind` and `rbtreepfind`. and give the following output:

```
key value = aa  rbtreefind yields (NULL)   rbtreepfind yields: (NULL)
key value = aargh  rbtreefind yields aargh   rbtreepfind yields: aargh
key value = magma  rbtreefind yields magma   rbtreepfind yields: magma
key value = magmi  rbtreefind yields (NULL)   rbtreepfind yields: magma
key value = flora  rbtreefind yields flora   rbtreepfind yields: flora
key value = flori  rbtreefind yields (NULL)   rbtreepfind yields: flora
key value = zul  rbtreefind yields (NULL)   rbtreepfind yields: (NULL)
```

Example 13.7 See program `rbt05.c` and command `rbt05.cmd` in directory `GHStests`. The program builds a binary search tree from a list of key values contained in file `bookrnd.klist` in directory `GHSgraphs`. Then the key values `KNABE, BESUCH, ANTON, ZANK, HAT, HARZ` are processed with `rbtreenext` and `rbtreeprevious` and give the following output:

```
vertex preceeding vertex KNABE = HUT     vertex following vertex KNABE = LAGE
vertex preceeding vertex BESUCH = ANTON     vertex following vertex BESUCH = DER
vertex preceeding vertex ANTON = (NULL)     vertex following vertex ANTON = BESUCH
vertex preceeding vertex ZANK = ZAHN     vertex following vertex ZANK = (NULL)
vertex preceeding vertex HAT = DER     vertex following vertex HAT = HUT
rbt05.c: HARZ is not element of the tree.
```

Example 13.8 See program `rbt04.c` and command `rbt04.cmd` in directory `GHStests`. The maximum vertex name und the minimum vertex name of graph `GHSwords` are searched for and printed. The output is

```
Max = zowie     Min = aargh
```

□

Example 13.9 See Graph1 of example 2.1 (figure 2.1, page 18). If the graph is read with `readgraphlist` and then function `printorder` is called:

```
printorder(fd, (RB *)(graph->grvtlist), SND, "This is a test string ");
```

the list of table 13.9 is printed. For the complete program see `rbt03.c` in directory `GHStests`. □

Example 13.10 Consider Graph0, figure 1.2 on page 5. `rbt10.c` reads and constructs Graph0. It then finds edge `_i` and finally prints its name together with the names of it end points. Table 13.10 □

Example 13.11 Example `rbtc11.cmd` and `rbtc11.c` in `GHStest` constructs graph `GHSwords` and decomposes in weak and strong components. Afterwords a list of all five classes of weak components is printed showing the charname and the hierarchical name for each component. Since `GHSwords` is an undirected graph, all but two of the classes are empty as they ought to be. □

```
/*********************************************************************/
/*     rbt01                                                         */
/*     Program  for constructing binary search trees.                */
/*********************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

int main(int argc, char **argv)
{ RB          *root;
  VERTEX      *vt;
  FILE        *fdin;
  int          n;
  char         inbuffer[1012];
  BOOLEAN      bl;

  fdin = fopen(argv[1], "r");
  if (fdin == NULL)
    { printf("rbt01: Input file %s cannot be opened.\n", argv[1]); exit(0); }
  root = NULL;
// Construction of a binary search tree using a list of key values.
  n = fscanf(fdin, "%s", inbuffer);
  while (n != EOF)
     { vt = mnewvt("newvt");
       vt->vtname = mnewname(inbuffer, "newname");
       bl = ntreeinsert(&root, (RB *)vt, SND);
       if (!bl) printf("rbt01: %s exists already.\n", vt->vtname);
                                        // Continue without the multiple value
       n = fscanf(fdin, "%s", inbuffer);
     }
// Trying to insert a key value twice
  vt = mnewvt("newvt1");
  vt->vtname = mnewname("HAT", "newname1");
  bl = ntreeinsert(&root, (RB *)vt, SND);
  if (!bl) printf("rbt01: %s exists already.\n", vt->vtname);
// Testing tree contructed
  bl = testtree(stdout, root, SND, 'S'); // Testing for correctness
  if (!bl) {printf("rbt01: No correct binary search tree has been built\n");
            exit(0);}
  else { printf("rbt01: Binary search tree is correct.\n");}
  bl = testtree(stdout, root, SND, 'R'); // Testing for correct red-black tree
  if (!bl) printf("rbt01: No correct red-black tree has been built.\n");
  printorder(stdout, root, SND, "   ");
  printrbtree(stdout, root, 0, SND);
}
```

Table 13.1: *Program* `rbt01.c`

```
rbt01: HAT exists already.
rbt01: Binary search tree is correct.
testtree: Blacklengths differ.
Blacklength to ANTON is 3, blacklength to BESUCH is 4
rbt01: No correct red-black tree has been built.
   ANTON
   BESUCH
   DER
   HAT
   HUT
   KNABE
   LAGE
   LUST
   NEIN
   NOCH
   PFAU
   QUARK
   ROT
   RUHEN
   SEHEN
   TANNE
   ZAHN
   ZANK
```

Table 13.2: *Results of Program* `rbt01.c` *(Part I)*

```
       k =   0     SEHEN   color = BLACK   Parent = (NULL)
left:  k =   1     DER   color = BLACK   Parent = SEHEN
left:  k =   2     ANTON   color = BLACK   Parent = DER
left:  k =   3     Leaf
right: k =   3     BESUCH   color = BLACK   Parent = ANTON
left:  k =   4     Leaf
right: k =   4     Leaf
right: k =   2     RUHEN   color = BLACK   Parent = DER
left:  k =   3     HAT   color = BLACK   Parent = RUHEN
left:  k =   4     Leaf
right: k =   4     HUT   color = BLACK   Parent = HAT
left:  k =   5     Leaf
right: k =   5     KNABE   color = BLACK   Parent = HUT
left:  k =   6     Leaf
right: k =   6     LAGE   color = BLACK   Parent = KNABE
left:  k =   7     Leaf
right: k =   7     ROT   color = BLACK   Parent = LAGE
left:  k =   8     QUARK   color = BLACK   Parent = ROT
left:  k =   9     PFAU   color = BLACK   Parent = QUARK
left:  k =  10     NOCH   color = BLACK   Parent = PFAU
left:  k =  11     NEIN   color = BLACK   Parent = NOCH
left:  k =  12     LUST   color = BLACK   Parent = NEIN
left:  k =  13     Leaf
right: k =  13     Leaf
right: k =  12     Leaf
right: k =  11     Leaf
right: k =  10     Leaf
right: k =   9     Leaf
right: k =   8     Leaf
right: k =   3     Leaf
right: k =   1     ZAHN   color = BLACK   Parent = SEHEN
left:  k =   2     TANNE   color = BLACK   Parent = ZAHN
left:  k =   3     Leaf
right: k =   3     Leaf
right: k =   2     ZANK   color = BLACK   Parent = ZAHN
left:  k =   3     Leaf
right: k =   3     Leaf
```

Table 13.3: *Results of Program* `rbt01.c` *(Part II)*

```
rbt02: Red-black tree is correct.
   ANTON
   BESUCH
   DER
   HAT
   HUT
   KNABE
   LAGE
   LUST
   NEIN
   NOCH
   PFAU
   QUARK
   ROT
   RUHEN
   SEHEN
   TANNE
   ZAHN
   ZANK
```

Table 13.4: *Results of Program* `rbt02.c` *(Part I)*

```
       k =   0     HUT   color = BLACK   Parent = (NULL)
left:  k =   1     BESUCH   color = BLACK   Parent = HUT
left:  k =   2     ANTON   color = BLACK   Parent = BESUCH
left:  k =   3     Leaf
right: k =   3     Leaf
right: k =   2     DER   color = BLACK   Parent = BESUCH
left:  k =   3     Leaf
right: k =   3     HAT   color = RED   Parent = DER
left:  k =   4     Leaf
right: k =   4     Leaf
right: k =   1     QUARK   color = RED   Parent = HUT
left:  k =   2     LAGE   color = BLACK   Parent = QUARK
left:  k =   3     KNABE   color = BLACK   Parent = LAGE
left:  k =   4     Leaf
right: k =   4     Leaf
right: k =   3     NOCH   color = RED   Parent = LAGE
left:  k =   4     NEIN   color = BLACK   Parent = NOCH
left:  k =   5     LUST   color = RED   Parent = NEIN
left:  k =   6     Leaf
right: k =   6     Leaf
right: k =   5     Leaf
right: k =   4     PFAU   color = BLACK   Parent = NOCH
left:  k =   5     Leaf
right: k =   5     Leaf
right: k =   2     RUHEN   color = BLACK   Parent = QUARK
left:  k =   3     ROT   color = BLACK   Parent = RUHEN
left:  k =   4     Leaf
right: k =   4     Leaf
right: k =   3     TANNE   color = RED   Parent = RUHEN
left:  k =   4     SEHEN   color = BLACK   Parent = TANNE
left:  k =   5     Leaf
right: k =   5     Leaf
right: k =   4     ZAHN   color = BLACK   Parent = TANNE
left:  k =   5     Leaf
right: k =   5     ZANK   color = RED   Parent = ZAHN
left:  k =   6     Leaf
right: k =   6     Leaf
```

Table 13.5: *Results of Program* `rbt02.c` *(Part II)*

```
/*********************************************************************/
/*     rbt07                                                         */
/*     Program  showing deletions in binary search trees             */
/*********************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

int main(int argc, char **argv)
{ RB          *root, *rbroot;
  VERTEX      *vt, *vt1;
  FILE        *fdin;
  int          n;
  char         inbuffer[1012];
  BOOLEAN      bl;

   ........................................
// Construction of a binary search tree using a list of key values.
   ........................................

// Testing tree contructed
   ........................................

// Deleting records
  fdin = fopen(argv[1], "r");   // repositioning file
  n = fscanf(fdin, "%s", inbuffer);
  while (n != EOF)
    {  bl = ntreedelete(&root, (RB *)inbuffer, SNN);
       if (!bl)
         { printf("rbt07: Record %s not in tree.\n", inbuffer);
           exit(0); }
       bl = testtree(stdout, root, SND, 'S'); // Testing for correctness
       if (bl) {printf("rbt07.c: Correct tree after deleting %s.\n", inbuffer);}
       else
         { printf("Incorrect search tree after deleting record %s\n", inbuffer);
           exit(0); }
       n = fscanf(fdin, "%s", inbuffer);
     }
   if (root == NULL) printf("rbt07.c: Final tree is empty.\n");
   else printf("rbt07.c: Final tree is not(!) empty.\n");
}
```

Table 13.6: *Program* `rbt07.c`

```
rbt07: Binary search tree is correct.
rbt07.c: Correct tree after deleting wasps.
rbt07.c: Correct tree after deleting infix.
rbt07.c: Correct tree after deleting toyon.
rbt07.c: Correct tree after deleting jihad.
rbt07.c: Correct tree after deleting teeth.
rbt07.c: Correct tree after deleting pores.
rbt07.c: Correct tree after deleting bract.
rbt07.c: Correct tree after deleting alert.
.....................................
.....................................
.....................................

rbt07.c: Correct tree after deleting cover.
rbt07.c: Correct tree after deleting maned.
rbt07.c: Correct tree after deleting peons.
rbt07.c: Correct tree after deleting shire.
rbt07.c: Correct tree after deleting vodka.
rbt07.c: Correct tree after deleting zowie.
rbt07.c: Correct tree after deleting teams.
rbt07.c: Correct tree after deleting sacks.
rbt07.c: Final tree is empty.
```

Table 13.7: *Results of Program* `rbt07.c`

```
stiege.thinkpad> rbt08.cmd
rbt08.c: The constructed red-black tree is correct.
rbt08.c: Dialog begins. Input $$$ to terminate program.
Key to be deleted?  ROT
rbt08.c: Correct rb-tree after deleting ROT.
Key to be deleted?  ANTON
rbt08.c: Correct rb-tree after deleting ANTON.
Key to be deleted?  KNABE
rbt08.c: Correct rb-tree after deleting KNABE.
Key to be deleted?  ZANK
rbt08.c: Correct rb-tree after deleting ZANK.
Key to be deleted?  HUT
rbt08.c: Correct rb-tree after deleting HUT.
Key to be deleted?  $$$
        k =   0     LAGE   color = BLACK   Parent = (NULL)
left:   k =   1     DER   color = BLACK   Parent = LAGE
left:   k =   2     BESUCH   color = BLACK   Parent = DER
left:   k =   3     Leaf
right:  k =   3     Leaf
right:  k =   2     HAT   color = BLACK   Parent = DER
left:   k =   3     Leaf
right:  k =   3     Leaf
right:  k =   1     QUARK   color = RED   Parent = LAGE
left:   k =   2     NOCH   color = BLACK   Parent = QUARK
left:   k =   3     LUST   color = BLACK   Parent = NOCH
left:   k =   4     Leaf
right:  k =   4     NEIN   color = RED   Parent = LUST
left:   k =   5     Leaf
right:  k =   5     Leaf
right:  k =   3     PFAU   color = BLACK   Parent = NOCH
left:   k =   4     Leaf
right:  k =   4     Leaf
right:  k =   2     TANNE   color = BLACK   Parent = QUARK
left:   k =   3     RUHEN   color = BLACK   Parent = TANNE
left:   k =   4     Leaf
right:  k =   4     SEHEN   color = RED   Parent = RUHEN
left:   k =   5     Leaf
right:  k =   5     Leaf
right:  k =   3     ZAHN   color = BLACK   Parent = TANNE
left:   k =   4     Leaf
right:  k =   4     Leaf
```

Table 13.8: *Results of Program* `rbt08.c`

```
This is a test string   K00
This is a test string   K01
This is a test string   K02
This is a test string   K03
This is a test string   K04
This is a test string   K05
This is a test string   K06
This is a test string   K07
This is a test string   K08
This is a test string   K09
This is a test string   K10
This is a test string   K11
This is a test string   K12
This is a test string   K13
This is a test string   K14
This is a test string   K15
This is a test string   K16
This is a test string   K17
This is a test string   K18
This is a test string   K19
This is a test string   K20
This is a test string   K21
```

Table 13.9: *The vertices of Graph1 printed with* ***printorder***

```
/*************************************************************************/
/*     rbt10                                                             */
/*     Program  showing an example for getname                           */
/*************************************************************************/
  #include <stdio.h>
  #include <GHSstructure.h>

int main(int argc, char **argv)
{ GRAPH       *graph;
  VERTEX      *vt1;
  EDGE        *ed;

  graph = readgraphlist(argv[1]);
  if (graph ==  NULL)
    { printf("Incorrect graph input\n");
      exit(0);
    }
  ed = (EDGE *)rbtreefind((RB *)(graph->gredlist), (RB *)"_i", SEN);
  getname((RB *)ed, SED1);
  printf("%s\n", namebuffer);
}
```

Program output

```
_i
  D
  F
```

Table 13.10: *Example for* `getname`

Chapter 14

Stacks and Queues

14.1 Problem Description

Stacks and queues are useful and important basic data structures, see for instance Cormen/-Leiserson/Rivest [CormLR1990]. In GHS, the decision has been made to implement them as doubly linked lists. When using stacks and queues in the implementation of graph algorithms two problems arise. At the one hand, it is sometimes useful to have a given record simultaneously as a member in different stacks or different queues or more than once in the same stack or in the same queue. At the other hand, it is often necessary to give the membership of a record in a stack (queue) an attribute valid for the duration of the membership.

`push` and `pop`, respectively `enqueue` and `dequeue`
are to be used when there are no membership attributes required. Allow simultaneous multiple membership.

`fpush` and `fpop`, respectively `fenqueue` and `fdequeue`
are to be used when membership attributes are required. Simultaneous multiple membership is not allowed.

`top`
returns a pointer to the uppermost record of a stack.

`qfront` and `qend`
return a pointer to the first /last record of a queue.

`qscreate` and `qsremove`
A `QS` record is created for / removed from each element of an RB-tree.

`mnewqusta`
A new `QUSTA` record is created.

14.2 Formats and Data Structures

To handle stacks and queues the data types `QUSTA` and `QS` have been introduced. Figure 14.1 shows these records and the links between them. In the upper part, a `QUSTA` record points to a queue (left linked list) and to a stack (right linked list). The uppermost `QS` record is the front of the queue, respectively the top of the stack. There is an additional pointer from the `QUSTA`

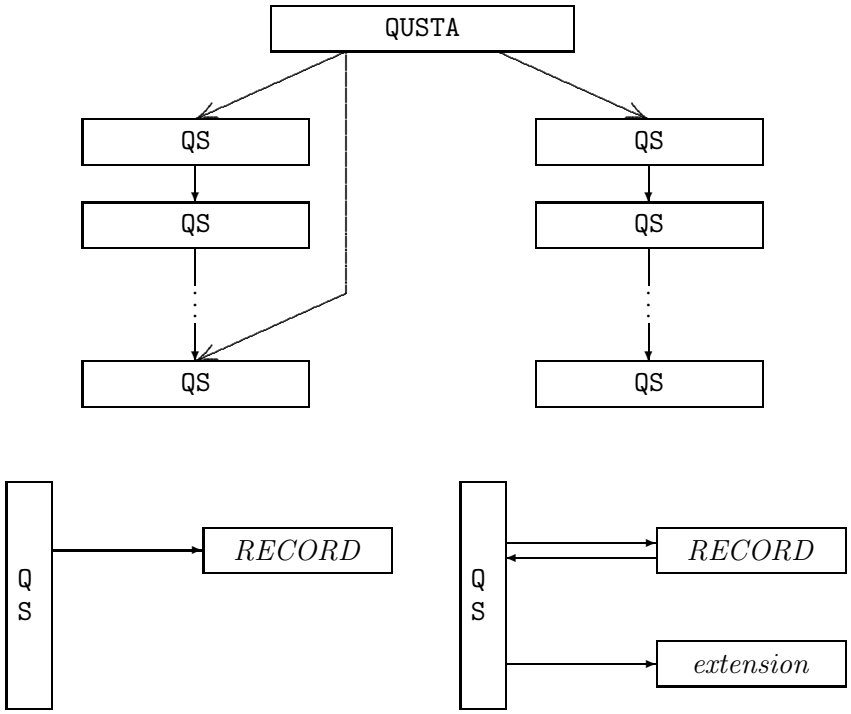


Figure 14.1: *Data structures* `QUSTA` *and* `QS`

record to the end of the queue. The backward links in the list of `QS` records are not shown in the figure.

In the lower part of figure 14.1 the left side shows the use of `QS` records with simultaneous multiple membership (`push, pop, enqueue, dequeue`) whereas the right side shows the use of `QS` records with membership attributes (`fpush, fpop, fenqueue, fdequeue`). In the first case, there is only a pointer from the `QS` record to the record it stands for. The `QS` records are created by `push` and `enqueue` and deleted by `pop` and `dequeue`. They are not visible for the calling program. In the second case, the `QS` record must be allocated and eventually released by the calling program. The calling program is also responsible for putting a link from the original record to the `QS` record in the original record's auxiliary field and a link from the `QS` record to the original record in the `QS` record's `qsp` field. Each `QS` record provides two integer fields, two Boolean fields and a void pointer in which the calling program may keep membership dependent information. In figure 14.1 only the void pointer (*extension*) is shown.

Remark 14.1 On the one hand, as figure 14.1 shows, each `QUSTA` record is both, header structure of a queue *and* header structure of a stack which can be operated on independently. On the other hand, each `QS` record has two forward links and two backward links (not shown in the figure). One pair is used in queues and the other pair is used in stacks. Therefore, the same `QS` record may appear in a queue and in a stack with identical or different `QUSTA` headers. *This case is not a multiple membership.*

14.3 Functions

14.3.1 push

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void push(QUSTA *stack, void *qelem)`

Description: Puts element `qelem` on top of `stack`. `qelem` may be an element of any GHS type given by its address casted to `void *`. `qelem` is not linked directly into the the stack list. Instead a new `QS` record is created which points to qelem. The `QS` record remains invisible to the calling program. Normally, elements put on the stack with `push` should be removed from the stack using `pop` (and not `fpop`). Multiple membership is allowed.

Error Exits: Error if NULL record is to be pushed onto stack.

Remarks: None.

Examples: See example `qusta01.c` in `GHStests`.

14.3.2 pop

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void *pop(QUSTA *stack)`

Description: The `QS` record on top of `stack` is removed and deleted. A pointer to the `qelem` record it stands for is returned. The `qelem` element is not touched.

Error Exits: None.

Remarks: If the stack is empty, NULL is returned.

Examples: See example `qusta01.c` in `GHStests`.

14.3.3 fpush

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void fpush(QUSTA *stack, void *qelem)`

Description: Puts element `qelem` on top of `stack`. `qelem` may be an element of any type given by its address, castes to void *. `qelem` is not linked directly into the stack list. Instead a `QS` record which is pointed to from the auxiliary field of `qelem` is used. The `QS` record must have been created and attached to the `qelem` element by the calling program. Function `qscreate` may be used for this purpose. Normally, elements put in a stack with `fpush` should be removed from the stack with `fpop` (and not `pop`). Multiple membership is not allowed.

Error Exits: Error if NULL record is to be pushed onto stack. Error if `QS` record missing. If an attempt for multiple stack membership is detected, an error message is printed, no stack insertion is done, and the function returns to the calling program.

Remarks: The integer fields `qshelp1` and `qshelp2` as well as the void pointer field `qsphelp` of the `QS` record may be used freely by the calling program.

Examples: See example `qusta01.c` in `GHStests`.

14.3.4 fpop

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void *fpop(QUSTA *stack)`

Description: The `QS` record on top of `stack` is removed, but not deleted. A pointer to the `qelem` element it stands for is returned. The `qelem` is not touched.

Error Exits: None.

Remarks: If the stack is empty, NULL is returned.

Examples: See example `qusta01.c` in `GHStests`.

14.3.5 top

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void *top(QUSTA *stack)`

Description: Returns a pointer to the top element of `stack`.

Error Exits: None.

Remarks: If the stack is empty, NULL is returned.

Examples: See example `qusta01.c` in `GHStests`.

14.3.6 enqueue

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void enqueue(QUSTA *queue, void *qelem)`

Description: Adds element `qelem` to the end of `queue`. `qelem` may be an element of any type given by its address casted to `void *`. `qelem` is not linked directly into the queue. Instead a new `QS` record is created which points to `qelem`. The `QS` record remains invisible to the calling program. Normally, elements put in a queue with `enqueue` should be removed from the queue using `dequeue` (and not `fdequeue`). Multiple membership is allowed.

Error Exits: Error if NULL record is to be added to queue.

Remarks: None.

Examples: See example `qusta01.c` in `GHStests`.

14.3.7 dequeue

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void *dequeue(QUSTA *queue)`

Description: The `QS` record at the front of `queue` is removed and deleted. A pointer to the `qelem` record it stands for is returned. The `qelem` element is not touched.

Error Exits: None.

Remarks: If the queue is empty, NULL is returned.

Examples: See example `qusta01.c` in `GHStests`.

14.3.8 fenqueue

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void fenqueue(QUSTA *queue, void *qelem)`

Description: Adds element `qelem` to the end of `queue`. `qelem` may be an element of any type given by its address casted to `void *`. `qelem` is not linked directly into the queue. Instead a `QS` record which is pointed to from the auxiliary field `qelem` is used. The `QS` record must have been created and attached to the `qelem` by the calling program. Function `qscreate` may be used for this purpose. Normally, elements put in a queue with `fenqueue` should be removed from the queue with `fdequeue` (and not `dequeue`). Multiple membership is not allowed.

Error Exits: Error if NULL record is to be added to queue. Error if `QS` record missing.
If an attempt for multiple queue membership is detected, an error message is printed, no queue insertion is made, and the function returns to the calling program.

Remarks: The integer fields `qshelp1` and `qshelp2` as well as the void pointer field `qsphelp` of the `QS` record may be used freely by the calling program.

Examples: See example `qusta01.c` in `GHStests`.

14.3.9 fdequeue

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void *fdequeue(QUSTA *queue)`

Description: Returns a pointer to the top element of `stack`.

Error Exits: None.

Remarks: If the queue is empty, NULL is returned.

Examples: See example `qusta01.c` in `GHStests`.

14.3.10 qfront

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void *qfront(QUSTA *queue)`

Description: Returns a pointer to the first element in `queue`.

Error Exits: None.

Remarks: If the queue is empty, NULL is returned.

Examples: See example `qusta01.c` in `GHStests`.

14.3.11 qend

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void *qend(QUSTA *queue)`

Description: Returns a pointer to the last element in `queue`.

Error Exits: None.

Remarks: If the queue is empty, NULL is returned.

Examples: See example `qusta01.c` in `GHStests`.

14.3.12 qscreate

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void qscreate(RB *rb)`

Description: Creates for each element of the red-black tree a `QS` record which represents the original element in a queue and/or a stack. The auxiliary field of the original element points to the `QS` record. The `qsp` field of the `QS` record points back to the original record.

Error Exits: None.

Remarks: None.

Examples: See example `qusta01.c` in `GHStests`.

14.3.13 qsremove

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void qsremove(RB *rb)`

Description: Removes and deletes all `QS` records pointed to by the elements of red-black tree `rb`.

Remarks: None.

Error Exits: None.

Examples: See example `qusta01.c` in `GHStests`.

14.3.14 mnewqusta

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `QUSTA *mnewqusta(char *message)`

Description: Creates a new `QUSTA` record.

Remarks: `message` is printed if there is not enough memory available.

Error Exits: System error if there is not enough memory available.

Examples: See example `qusta01.c` in `GHStests`.

14.3.15 releasequstalist

Program Author: Günther Stiege, Universität Oldenburg

Syntax: `void releasequstalist(QUSTA *qusta)`

Description: Releases a red-black tree of `QUSTA` records. Each `QUSTA` record is deleted.
Note: If there are `QS` records the `QUSTA` record points to (see figure 14.1) these are *not released*.

Remarks: To achieve a correct deallocation of memory one has to proceed the following way:
1. Simultaneously multiple membership. The user program must empty the stack/queue completely using `pop`/`dequeue`.
2.Membership attributes. The `QS` records must be deleted explicitly by calling `qsremove`.

Error Exits: None.

Examples: See example `qusta01.c` in `GHStests`.

14.4 Examples

See example `qusta01.c` in `GHStests`.

Part II

Appendix to User Manual

Appendix A

GHS Makefile

```
#** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * #
# Source: Makefile                                                     #
# $Revision: 2.12 $                                                     #
# $Date: 2009/02/01 18:52:35 $                                         #
# Author: G. Stiege                                                    #
# GHScore Makefile                                                     #
# * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * #

# * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * #
#                  Graph Handling System (GHS)                         #
#                   Copyright (c) 2000 - 2003                          #
#               Guenther Stiege and Sergej Alekseev                    #
#                Universitaet Oldenburg, Germany                       #
#                    All rights reserved.                              #
#                                                                      #
# Redistribution and use in source and binary forms,  with or without #
# modification, are permitted provided that:                           #
# (1) source code distributions retain the above copyright notice and #
# this paragraph in its entirety.                                      #
# (2) distributions including binary code include the above copyright #
# notice and this paragraph in its entirety  in  the documentation or #
# other materials provided with the distribution.                      #
# (3) all  advertising  materials  mentioning features or use of this #
# software display the following acknowledgement:                      #
# ‘‘This product includes software developed by Guenther Stiege et al.#
# and Sergej Alekseev, Universitaet Oldenburg, Germany.’’ Neither the #
# name of the University nor the names of its authors may be used  to #
# endorse or promote products  derived  from  this  software  without #
# specific prior written permission.                                   #
# (4) changes are  permissible  only if the changed files are given a #
# new name,  different from the names of  existing  files of GHS, and #
# only if the changed files are clearly  identified as not being part #
# of GHS.                                                              #
#                                                                      #
# THE AUTHORS HAVE TRIED  THEIR  BEST TO PRODUCE A CORRECT AND USEFUL #
```

```
# PROGRAM, IN ORDER TO HELP  PROMOTE  COMPUTER  SCIENCE RESEARCH, BUT #
# THIS  SOFTWARE  IS  PROVIDED ‘‘AS IS’’ AND  WITHOUT ANY  EXPRESS OR #
# IMPLIED  WARRANTIES,  INCLUDING,  WITHOUT  LIMITATION, THE  IMPLIED #
# WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. #
# * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * #

CC=gcc -Wall
SDIR=.
ODIR=GHSobjects
LIBDIR=../GHScorelib

ghs:                                                    \
        createobjectdir                                 \
                                                        \
        $(ODIR)/basics.o                                \
                                                        \
        $(ODIR)/gsets.o                                 \
                                                        \
        $(ODIR)/generate.o                              \
                                                        \
        $(ODIR)/cpchtp.o                                \
                                                        \
        $(ODIR)/util.o                                  \
                                                        \
        $(ODIR)/rbtree.o                                \
                                                        \
        $(ODIR)/qusta.o                                 \
                                                        \
        $(ODIR)/gcomponents.o                           \
                                                        \
        $(ODIR)/gstdutil.o                              \
                                                        \
        $(ODIR)/astd.o                                  \
                                                        \
        $(ODIR)/astdutil.o                              \
                                                        \
        $(ODIR)/path.o                                  \
                                                        \
        $(ODIR)/pathutil.o                              \
                                                        \
        $(ODIR)/menger.o                                \
                                                        \
        $(ODIR)/mengerutil.o                            \
                                                        \
createlib                                       \
                                                        \
```

```
createlib:
gcc -g -fPIC -shared -W1 -o libGHS.so $(ODIR)/*.o -lc
rm -f -r $(LIBDIR)
mkdir $(LIBDIR)
mv libGHS.so $(LIBDIR)
rm -r $(ODIR)

#   Directory for object files
createobjectdir:

rm -f -r GHSobjects
mkdir GHSobjects
#   Basic Graph Functions:
$(ODIR)/basics.o: $(SDIR)/basics.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/basics.o -c $(SDIR)/basics.c

#   Set Functions:
$(ODIR)/gsets.o: $(SDIR)/gsets.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/gsets.o -c $(SDIR)/gsets.c

#   Graph Generating Functions:
$(ODIR)/generate.o: $(SDIR)/generate.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/generate.o -c $(SDIR)/generate.c

$(ODIR)/cpchtp.o: $(SDIR)/cpchtp.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/cpchtp.o -c $(SDIR)/cpchtp.c

#   Functions for the Decomposition of General Graphs:
$(ODIR)/gcomponents.o: $(SDIR)/gcomponents.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/gcomponents.o -c $(SDIR)/gcomponents.c

$(ODIR)/astd.o: $(SDIR)/astd.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/astd.o -c $(SDIR)/astd.c

$(ODIR)/gstdutil.o: $(SDIR)/gstdutil.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/gstdutil.o -c $(SDIR)/gstdutil.c

$(ODIR)/astdutil.o: $(SDIR)/astdutil.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/astdutil.o -c $(SDIR)/astdutil.c

#   Path functions:
$(ODIR)/path.o: $(SDIR)/path.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/path.o -c $(SDIR)/path.c

$(ODIR)/pathutil.o: $(SDIR)/pathutil.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/pathutil.o -c $(SDIR)/pathutil.c

#   Menger functions:
```

```
$(ODIR)/menger.o: $(SDIR)/menger.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/menger.o -c $(SDIR)/menger.c

$(ODIR)/mengerutil.o: $(SDIR)/mengerutil.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/mengerutil.o -c $(SDIR)/mengerutil.c

#   General Utilities

 $(ODIR)/util.o: $(SDIR)/util.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/util.o -c $(SDIR)/util.c

#   Red-black trees

$(ODIR)/rbtree.o: $(SDIR)/rbtree.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/rbtree.o -c $(SDIR)/rbtree.c

#   Stacks and Queues

$(ODIR)/qusta.o: $(SDIR)/qusta.c $(SDIR)/GHSstructure.h
$(CC)  -fPIC -I. -o $(ODIR)/qusta.o -c $(SDIR)/qusta.c



clean:
#       rm -f $(ODIR)/*.o
rm -f ghs
```

Appendix B

GHS External Data Structures

B.1 File Format

Files for graphs are ASCII files and must be structured according to the following syntax (BNF). An informal description of graph files is given in subsection 2.1 on page 11.

<graphfile> := `$GRAPH`*<sep><graphname>*
<sep>`$TYPE`*<sep><graphtype>*
<sep>`$VERTICES`*<sep><vertexname>*[*<sep><vertexname>*]*
[*<sep>*`$EDGES`*<sep><edgespecification>*[*<sep><edgespecification>*]*]
[*<sep>*`$ARCS`*<sep><edgespecification>*[*<sep><edgespecification>*]*]
<sep>`$END`

<sep> := "characters separating string when reading with scanf"

<graphname> := "string not containing characters from *<sep>*,
not starting with $, and not starting with _"

<graphtype> := `GG` | `DG` | `DGS` | `DGSLF` | `UG` | `UGS` | `UGSLF`

<vertexname> := *<graphname>*

<edgespecification> := *<edgename><sep><vertexname><sep><vertexname>*

<edgename> := "string starting with _ and not containing characters from *<sep>*"

Appendix C

GHS Internal Data Structures

```
/*************************************************************************/
/*      Data Structures for the Graph Handling System (GHS)              */
/*                                                                       */
/*                   Guenther Stiege (Oldenburg)                         */
/*                                                                       */
/*************************************************************************/
/*                                                                       */
/*   GHSstructure.h                                                      */
/*                                                                       */
/*   Common Data Structure for GHS Programs                              */
/*                                                                       */
/*   (comments of type "LaTeX:  ..." are for texing purposes only)       */
/*************************************************************************/
```

C.1 General Organization Scheme

```
//   1.      Utilities
//   1.1     General Services
//   1.2     Red-Black Trees
//   1.3     Stacks and Queues
//   2.      Basic Graph Functions
//   3.      Sets
//   4.      Graph Generating Functions
//   5.      Weak and Strong Components
//   6.      Biblock Decomposition
//   7.      Additional DFS Structure
//   8.      General Partitions
//   9.      Paths
//   10.     Menger Structures
//   11.     Higher A-Decomposition
//   12.     Distances
```

C.2 Literal Constants and Type Definitions

```
/*    Literal Constants and Type Definitions */

   #include <stdio.h>
   #include <stdlib.h>
   #include <string.h>
   #include <math.h>

#ifndef _GHSSTRUCTURE_H_
#define _GHSSTRUCTURE_H_

//   1.      Utilities
//   1.1     General Services
  #define   TRUE   1
  #define   FALSE  0
      typedef    int    BOOLEAN;
      typedef    struct relemstr     RELEM;     /* Generic substructure   */
//   1.2     Red-Black Trees
  #define   BLACK   1
  #define   RED     0
      typedef    int    COLOR;
      typedef    struct  rbtree      RB;        /* generic RB element     */
//   1.3     Stacks and Queues
      typedef    struct  ghsqusta    QUSTA;     /* queue/stack of         */
                                                /* arbitrary elements     */
      typedef    struct  ghsqs       QS;        /* representative records */
                                                /* in queues and stacks   */
//   2.      Basic Graph Functions
  #define   TYPENO            7                /* number of graph types  */
      typedef    struct grphst       GRAPH;     /* graphs and subgraphs   */
      typedef    struct vtstr        VERTEX;    /* vertices               */
      typedef    struct rvtstr       RVERTEX;   /* references to a vertex */
      typedef    struct edstr        EDGE;      /* lines (line or arc)    */
                                                /* in the sequel 'edge'   */
                                                /* is sometimes used for  */
                                                /* 'line' and refers to   */
                                                /* both, edges and arcs.  */
      typedef    struct redstr       REDGE;     /* references to an line  */
      typedef    struct incstr       INC;       /* incidences             */
//   3.      Sets
      typedef    struct vtsetstr     VTSET;     /* Vertex set             */
      typedef    struct edsetstr     EDSET;     /* Vertex set             */
      typedef    struct gsetstr      GSET;      /* General set            */
      typedef    struct setelem      SETELEM;   /* Element of general set */
//   4.      Graph Generating Functions
//   5.      Weak and Strong Components
      typedef    struct gstdd        GSTDD;     /* decomposition into     */
```

```
                                                /* weak and strong       */
                                                /* components            */
      typedef    struct  wcomp       WCOMP;     /* weak component        */
      typedef    struct  scomp       SCOMP;     /* strong component      */
      typedef    struct  gvtstd      GVTSTD;    /* vertex in a weak /    */
                                                /* strong components     */
                                                /* decomposition of a    */
                                                /* general graph         */
      typedef    struct  gedstd      GEDSTD;    /* line in weak / strong */
                                                /* components            */
                                                /* decomposition of a    */
                                                /* general graph         */
      typedef    struct  gincsqrstr  GINCSQR;   /* list of incidences    */
//   6.      Biblock Decomposition
      typedef    struct  clcstr      CLC;       /* Cyclic weak component */
      typedef    struct  substr      SUB;       /* Subcomponent          */
      typedef    struct  blbstr      BLB;       /* Biblock               */
      typedef    struct  itstr       IT;        /* Internal tree         */
      typedef    struct  ptstr       PT;        /* Peripheral tree       */
      typedef    struct  avtstd      AVTSTD;    /* Extension of VERTEX   */
                                                /* record                */
      typedef    struct  aedstd      AEDSTD;    /* Extension of EDGE     */
                                                /* record                */
      typedef    struct  incsqrstr   INCSQR;    /* biblock of a vertex   */
//   7.      Additional DFS Structure
      typedef    struct  dfsvtstr    DFSVT;     /* Vertex extension for  */
                                                /* additional dfs        */
                                                /* structure             */
      typedef    struct  dfsedstr    DFSED;     /* Edge extension for    */
                                                /* additional dfs        */
                                                /* structure             */
//   8.      General Partitions
//   9.      Paths

      typedef    struct  phdrstr     PHDR;      /* Path header           */
      typedef    struct  pthastr     PTHA;      /* line on a path        */
//   10.     Menger Structures
      typedef     struct   mgdescrstr MGDESCR;  /* Descriptor for a      */
                                                /* Menger analysis       */
      typedef     struct   mgsepdescrstr SEPDESCR; /* Separating elements */
//   11.     Higher A-Decomposition
      typedef     struct   hstdd HSTDD;          /* Main record for      */
                                                 /* higher decomposition */
      typedef     struct   hcomp HCOMP;          /* Element of a higher  */
                                                 /* decompositions       */
      typedef     struct   hvtstd HVTSTD;        /* vertex record for    */
                                                 /* higher decompositions */
      typedef     struct   hedstd HEDSTD;        /* line record for      */
```

```
//   12.     Distances
                                                /* higher decompositions */
```

C.3 Functions and Global Variables

C.3.1 Explicit Functions

```
/*************************************************************************/
/*   Functions and global variables.                                     */
/*************************************************************************/

/*    Explicit functions: See user manual.                               */
/*               Basic Graph Functions:                                  */

  GRAPH      *readgraphlist(char *filename);           /* basics.c       */
  void        savegraphlist(GRAPH  *graph,             /* basics.c       */
                            char *filename);
  void        releasegraphlist(GRAPH *graph);          /* util.c         */
  void        printgrlist(GRAPH *graph, char *option,  /* basics.c       */
                          char *filename);
  void        printdfs(GRAPH *graph, char *vname,      /* basics.c       */
                       char* option, char *filename);
  void        printbfs(GRAPH *graph, char *vname,      /* basics.c       */
                       char* option, char *filename);
  void        dfsstructure(GRAPH *graph, char *vname,  /* basics.c       */
                         char *option);
  void        releasedfs(GRAPH *graph);                /* util.c         */

/*               Sets                                                    */

  VTSET      *readvtset(GRAPH *graph, char *filename); /* gsets.c        */
  VTSET      *gset2vtset(GSET *gset, GRAPH *graph);    /* gsets.c        */
  void        savevtset(VTSET *vtset, char * filename);/* gsets.c        */
  void        releasevtsetlist(VTSET *vtset);          /* util.c         */
  void        add2vtset(GRAPH * graph,                 /* gsets.c        */
                VTSET **pvtset, char *name);
  void        addvtset2vtset(GRAPH *graph,             /* gsets.c        */
                VTSET **pvtset, RVERTEX *rvt);

  EDSET      *readedset(GRAPH *graph, char *filename); /* gsets.c        */
  EDSET      *gset2edset(GSET *gset, GRAPH *graph);    /* gsets.c        */
  void        saveedset(EDSET *edset, char *filename); /* gsets.c        */
  void        releaseedsetlist(EDSET *edset);          /* util.c         */
  void        add2edset(GRAPH * graph,                 /* gsets.c        */
                EDSET **pedset, char *name);
  void        addedset2edset(GRAPH *graph,             /* gsets.c        */
                EDSET **pedset, REDGE *red);
```

```
  GSET       *readgset(char *filename);                /* gsets.c        */
  void        savegset(GSET *gseti, char *filenema);   /* gsets.c        */
  void        releasegsetlist(GSET *gset);             /* utilc         */
  BOOLEAN     add2gset(GSET **pgset, char *name);      /* gsets.c        */
  GSET       *gsetunion(RB *lista, int keya,           /* gsets.c        */
                        RB *listb, int keyb);
  GSET       *gsetintersect(RB *lista, int keya,       /* gsets.c        */
                          RB *listb, int keyb);
  GSET       *gsetdiff(RB *lista, int keya,            /* gsets.c        */
                       RB *listb, int keyb);

/*              Graph Generating Functions:                                */

 GRAPH      *generatefromvt(GRAPH *oldgraph,           /* generate.c     */
                 char *newname, RVERTEX *rvt);
 GRAPH      *generatefromed(GRAPH *oldgraph,           /* generate.c     */
                  char *newname, REDGE *red);
 GRAPH      *cpchtp(GRAPH *graph, char *newtype,       /* cpcht.c        */
               char *newname);                         /*                */
 void        degreedelete(GRAPH *oldgraph, GRAPH       /* generate.c     */
                 **delgraph, char *delname, GRAPH      /*                */
                 **remgraph, char *remname,            /*                */
                 VTSET **att, int dg);                 /*                */
 GRAPH      *generatefromcomp(GRAPH *oldgraph,         /* generate.c     */
                  char *gstdname, char *newgraph);
 GRAPH      *generatefromchnm(GRAPH *oldgraph,         /* generate.c     */
                  char *gstdname, char *newgraph);

/*              Weak and Strong Components                                 */

  void         gcomponents(GRAPH *graph);             /* gcomponents.c  */
  void         gprstd(GRAPH *graph, int stroption,    /* gstdutil.c     */
                    char *filename);
  void         gprstdvt(GRAPH *graph, int poption,    /* gstdutil.c     */
                      char *filename);

/*              Bibloc Decomposition                                       */

  void         astd(GRAPH *graph);                    /* astd.c         */
  BOOLEAN      generateblbgraph(GRAPH *graph,         /* astd.c         */
                            char *filename);
  void         aprstd(GRAPH *graph, int stroption,    /* astdutil.c     */
                    char *filename);
  void         aprstdvt(GRAPH *graph, int poption,    /* astdutil.c     */
                    char * filename);

/*              Path Functions                                             */
```

```
  PHDR         *newphdr(GRAPH *graph, char *direction,  /* path.c         */
                                char *name);

  PTHA         *insertline2path(PHDR *phdr,             /* path.c         */
                  EDGE *line, VERTEX *first,
                                char *position);
  BOOLEAN       insertpath2path(PHDR *phdr1,            /* path.c         */
                                PHDR *phdr2);
  void         removepath(PHDR *phdr);                  /* path.c         */
  void         releasepathlist(PHDR *phdr);             /* path.c         */
  PHDR        *readpath(GRAPH *graph,                   /* pathutil.c     */
                       char *filename);
  PHDR        *readpathlist(GRAPH *graph,               /* pathutil.c     */
                       char *filename);
  void         savepathlist(PHDR *list, char *filename);/* pathutil.c     */
  void         printpathlist(PHDR *phdr,                /* pathutil.c     */
                        char *stroption, char *filename);

/*               Menger Functions                                          */

  MGDESCR      *mengerstr(GRAPH *graph, VTSET * vtsource,   /* menger.c   */
                       VTSET *vtsink, char *mode,
                       char direction, char *name, int bound,
                       char *sep);
  void          mgprstd(MGDESCR *mgdescr, int stroption,    /* mengerutil.c  */
                         char *filename);
  void          releasesepdescrlist(SEPDESCR *sepdescr);

/*               Distance Functions                                        */

/*               Red-Black Trees                                           */

  BOOLEAN     ntreeinsert(RB **tree, RB *elem,          /* rbtree.c       */
                          int key);
  BOOLEAN     rbtreeinsert(RB **tree, RB *elem,         /* rbtree.c       */
                          int key);
  BOOLEAN     rbtreedelete(RB **tree, RB *elem,         /* rbtree.c       */
                           int key);
  BOOLEAN     ntreedelete(RB **tree, RB *elem,          /* rbtree.c       */
                          int key);
  RB         *rbtreefind(RB *tree, RB  *elem,           /* rbtree.c       */
                           int key);
  RB         *rbtreepfind(RB *tree, RB  *elem,          /* rbtree.c       */
                           int key);
  int         rbtreesize(RB *tree);                     /* rbtree.c       */
  RB         *rbtreemin(RB *tree);                      /* rbtree.c       */
  RB         *rbtreemax(RB *tree);                      /* rbtree.c       */
  RB         *rbtreenext(RB *tree, RB  *elem,           /* rbtree.c       */
```

```
                int key);
  RB          *rbtreeprevious(RB *tree, RB  *elem,       /* rbtree.c       */
                   int key);
  void         printorder(FILE *fd, RB *ndx,             /* rbtree.c       */
                          int key, char *ftext);
  void         printrbtree(FILE *fd, RB  *ndx,           /* rbtree.c       */
                          int level, int key);
  BOOLEAN      testtree(FILE *fd, RB *root,int key,       /* rbtree.c       */
                        char option);
  void         getname(RB *ndx, int key);                /* rbtree.c       */
  char         *getcharname(RB *ndx, char *comptype);    /* rbtree.c       */


/*                Queues and Stacks                                        */

  void         enqueue(QUSTA *qusta, void *qelem);       /* qusta.c        */
  void         fenqueue(QUSTA *qusta, void *qelem);      /* qusta.c        */
  void         push(QUSTA *qusta, void *qelem);          /* qusta.c        */
  void         fpush(QUSTA *qusta, void *qelem);         /* qusta.c        */
  void        *dequeue(QUSTA *qusta);                    /* qusta.c        */
  void        *fdequeue(QUSTA *qusta);                   /* qusta.c        */
  void        *qfront(QUSTA *qusta);                     /* qusta.c        */
  void        *qend(QUSTA *qusta);                       /* qusta.c        */
  void        *pop(QUSTA *qusta);                        /* qusta.c        */
  void        *fpop(QUSTA *qusta);                       /* qusta.c        */
  void        *top(QUSTA *qusta);                        /* qusta.c        */
  void         qscreate(RB *list);                       /* qusta.c        */
  void         qsremove(RB *list);                       /* qusta.c        */
  void         releasequstalist(QUSTA *qusta);           /* util.c         */
```

C.3.2 Implicit Functions

```
/*************************************************************************/
/*    Implicit functions in util.c:                                      */
/*************************************************************************/
//   B.      General Service Functions
  VERTEX      *otherend(EDGE *edge, VERTEX * vertex1);    /* util.c B.1    */
  void         resetaux(RB *rb);                          /* util.c B.2    */
  void        *collam(size_t size, char *message);        /* util.c B.3    */
  void         eerf(void *freepointer, size_t size);      /* util.c B.4    */
  char        *mnewname(char *string, char *message);     /* util.c B.5    */
  void         releasename(char *);                       /* util.c B.6    */
  BOOLEAN      boolstrcmp(char *str1, char *str2);        /* util.c B.7    */
  int          putsizeposition(RELEM *rel, int pos,       /* util.c B.8    */
                 int *possize);
  void         initheader(RB *record);                    /* util.c B.9    */
```

```
  int         gcd(int m, int n);                          /* util.c B.10   */
  void        mkqufromrb(RB *rb, QUSTA *qu);              /* util.c B.11   */
  void        mkstckfromrb(RB *rb, QUSTA *stck);          /* util.c B.12   */
  void        memstatistics(void);                        /* util.c B.13   */
  void       *findcomponent(GRAPH *graph,                 /* util.c B.14   */
                            char *gstdname);
  char       *colorname(int clr);                         /* util.c B.15   */
//   C.       Functions for Allocating Records
  RELEM      *mnewrelem(char *message);                   /* util.c C.1.1.1*/
  QUSTA      *mnewqusta(char *message);                   /* util.c C.1.3.1*/
  QS         *mnewqs(char *message);                      /* util.c C.1.3.2*/
  GRAPH      *mnewgraph(char *message);                   /* util.c C.2.1  */
  VERTEX     *mnewvt(char *message);                      /* util.c C.2.2  */
  RVERTEX    *mnewrvertex(char *message);                 /* util.c C.2.3  */
  EDGE       *mnewed(char *message);                      /* util.c C.2.4  */
  REDGE      *mnewredge(char *message);                   /* util.c C.2.5  */
  INC        *mnewinc(char *message);                     /* util.c C.2.6  */
  VTSET      *mnewvtset(char *message);                   /* util.c C.3.1  */
  EDSET      *mnewedset(char *message);                   /* util.c C.3.2  */
  GSET       *mnewgset(char *message);                    /* util.c C.3.3  */
  SETELEM    *mnewsetelem(char *message);                 /* util.c C.3.4  */
  GSTDD      *mnewgstdd(char *message);                   /* util.c C.4.1  */
  WCOMP      *mnewwcomp(char *message);                   /* util.c C.4.2  */
  SCOMP      *mnewscomp(char *message);                   /* util.c C.4.3  */
  GVTSTD     *mnewgvtstd(char *message);                  /* util.c C.4.4  */
  GEDSTD     *mnewgedstd(char *message);                  /* util.c C.4.5  */
  SUB        *mnewsub(char *message);                     /* util.c C.5.1  */
  BLB        *mnewblb(char *message);                     /* util.c C.5.2  */
  IT         *mnewit(char *message);                      /* util.c C.5.3  */
  PT         *mnewpt(char *message);                      /* util.c C.5.4  */
  AVTSTD     *mnewavtstd(char *message);                  /* util.c C.5.5  */
  AEDSTD     *mnewaedstd(char *message);                  /* util.c C.5.6  */
  INCSQR     *mnewincsqr(char *message);                  /* util.c C.5.7  */
  DFSVT      *mnewdfsvt(char *message);                   /* util.c C.6.1  */
  DFSED      *mnewdfsed(char *message);                   /* util.c C.6.2  */
  PHDR       *mnewphdr(char *message);                    /* util.c C.9.2  */
  PTHA       *mnewptha(char *message);                    /* util.c C.9.3  */
  MGDESCR    *mnewmgdescr(char *message);                 /* util.c C.10.1 */
//   D.       Functions for Releasing List of Records
  void        releaserellist(RELEM *qusta);               /* util.c D.1.1.1*/
  void        releaseqstalist(QUSTA *qusta);              /* util.c D.1.3.1*/
  void        releaseqslist(QS *qs);                      /* util.c D.1.3.2*/
  void        releasevtlist(VERTEX *vt);                  /* util.c D.2.2  */
  void        releaservtlist(RVERTEX *rvt);               /* util.c D.2.3  */
  void        releaseedlist(EDGE *ed);                    /* util.c D.2.4  */
  void        releaseredlist(REDGE *red);                 /* util.c D.2.5  */
  void        releaseinclist(INC *inc);                   /* util.c D.2.6  */
  void        releasesellist(SETELEM *rel);               /* util.c D.3.4  */
```

```
  void        releasegstddlist(GSTDD *gstdd);           /* util.c D.4.1  */
  void        releasewcomplist(WCOMP *wcomp);           /* util.c D.4.2  */
  void        releasescomplist(SCOMP *scomp);           /* util.c D.4.3  */
  void        releasegvtstdext(VERTEX *vt);             /* util.c D.4.4  */
  void        releasegedstdext(EDGE *ed);               /* util.c D.4.5  */
  void        releaseastd(GRAPH *graph);                /* util.c D.5.1  */
  void        releasesublist(SUB *sub);                 /* util.c D.5.2  */
  void        releaseblblist(BLB *blb);                 /* util.c D.5.3  */
  void        releaseitlist(IT *it);                    /* util.c D.5.4  */
  void        releaseptlist(PT *pt);                    /* util.c D.5.5  */
  void        releaseavtstdext(VERTEX *vt);             /* util.c D.5.6  */
  void        releaseaedstdext(EDGE *ed);               /* util.c D.5.7  */
  void        releaseincsqrlist(INCSQR *incsqr);        /* util.c D.5.8  */
  void        releasedfsvtext(VERTEX *vt);              /* util.c D.6.2  */
  void        releasedfsedext(EDGE *ed);                /* util.c D.6.2  */
  void        releasepdhrlist(PHDR *phdr);              /* util.c D.9.2  */
  void        releasephdrlist(PHDR *phdr);
  void        releasepthalist(PTHA *ptha);              /* util.c D.9.3  */
  void        releasemgdescrlist(MGDESCR *mgdescr);     /* util.c D.10.1 */
/*************************************************************************/
/*    Implicit functions in rbtree.c:                                    */
/*************************************************************************/
  int         rbtcomp(RB *ls1, RB *ls2,  int key);
  char       *keyname(int key);
```

C.3.3 Global Variables

```
/*                                                                       */
/*    Global variables                                                   */
  extern      long int ghsmemsize;       /* global counter of          */
                                         /* allocated memory           */
  extern      BOOLEAN  btesttest;        /* for test purposes          */
  extern      char     namebuffer[1001]; /* common buffer for          */
                                         /* strings                    */
  extern      char    *typearray[];      /* graph types as strings     */
  extern      char    *stypearray[];     /* graph subtypes as strings  */
  extern      char    *vtclassarray[];   /* vertex class               */
  extern      char    *vtstatarray[];    /* vertex processing status   */
  extern      char    *edclassarray[];   /* edge class                 */
  extern      char    *edstatarray[];    /* edge processing status     */
  extern      char    *poptionarray[];   /* print option               */
  extern      char    *edcolorarray[];   /* line color                 */
  extern      char    *BLBstring;        /* pointer to "BLN"           */
  extern      char    *ITstring;         /* pointer to "IT"            */
  extern      char    *PTstring;         /* pointer to "PT"            */
  extern      char    *ATTstring;        /* pointer to "ATT"           */
```

C.4 Data Structures for Utilities

C.4.1 Data Structures for General Sevices

```
/*************************************************************************/
/*                                                                       */
/*        RELEM       Generic substrucure.               C.1.1.1         */
/*                                                                       */
/*        RELEMCLASS  Type of elementary substructure                    */
/*                                                                       */
/*************************************************************************/

struct relemstr     /* RELEM Generic substructure                        */
{
   RELEM         *relleft;
   RELEM         *relright;
   RELEM         *relpar;
   COLOR          relcolor;
   void          *relauxiliary; /* for future extensions                */
   int            reltype;      /* type of substructure                 */
                                /* -1: no type assigned yet             */
   void          *relp;         /* pointer to substructure              */
};

   enum relemclass  /* RELEMCLASS Type of elementary substructure       */
        {      TYGRA,              /* Graph                             */
               TYSGR,              /* Subgraph                          */
               TYSUB,              /* Subcomponent                      */
               TYBLB,              /* BLB                               */
               TYIT,               /* IT                                */
               TYPT,               /* PT                                */
//             TYAP,               /* attachment point                  */
               TYWCOMP,            /* weakly connected component        */
               TYSCOMP,            /* strongly connected component      */
//             TYCLASS,            /* EDCLASS record                    */
//             TYHCOMP,            /* HCOMP record                      */
//             TYEDSTD,            /* EDSTD record                      */
               TYDUMMY             /* Points to no record class,        */
                                   /* but carries a value in the        */
                                   /* auxiliary field                   */
        };
   enum COLORS   /* Colors for processing and classifying             */
        { CLRNONE,               /* uncolored                         */
          CLRBROWN,              /* brown                             */
          CLRGREEN,              /* green                             */
          CLRRED,                /* red                               */
          CLRBLUE,               /* blue                              */
          CLRYELLOW,             /* yellow                            */
```

```
        CLRPINK,                /* pink                                 */
        CLRORANGE,              /* orange                               */
        CLRMAGENTA,             /* magenta                              */
        CLRWHITE,               /* white                                */
        CLRBLACK                /* black                                */
         };
```

C.4.2 Data Structures for Red-Black Trees

```
/*************************************************************************/
/*                                                                       */
/*                                                                       */
/*        RB        Generic for red-black trees.                         */
/*                                                                       */
/*        For SORTKEY (type of sortig key) see end of file.              */
/*                                                                       */
/*************************************************************************/
struct  rbtree  /* RB                                                    */
{       RB    *left;
        RB    *right;
        RB    *par;
        COLOR  color;
        void  *auxiliary;
};
```

C.4.3 Data Structures for Stacks and Queues

```
/*************************************************************************/
/*                                                                       */
/*        QUSTA     Stacks an queues for arbitrary GHS elements. C.1.3.1 */
/*                                                                       */
/*        QS        Elements in queues and stacks.              C.1.3.2  */
/*                  Are not the records to be queued/stacked but         */
/*                  their representatives.                               */
/*                                                                       */
/*************************************************************************/
struct  ghsqusta  /* QUSTA                                               */
{       QUSTA        *qustaleft;
        QUSTA        *qustaright;
        QUSTA        *qustapar;
        COLOR         qustacolor;
        void         *qustaauxiliary;
        QS           *qustastart;
        QS           *qustaend;
        QS           *qustatop;
};
struct  ghsqs     /* QS                                                  */
{       QS           *qsleft;
```

```
        QS          *qsright;
        QS          *qspar;
        COLOR        qscolor;
        void        *qsauxiliary;      // may be used only as pointer
                                       // to list of RVERTEX records
                                       // (see distance.c)
        QS          *qsqforward;       // forward pointer in queue
        QS          *qsqbackward;      // backward pointer in queue
        QS          *qssforward;       // forward pointer in stack
        QS          *qssbackward;      // backward pointer in stack
        void        *qsp;              // pointer to original record
        int          qshelp1;          // application dependent
        int          qshelp2;          // application dependent
        void        *qsphelp;          // application dependent
        BOOLEAN      qsinqueue;        // record in queue
        BOOLEAN      qsinstack;        // record in stack
};
```

C.5 Basic Graph Data Structures

```
/*************************************************************************/
/*                                                                       */
/*        GRAPH     Structure of graphs.                      C.2.1      */
/*                                                                       */
/*        GRAPHTYPE Indicator for type of graph.                         */
/*                                                                       */
/*        VERTEX    Vertices.                                 C.2.2      */
/*                                                                       */
/*        RVERTEX   Describes role of vertex.                 C.2.3      */
/*                                                                       */
/*        EDGE      Edges/arcs.                               C.2.4      */
/*                                                                       */
/*        REDGE     Describes role of line.                   C.2.5      */
/*                                                                       */
/*        INC       Incidences.                               C.2.6      */
/*                                                                       */
/*************************************************************************/

struct grphst   /* GRAPH graph description                             */
 {
   GRAPH         *grleft;
   GRAPH         *grright;
   GRAPH         *grpar;
   COLOR          grcolor;
   void          *grauxiliary;  /* for future extensions               */
   char          *grname;       /* name of the graph                   */
   int            grtype;       /* graph type                          */
```

```
int           grvtno;      /* number of vertices                    */
VERTEX       *grvtlist;    /* list of vertices                      */
int           gruedno;     /* number of undirected edges            */
EDGE         *gredlist;    /* list  of edges                        */
int           grdedno;     /* number of directed edges (arcs)       */
EDGE         *grdedlist;   /* list  of directed edges (arcs)        */
GSTDD        *grgcomp;     /* pointer to weak/strong decomposition  */
BOOLEAN       grastd;      /* = TRUE: Biblock decomposition exists  */
                           /* (The biblock decomposition, if it     */
                           /* exists, starts with the weak com-     */
                           /* ponents of the weak and strong        */
                           /* decomposition)                        */
BOOLEAN       grsga;       /* = TRUE: The higher decomposition into */
                           /*         maximal a-components exists   */
                           /* This decomposition, if it exists,     */
                           /* starts with the biblocks of the       */
                           /* biblock decomposition                 */
BOOLEAN       grsgal;      /* = TRUE: The higher decomposition into */
                           /*         maximal a-linecomponents      */
                           /*         exists                        */
                           /* This decomposition, if it exists,     */
                           /* starts with the subcomponents of the  */
                           /* biblock decomposition                 */
BOOLEAN       grvsa;       /* = TRUE: The higher decomposition into */
                           /*         maximal a-connected vertex    */
                           /*         sets exists                   */
                           /* This decomposition, if it exists,     */
                           /* starts with the biblocks of biblock   */
                           /* decomposition                         */
BOOLEAN       grvsal;      /* = TRUE: The higher decomposition into */
                           /*         maximal a-lineconnected vertex*/
                           /*         sets exists                   */
                           /* This decomposition, if it exists,     */
                           /* starts with the subcomponents of the  */
                           /* biblock decomposition                 */
BOOLEAN       grsgf;       /* = TRUE: The higher decomposition into */
                           /*         maximal f-components exists   */
                           /* This decomposition, if it exists,     */
                           /* starts with the strong components of  */
                           /* the weak and strong decomposition     */
BOOLEAN       grsgfl;      /* = TRUE: The higher decomposition into */
                           /*         maximal f-linecomponents      */
                           /*         exists                        */
                           /* This decomposition, if it exists,     */
                           /* starts with the strong components of  */
                           /* the weak and strong decomposition     */
BOOLEAN       grvsf;       /* = TRUE: The higher decomposition into */
                           /*         maximal f-connected vertex    */
```

```
                              /*         sets exists                    */
                              /* This decomposition, if it exists,      */
                              /* starts with the strong components of   */
                              /* the weak and strong decomposition      */
  BOOLEAN        grvsfl;      /* = TRUE: The higher decomposition into  */
                              /*         maximal f-lineconnected vertex */
                              /*         sets exists                    */
                              /* This decomposition, if it exists,      */
                              /* starts with the strong components of   */
                              /* the weak and strong decomposition      */
  int            grdfs;       /* = 0: there is no additional dfs tree   */
                              /*      structure                         */
                              /* = 1: there is an additional f-dfs      */
                              /*      tree structure                    */
                              /* = 2: there is an additional b-dfs      */
                              /*      tree structure                    */
                              /* = 3: there is an additional a-dfs      */
                              /*      tree structure                    */
  VERTEX        *grdfsroot;   /* starting vertex for dfs structure      */
  int            grsubtype;   /* graph subtype                          */
  void          *grsysattr;   /* pointer to system attribute structure  */
  VERTEX        *grroot;      /* root (optional)                        */
  void          *grattr;      /* pointer to user attribute structure    */
  int            grattrtype;  /* type of user attribute structure       */
                              /* = -1: No attributes                    */
                              /* = 0:  Direct paths                     */
};
  enum grtp   /* GRAPHTYPE */
       {      GG,             /* general graph, all kinds of lines      */
                              /* allowed                                */
              DG,             /* general digraph                        */
              DGS,            /* digraph, single arcs                   */
              DGSLF,          /* digraph, single arcs, loopfree         */
              UG,             /* general undirected graph               */
              UGS,            /* undirected graph, single edges         */
              UGSLF,          /* undirected graph, single edges,        */
                              /* loopfree                               */
       };
  enum grsubtp   /* GRAPHSUBTYPE */
       {      GSUBNA,         /* no subtype specified                   */
      GSUBDIR,         /* graph representing direct path         */
              GSUBDIST1,      /* distance graph centered in grroot      */
              GSUBDIST2,      /* distance graph from the root to a      */
                              /* second vertex                          */
              GSUBDIST3,      // Global distance graph. For each vertex
                              // the eccentricity and a list of most
                              // distant vertices are recorded in
                              // a QS record attached to the vertex.
```

```
                                // Field grroot points to a list
                                // of RVERTEX records ordered by
                                //     1. eccentricity
                                //     2. by vertex name
        };
struct  vtstr      /* VERTEX                                              */
{
   VERTEX        *vtleft;
   VERTEX        *vtright;
   VERTEX        *vtpar;
   COLOR          vtcolor;
   void          *vtauxiliary;    /* For future extensions               */
   char          *vtname;         /* name of vertex                      */
   GRAPH         *vtgraph;        /* graph vertex belongs to             */
   int            vtdg;           /* degree of vertex: no. of            */
                                  /* undirected edges.                   */
                                     /*********************************
                                     * Undirected loops are counted  *
                                     * only once !!!                 *
                                     *********************************/
   int            vtidg;          /* indegree of vertex                  */
   int            vtodg;          /* outdegree of vertex                 */
   INC           *vtinclist;      /* pointer to incindence list          */
                                  /* (undirected edges)                  */
   INC           *vtiinclist;     /* pointer to incindence list          */
                                  /* (incoming arcs)                     */
   INC           *vtoinclist;     /* pointer to incindence list          */
                                  /* (outgoing arcs)                     */
   GVTSTD        *vtptgstd;       /* decomposition extension of record:  */
                                  /* weak and strong components          */
   AVTSTD        *vtptastd;       /* decomposition extension of record:  */
                                  /* biblock decomposition               */
   void          *vtpthstd;       /* decomposition extension of record:  */
                                  /* higher decompositions               */
   DFSVT         *vtptdfs;        /* extension record for additional     */
                                  /* dfs structure                       */
   void          *vtattr;         /* pointer to attribute structure      */
   int            vtattrtype;     /* type of attribute structure         */
                                  /* = -1: No attribute structure        */
                                  /* =  0: User specific attributes,     */
                                  /*       not supported by GHS          */
                                  /* Attributes supported by GHS:        */
                                  /*    none at the time                 */
};
struct  rvtstr     /* RVERTEX                                             */
{
   RVERTEX       *rvtleft;
   RVERTEX       *rvtright;
```

```
   RVERTEX       *rvtpat;
   COLOR          rvtcolor;
   void          *rvtauxiliary;    /* for future extensions               */
   VERTEX        *rvtp;            /* pointer to vertex                   */
};
struct  edstr      /* EDGE                                                */
{
   EDGE          *edleft;
   EDGE          *edright;
   EDGE          *edpar;

   COLOR          edcolor;
   void          *edauxiliary;     /* for future extensions               */
   char          *edname;          /* line name                           */
   char           edtype;          /* type (directed or undirected)       */
                                   /*    'u' for undirected               */
                                   /*    'd' for directed                 */
   VERTEX        *edfirst;         /* first vertex resp. head             */
   VERTEX        *edsec;           /* second vertex resp. tail            */
   GEDSTD        *edptgstd;        /* decomposition extension of record   */
                                   /* weak and strong components          */
   AEDSTD        *edptastd;        /* decomposition extension of record   */
                                   /* biblock decomposition               */
   DFSED         *edptdfs;         /* extension of record for additional  */
                                   /* dfs structure                       */
   void          *edattr;          /* pointer to attribute structure      */
   int            edattrtype;      /* type of attribute structure         */
                                   /* = -1: No attributes                 */
                                   /* =  0: User specific attributes,     */
                                   /*       not supported by GHS          */
};
struct  redstr      /* REDGE */
{
   REDGE         *redleft;
   REDGE         *redright;
   REDGE         *redpar;
   COLOR          redcolor;
   void          *redauxiliary;    /* for future extensions */
   EDGE          *redp;            /* Pointer to line */
};
struct  incstr     /* INC  Incidence entry */
{
   INC         *incleft;
   INC         *incright;
   INC         *incpar;
   COLOR        incolor;
   void        *incauxiliary;      /* for future extensions */
   EDGE        *incedge;           /* pointer to line */
```

```
};
```

C.6 Data Structures for Sets

```
/*************************************************************************/
/*                                                                       */
/*        VTSET     Vertex set.                        C.3.1             */
/*        EDSET     Edge set.                          C.3.2             */
/*        GSET      General set.                       C.3.3             */
/*        SETELEM   Element of general set.            C.3.4             */
/*                                                                       */
/*************************************************************************/
struct vtsetstr     /* VTSET Vertex set */
{
   VTSET         *vtsetleft;
   VTSET         *vtsetright;
   VTSET         *vtsetpar;
   COLOR   vtsetcolor;
   void          *vtsetauxiliary; /* for future extensions */
   int            vtsetcard;      /* No. of elements */
   RVERTEX       *vtsetlist;      /* pointer to vertices */
};
struct edsetstr     /* EDSET Edges set */
{
   EDSET         *edsetleft;
   EDSET         *edsetright;
   EDSET         *edsetpar;
   COLOR          edsetcolor;
   void          *edsetauxiliary; /* for future extensions */
   int            edsetcard;      /* No. of elements */
   REDGE         *edsetlist;      /* pointer to lines */
};
struct gsetstr      /* GSET General set */
{
   GSET          *gsetleft;
   GSET          *gsetright;
   GSET          *gsetpar;
   COLOR          gsetcolor;
   void          *gsetauxiliary; /* for future extensions */
   int            gsetcard;      /* No. of elements */
   SETELEM       *gsetlist;      /* pointer to vertices */
};
struct setelem      /* SETELEM Element of general set */
{
   SETELEM       *selleft;
   SETELEM       *selright;
   SETELEM       *selpar;
```

```
   COLOR          selcolor;
   void          *selauxiliary;
   char          *selname;
};
```

C.7 Data Structures for Weak and Strong Components

```
/*************************************************************************/
/*                                                                       */
/*        GSTDD       Weak and strong components          C.4.1          */
/*        WCOMP       Weakly connected component.         C.4.2          */
/*        SCOMP       Strongly connected component.       C.4.3          */
/*        GVTSTD      Vertex extension for weak and       C.4.4          */
/*                    strong components                                  */
/*        GEDSTD      Edge/arc extension for weak         C.4.5          */
/*                    and strong components                              */
/*************************************************************************/
struct gstdd        /* GSTDD Weak and strong components                  */
 {
   GSTDD         *gstdleft;
   GSTDD         *gstdright;
   GSTDD         *gstdpar;
   COLOR          gstdcolor;
   void          *gstdauxiliary; /* for future extensions               */
   GRAPH         *gstdgraph;     /* pointer to graph                    */
   GVTSTD        *gvtstdlist;    /* red-black tree of GVTSTD records    */
   int            gisvtno;       /* number of isolated vertices         */
   RVERTEX       *gisvtlist;     /* list of isolated vertices           */
   int            wcompno;       /* number of weak components           */
   int            wacno;         /* number of a-acyclic weak components */
                                 /* without strong components           */
   WCOMP         *waclist;       /* list of a-acyclic weak components   */
                                 /* without strong components           */
   int            wacsno;        /* number of a-acyclic weak components */
                                 /* with strong components              */
   WCOMP         *wacslist;      /* list of a-acyclic weak components   */
                                 /* with strong components              */
   int            wccno;         /* number of a-cyclic weak components  */
                                 /* without strong components           */
   WCOMP         *wcclist;       /* list of a-cyclic weak components    */
                                 /* without strong components           */
   int            wccsno;        /* number of a-cyclic weak components  */
                                 /* with all strong components f-acyclic*/
   WCOMP         *wccslist;      /* list of a-cyclic weak components    */
                                 /* with all strong components f-acyclic*/
   int           wccsfno;        /* number of a-cyclic weak components  */
                                 /* with at least one f-cyclic strong   */
```

```
                                /* component                                */
   WCOMP         *wccsflist;     /* list of a-cyclic weak components         */
                                /* with at least one f-cyclic strong        */
                                /* component                                */
   BOOLEAN        gstdastd;      /* = TRUE: The biblock decomposition        */
                                /*         of the graph has been            */
                                /*         determined                       */
 };
  struct  wcomp    /* WCOMP weak component                                   */
{
   WCOMP       *wcompleft;
   WCOMP       *wcompright;
   WCOMP       *wcomppar;
   COLOR        wcompcolor;
   void        *wcompauxiliary;
   int          wcomptype;      /* type of weak component                   */
                                /* = -1: not yet assigned                   */
                                /* = 1:  a-acyclic, no strong components    */
                                /* = 2:  a-acyclic, strong components       */
                                /* = 3:  a-cyclic, no strong components     */
                                /* = 4:  a-cyclic, strong components,       */
                                /*       all strong components f-acyclic    */
                                /* = 5:  a-cyclic, strong components,       */
                                /*       at least one f-cyclic strong       */
                                /*       component                          */
   int          wcompnumber;    /* number of weak component                 */
   GSTDD       *wcompgstdd;     /* pointer to weak and strong               */
                                /* decomposition.                           */
   int          wcompvtno;      /* number of vertices                       */
   RVERTEX     *wcompvtlist;    /* list of vertices                         */
   int          wcompedno;      /* number of edges                          */
   REDGE       *wcompedlist;    /* list of edges                            */
   int          wcompdedno;     /* number of arcs                           */
   REDGE       *wcompdedlist;   /* list of arcs                             */
   int          wcompexdvtno;   /* number of vertices of external dag       */
   RVERTEX     *wcompexdvtlist; /* list of vertices of external dag        */
   int          wcompexddedno;  /* number of arcs of external dag           */
   REDGE       *wcompexddedlist; /* list of arcs of external dag            */
   int          wcompwattno;    /* number of weak attachment points         */
   RVERTEX     *wcompwattlist;  /* list of weak attachment points           */
   int          wcompstrongno;  /* number of strong components              */
   int          wcompsno;       /* number of f-acyclic strong               */
                                /* components                               */
   SCOMP       *wcompslist;     /* list of f-acyclic strong components      */
   int          wcompsfno;      /* number of f-cyclic strong components     */
   SCOMP       *wcompsflist;    /* list of f-cyclic strong components       */
   int          wcompperiod;    /* a-period of weak component               */
                                /* = -1:   not yet assigned                 */
```

```
   int          wcomproot;      /* root type of weak component:           */
                                /* = -1:   not yet assigned               */
                                /* = 0:    component is not rooted        */
                                /* = 1:    there is a vertex of no        */
                                /*         return with root property      */
                                /* = 2:    there is a strong component    */
                                /*         with root property             */
   void        *wcomprootp;     /* = NULL: Not applicable                 */
                                /* !=NULL: pointer to vertex              */
                                /*         if wcomproot == 1.             */
                                /*         pointer to strong component    */
                                /*         if wcomproot == 2.             */
   int          wcompstpvtno;   /* number of vertices of stopfree kernel  */
   RVERTEX     *wcompstpvtlist; /* list of vertices of stopfree kernel    */
   int          wcompstpedno;   /* number of lines of stopfree kernel     */
   REDGE       *wcompstpedlist; /* list of lines of stopfree kernel       */
   int          wcompstpdedno;  /* number of arcs of stopfree kernel      */
   REDGE       *wcompstpdedlist;/* list of arcs of stopfree kernel        */
   int          wcompsubno;     /* number of subcomponents                */
   SUB         *wcompsublist;   /* list of subcomponents                  */
   int          wcompblbno;     /* number of biblocks                     */
   RELEM       *wcompblblist;   /* list of biblocks                       */
   int          wcompitno;      /* number of internal trees               */
   IT          *wcompitlist;    /* list of internal trees                 */
   int          wcompptno;      /* number of peripheral trees             */
   PT          *wcompptlist;    /* pointer to list of peripheral trees    */
   int          wcompattno;     /* number of attachment points            */
   RVERTEX     *wcompattlist;   /* list of attachment points              */
   int          wcompborno;     /* number of border points                */
   RVERTEX     *wcompborlist;   /* list of border points                  */
   int          wcompcheno;     /* number of check points                 */
   RVERTEX     *wcompchelist;   /* list of check points                   */
   int          wcomphinno;     /* number of check points                 */
   RVERTEX     *wcomphinlist;   /* list of hinge points                   */
};
  struct  scomp     /* C.3.3  SCOMP strong component                      */
{
   SCOMP       *scompleft;
   SCOMP       *scompright;
   SCOMP       *scomppar;
   COLOR        scompcolor;
   void        *scompauxiliary;
   int          scomptype;      /* type of strong component               */
                                /* = -1: not yet assigned                 */
                                /* = 1:  f-acyclic                        */
                                /* = 2:  f-cyclic                         */
   int          scompnumber;    /* number of strong component             */
                                /* = -1: not yet assigned                 */
```

```
   WCOMP       *scompwcomp;     /* pointer to weak component             */
   int          scompvtno;      /* number of vertices                    */
   RVERTEX     *scompvtlist;    /* list of vertices                      */
   int          scompedno;      /* number of edges                       */
   REDGE       *scompedlist;    /* list of edges                         */
   int          scompdedno;     /* number of arcs                        */
   REDGE       *scompdedlist;   /* list of arcs                          */
   int          scompwattno;    /* number of weak attachment points      */
   RVERTEX     *scompwattlist;  /* list of weak attachment poits         */
   int          scomplevel;     /* level number                          */
                                /* = -1: not yet assigned                */
                                /* level numbering starts with 0         */
   int          scompperiod;    /* f-period of strong component          */
                                /* = -1: not yet assigned                */
};
  struct  gvtstd       /* GVTSTD Vertex in a weak and strong             */
                       /* decomposition of a general graph               */
 {
   GVTSTD        *gvtstdleft;
   GVTSTD        *gvtstdright;
   GVTSTD        *gvtstdpar;
   COLOR          gvtstdcolor;
   void          *gvtstdauxiliary;/* for future extension           */
   VERTEX        *gvtstdp;        /* pointer to vertex              */
   WCOMP         *gvtstdwcomp;    /* pointer to weak component      */
                                  /* = NULL: isolated vertex        */
   int            gvtstdtype;     /* type of vertex                 */
                                  /* = -1: not yet assigned         */
                                  /* = 0:  isolated vertex          */
                                  /* = 1:  internal vertex of       */
                                  /*       external dag (no return) */
                                  /* = 2:  internal vertex of       */
                                  /*       strong component         */
                                  /* = 3:  weak attachment point    */
   SCOMP         *gvtstdscomp;    /* pointer to strong component    */
                                  /* = NULL: not applicable         */
   int            gvtstdexdino;   /* number of incoming external    */
                                  /* dag arcs                       */
   REDGE         *gvtstdexdilist; /* list of incoming external      */
                                  /* dag arcs                       */
   int            gvtstdexdono;   /* number of outgoing external    */
                                  /* dag arcs                       */
   REDGE         *gvtstdexdolist; /* list of outgoing external      */
                                  /* dag arcs                       */
   int            gvtstdlevel;    /* level number                   */
                                  /* = -1: Not yet assigned         */
                                  /* all vertices of a strong       */
                                  /* component have the level number */
```

```
                                /* of that component              */
   BOOLEAN        gvtstdmarked;  /* General mark indicator         */

};
struct  gedstd       /* GEDSTD line in a weak and strong           */
                     /* decomposition                              */
{
   GEDSTD        *gedstdleft;
   GEDSTD        *gedstdright;
   GEDSTD        *gedstdpar;
   COLOR          gedstdcolor;
   void          *gedstdauxiliary;
   EDGE          *gedstdp;       /* pointer to arc                  */
   int            gedstdtype;    /* type of line                    */
                                 /* = -1: not yet assigned          */
                                 /* = 1:  edge; gedstdelcmp points  */
                                 /*       to strong component       */
                                 /* = 2:  arc of a strong component;*/
                                 /*       gedstdelcmp points        */
                                 /*       to strong component       */
                                 /* = 3:  arc of an external dag;   */
                                 /*       gedstdelcmp points        */
                                 /*       to weak component         */
   void          *gedstdelcmp;  /* pointer to elementary component */
                                 /* edge/arc belongs to             */
   BOOLEAN        gedstdmarked;  /* general mark indicator          */
};
```

C.8 Data Structures for the Biblock Decomposition

```
/*************************************************************************/
/*                                                                       */
/*        SUB         Subcomponent.                                      */
/*                                                                       */
/*        BLB         Biblock.                                           */
/*                                                                       */
/*        IT          Internal tree.                                     */
/*                                                                       */
/*        PT          Peripheral tree.                                   */
/*                                                                       */
/*        AVTSTD      Vertex in a biblock decomposition.                 */
/*                                                                       */
/*        RVSTDCLASS  Vertex classification.                             */
/*                                                                       */
/*        RVSTDSTAT   Vertex processing status.                          */
/*                                                                       */
/*        AEDSTD      Edge in a biblock decomposition.                   */
```

```
/*                                                                    */
/*        EDCLASS     Edge classification.                            */
/*                                                                    */
/*        EDSTAT      Edge processing status.                         */
/*                                                                    */
/*        INCSQR      List of incidences in biblocks.                 */
/*                    (Also used for general line partitions)         */
/*                                                                    */
/*        POPTION     Print options for vertex list (prstdvt).        */
/*                                                                    */
/*        STROPTION   Print options for the weak and strong decompostion */
/*                    and the biblock decomposition                   */
/**********************************************************************/
struct substr       /* SUB Subcomponent                               */
{
   SUB           *subleft;
   SUB           *subright;
   SUB           *subpar;
   COLOR          subcolor;
   void          *subauxiliary; /* for future extensions             */
   WCOMP         *subwcomp;     /* pointer to WCOMP                  */
   int            subnumber;    /* identifying number SUB            */
   int            subvtno;      /* number of vertices                */
   RVERTEX       *subvtlist;    /* list of vertices                  */
   int            subedno;      /* number of edges                   */
   REDGE         *subedlist;    /* list of edges                     */
   int            subdedno;     /* number of arcs                    */
   REDGE         *subdedlist;   /* list of arcs                      */
   int            subblbno;     /* number of biblocks                */
   BLB           *subblblist;   /* pointer to list of biblocks       */
   int            subattno;     /* number of attachment points       */
   RVERTEX       *subattlist;   /* list of attachment points         */
   int            subborno;     /* number of border points           */
   RVERTEX       *subborlist;   /* pointer to list of border points  */
   int            subhinno;     /* number of hinge points            */
   RVERTEX       *subhinlist;   /* pointer to list of hinge points   */
   int            subcheno;     /* number of check points            */
   RVERTEX       *subchelist;   /* pointer to list of checkpoints    */
};
struct blbstr       /* BLB Biblock                                    */
{
   BLB           *blbleft;
   BLB           *blbright;
   BLB           *blbpar;
   COLOR          blbcolor;
   void          *blbauxiliary; /* for future extensions             */
   SUB           *blbsub;       /* pointer to SUB                    */
   int            blbnumber;    /* identifying number BLB            */
```

```
   int            blbvtno;      /* number of vertices                    */
   RVERTEX       *blbvtlist;    /* pointer to vertex list                */
   int            blbedno;      /* number of edges                       */
   REDGE         *blbedlist;    /* list of edges                         */
   int            blbdedno;     /* number of arcs                        */
   REDGE         *blbdedlist;   /* list of arcs                          */
   int            blbattno;     /* number of attachment points           */
   RVERTEX       *blbattlist;   /* list of attachment points             */
   int            blbborno;     /* number of hinge points                */
   RVERTEX       *blbborlist;   /* pointer to list of hinge points       */
   int            blbcheno;     /* number of hinge points                */
   RVERTEX       *blbchelist;   /* pointer to list of hinge points       */
   int            blbhinno;     /* number of hinge points                */
   RVERTEX       *blbhinlist;   /* pointer to list of hinge points       */
};
struct itstr        /* IT Internal Tree                                  */
{
   IT            *itleft;
   IT            *itright;
   IT            *itpar;
   COLOR          itcolor;
   void          *itauxiliary;  /* for future extensions                 */
   WCOMP         *itwcomp;      /* pointer to WCOMP                      */
   int            itnumber;     /* identifying number IT                 */
   int            itvtno;       /* number of vertices                    */
   RVERTEX       *itvtlist;     /* pointer to vertex list                */
   int            itedno;       /* number of edges                       */
   REDGE         *itedlist;     /* pointer to list of edges              */
   int            itdedno;      /* number of arcs                        */
   REDGE         *itdedlist;    /* pointer to list of arcs               */
   int            itattno;      /* number of attachment points           */
   RVERTEX       *itattlist;    /* list of attachment points             */
   int            itborno;      /* number of border points               */
   RVERTEX       *itborlist;    /* pointer to list of border points      */
   int            itcheno;      /* number of check points                */
   RVERTEX       *itchelist;    /* pointer to list of check points       */
   int            ithinno;      /* number of hinge points                */
   RVERTEX       *ithinlist;    /* pointer to list of hinge points       */
};

struct ptstr        /* PT Peripheral Tree                                */
{
   PT            *ptleft;
   PT            *ptright;
   PT            *ptpar;
   COLOR          ptcolor;
   void          *ptauxiliary;  /* for future extensions                 */
   WCOMP         *ptwcomp;      /* pointer to WCOMP                      */
```

```
   int            ptnumber;     /* identifying number PT                */
   int            ptvtno;       /* number of vertices                   */
   RVERTEX       *ptvtlist;     /* pointer to vertex list               */
   int            ptedno;       /* number of edges                      */
   REDGE         *ptedlist;     /* pointer to list of edges             */
   int            ptdedno;      /* number of arcs                       */
   REDGE         *ptdedlist;    /* pointer to list of arcs              */
   VERTEX        *ptbor;        /* border point                         */
};

struct  avtstd      /* AVTSTD vertex in a biblock decomposition          */
{
   AVTSTD        *avtstdleft;
   AVTSTD        *avtstdright;
   AVTSTD        *avtstdpar;
   COLOR          avtstdcolor;
   void          *avtstdauxiliary; /* for future extension             */
   VERTEX        *avtstdp;         /* pointer to vertex                */
   int            avtstdclass;     /* class of vertex                  */
                                   /* see enum RVSTDCLASS              */
   BOOLEAN        avtstdhinge;     /* hinge point                      */
   BOOLEAN        avtstdcheck;     /* check point                      */
   BOOLEAN        avtstdborder;    /* border point                     */
   int            avtstdptedno;    /* number of peripheral tree edges  */
   REDGE         *avtstdptedlist;  /* pointer to incidence list of     */
                                   /* peripheral tree edges            */
   int            avtstdptiedno;   /* number of incoming peripheral    */
                                   /* tree arcs                        */
   REDGE         *avtstdptiedlist; /* pointer to incidence list of     */
                                   /* incoming peripheral tree arcs    */
   int            avtstdptoedno;   /* number of outgoing peripheral    */
                                   /* tree arcs                        */
   REDGE         *avtstdptoedlist; /* pointer to incidence list of     */
                                   /* outgoing peripheral tree arcs    */
   PT            *avtstdpt;        /* pointer to peripheral tree       */
   int            avtstditedno;    /* number of internal tree edges    */
   REDGE         *avtstditedlist;  /* list of internal tree edges      */
   int            avtstditiedno;   /* number of incoming internal      */
                                   /* tree arcs                        */
   REDGE         *avtstditiedlist; /* list of incoming internal        */
                                   /* tree arcs                        */
   int            avtstditoedno;   /* number of outgoing internal      */
                                   /* tree arcs                        */
   REDGE         *avtstditoedlist; /* list of outgoing internal        */
                                   /* tree arcs                        */
   IT            *avtstdit;        /* pointer to internal tree         */
   int            avtstdblbno;     /* number of biblocks containing    */
                                   /* the vertex                       */
```

```
   INCSQR        *avtstdblblist;   /* pointer to list of biblock          */
                                   /* edge lists                          */
   BOOLEAN        avtstdgreen;     /* TRUE: vertex is incident with an    */
                                   /* edge/arc of an internal tree        */
};
enum rvstdclass  /* RVSTDCLASS Classification                              */
      {      RVDNY,               /* not yet classified                  */
             RVDPV,               /*  vertex in peripheral tree          */
                                  /* (not attachement point)             */
             RVDIV,               /* vertex in internal tree             */
                                  /* (not attachment point)              */
             RVDBV,               /* vertex in biblock                   */
                                  /* (not attachment point)              */
             RVDAP                /* attachment point                    */
      };
struct  aedstd      /* AEDSTD line in a biblock decomposition             */
{
   AEDSTD        *aedstdleft;
   AEDSTD        *aedstdright;
   AEDSTD        *aedstdpar;
   COLOR          aedstdcolor;
   void          *aedstdauxiliary; /* for future extensions              */
   EDGE          *aedstdp;         /* pointer to edge                    */
   int            aedstdedcolor;   /* coloring for classifying           */
   int            aedstdclass;     /* class of edge/arc                  */
   void          *aedstdelcmp;     /* pointer to elementary component    */
                                   /* edge/arc belongs to                */
// int            edstdcompno;     /* Number of higher connected         */
//                                 /* components (status FINAL or OPEN)  */
//                                 /* or cocomponents the edge is        */
//                                 /* element of.                        */
// RELEM         *edstdcomplist;   /* List of higher connected components */
//                                 /* (status FINAL or OPEN) or          */
//                                 /* cocomponents the edge is           */
//                                 /* element of. The connected component */
//                                 /* is of order at least 3.            */
};

   enum edclass    /* EDCLASS classification                              */
        {      EDNY,               /* not yet classified                  */
               EDPT,               /* peripheral tree line                */
               EDIT,               /* internal tree line                  */
               EDBB                /* biblock line                        */
        };
struct  incsqrstr  /* INCSQR List of incidences                          */
{
   INCSQR      *incsqrleft;
   INCSQR      *incsqrright;
```

```
   INCSQR      *incsqrpar;
   COLOR        incsqrcolor;
   void        *incsqrauxiliary;   /* for future extensions              */
   int          incsqredno;        /* number of edges                    */
   REDGE       *incsqredlist;      /* pointer to list of edges           */
   int          incsqriedno;       /* number of incoming arcs            */
   REDGE       *incsqriedlist;     /* pointer to list of incoming arcs   */
   int          incsqroedno;       /* number of aoutgoing arcs           */
   REDGE       *incsqroedlist;     /* pointer to list of aoutgoing arcs  */
   BLB         *incsqrblb;         /* pointer to blb                     */
};
   enum poption   /* POPTION */
        {      VPV,                 /* all vertices                       */
               VPA,                 /* all attachment points              */
               VPB,                 /* all border points                  */
               VPC,                 /* all check points                   */
               VPH,                 /* all hinge ponts                    */
               VPBC,                /* all vertices which are             */
                                    /* border points and check points,    */
                                    /* but not hinge points               */
               VPBH,                /* all vertices which are             */
                                    /* border points and hinge points,    */
                                    /* but not checkpoints                */
               VPBCH,               /* all vertices which are             */
                                    /* border points, check points and    */
                                    /* hinge points                       */
               VPCH                 /* all vertices which are             */
                                    /* check points and hinge points,     */
                                    /* but not border points              */
        };

   enum stroption  /*STROPTION */
        {      STR,                 /* complete structure                 */
               STRA,                /* print attachment points,           */
                                    /* but no other vertices              */
                                    /* and no lines                       */
               STRR,                /* reduced structure: no vertices,    */
                                    /* no lines, only summary of          */
                                    /* number of attachment points        */
               STRCS,               /* print condensed statistics only    */
               STAT,                /* for partitions with paintings      */
                                    /* only: size/color statistics of     */
                                    /* classes                            */
               STATCS,              /* for partitions with paintings      */
                                    /* only: condensed size/color         */
                                    /* statistics                         */
        };
```

C.9 Data Structures for Additional DFS Structure

```
/*************************************************************************/
/*                                                                       */
/*        DFSVT       Vertex extension.                                  */
/*                                                                       */
/*        DFSED       Edge extension.                                    */
/*                                                                       */
/*************************************************************************/
struct  dfsvtstr  /* DFSVT vertex extension for dfs structure            */
{
   DFSVT       *dfsvtleft;
   DFSVT       *dfsvtright;
   DFSVT       *dfsvtpar;
   COLOR        dfsvtcolor;
   void        *dfsvtauxiliary;    /* for future extensions              */
   VERTEX      *dfsvtp;            /* pointer to vertex                  */
   int          dfslevel;          /* do not confound with gvtstdlevel!  */
                                   /* = -1: not yet assigned             */
   EDGE        *dfsiinctree;       /* incoming dfs tree arc              */
                                   /* (at most one!)                     */
   int          dfsiincforno;      /* no. of incoming dfs forward arcs   */
   INC         *dfsiincforlist;    /* list of incoming dfs forward arcs  */
   int          dfsiincbackno;     /* no. of incoming dfs backward arcs  */
   INC         *dfsiincbacklist;   /* list of incoming dfs backward arcs */
   int          dfsiinccrossno;    /* no. of incoming dfs cross arcs     */
   INC         *dfsiinccrosslist;  /* list of incoming dfs cross arcs    */
   int          dfsoinctreeno;     /* no. of outgoing dfs tree arcs      */
   INC         *dfsoinctreelist;   /* list of outgoing dfs forward arcs  */
   int          dfsoincforno;      /* no. of outgoing dfs forward arcs   */
   INC         *dfsoincforlist;    /* list of outgoing dfs forward arcs  */
   int          dfsoincbackno;     /* no. of outgoing dfs backward arcs  */
   INC         *dfsoincbacklist;   /* list of outgoing dfs backward arcs */
   int          dfsoinccrossno;    /* no. of outgoing dfs cross arcs     */
   INC         *dfsoinccrosslist;  /* list of outgoing dfs cross arcs    */
   VERTEX      *dfsroot;           /* root of corrsponding dfs tree      */
};
struct  dfsedstr  /* DFSED line extension for dfs structure              */
{
   DFSED       *dfsedleft;
   DFSED       *dfsedright;
   DFSED       *dfsedpar;
   COLOR        dfsedcolor;
   void        *dfsedauxiliary;    /* for future extensions              */
   EDGE        *dfsedp;            /* pointer to line                    */
   DFSVT       *dfsedhead;         /* head of dfs arc                    */
   DFSVT       *dfsedtail;         /* tail of dfs arc                    */
   int          dfsedtype;         /* type of dfs arc                    */
```

```
};
   enum dfsedtype   /* DFSEDTYPE */
        {      DFSNY,               /* not yet assigned                */
               DFSTREE,             /* dfs tree arc                    */
               DFSFOR,              /* dfs forward arc                 */
               DFSBACK,             /* dfs backward arc                */
               DFSCROSS             /* dfs cross arc                   */
        };
```

C.10 Data Structures for General Partitions

```
/*************************************************************************/
/*                                                                       */
/*        EDPART      Line partition.                                    */
/*                                                                       */
/*        EDCLASS     Line class of a partition.                         */
/*                                                                       */
/*        REDCLASS    Role of line class in a partition.                 */
/*                                                                       */
/*        PARTVT      Vertex in line partition.                          */
/*                                                                       */
/*        RPARTVT     Role of vertex in an line partition.               */
/*                                                                       */
/*        PARTED      Edge in a an line partition.                       */
/*                                                                       */
/*        RPARTED     Role of line in a partition.                       */
/*                                                                       */
/*************************************************************************/
```

C.11 Data Structures for Paths

```
/*************************************************************************/
/*                                                                       */
/*        PHDR      Path header.                                       */
/*                                                                       */
/*        PTHA      Arc on a path.                                     */
/*                                                                       */
/*************************************************************************/
struct  phdrstr  /* PHDR header record of a path                       */
{
   PHDR          *phleft;
   PHDR          *phright;
   PHDR          *phpar;
   COLOR          phcolor;
   void          *phauxiliary;  /* for future extensions              */
   char          *phname;       /* name of the path                   */
   GRAPH         *phgraph;      /* pointer to graph                   */
```

```
   int            phtype;  /* = -1: no type assigned yet                  */
                           /* = 0:  a-path                                */
                           /* = 1:  f-path                                */
                           /* = 2:  b-path                                */
   PTHA          *phfirsted;    /* first line of path                     */
   PTHA          *phlasted;     /* last line of path                      */
   int            phlength;     /* No of lines in path                    */
   PTHA          *phspeced;     /* special line of path                   */
                                /* for various applications               */
   PTHA          *phspeced1;    /* special line of path                   */
                                /* for various applications               */
};
struct  pthastr  /* PTHA line in path                                      */
{
   PTHA          *pthaleft;
   PTHA          *ptharight;
   PTHA          *pthapar;
   COLOR          pthacolor;
   void          *pthaauxiliary; /* for future extensions                 */
   EDGE          *pthaed;        /* line record                           */
   VERTEX        *pthafirst;     /* first vertex in path direction        */
   VERTEX        *pthasecond;    /* second vertex in path direction       */
   PTHA          *pthanext;      /* next line in path                     */
   PTHA          *pthaprevious;  /* previous line in path                 */
   PHDR          *pthahdr;       /* pointer to path header                */
};
```

C.12 Data Structures for Menger Structures

```
/*************************************************************************/
/*                                                                       */
/*        MGDESCR   Menger descriptor.                                   */
/*        SEPDESCR  Separating                                           */
/*                                                                       */
/*************************************************************************/
struct  mgdescrstr  /* MGDESCR descriptor for Menger analysis            */
{
   MGDESCR       *mgleft;
   MGDESCR       *mgright;
   MGDESCR       *mgpar;
   COLOR          mgcolor;
   void          *mgauxiliary;  /* for future extensions                 */
   char          *mgname;       /* name of the Menger structure          */
   GRAPH         *mggraph;      /* pointer to graph                      */
   int            mgmode;  /* = -1: No mode assigned yet                 */
                           /* = 0:  internally disjoint paths (vt)       */
                           /* = 1:  line-disjoint paths       (vl)       */
```

```
                          /* = 2:  externally disjoint paths (evt)       */
                          /*       in the extended graph                 */
                          /* = 3:  line-disjoint paths       (eli)       */
                          /*       in the extended graph                 */
   int            mgdir;  /* = -1: No direction  assigned yet            */
                          /* = 0:  a-paths                               */
                          /* = 1:  f-paths                               */
   VTSET         *mgvtsource;  /* vertices of vtsource                   */
   VTSET         *mgvtsink;    /* vertices of vtsink                     */
   VTSET         *mgvtcommon;  /* vertices of vtcommon                   */
   EDSET         *mgdiredges;  /* directly linking edges                 */
   EDSET         *mgdirarcs;   /* directly linking arcs                  */
// PDESCR        *mgpaths;     /* Menger paths                           */
   void          *mgattr;      /* pointer to user attribute structure    */
   int            mgattrtype;  /* type of user attribute structure       */
                               /* = -1: No attributes                    */
   int            mgbound;     /* < 0 : All Menger paths from vtsource   */
                               /*       to vtsink sre determined         */
                               /* 0 <=: At most this number of paths are */
                               /*       determined                       */
   BOOLEAN        mgsep;       /* TRUE:   All Menger vertices (lines)    */
                               /*         on all pertinent paths         */
                               /*         together with its minimal      */
                               /*         Menger separating sets are     */
                               /*         recorded                       */
                               /* FALSE:  Only the first Menger          */
                               /*         separating set (in direction   */
                               /*         from vtsource to vtsink) is    */
                               /*         recorded                       */

   SEPDESCR      *mgsepdescr;  /* Separating elements                    */

};
struct  mgsepdescrstr  /* SEPDESCR descriptor for separating elements    */
{
   SEPDESCR       *sepleft;
   SEPDESCR       *sepright;
   SEPDESCR       *seppar;
   COLOR           sepcolor;
   void           *sepauxiliary;  /* for future extensions              */
};
```

C.13 Data Structures for the Higher A-Decomposition

```
/*************************************************************************/
/*                                                                       */
/*        HSTDD      Weak and strong components              C.11.1      */
```

```
/*        HCOMP      Component or maximal vertex set in a    C.11.2       */
/*                    higher decomposition.                                 */
/*         TBP        "to be processed" records                           */
/*                                                                          */
/****************************************************************************/
struct hstdd        /* GSTDD Weak and strong components                     */
 {
   HSTDD         *hstdleft;
   HSTDD         *hstdright;
   HSTDD         *hstdpar;
   COLOR          hstdcolor;
   void          *hstdauxiliary; /* for future extensions                 */
   GRAPH         *hstdgraph;     /* pointer to graph                      */
   HVTSTD        *hvtstdlist;    /* red-black tree of HVTSTD records      */
   int            hsggano;       /* number of a-components                */
   BLB           *hsggalist;     /* list of biblocks containing           */
                                 /* a-components                          */
   int            hsggalno;      /* number of a-linecomponents            */
   SUB           *hsggallist;    /* list of subcomponents containing      */
                                 /* a-linecomponents                      */
   int            hvsgano;       /* number of maximal a-connected         */
                                 /* vertex sets                           */
   BLB           *hvsgalist;     /* list of biblocks containing           */
                                 /* maximal a-connected vertex sets       */
   int            hvsgalno;      /* number of maximal a-lineconnectged    */
                                 /* vertex sets                           */
   SUB           *hvsgallist;    /* list of subcomponents containing      */
                                 /* maximal a-lineconnected vertex sets   */
   int            hsggfno;       /* number of f-components                */
   BLB           *hsggflist;     /* list of strong components containing  */
                                 /* f-components                          */
   int            hsggflno;      /* number of f-linecomponents            */
   SUB           *hsggfllist;    /* list of strong components containing  */
                                 /* f-linecomponents                      */
   int            hvsgfno;       /* number of maximal f-connected         */
                                 /* vertex sets                           */
   BLB           *hvsgflist;     /* list of strong components containing  */
                                 /* maximal f-connected vertex sets       */
   int            hvsgflno;      /* number of maximal f-lineconnectged    */
                                 /* vertex sets                           */
   SUB           *hvsgfllist;    /* list of strong components containing  */
                                 /* maximal f-lineconnected vertex sets   */
 };

  struct  hdccomp  /* HCOMP Higher decomposition element                   */
{
   HCOMP       *hcompleft;
   HCOMP       *hcompright;
```

```
   HCOMP       *hcomppar;
   COLOR        hcompcolor;
   void        *hcompauxiliary;
   int          hcomptype;      /* type of higher component              */
                                /* = -1: not yet assigned                */
                                /* = 1:  subgraph, internally disjoint   */
                                /*       a-paths                         */
                                /* = 2:  subgraph, line-disjoint         */
                                /*       a-paths                         */
                                /* = 3:  vertex set, internally disjoint */
                                /*       a-paths                         */
                                /* = 4:  vertex set, line-disjoint       */
                                /*       a-paths                         */
                                /* = 5:  subgraph, internally disjoint   */
                                /*       f-paths                         */
                                /* = 6:  subgraph, line-disjoint         */
                                /*       f-paths                         */
                                /* = 7   vertex set, internally disjoint */
                                /*       f-paths                         */
                                /* = 8   vertex set, line-disjoint       */
                                /*       f-paths                         */
   int          hcompdegree;    /* degree of connectedness (k)           */
   int          hcompnumber;    /* number of decomposition element in    */
                                /* element of next lower degree of       */
                                /* connectedness                         */
   void        *hcomplowerel;   /* next lower decomposition element      */
                                /* of type HCOMP or                      */
                                /* -  type BLN (modes 1, 3)              */
                                /* -  type SUB (modes 2, 4)              */
                                /* -  type SCOMP (modes 5, 6, 7, 8)      */
   HSTDD       *hcomphstdd;     /* pointer main higher decomposition     */
                                /* record                                */
                                /* decomposition.                        */
   int          hcompvtno;      /* number of vertices                    */
   int          hcompedno;      /* number of edges                       */
   int          hcompdedno;     /* number of arcs                        */
   int          hcomcovtno;     /* number of vertices in coelement       */
   int          hcomcoedno;     /* number of edges in coelement          */
   int          hcomcoeddno;    /* number of arcs in coelement           */
   int          hcompnextlvno;  /* number of decomposition elements with */
                                /* next higher degree of connectedness   */
   HCOMP       *hcompnextlvli;  /* list of decomposition elements with   */
                                /* next higher degree of connectedness   */
};
  struct  hvtstd       /* HVTSTD Vertex in a higher decomposition of     */
                       /* a general graph                                */
 {
   HVTSTD        *hvtstdleft;
```

```
   HVTSTD        *hvtstdright;
   HVTSTD        *hvtstdpar;
   COLOR          hvtstdcolor;
   void          *hvtstdauxiliary;/* for future extension                */
   VERTEX        *hvtstdp;        /* pointer to vertex                   */
   int            hvtstdhcompno;  /* number of most connected            */
                                  /* decomposition elements (of a given  */
                                  /* type) vertex is contained in        */
   HCOMP         *hvtstdwcomp;    /* list of most connected decomposition */
                                  /* elements (of a given typevertex is contained in      */
   int            gvtstdtype;     /* type of vertex                   */
                                  /* = -1: not yet assigned           */
                                  /* = 0:  isolated vertex            */
                                  /* = 1:  internal vertex of         */
                                  /*       external dag (no return)   */
                                  /* = 2:  internal vertex of         */
                                  /*       strong component           */
                                  /* = 3:  weak attachment point      */
   SCOMP         *gvtstdscomp;    /* pointer to strong component      */
                                  /* = NULL: not applicable           */
   int            gvtstdexdino;   /* number of incoming external      */
                                  /* dag arcs                         */
   REDGE         *gvtstdexdilist; /* list of incoming external        */
                                  /* dag arcs                         */
   int            gvtstdexdono;   /* number of outgoing external      */
                                  /* dag arcs                         */
   REDGE         *gvtstdexdolist; /* list of outgoing external        */
                                  /* dag arcs                         */
   int            gvtstdlevel;    /* level number                     */
                                  /* = -1: Not yet assigned           */
                                  /* all vertices of a strong         */
                                  /* component have the level number  */
                                  /* of that component                */
   BOOLEAN        gvtstdmarked;   /* General mark indicator           */

};
struct  hedstd       /* HEDSTD line in a weak and strong              */
                     /* decomposition                                 */
{
   HEDSTD        *hedstdleft;
   HEDSTD        *hedstdright;
   HEDSTD        *hedstdpar;
   COLOR          hedstdcolor;
   void          *hedstdauxiliary;
   EDGE          *hedstdp;        /* pointer to line                   */
   int            hedstdtype;     /* type of line                      */
                                  /* = -1: not yet assigned            */
                                  /* = 1:  edge; gedstdelcmp points    */
```

```
                                /*       to strong component         */
                                /* = 2:  arc of a strong component;  */
                                /*       gedstdelcmp points          */
                                /*       to strong component         */
                                /* = 3:  arc of an external dag;     */
                                /*       gedstdelcmp points          */
                                /*       to weak component           */
   void          *hedstdelcmp;    /* pointer to elementary component   */
                                /* edge/arc belongs to               */
   BOOLEAN        hedstdmarked;   /* general mark indicator            */
};
```

C.14 Sort Keys for Red-Black Trees

```
/*****************************************************************************/
/*                                                                           */
/*        SORTKEY  Sorting key for red-black trees.                          */
/*                                                                           */
/*****************************************************************************/
   enum sortkey   /* SORTKEY   Type of sorting key */
{
/*       ********************************************************         */
/*       * Extension (2008/8/17): The function rbtreedelete is  *         */
/*       * implicitly added to all keys which allow function    *         */
/*       * rbtcomp.                                             *         */
/*       ********************************************************         */
/*                                                                         */
/*       ************************************************************      */
/*       * Extension (2009/12/16): All keys which allow rbtreeinsert *      */
/*       * also allow ntreeinsert. All keys which allow rbtreefind   *      */
/*       * also allow rbtreepfind, rbtreenext and rbtreeprevious     *      */
/*       ************************************************************      */
//   1.      Utilities
//   1.1     General Services
    CRON,     /* rbtreeinsert   Insertion in the order of appearance.     */
              /* Valid for any record type.  Used to contruct linear      */
              /* linked lists.                                            */
    PCLSIZE,  /* rbtcomp        RELEM records representing sizes of       */
              /* rbtreefind     partition classes. The record's           */
              /* rbtreeinsert   auxiliary field contains the size.        */
              /* getname        Comparison of the records according       */
              /*                to size.                                  */
              /*                NOTE: This sortkey may also be used with  */
              /*                other RELEM records, as long as an        */
              /*                integer value in the auxiliary field is   */
              /*                used as the primary RB key.               */
    PECLSIZE, /* rbtcomp        As PCLSIZE.                               */
```

```
            /* rbtreefind     Records are compared to a value of type   */
            /*                int. These must be specified as pointers  */
            /*                to an integer variable (type *int) to     */
            /*                avoid confusions between value 0 and      */
            /*                NULL pointer.                             */
//   1.2    Red-Black Trees
//   1.3    Stacks and Queues
//   2.     Basic Graph Functions
    GRNM,   /* rbtcomp        GRAPH compared to GRAPH                   */
            /* rbtreefind     by grname                                 */
            /* rbtreeinsert                                             */
            /* getname                                                  */
    GRNC,   /* rbtcomp        GRAPH->grname compared to char*           */
            /* rbtreefind                                               */
    SND,    /* rbtcomp        VERTEX compared to VERTEX by vtname       */
            /* rbtreefind                                               */
            /* rbtreeinsert                                             */
            /* getname                                                  */
    SNN,    /* rbtcomp        VERTEX->vtname compared to char*          */
            /* rbtreefind                                               */
    RSND,   /* rbtcomp        RVERTEX compared to RVERTEX by vtname     */
            /* rbtreefind                                               */
            /* rbtreeinsert                                             */
            /* getname                                                  */
    RSNN,   /* rbtcomp        RVERTEX->rvtp->vtname compared to char*   */
            /* rbtreefind                                               */
    RSDND,  /* rbtcomp        RVERTEX compared to                       */
            /* rbtreefind     RVERTEX by                                */
            /* rbtreeinsert   a. total degree                           */
            /*                b. vtname                                 */
    SED,    /* rbtcomp        EDGE by edname                            */
            /* rbtreefind                                               */
            /* rbtreeinsert                                             */
            /* getname                                                  */
    SEN,    /* rbtcomp        EDGE->edname compared/ to char *          */
            /* rbtreefind                                               */
    RSED,   /* rbtcomp        REDGE compared to REDGE by line name      */
            /* rbtreefind                                               */
            /* rbtreeinsert                                             */
            /* getname                                                  */
    SED1,   /* getname        Uses EDGE record. Provides line name and  */
            /*                names of end vertices in a 3 lines        */
            /*                print format                              */
    RSEN,   /* rbtcomp        REDGE->redp->edname compared              */
            /* rbtreefind     to char*                                  */
    SIN,    /* rbtcomp        INC by linename                           */
            /* rbtreefind                                               */
            /* rbtreeinsert                                             */
```

```
             /* getname                                                 */
//   3.      Sets
    SSEL,    /* rbtcomp        SETELEM compared to                      */
             /* rbtreefind     SETELEM by selname                       */
             /* rbtreeinsert                                            */
             /* getname                                                 */
    SSCEL,   /* rbtcomp        SETELEM->selname                         */
             /* rbtreefind     compared to *char                        */
//   4.      Graph Generating Functions
//   5.      Weak and Strong Components
    RGSTDD,  /* rbtcomp        Two GSTDD records are compared by the    */
             /* rbtreefind     names of the graphs they point to        */
             /* rbtreeinsert                                            */
             /* getname                                                 */
    NGSTDD,  /* rbtcomp        Name of the graph the GSTDD record       */
             /* rbtreefind     points to is compared with char *        */
    RWCOMP,  /* rbtcomp        WCOMP compared to WCOMP by wcompnumber   */
             /* rbtreefind                                              */
             /* rbtreeinsert                                            */
             /* getname                                                 */
    NWCOMP,  /* rbtcomp        wcompnumber compared to int *            */
             /* rbtreefind     Note: Do not use parameters of mode int! */
    RSCOMP,  /* rbtcomp        SCOMP compared to SCOMP by scompnumber   */
             /* rbtreefind                                              */
             /* rbtreeinsert                                            */
             /* getname                                                 */
    NSCOMP,  /* rbtcomp        scompnumber compared to int *            */
             /* rbtreefind     Note: Do not use parameters of mode int! */
    REXD,    /* getname        Name of wcomp with suffix EXD            */
    RGVTSTD, /* rbtcomp        Two GVSTD records are compared by        */
             /* rbtreefind        a. total deegree                      */
             /* rbtreeinsert      b. vertex name                        */
             /* getname                                                 */
             /*********************************************************/
             /* NOTE: RGVTSTD must be used with the pointer gvtstdlist  */
             /* in a GSTDD record, since that is the only list where    */
             /* an RB-Tree of GVTSTD records is contructed.             */
             /* To find a GVTSTD record given the name of its           */
             /* corresponding VERTEX record, search the vertex directly */
             /* and follow the pointer vtptgstd                         */
             /*********************************************************/
    RGEDSTD, /* rbtcomp        Two GEDSTD records are compared by the   */
             /* rbtreefind     names of the edges/arcs they point to    */
             /* rbtreeinsert                                            */
             /* getname                                                 */
    NGEDSTD, /* rbtcomp        Name of the vertex the GEDSTD record     */
             /* rbtreefind     points to is compared with char *        */
//   6.      Biblock Decomposition
```

```
    RSTP,     /* getname        Name of wcomp with suffix STP             */
    RSUB,     /* rbtcomp        Subcomponent sorted by subnumber          */
              /* rbtreefind                                               */
              /* rbtreeinsert                                             */
              /* getname                                                  */
    NSUB,     /* rbtcomp        subnumber compared to int *               */
              /* rbtreefind     Note: Do not use parameters of mode int!  */
    RBLB,     /* rbtcomp        Biblock sorted by blbnumber               */
              /* rbtreefind                                               */
              /* rbtreeinsert                                             */
              /* getname                                                  */
    NBLB,     /* rbtcomp        blbnumber compared to int *               */
              /* rbtreefind     Note: Do not use parameters of mode int!  */
    RELBLB,   /* rbtcomp        RELEM of BLB type sorted by blb number    */
              /* rbtreefind                                               */
              /* rbtreeinsert                                             */
              /* getname                                                  */
    RLBLBSUB, /* rbtcomp        RELEM of BLB type sorted by sub number    */
              /* rbtreefind     and by blb number within sub number       */
              /* rbtreeinsert                                             */
    NELBLB,   /* rbtcomp        RELEM of BLB; blbnumber compared          */
              /* rbtreefind     to int * (Do not use mode int!)           */
    RIT,      /* rbtcomp        Internal tree sorted by itnumber          */
              /* rbtreefind                                               */
              /* rbtreeinsert                                             */
              /* getname                                                  */
    RPT,      /* rbtcomp        Peripheral tree sorted by ptnumber        */
              /* rbtreefind                                               */
              /* rbtreeinsert                                             */
              /* getname                                                  */
    RINCSQR,  /* rbtcomp        INCSQR sorted by blbnumber                */
              /* rbtreefind                                               */
              /* rbtreeinsert                                             */
              /* getname                                                  */
    NINCSQR,  /* rbtcomp        INCSQR compared to blbnumber int *        */
              /*rbtreefind      Do not use mode int!                      */
              /* rbtreefind                                               */
//   7.      Higher A-Decomposition
//   8.      General Partitions
//   9.      Paths
    PTNM,     /* rbtcomp        PHDR compared to PHDR                     */
              /* rbtreefind     by phname                                 */
              /* rbtreeinsert                                             */
              /* getname                                                  */
    PTNC,     /* rbtcomp        PHDR->phname compared to char*            */
              /* rbtreefind                                               */
//   10.     Menger Structures
```

```
/*************************************************************************/
/****************** Not yet updated **************************************/
/*************************************************************************/
    RSLND,    /* rbtcomp        RVERTEX compared to RVERTEX by
                 rbtreefind     a. distance to center (in
                 rbtreeinsert   auxiliary field)
                                b. vtname */
    RSTAT,    /* rbtrcomp       RVERTEX compared to RVERTEX by
                 rbtreefind     a. eccentricity (in the rvtauxiliary
                                   field of the RVERTEX record)
                                b. vtname */
//    RDMND,    /* rbtcomp        RVERTEX compared to RVERTEX
//                 rbtreefind     by oldname (pointed to by
//                 rbtreeinsert   auxiliary field)
//                 getname        */
    RRVSTD,   /* rbtcomp        RVSTD by  total degree and
                 rbtreefind     vertex name
                 rbtreeinsert
                 getname        */
    RDRVSTD,  /* rbtcomp        DRVSTD by total degree and
                 rbtreefind     vertex name
                 rbtreeinsert
                 getname        */
    PPARTVT,  /* rbtcomp        PARTVT compared to PARTVT by
                 rbtreefind     vertex name
                 rbtreeinsert
                 getname        */
    PEPARTVT, /* rbtrcomp       PARTVT compared to char *
                 rbtreefind     */
    PRPARTVT, /* rbtcomp        RPARTVT compared to
                 rbtreefind     RPARTVT by vtname
                 rbtreeinsert
                 getname        */
    PERPARTVT,/* rbtcomp        RPARTVT comparted to char *
                 rbtreefind     */
//  RSLED,    /* rbtcomp        REDGE compared to REDGE by
//               rbtreefind     a. level number
//               rbtreeinsert      (in auxiliary field)
//                              b. edname */
    EEDSTD,   /* rbtcomp        EDSTD by line name
                 rbtreefind
                 rbtreeinsert
                 getname        */
    RREDSTD,   /* rbtreecomp    EDSTD by
                                 a. edstdcompno
                                 b. line name */
    SSOINC,   /* rbtcomp        SOINC compared to SOINC by
```

```
                rbtreeinsert      a. depth of subtree
                                  b. breadth of subtree
                                  c. weight of subtree (vtno + edno)
                                  d. outdegree of end vertex of arc
                                  d. name of line */
    PPARTED,  /* rbtcomp         PARTED compared to PARTED
                 rbtreefind
                 rbtreeinsert
                 getname          */
    PEPARTED, /* rbtcomp         PARTED compared to char *
                 rbtreefind       */
    PRPARTED, /* rbtcomp         RPARTED compared to RPARTED
                 rbtreefind
                 rbtreeinsert
                 getname          */
    PERPARTED,/* rbtcomp         RPARTED compared to char *
                 rbtreefind       */
    CCFR,     /* rbtcomp         CFR sorted by
                 rbtreefind       a. number of vertices
                 rbtreeinsert     b. smallest line name
                 getname          */
    WCCFR,    /* rbtcomp         WCFR sorted as CCFR
                 rbtreefind
                 rbtreeinsert
                 getname          */
    SCCFR,    /* rbtcomp         SCFR sorted as CCFR
                 rbtreefind
                 rbtreeinsert
                 getname          */
    MCFR,     /* getname         Minimum line name of cfr */
    CCLC,     /* rbtcomp         CLC sorted by
                 rbtreefind       a. number of vertices of CLC
                 rbtreeinsert     b. number of lines of CLC
                 getname          c. number of vertices of stopfree kernel
                                  d. number of line of stopfree kernel
                                  e. number of peripheral trees
                                  f. number of subcomponents
                                  g. number of internal trees
                                  h. name of 'largest' biblock */
    WCCLC,    /* rbtrcomp        WCLC sorted as CCLC
                 rbtreefind
                 rbtreeinsert
                 getname          */
    SCCLC,    /* rbtrcomp        SCLC sortes as CCLC
                 rbtreefind
                 rbtreeinsert
                 getname          */
    IIT,      /* rbtcomp         Internal tree sorted by
```

```
                rbtreefind     a. number of vertices
                rbtreeinsert   b. smallest line name
                getname        */
    IINCSQR,  /* rbtcomp        Cronological order
                rbtreefind
                rbtreeinsert   */
//    RREL,     /* rbtcomp        List of substructures sorted by
//                  rbtreefind     a. ordering given in RELEMCLASS
//                  rbtreeinsert   b. order of substructure.
//                                    (not yet implemented for
//                                     all substructures)
//                                 See also RPCLASS  and
//                                 RREDSTD */
    PCLASS,   /* rbtcomp        Partition classes ordered by class
                rbtreefind     name. The names are not ordered
                rbtreeinsert   lexicographically, but numerically
                getname        */
    PECLASS,  /* rbtcom         EDCLASS by classname (*char)
                rbreefind      */
    PRCLASS,  /* rbtcomp        Partition classes ordered by class
                rbtreefind     name. The names are not ordered
                rbtreeinsert   lexicographically, but numerically
                getname        */
    PRECLASS, /* rbtcom         REDCLASS by classname (*char)
                rbreefind      */
    RPCLASS,  /* rbtcomp        Partition classes -- represented by
                rbtreefind     RELEM records - ordered by
                rbtreeinsert   class name.
                getname        */
    SSTCS,    /* rbtcomp        STCS compared to
                rbtreefind     STCS
                rbtreeinsert
                getname        */
    SSCTCS,   /* rbtcomp        STCS
                rbtreefind     compared to *char */

    HHDC,     /* rbtcomp        HCOMP compared to
                rbtreefind     HCOMP by HCOMP
                rbtreeinsert   name.
                getname        */
/* Note: The name of a HCOMP record is a character string
   of a hierarchical structure. It consists of two or more
   substrings separated by one or more blancs. The first
   substring is the biblock name. Then a
   (possibly empty) sequence of component/candidate numbers
   In case of a candidate, the number is inmediatly
   followed by the string CAND.
   A cocomponent or cocandidate has no number but
```

```
   the string C as name.
   The component numbers are
   noted as strings but have numerical sort order. E.g.
          nameofbiblock 1 3 12
   is greater than
          nameofbiblock 1 3 4.   */
    TTBP,     /* getname        for name purposes only */

    NHDC     /* rbtcomp        HCOMP compared to
                                *char */
        };
#endif
```

List of Figures

List of Tables

Bibliography

Pages where cited are in parentheses.

[CharL1996] Chartrand, Gary · Lesniak, Linda. *Graphs & Digraphs*. Chapman & Hall, 3 edition, 1996. *(51)*

[CormLR1990] Cormen, Thomas. H · Leiserson, Charles E. · Rivest, Ronald L. *Introduction to Algorithms*. The MIT Electrical and Computer Science Series. The MIT Press / McGraw-Hill Book Company, 1990. *(115, 119, 137)*

[Hara1969] Harary, Frank. *Graph Theory*. Addison-Wesley Series in Mathematics. Addison-Wesley Publishing Company, 1969. *(51, 61)*

[HopcT1973] Hopcroft, John E. · Tarjan, Robert Endre. Efficient Algorithms for Graph Manipulation. *Communications of the ACM*, 16(6):372–378, 1973. *(61)*

[Knut1993] Knuth, Donald E. *The Stanford GraphBase*. Addison-Wesley Publishing Company, 1993. A Platform for Combinatorial Computing. *(4, 124)*

[Stie1998] Stiege, Günther. Edge Partitions in Undirected Graphs. Berichte aus dem Fachbereich Informatik 5/98, Universität Oldenburg, 1998. Available from *http://www-bvs.informatik.uni-oldenburg.de/Literatur/Berichte/oib98-05.html*. *(81)*

[Stie2006] Stiege, Günther. *Graphen und Graphalgorithmen*. Reihe Informatik. Shaker Verlag, 2006. *(51, 53, 61, 94, 99)*

[Stie2007a] Stiege, Günther. General Graphs. Berichte aus dem Department für Informatik 02/07, Universität Oldenburg, 2007. Available from *http://www-bvs.informatik.uni-oldenburg.de/Literatur/Berichte/oib07-02.html*. *(51, 53, 61, 94, 99)*

[Stie2009] Stiege, Günther. *Einführung in die Informatik*. Reihe Informatik. Shaker Verlag, 2009. to appear. *(115)*

[Tarj1972] Tarjan, Robert Endre. Depth-First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2):215–225, 1972. *(61)*

[Volk1996] Volkmann, Lutz. *Fundamente der Graphentheorie*. Springer Lehrbuch Mathematik. Springer-Verlag, 1996. *(51)*

Index

V

W